Exemplar Programming Guide

for HP-UX Systems

First Edition

HEWLETT® PACKARD

B6056-96002

Fortran 90, Fortran 77, C, aC++

Customer Order Number: B6056-90002

December 1997

Revision History

Edition: First

Document Number: B6056-90002
Remarks: This is the first edition of this book for HP-UX platforms.

Notice

Contents

Figures

Tables

How to use this guide

Purpose and audience

This guide describes efficient methods for shared-memory programming using an Exemplar compiler: HP Fortran 90, HP aC++ (ANSI C++), HP Fortran 77, or HP C on computers running the HP-UX operating system. The first four chapters cover basic concepts, including automatic optimizations and simple manual optimizations that require minimal programmer intervention. In the following chapters, more progressive topics are covered, including advanced manual optimizations and the Compiler Parallel Support Library.

The *Exemplar Programming Guide* is for experienced Fortran 90, Fortran 77, C, and C++ programmers. Readers need not be familiar with the Exemplar parallel architectures, programming model, or optimization concepts; this book addresses these topics in the necessary detail.

Scope

This guide covers programming methods for the following Exemplar compilers on V2200 (or V-Class) and K-Class machines running HP-UX 11.0 and higher:

- Exemplar HP Fortran 90 Version 2.0 (and higher)
- Exemplar HP aC++ Version 1.0 (and higher)
- Exemplar HP Fortran 77 Version 1.2.3 (and higher)
- Exemplar HP C Version 1.2.3 (and higher)

The Exemplar compilers are the same as the standard HP compilers but also support the Exemplar programming model.

HP-UX 11.0 and higher includes the required assembler, linker, and libraries. These utilities are also included in SPP-UX Version 5.2 and higher. SPP-UX is the operating system on X2000 servers. (X2000 servers are also known as X-Class servers. Machines running SPP-UX are used to discuss multidimensional parallelism.)

This guide is concerned with producing programs that efficiently exploit the features of Hewlett-Packard Exemplar architectures and the compilers that run on them. Producing an efficient program requires efficient algorithms and implementation. The techniques of writing an efficient algorithm are beyond the scope of this guide. This guide assumes that you have chosen the best possible algorithm for your problem and helps you obtain the best possible performance from that algorithm.

Suggested reading order

This book takes the following approach to presenting information.

- Chapters 1, 2 and 3 provide background information that helps you understand Exemplar architectures and how HP compilers optimize your code.
- Chapter 4 tells you how to derive performance gains with minimal intervention.
- Chapters 5 and 6 explain how to use more advanced programming techniques to further improve performance.
- Chapter 7 discusses message passing on HP-UX machines, requiring even more manual intervention.
- Chapter 8 presents coding tips and tells you about problems you may encounter when using the techniques of the previous chapters and how to enable even more aggressive optimizations.
- The appendixes contain mostly reference information, including a discussion of the Compiler Parallel Support Library (CPSlib).

If you are interested in a general, comprehensive overview of programming for Exemplar servers, read the chapters in order.

If you are interested in simply compiling existing programs and getting them to run with minimal effort, start with chapters 3 and 8. Following the cross-references that interest you will probably expose you to as much of the rest of the book as is necessary.

If you are interested in getting maximum performance gains for minimum programming effort, read chapters 3 and 4, then proceed if necessary.

If you are willing to spend some time adding directives and rewriting some of your code to realize significant performance benefits (especially if your Exemplar server is equipped with multiple hypernodes), read at least chapters 2 through 6.

If you are interested in running message-passing codes on your Exemplar system, refer to Chapter 7, “Message-passing programming.” You may also want to read chapters 2 through 6 to see how the compilers can help you with automatic optimizations.

If you are interested in very low-level control over parallelism using the Compiler Parallel Support Library, start with Appendix F. Again, you may want to refer to the other chapters to see how the compiler can help with automatic optimizations.

Notational conventions

This section discusses notational conventions used in this book.

General conventions

In general, the following conventions are used in this guide:

- Fortran

  The term “Fortran” refers to both Fortran 90 and Fortran 77. When functionality differs between the two compilers, the terms “Fortran 77” and “Fortran 90” will be used.

- *Italic*

  – Designates user-supplied variables in a command line or code example

  – Introduces new and important terms

  – Identifies variables in mathematical equations

  – Indicates document titles

- `Constant-width font` designates input and output, including

  – Command names and options

  – System calls

  – Data structures and types

  – Variables and arrays

  – Function and subroutine names

  – Directives, program statements, display examples, printout examples, and error messages returned

  Except where noted, the directives and pragmas described in this book can be used with the Fortran 90, Fortran 77 and C compilers. (The aC++ compiler does not support the pragmas, but does support the memory classes.) In general discussion, these directives and pragmas are presented in lowercase type, but each compiler will recognize them regardless of their case.

- **`Bold constant-width font`** designates text that must be input by the user.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipsis shows that lines of code have been left out of an example.

References to man pages appear in the form mnpgname(1), where “mnpgname” is the name of the man page and is followed by its section number enclosed in parentheses. To view this man page, type:

```
% man 1 mnpgname
```

NOTE A Note highlights important supplemental information.

Command syntax

Consider this example:

`COMMAND` *input_file* `[...] {a | b} [`*output_file*`]`

`COMMAND` must be typed as it appears.

input_file indicates a file name that must be supplied by the user.

The horizontal ellipsis in brackets indicates that additional, optional input file names may be supplied.

Either `a` or `b` must be supplied.

[*output_file*] indicates an optional file name.

Associated documents

Hewlett-Packard Company provides the following documents to help you use the compilers and associated tools:

- *Fortran 90 Programmer's Guide* (B6056-90003)—Provides extensive usage information (including how to compile and link), suggestions and tools for migrating to HP Fortran 90, and how to call C and HP-UX routines for HP Fortran 90.
- *Fortran 90 Programmer's Reference* (B5876-90001)—Presents complete Fortran 90 language reference information. It also covers compiler options, compiler directives, and library information.
- *HP aC++ Online Programmer's Guide* (This manual is accessed by specifying `aCC` with the `+help` command-line option.)—Presents reference and tutorial information on aC++.
- *HP MPI User's Guide* (B6011-90001)—This book discusses message-passing programming using Hewlett-Packard's Message-Passing Interface library.
- *Programming with Threads on HP-UX* (B2355-90060)—Discusses programming with POSIX threads.
- *Exemplar C and Fortran 77 Programmer's Guide for HP-UX Systems* (B6057-90002)—Describes the extensions to the standard Hewlett-Packard compilers in the Exemplar C and Fortran 77 compilers that support the Exemplar programming model.

- *HP C/HP-UX Reference Manual* (92453-90024)—Presents reference information on the C programming language, as implemented by Hewlett-Packard.
- *HP C/HP-UX Programmer's Guide* (92434-90002)—Contains detailed discussions of selected C topics.
- *FORTRAN/9000 Programmer's Reference* (B3906-90002)—Presents information on Hewlett-Packard Fortran 77 and can be used as a language reference.
- *FORTRAN/9000 Programmer's Guide* (B3906-90001)—Describes features and requirements in terms of the tasks a Fortran 77 programmer might perform. These tasks include how to compile, link, run, debug, and optimize programs.
- *Programming on HP-UX* (B2355-90652)—Describes how to develop software on HP-UX, using the HP compilers, assemblers, linker, libraries, and object files.
- *Managing Systems and Workgroups* (B2355-90157)—Describes how to perform various system administration tasks.

Ordering documentation

To order additional copies of this document or other documents listed in the "Associated documents" section, call 1-800-227-8164 between 6 a.m. and 5 p.m. PST.

To place an order from outside the United States, or if you cannot use the 1-800 number, call 415-857-5027.

Please have the order number (xxxxx-9xxxx) and the exact title of the document available when ordering.

1 Introduction

Hewlett-Packard compilers generate efficient parallel code with little intervention on your part; however, you can increase this efficiency by using the techniques discussed in this book.

This chapter provides a general overview of the:

- Exemplar architectures as compared to other parallel architectures
- Applicable programming models
- HP compiler optimizations

Exemplar SMP architectures

Hewlett-Packard offers single-processor systems and symmetric multiprocessor (SMP) systems. The SMP systems, known as Exemplar servers, can be either nonscalable or scalable systems. The remainder of this section discusses the scalability of SMPs.

Nonscalable SMPs

Hewlett-Packard's nonscalable SMPs are single-hypernode systems. (For nonscalable SMPs, a *hypernode* is simply the set of processors and physical memory.) Memory is shared among all the processors, with a bus serving as the interconnect. The shared-memory architecture has a uniform access time from each processor. For example, D-Class servers are nonscalable SMPs.

Scalable SMPs

HP's scalable Exemplar systems implement parallel processing using scalable parallel processing technology. Scalable parallel machines can be scaled to meet your specific needs. Current configurations range from one to four hypernodes (or nodes), with the system having from 4 to 64 processors.

Processors communicate with each other, with memory, and with I/O devices via a nonblocking crossbar on each hypernode for intrahypernode communication and eight high-speed CTI rings that link the hypernodes together for interhypernode communication. (CTI stands for Coherent Toroidal Interconnect.) The CTI ring design is derived from the IEEE standard 1596-1992, SCI (Scalable Coherent Interface), but the Exemplar implementation sacrifices complete SCI compatibility to provide lower latencies.

Physical memory is also scalable. V2200 and X2000 servers support up to 16 Gbytes of memory.

Each process on an HP-UX 11.0 system can access a 16-terabyte (Tbyte) virtual address space.

Exemplar vs. vector/parallel architectures

Scalable parallel processing represents a departure from traditional vector/parallel supercomputers like the Convex C Series. The C Series architecture is used to illustrate the difference between traditional and Exemplar architectures below, but the same differences apply in principle to all vector/parallel machines.

Architectural differences

Convex C Series machines contain a limited number (1-8) of custom processors connected by a high-speed crossbar to a large, shared memory. For connecting small numbers of processors such as these to memory, crossbars are cost-effective and fast, allowing all processors to access all memory with equally high speed. Each processor is equipped with one or more vector units that speed loop computations involving arrays by performing array arithmetic on up to 128 elements per vector instruction. Machines containing multiple processors can further reduce time-to-solution by adding parallelism at the process, loop, and task level.

The Exemplar architectures take a different approach. Rather than using vector units to exploit fine-grained parallelism, the processors in an Exemplar server speed scalar processing by using a reduced set of high-speed instructions coupled with pipelining, high-speed instruction and data caches, and a large register set.

Two-dimensional parallelism, which can benefit nested parallel structures, is also possible on multihypernode Exemplar servers. Rather than implementing the first dimension in the vector unit and the second across processors (as in C Series), Exemplar servers can implement the first level within a hypernode and the second across hypernodes. Single-dimensional parallelism that spans hypernodes can also be implemented.

Memory

Because of the potentially large number of processors available on a multihypernode Exemplar server, memory access via a system-wide crossbar is not practical. Instead, low latency, high-bandwidth memory access is provided by shared memory. In this model, physical memory is distributed among all hypernodes, and the entire virtual address space of a process is accessible by every processor. Processors within a hypernode can access hypernode-local memory via the crossbar regardless of whether the address space is on one or more hypernodes; memory in another hypernode can be accessed via the CTI rings. Of course, interhypernode accesses take longer than intrahypernode accesses. However, part of every hypernode's memory is dedicated to act as a CTIcache, which holds copies of recently used data from other hypernodes. These CTIcaches and the processor caches are *coherent*, meaning that when a thread references a data item via its virtual address, the value it receives will be the most recently-assigned value. By holding frequently referenced data close to its referencing processes, regardless of the actual memory location of the data, these caches provide excellent data distribution.

Optimizing compilers

Programs that optimize well on traditional vector/parallel machines optimize well on Exemplar systems with little manual intervention. Exemplar compilers automatically exploit opportunities for parallelism and data localization in programs written for shared-memory machines. Chapters 3 through 6 discuss manual optimizations that can yield even more performance from such programs.

HP SMP architectures vs. clustered workstations

While the Exemplar architectures use the same processors found in HP workstations, the following features sharply distinguish the Exemplar servers from clustered workstations:

- Exemplar architectures' low-latency shared memory
- Automatic optimizing compilers
- High-speed interconnections
- Shared peripherals
- User-configurability

The subsections below discuss each distinguishing feature in detail.

Memory

Each workstation in a cluster has its own private memory; there is no shared memory. That is, any data shared among processors must be passed over the low-performance network that connects them. While an Exemplar server can support this method of programming, it offers the many advantages of shared memory, as described in the "Exemplar vs. vector/parallel architectures" section on page 2.

Many workstation operating systems reserve a large amount of memory for system use, restricting user processes to what is left. The HP-UX operating system requires only a small fraction of each processor's memory, leaving a large majority of it for user processes, whether they are using shared memory or message passing.

Optimizing compilers

Programs for clustered workstations are compiled using the workstations' compilers. If the cluster contains workstations that require different executables (that is, if it is a *heterogeneous* cluster), the programmer must generate the executables using the proper compiler. Homogeneous clusters eliminate this requirement, but automatic parallelization is nevertheless unavailable on any type of cluster. The compilers used may generate efficient code for each processor, but any parallelism or process coordination must be explicitly implemented by the programmer via message passing.

Exemplar compilers provide fully automatic parallelism and several new data localization optimizations designed to improve memory usage and aid parallelization. Additionally, directives allow you to further enhance the automatic optimizations performed on your shared-memory program.

Exemplar compilers give the highest performance—with little or no programmer intervention—for generic programs that exploit shared memory. Message-passing programs, with their parallelism explicitly coded, also benefit from Exemplar compiler optimizations.

Interprocess communication

To communicate among themselves or access each other's data, processors in a cluster of workstations must communicate over low-performance networks and access distributed memory. Communication can be handled only by passing explicit messages between workstations over the network; because of the distributed memory and absence of parallelizing compilers, programmers must explicitly code parallelism. Parallel tasks running on clusters, then, must be fairly autonomous to avoid wasting time waiting for data or synchronization instructions to travel over the network. Clusters are best suited to coarse-grained parallelism, such as that possible at the process level, or to manually parallelizable algorithms that contain a large ratio of computation to communication. In these cases, task chunks or processes and their data are parcelled out to underused workstations, run to completion, and the results are sent back to the parent.

Fine-grained, loop-level parallelism is difficult to efficiently perform on clusters because of the need for frequent data accesses and synchronization.

Exemplar servers are suitable for both coarse- and fine-grained parallelism. Programs containing potential parallelism, when compiled with Exemplar compilers, automatically exploit the parallelism, spawning threads to run on as many processors as are available and rejoining these threads upon completion. This fine-grained parallelism takes full advantage of the fully coherent memory caches and high-speed interconnects available on an Exemplar system.

While message passing is supported and can be used to speed certain applications (refer to Chapter 7, “Message-passing programming”), with shared memory, it is not necessary for most programs. When message passing is used on an Exemplar server, the high-speed interconnects can give a substantial performance increase over traditional networks. This makes message-passing programs that exploit finer-grained parallelism practical.

HP-UX automatically schedules threads within a hypernode to execute on idle and underused processors as necessary. This ensures a balanced machine load and exploits both thread- and process-level parallelism.

Peripherals

Peripheral devices connected to an Exemplar server can be accessed from any processor on the machine. On clustered workstations, peripherals are processor-dependent. Programs running on Exemplar systems, therefore, have access to potentially greater mass storage space.

Configurability

In terms of configuring hardware, adding processors to a cluster can actually degrade performance because of the low-performance network and private memory. The network can present a bottleneck when parallelism increases to exploit the new processors; to overcome this, coarser granularity can be used—and this can require more private memory than the processors can address. The absolute performance of an Exemplar server, on the other hand, increases unhindered by a traditional network or private-memory limits. Adding peripherals and memory to an Exemplar server can also provide improved absolute performance, because all processors can access both, whereas memory and peripherals are processor-specific on clusters.

Exemplar programming model

The Exemplar programming model provides three perspectives from which a programmer can write (or adapt) code to run on an Exemplar system. Those perspectives are the shared-memory, message-passing, and shared-memory/message-passing hybrid paradigms. This book focuses on using the shared-memory paradigm but also provides some information on the other two paradigms.

The shared-memory paradigm

In the shared-memory paradigm, the compilers handle optimizations, and, if requested, parallelization. Numerous compiler directives and pragmas (discussed in detail in Chapter 4, “Basic shared-memory programming,” and Chapter 5, “Memory classes,” and listed in Appendix B, “Exemplar compiler directives and pragmas”) are available to further increase optimization opportunities.

Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” cover shared-memory programming in detail.

The message-passing paradigm

Hewlett Packard has implemented a version of the MPI standard known as HP MPI. This version is finely tuned for HP technical servers.

Under the message-passing paradigm, the programmer uses functions to explicitly spawn parallel processes, share data among them, and coordinate their activities. There is no shared memory; each process has its own private 16-terabyte (Tbyte) address space, and any data that must be shared must be explicitly passed between processes.

Support of message passing allows programs written under this paradigm for distributed-memory machines to be easily ported to HP servers. Programmers familiar with message passing may choose to write new programs using this paradigm rather than shared memory and can realize a substantial performance boost over conventional message-passing machines, even when coding finer-grained parallelism.

The few programs that require more per-process memory than possible using shared memory will benefit from the manually-tuned message-passing style.

For more information, see Chapter 7, “Message-passing programming” or the book *HP MPI User’s Guide*.

Message-passing/shared-memory hybrids

Some programs may benefit from combining the paradigms to allow several shared-memory processes to coordinate their activities via message passing. This model allows the majority of the program to be written in the familiar shared-memory style while exploiting the process-private memory benefits of message passing.

Overview of Exemplar optimizations

Exemplar compilers perform a range of user-selectable optimizations. These optimizations, which are specified via compiler command-line options, are briefly introduced here. A more thorough discussion, including the options associated with each, is given in Chapter 3, “Compiler optimizations.”

Basic scalar optimizations

Basic scalar optimizations improve performance at the basic block and program unit level.

A basic block is a sequence of statements that has a single entry point and a single exit. Branches do not exist within the body of a basic block. A program unit is a subroutine, function, or `main` program in Fortran or a function (including `main`) in C; program units are also often generically referred to as procedures. Basic blocks are contained within program units; program unit-level optimizations span basic blocks.

To improve performance, basic scalar optimizations:

- Fully exploit the processor's functional units and registers
- Reduce the number of times memory is accessed
- Simplify expressions
- Eliminate redundant operations
- Replace variables with constants
- Replace slow operations with faster equivalents

Advanced scalar optimizations

Advanced scalar optimizations are primarily intended to maximize processor data cache usage. This is referred to as data localization. Concentrating on loops, these optimizations strive to encache the data most frequently used by the loop and keep it encached so as to avoid costly memory accesses.

Advanced scalar optimizations include several loop transformations; many of them either facilitate more efficient strip mining or are performed on strip mined loops to optimize processor data cache usage. All of these optimizations are covered in Chapter 3, “Compiler optimizations.”

Advanced scalar optimizations implicitly include all basic scalar optimizations.

Parallelization

Through parallelization you can realize the full power of a scalable parallel computer like the Exemplar servers. Parallelization allows a program to be executed by as many processors as are available within its system, in most cases significantly reducing time-to-solution. Exemplar compilers can automatically locate and exploit loop-level parallelism in most programs, and, using the techniques described in Chapter 5, “Memory classes,” you can assist the compilers in finding even more parallelism in your programs.

Loops that have been data-localized are prime candidates for parallelization; individual iterations of inner loops that contain strips of localizable data can be parcelled out among several processors and run simultaneously. The maximum number of processors that can be used is limited by the number of iterations of the outer loop, and, of course, by processor availability.

While most parallelization is done on nested, data-localized loops, other code can also be parallelized. For example, through the use of manually inserted compiler directives, sections of code outside of loops can also be parallelized.

Parallelization optimizations implicitly include all scalar optimizations.

2 Architecture overview

This chapter provides an overview of Hewlett-Packard's multiprocessor architectures in terms of scalability. HP servers employ either a nonscalable or scalable SMP architecture. These overviews focus on the information most useful for programmers.

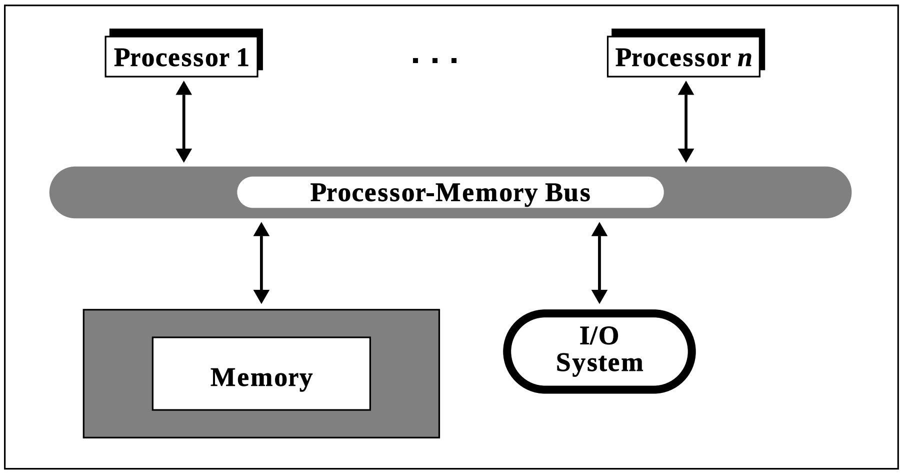
For more information on V2200 servers, see the *V-Class Architecture* manual (order number A3725-90004).

Although HP-UX is not used on X2000 servers, information on that architecture is provided throughout this book to facilitate the discussion of the Exemplar programming model on multinode SMPs. For additional information on X2000 servers, see the *Exemplar Architecture: S-Class and X-Class Servers* manual (order number A4716-90001).

System organization: nonscalable SMPs

Hewlett-Packard’s nonscalable SMPs are single-node, shared-memory machines that have a single level of memory latency. Processors communicate with each other, with memory, and with peripherals via a bus. Figure 1 gives an overview of a nonscalable SMP.

Figure 1 **Nonscalable SMP overview**



NOTE Although, V-Class servers are single-node machines, they are considered scalable SMPs because the memory bandwidth on V-Class servers, which use crossbar interconnects, is significantly greater than that on other single-node servers.

Memory

Memory is discussed in terms of physical memory and virtual memory. The following two sections describe these types of memories for nonscalable SMPs.

Physical memory

Memory configurations on HP’s nonscalable SMPs varies widely by machine. However, each of these machines uses memory interleaving to improve performance. For an explanation, see the section “Interleaving” on page 33.

HP-UX 11.0 provides variable-sized pages to improve performance. For more information on this feature, see the section “Variable-sized pages” on page 37.

Virtual memory

Virtual memory is divided into five classes. For nonscalable SMPs, only two of these classes are needed: `thread_private` and `node_private`. The three remaining classes are automatically mapped to the `node_private` class.

NOTE For applications that will be ported to Hewlett-Packard scalable SMPs, all five virtual memory classes can be useful. For information on using the memory classes on a scalable SMP, see the section “Virtual memory” on page 24.

A brief description of the virtual memory classes follow:

`thread_private`

This memory is private to each thread of a process. A `thread_private` data object has a unique virtual address for each thread. These addresses map to unique physical addresses in physical memory. Threads access the physical copies of `thread_private` data when they access `thread_private` virtual addresses.

`node_private`

This memory is shared among the threads running on a hypernode. (For nonscalable SMPs, a *hypernode* is the set of processors and physical memory.) Data objects of the class `node_private` have a single virtual address by which they can be accessed from any processor in the hypernode.

`near_shared`

This memory class is mapped to the `node_private` memory class for nonscalable SMPs.

`far_shared`

This memory class is mapped to the `node_private` memory class for nonscalable SMPs.

`block_shared`

This memory class is mapped to the `node_private` memory class for nonscalable SMPs.

Data caches

Hewlett-Packard systems use caches to enhance performance. Cache sizes, as well as cache line sizes, vary with the processor used. Data is moved between the cache and memory using *cache lines*. A cache line describes the size of a chunk of contiguous data that must be copied into or out of a cache in one operation.

When a processor experiences a cache miss—that is, requests data that is not already encached—the cache line containing the address of the requested data is moved to the cache. This cache line also contains a number of other data objects that were not specifically requested.

One reason cache lines are employed is to allow for *data reuse*. Data in a cache line is subject to reuse if, while the line is encached, any of the data elements contained in the line besides the originally requested element are referenced by the program, or if the originally requested element is referenced more than once.

Because data can only be moved to and from memory as part of a cache line, both load and store operations cause their operands to be encached. Cache-coherency hardware invalidates cache lines in other processors when they are stored to by a particular processor. This indicates to other processors that they must load the cache line from memory the next time they reference its data.

For information on avoiding inefficient use of data, see the section “Cache thrashing” on page 29.

System organization: scalable SMPs

NOTE HP-UX is the operating system on V2200 servers. Only the SPP-UX operating system runs on Hewlett-Packard X2000 servers.

Think of a scalable Exemplar SMP as a shared-memory computer with two levels of memory latency. Memory available on the current hypernode (accessed through the crossbar) constitutes the first level, and all other memory (accessed through the CTI rings) constitutes the second.

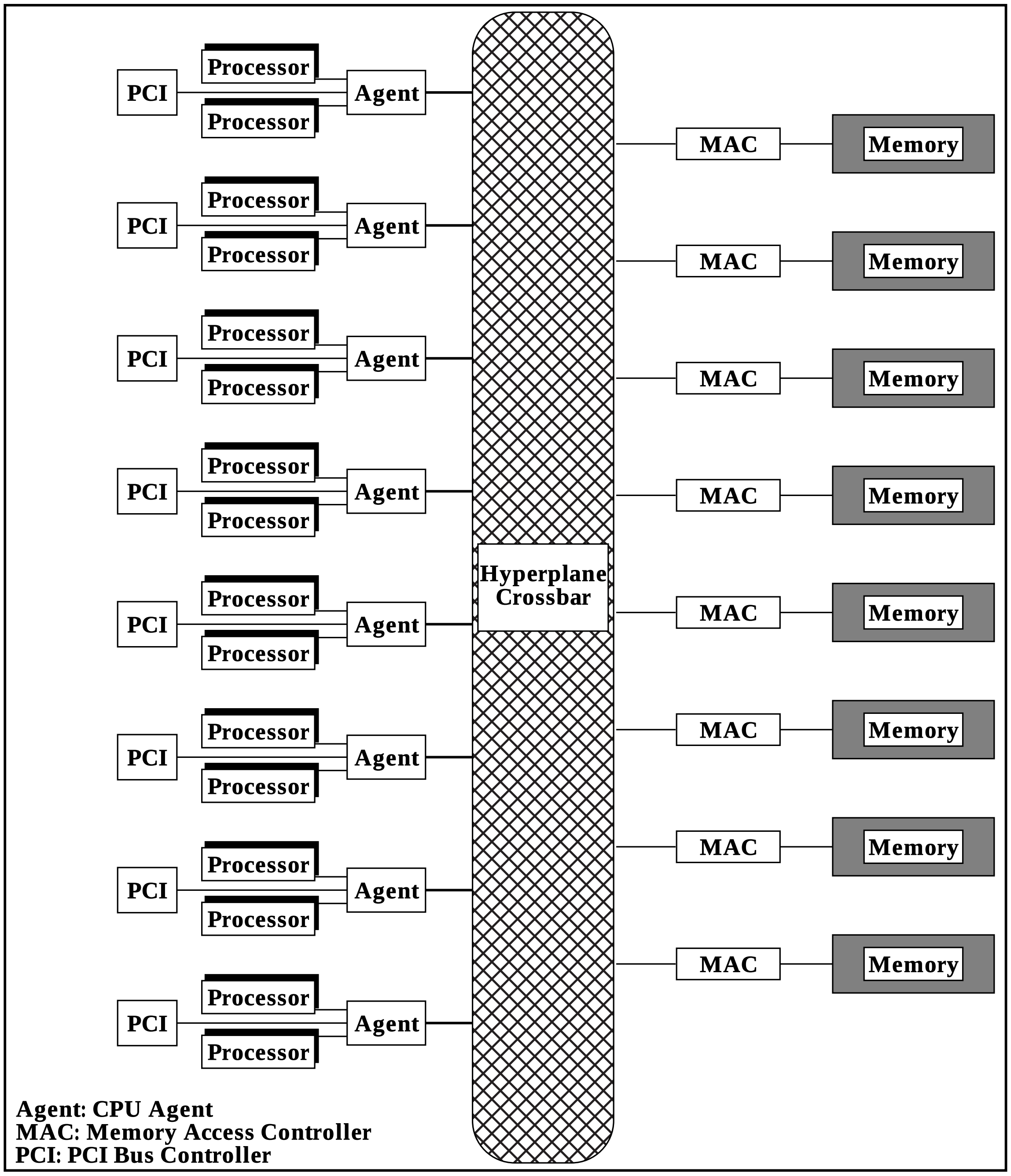
Exemplar V2200 servers consist of one hypernode that has 4 to 16 PA-8200 processors and 256 Mbytes to 16 Gbytes of physical memory. The X2000 servers consist of 1 to 4 hypernodes with a total of 16 to 64 PA-8000 processors and 16 Gbytes to 64 Gbytes of memory.

Processors within a hypernode communicate with each other, with memory, and with peripherals via a nonblocking crossbar. V2200 servers feature the HP HyperPlane crossbar. Figure 2 shows the V2200 crossbar configuration. Figure 3 shows the crossbar configuration for X2000 servers. Processors in different hypernodes communicate via CTI rings. These rings are configured in a one-dimensional interconnect for X2000 systems consisting of two or three hypernodes and in two-dimensional interconnects for systems with four hypernodes.

Figure 2 and Figure 3 show overviews of a V2200 hypernode and a single X2000 hypernode, respectively. These servers are the same except for the V2200's HyperPlane crossbar and the X2000's CTI controllers. Two CPUs and a PCI bus controller share a single CPU agent. The CPUs communicate with the rest of the machine through the CPU agent. The Memory Access Controllers (MACs) provide the interface between the memory banks and the rest of the machine. All intrahypernode memory accesses take approximately 510 nanoseconds on X2000 servers, regardless of location, because they must traverse the crossbar, which gives equal access to all hypernode memory from all CPUs. The CTIrings are used for internode communication.

Figure 4 shows a more detailed view of the connections between the CPU agents, the crossbar, and the Memory Controllers in an X2000 server.

Figure 2 V2200 hypernode overview



Agent: CPU Agent
MAC: Memory Access Controller
PCI: PCI Bus Controller

Figure 3 **X2000 hypernode overview**

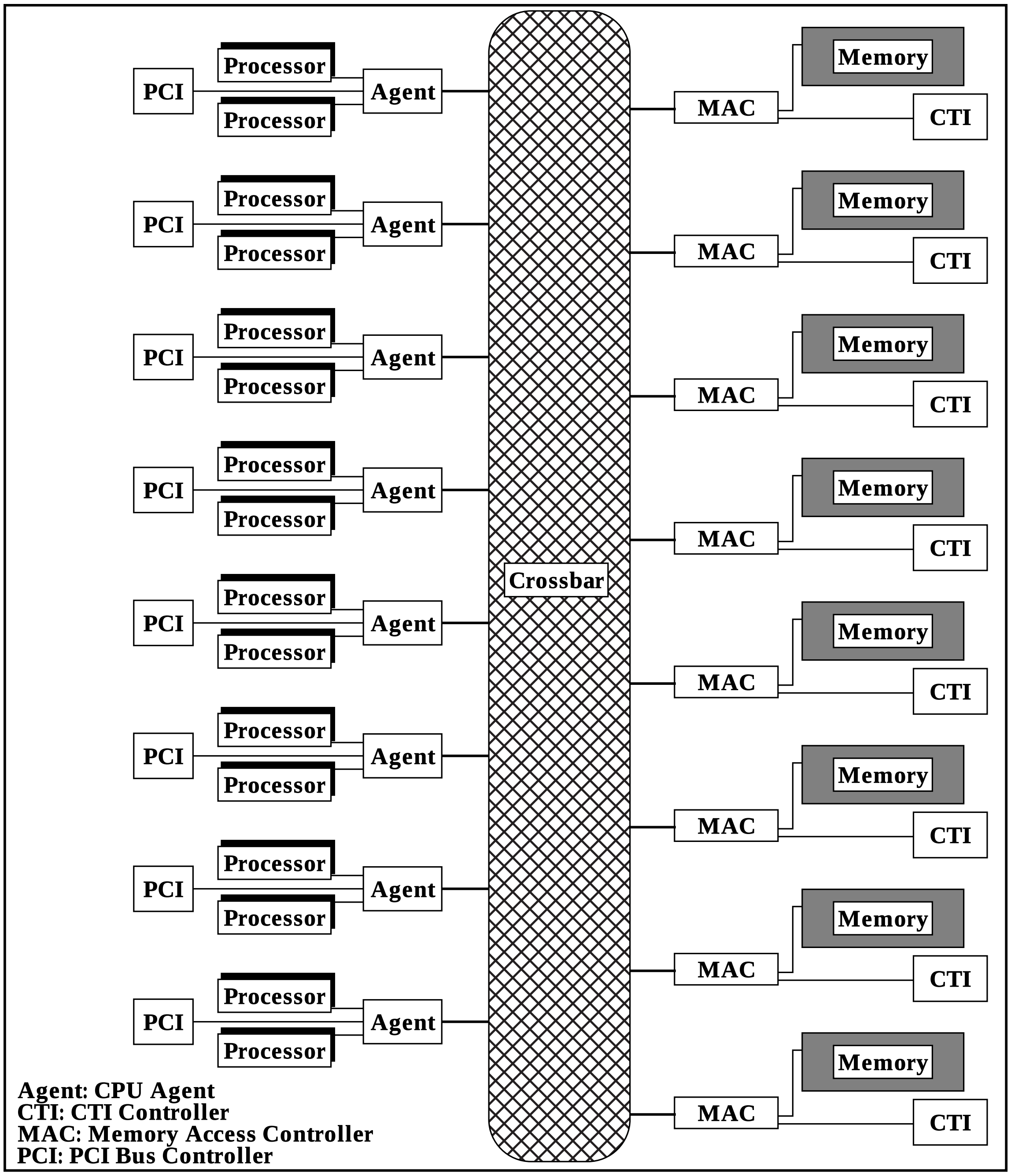
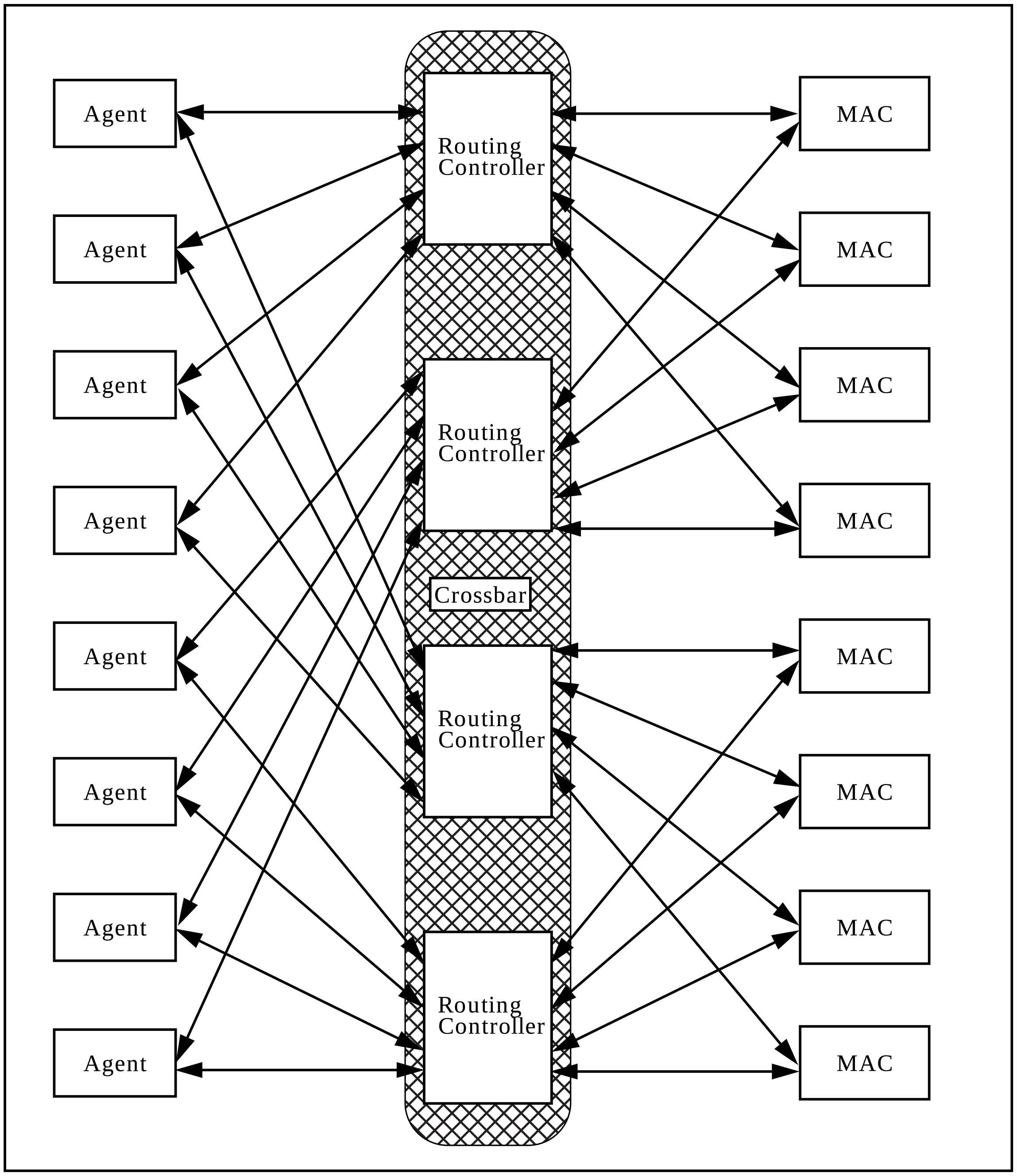


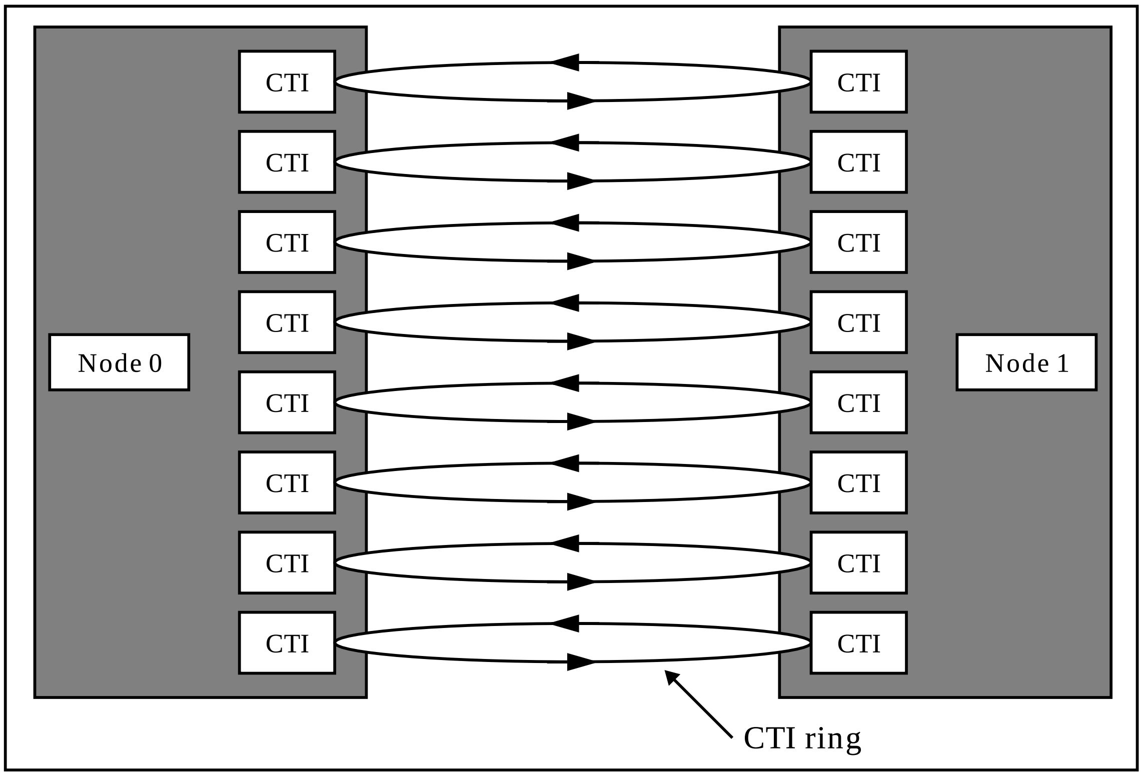
Figure 4 **X2000 crossbar connections**



Any processor can access memory on another hypernode by routing its request through its own crossbar to a CTI ring that attaches to that hypernode. Data is returned via a CTI ring and then routed via the crossbar back to the requesting processor.

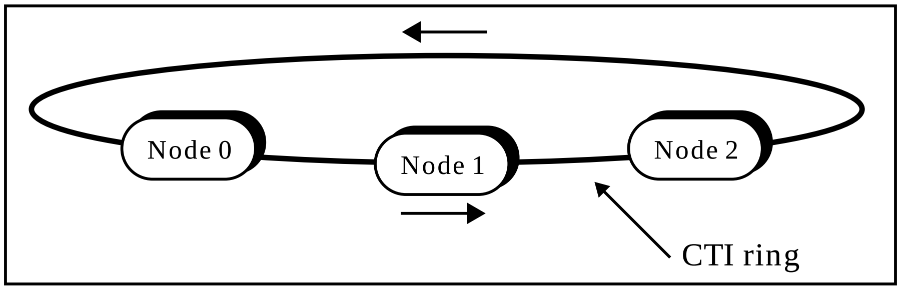
Figure 5 shows the CTI ring connections between two X2000 hypernodes. See Figure 3 on page 18 for details not available in the figure below.

Figure 5 CTI ring connections for two-hypernode X2000 server



CTI rings are unidirectional. That is, packets can only move in one direction on the rings. Consider the three-hypernode X2000 server illustrated in Figure 6; for simplicity, only one of the eight rings is shown. If Node 0 initiates communication with Node 2, it goes through the CTI controller on Node 1 to get to Node 2. Responses from Node 2 to Node 0 travel in the same direction as the request and cover the remainder of the ring.

Figure 6 **Unidirectional flow on a CTI ring**



As a system scales, a one-dimensional interconnect becomes less efficient because the ring grows to include a CTI controller for every hypernode in the system. For X2000 servers, when the number of nodes exceeds three, a one-dimensional interconnect is no longer optimal. A two-dimensional interconnect is then used to shorten paths between requesting and responding nodes.

The two-dimensional interconnect uses dimension-order routing to determine the path taken by a packet. A request packet first travels the required distance on the X-dimension ring then, if needed, the Y-dimension ring. On the return path, the response packet again travels the X-dimension ring first, then the Y-direction ring. Thus, the response packet does not necessarily follow the same path as the request packet.

Figure 7 shows a four-hypernode X2000 server using a two-dimensional interconnect. The node IDs (0, 1, 8, and 9 in the figure) are represented in 5-bit fields, where the first three bits represent the X dimension and the last two bits represent the Y dimension.

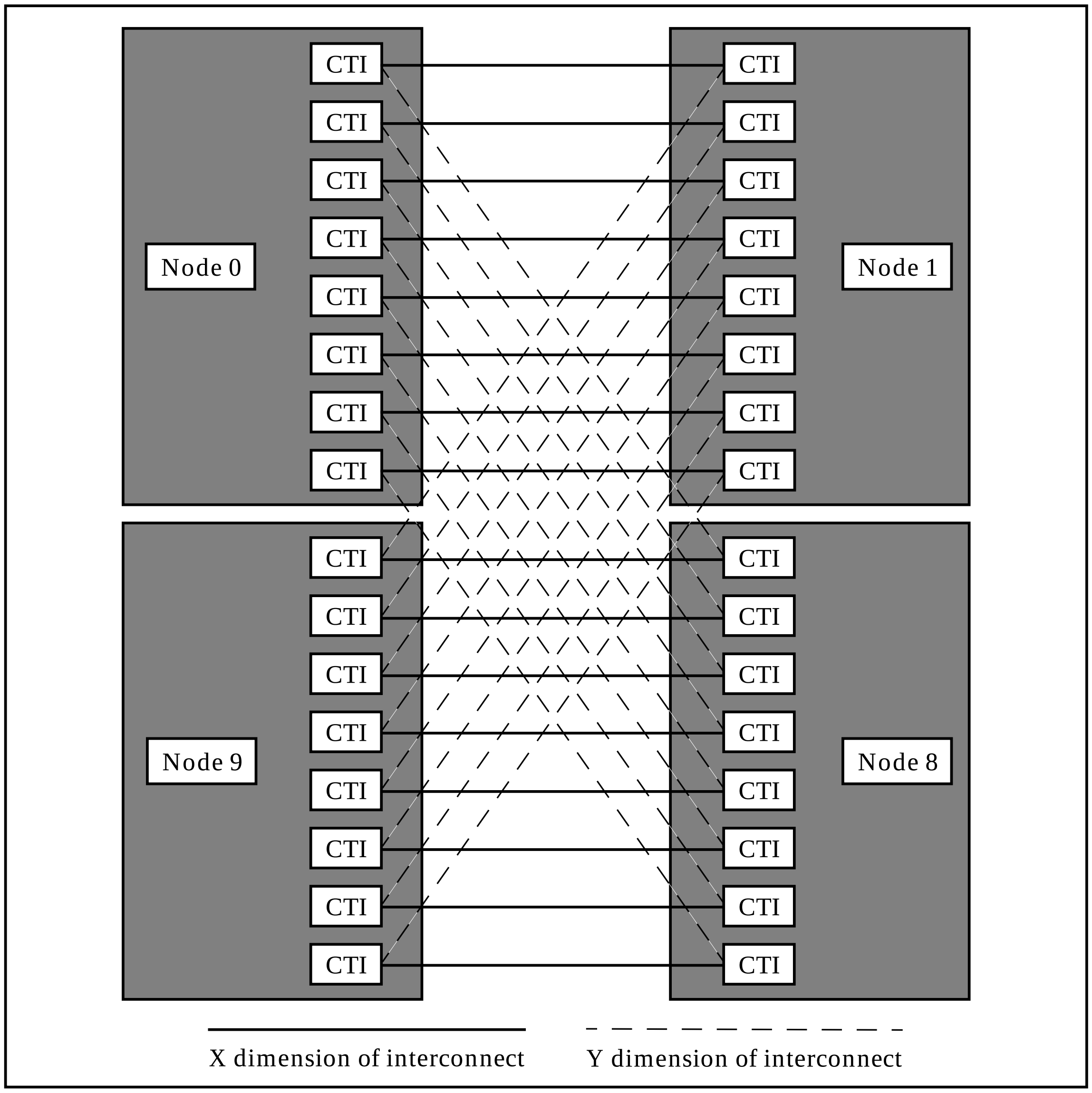
Nodes connected in the X dimension are:

- Node 0 (ID:00000) and node 1 (ID:00001)
- Node 8 (ID:01000) and node 9 (ID:01001)

Nodes connected in the Y dimension are:

- Node 0 (ID:00000) and node 8 (ID:01000)
- Node 1 (ID:00001) and node 9 (ID:01001)

Figure 7 **CTI ring connections for four-hypernode X2000 server**



CPUs communicate directly with their own instruction and data caches, which can be accessed by the processor in one clock (assuming a full pipeline). X2000 servers use 1-Mbyte off-chip instruction caches and data caches. V2200 servers use 2-Mbyte off-chip instruction caches and data caches.

Memory

Each process running on a V-Class or K-Class server (running HP-UX 11.0 and above) accesses its own 16-Tbyte virtual address space. Almost all of this space is available to hold program text, data, and the stack; the space used by the operating system is negligible.

On X2000 servers running the SPP-UX operating system, each process can accesses its own 4-Gbyte virtual address space. Again, most of this space is available to program text, data, and the stack with only a negligible amount of space used by the operating system.

The stack size is configurable; refer to the section “Default stack size” on page 152 for more information.

Processes cannot access each other’s virtual address spaces. This virtual memory maps to the physical memory of the system on which the process is running.

Physical memory

All memory (excluding processor caches) on V2200 servers and X2000 servers is implemented in memory banks. In 16-processor V2200 servers and X2000 servers, each hypernode consists of 32 memory banks. This memory is typically partitioned (by the system administrator) into hypernode-local, system-global, CTIcache (on multinode systems), and buffer cache. It is also interleaved as described in the “Interleaving” section later in this chapter.

Hypernode-local memory, as its name implies, is local to its hypernode, and cannot be accessed by other hypernodes. This is where application and operating-system executables, as well as user process data that has been explicitly declared private, reside.

System-global memory is accessible by all processors in a given system.

The CTIcache is used to store copies of global data fetched from other hypernodes.

The buffer cache is a file system cache and is used to encache items that have been read from disk and items that are to be written to disk.

Virtual memory

Virtual memory is divided into five classes. The compilers choose default classes to provide your programs with normal SMP memory-transaction semantics. You can also manually assign data to memory classes to improve data locality and further increase performance. However, doing so also requires some other aspects of optimization, particularly loop parallelization, to be handled manually.

Brief descriptions of the virtual memory classes and their physical memory mappings follow:

`thread_private`

This memory is private to each thread of a process. A `thread_private` data object has a unique virtual address for each thread within its hypernode. These addresses map to unique physical addresses in hypernode-local physical memory on each hypernode. Threads access the physical copies of `thread_private` data residing on their own hypernode when they access `thread_private` virtual addresses.

`node_private`

This memory is shared among the threads running on a given hypernode but is inaccessible from other hypernodes. A `node_private` data object has a unique virtual address by which all threads on all hypernodes access it. This address maps to one physical address per hypernode; when a thread accesses the data, it receives the value contained in the physical memory of its own hypernode.

`near_shared`

Data objects of the `near_shared` class have a single virtual address by which they can be accessed from any hypernode in the system. Physically, `near_shared` data is stored entirely within the memory of a particular hypernode. All data of a `near_shared` object maps to physical addresses on that hypernode.

`far_shared`

Data objects of the `far_shared` class have a single virtual address by which they can be accessed from any hypernode in the system. Physically, `far_shared` data is distributed by pages, in a manner that is approximately round-robin, to all the hypernodes in the system, so the virtual address maps to a single physical address located on one of the hypernodes.

`block_shared`

Data objects of the `block_shared` class have a single virtual address by which they can be accessed from any hypernode in the system. Physically, `block_shared` data is distributed in blocks equally among the hypernodes on which the process is executing, one block per hypernode. `block_shared` memory must be dynamically allocated; the programmer can then easily ensure that threads on a hypernode make most of their accesses to the block residing on their hypernode.

Using these memory classes is discussed in detail in Chapter 5, “Memory classes.”

Data caches

V2200 servers and X2000 servers use high-speed data caches to improve performance, but the architectures differ in their implementations of the cache. *CTIcaches* are used to improve performance on multihypernode systems. (A CTIcache is a partition of physical memory that exists on each hypernode and is used to store copies of global data fetched from other hypernodes.)

Cache lines

Before examining the specifics of caches, you must understand how data is moved between the cache and memory. A *cache line* describes the size of a chunk of contiguous data that must be copied into or out of a cache in one operation. V2200 servers use *processor cache lines*; X2000 servers use processor cache lines and *CTIcache lines*.

When a processor experiences a cache miss—that is, requests data that is not already encached—the cache line containing the address of the requested data is moved to the cache. This cache line also contains some number of other data objects that were not specifically requested; this number varies according to the object size and the type of cache line in question.

A CTIcache line moves data from shared memory to the CTIcache when a CTIcache miss occurs. For X2000 servers, the CTIcache line is 32 bytes, and each CTIcache line matches one-to-one to a 32-byte processor cache line. When a processor cache miss occurs, the requested data is fetched as part of a contiguous 32-byte cache line. If this data resides in any memory on the processor's hypernode, it need not traverse the CTIcache; if it resides in the memory of another hypernode, it will be fetched through the CTIcache.

All processor-encached data not residing on the processor's hypernode must pass through the CTIcache, so if this data is contained in processor cache, it is also resident in the CTIcache.

One reason cache lines are employed is to allow for *data reuse*. Data in a cache line is subject to reuse if, while the line is encached, any of the data elements contained in the line besides the requested element are referenced by the program, or if the requested element is referenced more than once.

Because data can only be moved to and from memory as part of a cache line, both load and store operations cause their operands to be encached. Cache-coherency hardware invalidates cache lines in other processors when they are stored to by a particular processor. This indicates to other processors that they must load the cache line from memory the next time they reference its data.

Direct-mapped data caches

V2200 servers use 2-Mbyte off-chip write-back direct-mapped data caches. In a direct-mapped cache, the cache address for a given data object is a function of the object's full virtual address. For V2200 systems, cache addresses are computed within a process using the following formula:

cache_address = `MOD(`*virtual_address*`,`2^{21}`)`

Where the `MOD` function yields the remainder when *virtual_address* is divided by 2^{21}. The value of 2^{21} is 2,097,152, or 2 Mbytes. Thus, a data object's cache address is the least-significant 21 bits of its virtual address.

X2000 servers use 1-Mbyte off-chip write-back direct-mapped data caches. For X2000 systems, cache addresses are computed within a process using the following formula:

cache_address = `MOD(`*virtual_address*`,`2^{20}`)`

Where the `MOD` function yields the remainder when *virtual_address* is divided by 2^{20}. The value of 2^{20} is 1,048,576, or 1 Mbyte. Thus, a data object's cache address is the least-significant 20 bits of its virtual address.

This mapping scheme can result in *cache thrashing*, which is discussed in the section "Cache thrashing" on page 29.

Prefetching with the `+Odataprefetch` compiler option

Prefetching is supported through the command-line option `+Odataprefetch`. Prefetching encaches data that will be used in future iterations of a loop, while the processor is executing current iterations. The prefetch distance (distance in terms of the number of processor cycles) varies and is tuned to the target machine architecture. Prefetching is not beneficial to loops whose data fits in the cache. For loops whose data does not fit in the cache, performance improvement can be substantial. This option is off (`+Onodataprefetch`) by default. For additional information on this option, see Appendix D, "Optimization options."

Data alignment

Aligning data addresses on cache line boundaries allows for efficient data reuse in loops (refer to Chapter 3, “Compiler optimizations”). The linker automatically aligns data over 32 bytes on a 32-byte boundary. Also, it aligns data greater in size than a page on a 64-byte boundary.

You can align data on 64-byte boundaries by:

- Using Fortran `ALLOCATE` statements. (Applies only to parallel executables.)
- Using the C functions `malloc` or `memory_class_malloc`. (Applies only to parallel executables.)

Only the first item in a list of data objects appearing in any of these statements is aligned on a cache line boundary. To make most efficient use of available memory, the total size, in bytes, of any array appearing in one of these statements should be an integral multiple of 32. Sizing your arrays this way prevents data following the first array from becoming misaligned. Scalar variables should be listed after arrays and ordered from longest data type to shortest (for example, `REAL*8` scalars should precede `REAL*4` scalars).

NOTE Aliases can inhibit data alignment. Be especially careful when equivalencing arrays in Fortran.

You can force CTIcache boundary alignment for specific scalar variables or arrays by using the `align_cti` directive or pragma. The Fortran directive has the form:

`C$DIR ALIGN_CTI(`*namelist*`)`

In C it has the form:

`#pragma _CNX align_cti(`*namelist*`)`

where *namelist* is a list of arrays and/or scalars that will be aligned on CTIcache boundaries.

Cache thrashing

Cache thrashing occurs when two or more data items that are needed by the program both map to the same cache address. Each time one of the items is encached, it overwrites another needed item, causing cache misses and impairing data reuse. This section explains how thrashing happens on X2000 servers.

A type of thrashing known as *false cache line sharing* is discussed in the section “False cache line sharing” on page 274.

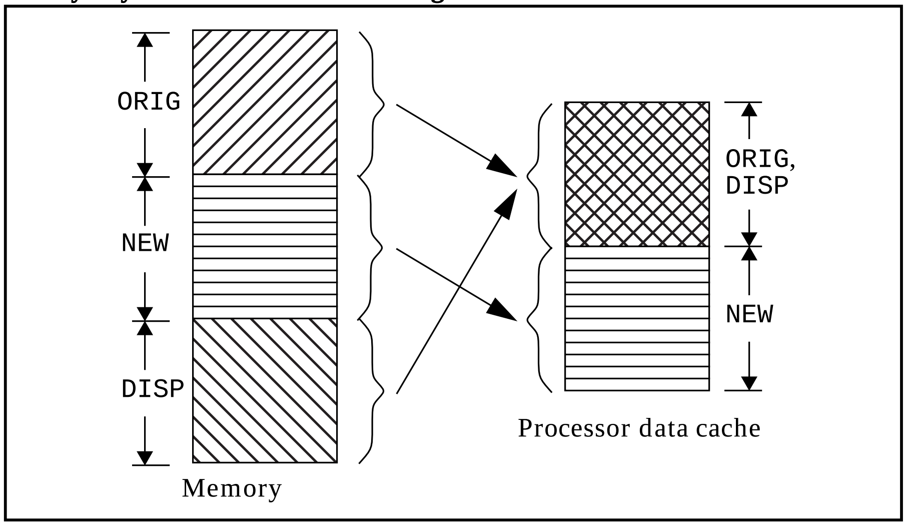
X2000 servers use a 1-Mbyte direct-mapped data cache. Thus, cache thrashing can become a problem on X2000 servers when two encachable data objects are exactly a multiple of 1 Mbyte apart in virtual memory. To eliminate the problem, you must ensure that your data is not spaced this way.

Consider the following Fortran example:

```
REAL*8 ORIG(65536), NEW(65536), DISP(65536)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
DO I = 1, N
  NEW(I) = ORIG(I) + DISP(I)
ENDDO
```

In this example, the arrays `ORIG` and `DISP` overwrite each other in a 1-Mbyte cache. Because the arrays are in a `COMMON` block, we know that they will be allocated in contiguous memory in the order shown. Each array element occupies 8 bytes, so each array occupies 0.5 Mbyte ($8 \times 65536 = 524288$ bytes); therefore arrays `ORIG` and `DISP` are exactly 1 Mbyte apart in memory, and all their elements have identical cache addresses. The layout of the arrays in memory and in the data cache is shown in Figure 8.

Figure 8 **Array layouts—cache-thrashing**



When the addition in the body of the loop executes, the current elements of both `ORIG` and `DISP` must be fetched from memory into the cache. Because these elements have identical cache addresses, whichever is fetched last will overwrite the first. Remember that processor cache data is fetched 32 bytes at a time; to efficiently execute a loop such as this, the unused elements in the fetched cache line (3 extra `REAL*8` elements are fetched in this case) must remain encached until they can be used in subsequent iterations of the loop. Because `ORIG` and `DISP` thrash each other, this reuse is never possible; every cache line of `ORIG` that is fetched is overwritten by the cache line of `DISP` that is subsequently fetched, and vice versa. The cache line is overwritten on every iteration; typically, in a loop like this, it would not be overwritten until all of its elements were used.

Because memory accesses take substantially longer than cache accesses, this severely degrades performance. Even if the overwriting involved the `NEW` array, which is stored rather than loaded on each iteration, thrashing would occur, because stores overwrite entire cache lines the same way loads do.

The problem is easily fixed by increasing the distance between the arrays. You can accomplish this by either increasing the array sizes or inserting a padding array.

The following example illustrates the padding approach:

```
REAL*8 ORIG(65536), NEW(65536), P(4),DISP(65536)
COMMON /BLK1/ ORIG, NEW, P, DISP
.
.
.
```

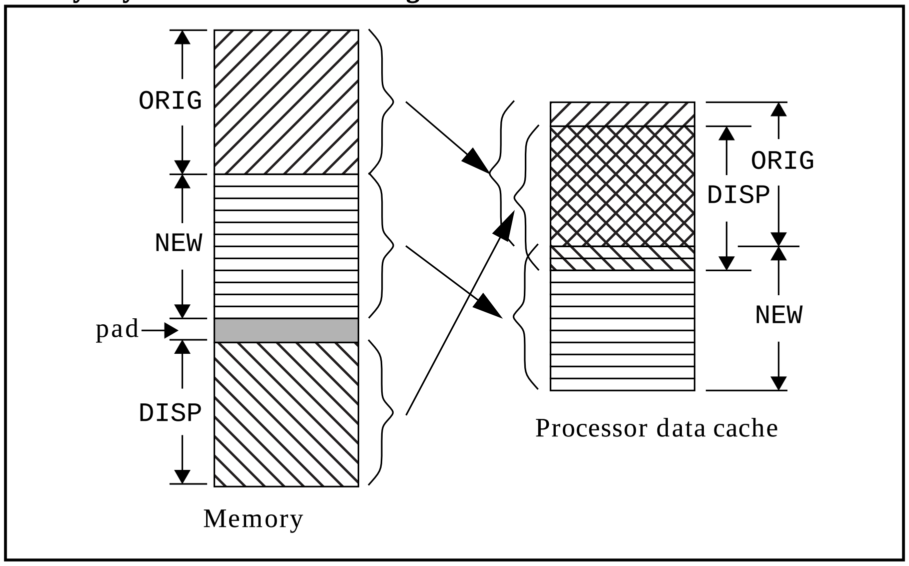
Here, the array `P(4)` moves `DISP` 32 bytes further from `ORIG` in memory. Now no two elements of the same index share a cache address, and for the given loop, this postpones cache overwriting until the entire current cache line is completely exploited. `P` is 4 elements, or 32 bytes, which prevents both processor cache thrashing and CTIcache thrashing on X2000 servers.

The alternate approach involves increasing the size of `ORIG` or `NEW` by 4 elements (32 bytes), as shown in the following example:

```
REAL*8 ORIG(65536), NEW(65540), DISP(65536)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
```

Here, `NEW` has been increased by 4 elements, providing the padding necessary to prevent `ORIG` from sharing cache addresses with `DISP`. Figure 9 shows how both solutions prevent thrashing.

Figure 9 **Array layouts—non-thrashing**



It is important to note that this is a highly simplified, worst-case example. On X2000 servers, thrashing can happen any time two data items that are referenced in the same loop are an integral multiple of 1 Mbyte apart in virtual memory. This can happen with data that is not stored in `COMMON`, in which case it is much more difficult to see, as such data can be stored noncontiguously and may be intermixed with completely unrelated data items.

The loop blocking optimization (described in Chapter 3, “Compiler optimizations”) will eliminate thrashing from certain nested loops, but not from all loops. Declaring arrays with dimensions that are not powers of two can help, but it will not necessarily eliminate the problem completely.

Using `COMMON` blocks in Fortran can also help; it allows you to accurately measure distances between data items, making thrashing problems easier to spot before they happen.

Interleaving

Physical pages are interleaved across the memory banks of a hypernode on a cache-line basis. (There are 32 banks per node in V2200 servers and X2000 servers). Contiguous cache lines are assigned in round-robin fashion, first to the even banks, then to the odd, as shown in Figure 10 for V2200 servers and X2000 servers.

Interleaving speeds memory accesses by allowing several processors to access contiguous data simultaneously. This is beneficial when a loop that manipulates arrays is split among many processors; in the best case, threads will access data in patterns with no bank contention. Even in the worst case, where each thread initially needs the same data from the same bank, after the initial contention delay, the accesses will be spread out among the banks.

Interleaving example

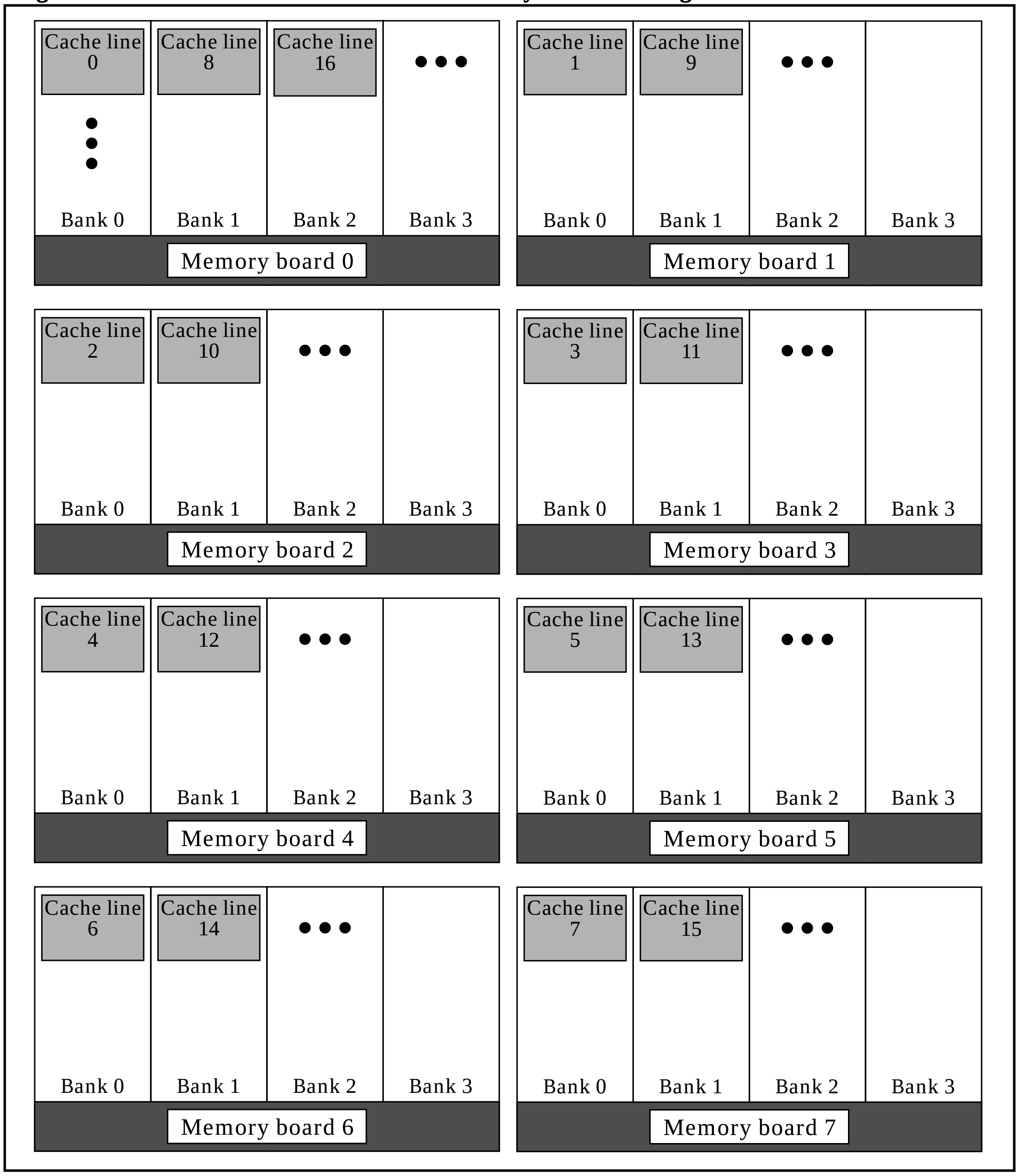
The following example illustrates a nested loop that accesses memory with very little contention. This example is greatly simplified for illustrative purposes, but the concepts apply to arrays of any size.

```
REAL*8 A(12,12), B(12,12)
...
DO J = 1, N
  DO I = 1, N
    A(I,J) = B(I,J)
  ENDDO
ENDDO
```

Assume that arrays `A` and `B` are stored contiguously in memory, with `A` starting in bank 0, CTIcache line 0, processor cache line 0, as shown in Figure 11 on page 36 for V2200 servers and X2000 servers.

Assume the Exemplar Fortran 90 compiler parallelizes the `J` loop to run on as many processors as are available in the system (up to `N`). Assuming `N=12` and there are four processors available when the program is run, the `J` loop could be divided into four new loops, each with 3 iterations. Each new loop would run to completion on a separate processor. We will refer to these four processors as CPU0 through CPU3.

Figure 10 V2200 and X2000 memory interleaving



NOTE This example is designed to simplify illustration. In reality, the dynamic selection optimization (discussed in Chapter 3, “Compiler optimizations”) would, given the iteration count and available number of processors described, cause this loop to run serially. The overhead of going parallel would outweigh the benefits.

In order to execute the body of the `I` loop, `A` and `B` must be fetched from memory and encached. Each of the four processors running the `J` loop will attempt to fetch its portion of the arrays, most likely simultaneously.

This means CPU0 will attempt to read arrays `A` and `B` starting at elements `(1,1)`, CPU1 will attempt to start at elements `(1,4)` and so on. For clarity, Figure 11 shows the first 32 CTIcache lines consecutively; after these, only the initial cache lines for each processor are shown. Each processor’s initial cache line is shaded.

Because of the number of memory banks in the V2200 and X2000 architecture, interleaving removes the contention from the beginning of the loop from the example, as shown in Figure 11.

CPU0 needs `A(1:12,1:3)` and `B(1:12,1:3)`

CPU1 needs `A(1:12,4:6)` and `B(1:12,4:6)`

CPU2 needs `A(1:12,7:9)` and `B(1:12,7:9)`

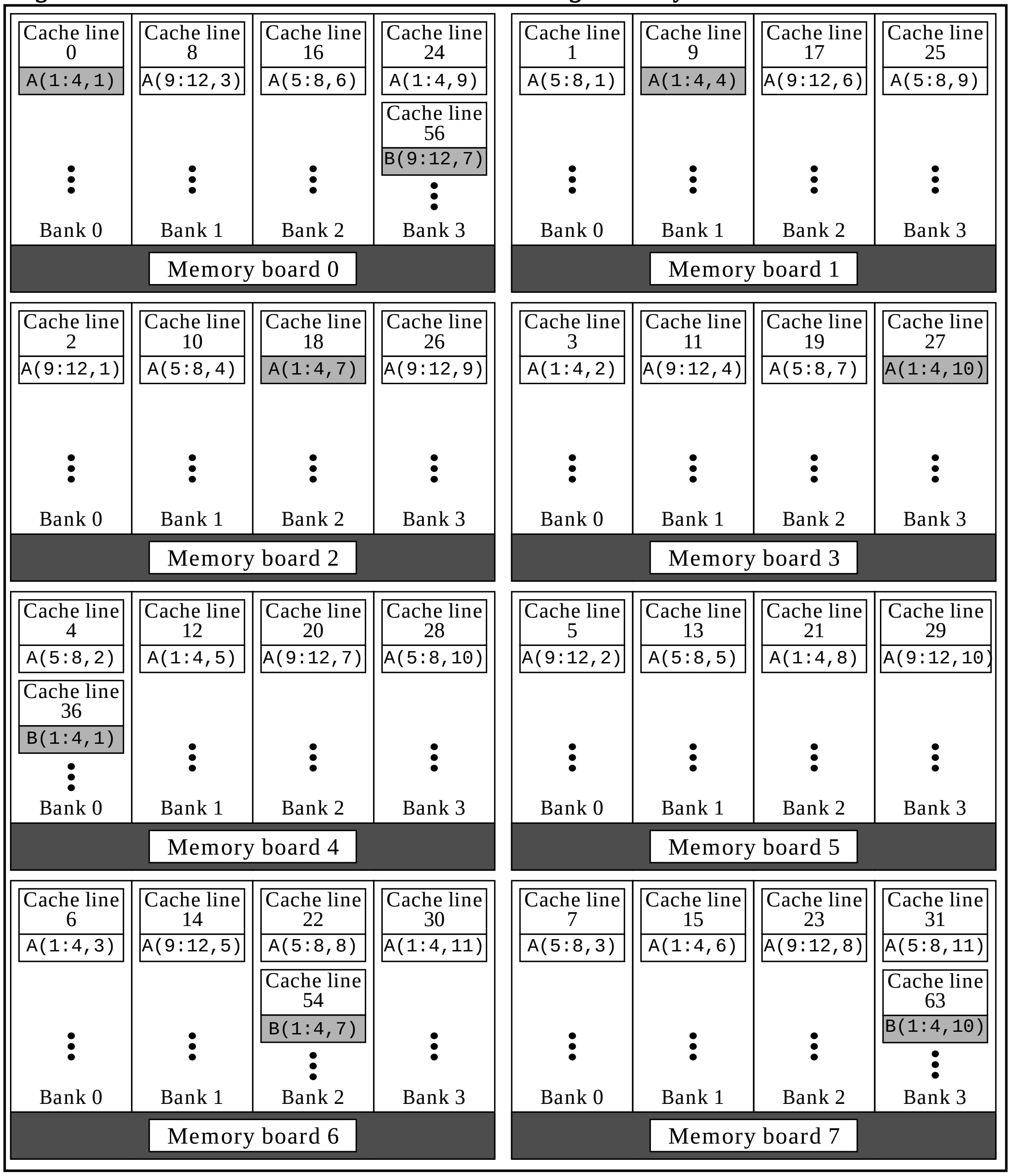
CPU3 needs `A(1:12,10:12)` and `B(1:12,10:12)`

The data from the V2200/X2000 example above is spread out on different memory banks as described below:

- `A(1,1)`, the first element of the chunk needed by CPU0, is on cache line 0 in bank 0 on board 0
- `A(1,4)`, the first element needed by CPU1, is on cache line 9 in bank 1 on board 1
- `A(1,7)`, the first element needed by CPU2, is on cache line 18 in bank 2 on board 2
- `A(1,10)` the first element needed by CPU3, is on cache line 27 in bank 3 on board 3

Because of interleaving, no contention exists between the processors when trying to read their respective portions of the arrays. Contention may surface occasionally as the processors make their way through the data, but the resulting delays are minimal compared to what could be expected without interleaving.

Figure 11 V2200 and X2000 interleaving of arrays A and B



Variable-sized pages

Variable-sized pages are used to reduce *Translation Lookaside Buffer* (TLB) misses and consequently improve performance. With variable-sized pages, each TLB entry used can map a larger portion of an application's virtual address space. Thus, applications with large reference sets can be mapped using fewer TLB entries, resulting in fewer TLB misses. (A TLB is a hardware entity used to translate a virtual memory reference to a physical page.)

If an application is not experiencing performance degradation due to TLB misses, using a different page size does not help. Also, if an application uses too large a page size, fewer pages will be available to other applications on the system, potentially resulting in increased paging activity and performance degradation.

Valid page sizes on the PA-8000 and PA-8200 processors are 4K, 16K, 64K, 256K, 1 Mbyte, 4 Mbytes, 16 Mbytes, 64 Mbytes, and 256 Mbytes. (The default size, which can be configured, is 4K.) Methods for specifying a page size are described below. However, the user-specified page size is only a request for a specific size; the operating system takes various factors into account when selecting the page size.

The following command options and configurable kernel parameters allow you to specify information regarding page sizes.

- Options to the `chatr` utility:
  - `+pi`: affects the page size for the application's text segment
  - `+pd`: affects the page size for the application's data segment
- Configurable kernel parameters:
  - `vps_pagesize`: represents the default or minimum page size (in kilobytes) if the user has not used `chatr` to specify a value; the default is 4K
  - `vps_ceiling`: represents the maximum page size (in kilobytes) if the user has not used `chatr` to specify a value; the default is 16K
  - `vps_chatr_ceiling`: places a restriction on the largest value (in kilobytes) a user can specify using `chatr`; the default is 64 Mbytes

For more information on the `chatr` utility, see the chatr(1) man page.

3 Compiler optimizations

This chapter discusses the various optimization levels available with the Exemplar compilers and explains the optimizations performed at each level.

The Fortran 90 compiler is located at /opt/fortran90/bin/f90.

The two Fortran 77 compilers are:

- `f77` is the Fortran 77 compiler and is located at /opt/fortran/bin/f77
- `fort77` is the POSIX-conforming Fortran 77 compiler and is located at /opt/fortran/bin/fort77

The remainder of this manual refers to the `f77` compiler. Any `f77` example applies to the `fort77` compiler.

The two C compilers are:

- `cc` is the C compiler and is located at /opt/ansic/bin/cc
- `c89` is the POSIX-conforming C compiler and is located at /opt/ansic/bin/c89

The remainder of this manual refers to the `cc` compiler. Any `cc` example applies to the `c89` compiler.

The aC++ compiler is located at /opt/aCC/bin/aCC.

Optimization levels

Five optimization levels are available for use with the Exemplar compilers. These options have identical names and perform identical optimizations, regardless of which compiler you are using. They are specified on the compiler command line along with any other options you wish to use. Exemplar compiler optimization levels are summarized in Table 1.

Table 1 **Compiler optimization levels**

Option	Description
`+O0` (default)	(Machine instruction-level optimizations) Constant folding and simple register assignment
`+O1`	(Block-level optimizations) `+O0` optimizations, plus instruction scheduling and optimizations on basic blocks (A basic block is a linear sequence of machine instructions with a single entry and a single exit.)
`+O2`	(Routine-level optimizations) `+O1` optimizations, plus optimizations within subprograms in a single file; loop optimizations to reduce pipeline stalls; analysis of data flow, memory usage, loops, and expressions
`+O3`	(File-level optimizations) `+O2` optimizations, plus full optimizations across all subprograms (including inlining) within a single file; use of parallelism-related directives and pragmas from the Exemplar programming model when `+Oparallel` is also specified
`+O4`*	(Cross-module optimizations) `+O3` optimizations, plus full optimizations across the entire application; optimizations include inlining across the application; the `+O4` optimizations are performed at link time

*The `+O4` option is not available in Fortran 90.

These options are cumulative; each option retains the optimizations of the previous option. For example, entering the following command line compiles the Fortran program foo.f with all `+O2`, `+O1`, and `+O0` optimizations shown in Table 1.

```
% f90 +O2 foo.f
```

In addition to these options, the `+Oparallel` option is available for use at `+O3` and above. (`+Onoparallel` is the default.) When the `+Oparallel` option is specified, the compiler:

- Looks for opportunities for parallel execution in loops.
- Honors the parallelism-related directives and pragmas of the Exemplar programming model. When using Exemplar Fortran 77 Version 1.2.3 or Exemplar C Version 1.2.3, `+Oexemplar_model` (the default) must also be in effect for these directives and pragmas to be enabled.

The `+Onoautopar` (no automatic parallelization) option is available for use with `+Oparallel` at `+O3` and above; `+Oautopar` is the default. `+Onoautopar` causes the compiler to parallelize only those loops that are immediately preceded by `loop_parallel` or `prefer_parallel` directives or pragmas; for more information, refer to Chapter 4, “Basic shared-memory programming.”

The `+Onodepar` (node-parallelism) option is also available for use with `+Oparallel` at `+O3` and above. This option causes the compiler to generate node-parallel code (indicated by directives and pragmas that use the `nodes` attribute) for a multinode, scalable SMP. (See Chapter 4, “Basic shared-memory programming,” for information on attributes.)

The `+Ononodepar` option (the default) causes the compiler to generate code for a single-node machine. When this option is used, serial code is generated for node-parallel constructs; thus, node-parallelism is not implemented. Thread-parallelism—both automatic and directive-specified—is still implemented.

Using the optimizer

Before exploring the various optimizations that are performed, we should examine what coding guidelines can be followed to assist the optimizer. This section is broken down into the following subsections:

- General guidelines
- Fortran 90 and Fortran 77 guidelines
- C and C++ guidelines

General guidelines

The coding guidelines presented in this section help the optimizer to optimize your program, regardless of the language the program is written in.

- Where possible, use local variables to help the optimizer promote variables to registers.
- Do not use local variables before they are initialized. When you request `+O2`, `+O3`, or `+O4` optimizations, the compiler tries to detect and indicate violations of this rule. See the section "`+O[no]initcheck`" on page 373 for related information.
- Where possible, use constants instead of variables in arithmetic expressions such as shift, multiplication, division, or remainder operations.
- If a loop contains only a procedure call, position the loop inside the procedure or use a directive to call the loop in parallel—if appropriate.
- The code generated for a loop termination test is more efficient with a test against zero than with a test against some other value. Therefore, where possible, construct loops so the control variable increases or decreases toward zero.
- Avoid referencing outside the bounds of an array. (Fortran provides the `-C` option to check whether your program references outside array bounds.)
- Avoid passing an incorrect number of arguments to functions.

Fortran guidelines

The coding guidelines presented in this section help the optimizer to optimize Fortran programs.

As part of the optimization process, the compiler gathers information about the use of variables and passes this information to the optimizer. The optimizer uses this information to ensure that every code transformation maintains the correctness of the program, at least to the extent that the original unoptimized program is correct.

When gathering this information, the compiler assumes that inside a routine (either a function or a subroutine) the only variables that can be accessed (directly or indirectly) are:

- `COMMON` variables declared in the routine
- Local variables
- Parameters to this routine

Local variables include all static and nonstatic variables.

In general, you do not need to be concerned about this assumption. However, if you have code that violates the assumption, the optimizer can change the behavior of the program in an undesirable way.

One guideline is to avoid using variables that can be accessed by a process other than the program. The compiler assumes that the program is the only process accessing its data. The only exception is the shared `COMMON` variable. In this case, optimization will be correct if you properly use the `$OPTIMIZE ASSUME_NO_SHARED_COMMON_PARMS` Fortran 77 directive. For more information on `OPTIMIZE` directives, see Appendix A, “Standard HP compiler directives and pragmas.”

A final guideline is to avoid using extensive equivalencing and memory-mapping schemes, where possible.

See the section “General guidelines” on page 42 for additional guidelines.

C and C++ guidelines

The coding guidelines presented in this section help the optimizer to optimize your C and C++ programs.

- Use `do` loops and `for` loops in place of `while` loops. `do` loops and `for` loops are more efficient because opportunities for removing loop-invariant code are greater.
- Use `register` variables where needed.
- When using `short` or `char` variables or bit-fields, it is more efficient to use unsigned variables rather than signed because a signed variable causes an extra instruction to be generated.
- Whenever possible, pass and return pointers to large structs instead of passing and returning large structs by value.
- Use type-checking tools like `lint` to help eliminate semantic errors.
- Use local variables for the upper bounds (stop values) of loops; using local variables may enable the compiler to optimize the loop.

During optimization, the compiler gathers information about the use of variables and passes this information to the optimizer. The optimizer uses this information to ensure that every code transformation maintains the correctness of the program, at least to the extent that the original unoptimized program is correct.

When gathering this information, the compiler assumes that while inside a function, the only variables that can be accessed indirectly through a pointer or by another function call are:

- Global variables (that is, all variables with file scope)
- Local variables that have had their addresses taken either explicitly by the & operator, or implicitly by the automatic conversion of array references to pointers

In general, you do not need to be concerned about this assumption. Standard-compliant C and C++ programs do not violate this assumption. However, if you have code that does violate this assumption, the optimizer can change the behavior of the program in an undesirable way. In particular, you should follow the coding practices below to ensure correct program execution for optimized code:

- Avoid using variables that are accessed by external processes. Unless a variable is declared with the `volatile` attribute, the compiler will assume that a program's data is accessed only by that program. Using the `volatile` attribute may significantly slow down a program.
- Avoid accessing an array other than the one being subscripted. For example, the construct `a[b-a]`, where `a` and `b` are the same type of array, actually references the array `b`, because it is equivalent to `*(a+(b-a))`, which is equivalent to `*b`. Using this construct might yield unexpected optimization results.
- Avoid referencing outside the bounds of the objects a pointer is pointing to. All references of the form `*(p+i)` are assumed to remain within the bounds of the variable or variables that `p` was assigned to point to.
- Do not rely on the memory layout scheme when manipulating pointers; incorrect optimizations may result. For example, if `p` is pointing to the first member of a structure, do not assume that `p+1` points to the second member of the structure. Another example: if `p` is pointing to the first in a list of declared variables, `p+1` is not necessarily pointing to the second variable in the list.

See the section “General guidelines” on page 42 for additional guidelines.

+O0 level optimizations

At optimization level `+O0`, the compiler performs the following optimizations that span no more than a single source statement:

- Constant folding
- Partial evaluation of test conditions
- Simple register assignment
- Data alignment on natural boundaries

The default optimization level is `+O0`.

Constant folding

Constant folding is the replacement of operations on constants with the result of the operation. For example, `Y=5+7` is replaced with `Y=12`.

More advanced constant folding is performed at optimization level `+O2`. See the section “Advanced constant folding and propagation” on page 54 for more information.

Partial evaluation of test conditions

Where possible, the compiler determines the truth value of a logical expression without evaluating all the operands (also known as short-circuiting). Consider the Fortran example below:

```
IF ((I .EQ. J) .OR. (I .EQ. K)) GOTO 100
```

If `(I .EQ. J)` is true, control immediately goes to `100`; otherwise, `(I .EQ. K)` must be evaluated before control can go to `100` or the following statement.

Do not rely upon partial evaluation if you use function calls in the logical expression because:

- There is no guarantee on the order of evaluation.
- A procedure or function call can have side effects on variable values that may or may not be partially evaluated correctly.

Simple register assignment

The compiler may place frequently used variables in registers to avoid more costly accesses to memory.

A more advanced register assignment algorithm is used at optimization level +O2. See the section "Global register allocation" on page 52 for more information.

Data alignment on natural boundaries

The compiler automatically aligns data objects to their natural boundaries in memory, providing more efficient access to data. This means that a data object's address is integrally divisible by the length of its data type; for example, REAL*8 objects have addresses integrally divisible by 8 bytes.

NOTE Aliases can inhibit data alignment. Be especially careful when equivalencing arrays in Fortran.

You should declare scalar variables in order from longest to shortest data length to ensure the efficient layout of such aligned data in memory. This minimizes the amount of padding the compiler has to do to get the data onto its natural boundary.

Consider the following Fortran example:

```
C     CAUTION: POORLY ORDERED DATA FOLLOWS:
      LOGICAL*2 BOOL
      INTEGER*8 A, B
      REAL*4 C
      REAL*8 D
```

Here, the compiler must insert 6 blank bytes after BOOL in order to correctly align A, and it must insert 4 blank bytes after C to correctly align D.

The same data is more efficiently ordered as shown in the following example:

```
C     PROPERLY ORDERED DATA FOLLOWS:
      INTEGER*8 A, B
      REAL*8 D
      REAL*4 C
      LOGICAL*2 BOOL
```

Natural boundary alignment is performed on all data. Do not confuse It with cache line boundary alignment, which is performed as described in the section "Data alignment" on page 28. Also discussed in Chapter 2 are the `align_cti` directive and pragma, which facilitate CTIcache line boundary alignment.

+O1 level optimizations

At optimization level `+O1`, the compiler performs optimizations on a block level. The compiler also continues to perform the optimizations performed at `+O0`.

The `+O1` optimizations are:

- Branch optimization
- Dead code elimination
- Faster register allocation
- Instruction scheduler
- Peephole optimizations

Branch optimization

The branch optimization involves traversing the procedure and transforming branch instruction sequences into more efficient sequences where possible. Examples of possible transformations are:

- Deleting branches whose target is the fall-through instruction (in other words, the target is two instructions away).
- Changing the target of the first branch to be the target of the second (unconditional) branch when the target of a branch is an unconditional branch.
- Transforming an unconditional branch at the bottom of a loop that branches to a conditional branch at the top of the loop into a conditional branch at the bottom of the loop.
- Changing an unconditional branch to the exit of a procedure into an exit sequence where possible.
- Changing conditional or unconditional branch instructions that branch over a single instruction into a conditional nullification in the following instruction.
- Looking for conditional branches over unconditional branches, where the sense of the first branch could be inverted and the second branch deleted. These result from null `THEN` clauses and from `THEN` clauses that only contain `GOTO` statements.

  For example, in the following Fortran example:

```
      IF (L) THEN
        A=A*2
      ELSE
        GOTO 100
      ENDIF
      B=A+1
100   C=A*10
```

becomes:

```
      IF (.NOT. L) GOTO 100
      A=A*2
      B=A+1
100   C=A*10
```

Dead code elimination

Dead code elimination removes unreachable code that is never executed.

For example, in C:

```
if(0)
  a = 1;
else
  a = 2;
```

becomes:

```
  a = 2;
```

Faster register allocation

Faster register allocation involves:

- Inserting entry and exit code
- Generating code for operations such as multiplication and division
- Eliminating unnecessary copy instructions
- Allocating actual registers to the dummy registers in instructions

Faster register allocation, when used at `+O0` or `+O1`, analyzes register use faster than the global register allocation performed at `+O2`.

Instruction scheduler

The instruction scheduler optimization performs the following:

- Reordering the instructions in a basic block to improve memory pipelining. For example, where possible, a load instruction is separated from the use of the loaded register.
- Following a branch instruction with an instruction that can be executed as the branch occurs, where possible.
- Scheduling floating-point instructions.

Peephole optimizations

A peephole optimization is a machine-dependent optimization that makes a pass through low-level assembly-like instruction sequences of the program, applying patterns to a small window (peephole) of code looking for optimization opportunities. The optimizations performed are:

- Changing the addressing mode of instructions so they use shorter sequences
- Replacing low-level assembly-like instruction sequences with faster (usually shorter) sequences, and removing redundant register loads and stores

+O2 level optimizations

At optimization level `+O2`, the compiler performs optimizations on a routine level. The compiler continues to perform the optimizations performed at `+O1`, with the following additions:

- Global register allocation
- Strength reduction of induction variables and constants
- Common subexpression elimination
- Advanced constant folding and propagation (Simple constant folding is done at `+O0`.)
- Loop-invariant code motion
- Store/copy optimization
- Unused definition elimination
- Software pipelining
- Register reassociation
- Loop unrolling

Global register allocation

Scalar variables can often be stored in registers, eliminating the need for costly memory accesses. Global register allocation (GRA) attempts to store commonly referenced scalar variables in registers throughout the code in which they are most frequently accessed.

The compiler automatically determines which scalar variables are the best candidates for GRA and allocates registers accordingly.

GRA can sometimes cause problems when parallel threads attempt to update a shared variable that has been allocated a register. In this case, each parallel thread will allocate a register for the shared variable; it is then unlikely that the copy in memory will be updated correctly as each thread executes.

Parallel assignments to the same shared variables from multiple threads make sense only if the assignments are contained inside critical or ordered sections, or are executed conditionally based on thread ID. GRA will not allocate registers for shared variables that are assigned within critical or ordered sections, as long as the sections are implemented using compiler directives or `sync_routine`-defined functions (refer to Chapter 6, “Advanced shared-memory programming”). However, for conditional assignments based on thread ID, GRA may allocate registers that may cause wrong answers when stored.

In such cases, GRA can be disabled only for shared variables that are visible to multiple threads by specifying the `+Onosharedgra` compiler option.

In procedures with large numbers of loops, GRA can contribute to long compile times; therefore, GRA is only performed if the number of loops in the procedure is below a predetermined limit. You can remove this limit (and possibly increase compile time) by specifying the `+Onolimit` compiler option.

This optimization is also known as *coloring register allocation* because of the similarity to map-coloring algorithms in graph theory.

Register allocation in C and C++

In C and C++, you can help the optimizer understand when certain variables are heavily used within a function by declaring these variables with the `register` qualifier.

The global register allocator may override your choices and promote a variable not declared `register` to a register over a variable that is declared `register`, based on estimated speed improvements.

Strength reduction of induction variables and constants

This optimization removes expressions that are linear functions of a loop counter and replaces each of them with a variable that contains the value of the function. Variables of the same linear function are computed only once. This optimization also replaces multiplication instructions with addition instructions wherever possible.

For example, in the following C/C++ code:

```
for (i=0; i<25; i++) {
  r[i] = i * k;
}
```

becomes:

```
t1 = 0;
for (i=0; i<25; i++) {
  r[i] = t1;
  t1 += k;
}
```

Common subexpression elimination

The common subexpression elimination optimization identifies expressions that appear more than once and have the same result, computes the result, and substitutes the result for each occurrence of the expression. The subexpression types include instructions that load values from memory, as well as arithmetic evaluation.

In Fortran, for example, the code:

```
A = X + Y + Z
B = X + Y + W
```

becomes:

```
T1 = X + Y
A = T1 + Z
B = T1 + W
```

Advanced constant folding and propagation

Constant folding computes the value of a constant expression at compile time. Constant propagation is the automatic compile-time replacement of variable references with a constant value previously assigned to that variable.

For example, consider the following C/C++ code:

```
a = 10;
b = a + 5;
c = 4 * b;
```

Once `a` is assigned, its value is propagated to the statement where `b` is assigned so that the assignment reads:

```
b = 10 + 5;
```

The expression `10 + 5` can then be folded. Now that `b` has been assigned a constant, the value of `b` is propagated to the statement where `c` is assigned. After all the folding and propagation, the original code is replaced by:

```
a = 10;
b = 15;
c = 60;
```

Loop-invariant code motion

The loop-invariant code motion optimization recognizes instructions inside a loop whose results do not change and then moves the instructions outside the loop. This optimization ensures that the invariant code is only executed once.

For example, the C/C++ code:

```
x = z;
for(i=0; i<10; i++)
  a[i] = 4 * x + i;
```

becomes:

```
x = z;
t1 = 4 * x;
for(i=0; i<10; i++)
  a[i] = t1 + i;
```

Store/copy optimization

Where possible, the store/copy optimization substitutes registers for memory locations, by replacing store instructions with copy instructions and deleting load instructions.

Unused definition elimination

The unused definition elimination optimization removes unused memory location and register definitions. These definitions are often a result of transformations made by other optimizations.

For example, the function:

```
f(int x){
  int a,b,c;

  a = 1;
  b = 2;
  c = x * b;
  return c;
}
```

becomes:

```
f(int x) {
  int a,b,c;

  c = x * 2;
  return c;
}
```

The assignment `a = 1` is removed because `a` is not used after it is defined. Due to another +O2 optimization (constant propagation), the `c = x * b` statement becomes `c = x * 2`. The assignment `b = 2` is then removed as well.

Software pipelining

Software pipelining is a code transformation that optimizes program loops. It rearranges the order in which instructions are executed in a loop. It generates code that overlaps operations from different loop iterations. Software pipelining is particularly useful for loops that contain arithmetic operations on `REAL*4` and `REAL*8` data in Fortran or on `float` and `double` data in C or C++.

The goal of this optimization is to avoid processor stalls due to memory or hardware pipeline latencies. The software pipelining transformation partially unrolls a loop and adds code before and after the loop to achieve a high degree of optimization within the loop.

You can enable [disable] software pipelining using the `+O[no]pipeline` command-line option at `+O2` and above. The default is `+Opipeline`. Use `+Onopipeline` if a smaller program size and/or faster compile time is more important than faster execution speed.

The following pseudo-code shows a loop before and after the software pipelining optimization. Four significant things happen:

- A portion of the first iteration of the loop is performed before the loop.
- A portion of the last iteration of the loop is performed after the loop.
- The loop is unrolled twice.
- Operations from different loop iterations are interleaved with each other.

Consider the following C/C++ `for` loop:

```
#define SIZ 10000
float x[SIZ], y[SIZ];
int i;
init();
for (i = 0;i<= SIZ;i++)
    x[i] = x[i] / y[i] + 4.00;
```

When this loop is compiled with software pipelining, the optimization can be expressed in pseudo-code as follows:

```
R1 = 0;                          Initialize array index
R2 = 4.00;                       Load constant value
R3 = X[0];                       Load first X value
R4 = Y[0];                       Load first Y value
R5 = R3 / R4;                    Perform division on first element: n = X[0]/Y[0]
  do {                           Begin loop
       R6 = R1;                  Save current array index
       R1++;                     Increment array index
       R7 = X[R1];               Load current X value
       R8 = Y[R1];               Load current Y value
       R9 = R5 + R2;             Perform addition on prior row: X[i] = n + 4.00
       R10 = R7 / R8;            Perform division on current row: m = X[i+1]/Y[i+1]
       X[R6] = R9;               Save result of operations on prior row
       R6 = R1;                  Save current array index
       R1++;                     Increment array index
       R3 = X[R1];               Load next X value
       R4 = Y[R1];               Load next Y value
       R11 = R10 + R2;           Perform addition on current row: X[i+1] = m + 4.00
       R5 = R3 / R4;             Perform division on next row: n = X[i+2]/Y[i+2]
       X[R6] = R11;              Save result of operations on current row
} while (R1 <= 100);             End loop
R9 = R5 + R2;                    Perform addition on last row: X[i+2] = n + 4.00
X[R6] = R9;                      Save result of operations on last row
```

This transformation stores intermediate results of the division instructions in unique registers (noted as `n` and `m`). These registers are not referenced until several instructions after the division operations. This decreases the possibility that the long latency period of the division instructions will stall the instruction pipeline and cause processing delays.

Prerequisites of Pipelining

Software pipelining is attempted on a loop that meets the following criteria:

- It is the innermost loop
- There are no branches or function calls within the loop
- The loop is of moderate size

This optimization produces slightly larger program files and increases compile time. It is most beneficial in programs containing loops that are executed a large number of times.

Register reassociation

Array references often require one or more instructions to compute the virtual memory address of the array element specified by the subscript expression. The register reassociation optimization implemented in PA-RISC compilers tries to reduce the cost of computing the virtual memory address expression for array references found in loops.

Within loops, the virtual memory address expression can be rearranged and separated into a loop-variant term and a loop-invariant term. Loop-variant terms are those items whose values may change from one iteration of the loop to another. Loop-invariant terms are those items whose values are constant throughout all iterations of the loop. The loop-variant term corresponds to the difference in the virtual memory address associated with a particular array reference from one iteration of the loop to the next.

The register reassociation optimization dedicates a register to track the value of the virtual memory address expression for one or more array references in a loop and updates the register appropriately in each iteration of a loop.

The register is initialized outside the loop to the loop-invariant portion of the virtual memory address expression, and the register is incremented or decremented within the loop by the loop-variant portion of the virtual memory address expression.

The net result is that array references in loops are converted into equivalent, but more efficient, pointer dereferences.

Consider the following C/C++ code:

```
int a[10][20][30];

void example (void)
{
    int i, j, k;

    for (k = 0; k < 10; k++)
      for (j = 0; j < 10;j++)
        for (i = 0; i < 10; i++)
           a[i][j][k] = 1;
}
```

After register reassociation is applied, the innermost loop becomes:

```
int a[10][20][30];

void example (void)
{
    int i, j, k;
    register int (*p)[20][30];

    for (k = 0; k < 10; k++)
      for (j = 0; j < 10; j++)
        for (p = (int (*)[20][30]) &a[0][j][k], i = 0; i < 10; i++)
            *(p++[0][0]) = 1;
}
```

In the above example, the compiler-generated temporary register variable, `p`, strides through the array `a` in the innermost loop. This register pointer variable is initialized outside the innermost loop and auto-incremented within the innermost loop as a side-effect of the pointer dereference.

Register reassociation can often enable another loop optimization. After performing the register reassociation optimization, the loop variable may be needed only to control the iteration count of the loop. If this is the case, the original loop variable can be eliminated altogether by using the PA-RISC `ADDIB` and `ADDB` machine instructions to control the loop iteration count.

You can enable [disable] register reassociation using the `+O[no]regreassoc` command-line option at `+O2` and above. The default is `+Oregreassoc`.

Loop unrolling

Loop unrolling increases a loop’s step value and replicates the loop body. Each replication is appropriately offset from the induction variable so that all iterations are performed, given the new step.

Unrolling can be total or partial. Total unrolling involves eliminating the loop structure completely by replicating the loop body a number of times equal to the iteration count and replacing the iteration variable with constants. This makes sense only for loops with small iteration counts.

Consider the following Fortran example:

```
SUBROUTINE FOO(A,B)
REAL A(10,10), B(10,10)
DO J=1, 4
  DO I=1, 4
    A(I,J) = B(I,J)
  ENDDO
ENDDO
END
```

This loop nest is completely unrolled as shown below:

```
A(1,1) = B(1,1)
A(2,1) = B(2,1)
A(3,1) = B(3,1)
A(4,1) = B(4,1)

A(1,2) = B(1,2)
A(2,2) = B(2,2)
A(3,2) = B(3,2)
A(4,2) = B(4,2)

A(1,3) = B(1,3)
A(2,3) = B(2,3)
A(3,3) = B(3,3)
A(4,3) = B(4,3)

A(1,4) = B(1,4)
A(2,4) = B(2,4)
A(3,4) = B(3,4)
A(4,4) = B(4,4)
```

Partial unrolling is performed on loops with larger or unknown iteration counts. It retains the loop structure, but replicates the body a number of times equal to the *unroll factor* and adjusts references to the iteration variable accordingly.

Consider the following Fortran example:

```
DO I = 1, 100
  A(I) = B(I) + C(I)
ENDDO
```

This example can be unrolled to a depth of four as shown below:

```
DO I = 1, 100, 4
  A(I) = B(I) + C(I)
  A(I+1) = B(I+1) + C(I+1)
  A(I+2) = B(I+2) + C(I+2)
  A(I+3) = B(I+3) + C(I+3)
ENDDO
```

Each iteration of the loop now computes four values of A instead of one value. The compiler also generates code for the case where the range is not evenly divisible by the unroll factor.

Loop unrolling and the unroll factor can be controlled using the `+O[no]loop_unroll[=`*unroll_factor*`]` option. See Appendix D, “Optimization options,” for more information on this option.

Some loop transformations cause loops to be fully or partially replicated. Because unlimited loop replication can significantly increase compile times, loop replication is limited by default. You can increase this limit (and possibly increase your program’s compile time and code size) by specifying the `+Onosize` and `+Onolimit` compiler options.

+O3 level optimizations

The +O3 optimizations include the +O2 optimizations, plus full optimization across all subprograms within a single file. The +O3 optimizations include:

- Inlining within a single source file
- Cloning within a single source file
- Test promotion
- Data localization
- Strip mining
- Loop distribution
- Loop interchange
- Loop blocking
- Loop fusion
- Loop unroll and jam
- Parallelization

Also, +O3 is the first optimization level where `+Oparallel` is available. Using `+Oparallel` at this optimization level (and at +O4) enables:

- Automatic and directive-specified loop parallelization
- Directive-specified task parallelization
- Directive-specified region parallelization

NOTE The HP Exemplar aC++ compiler does not support directive-specified parallelization. It does however support the compiler parallelization generated using the `+Oparallel` option.

At +O3, all the directives and pragmas of the Exemplar programming model are available in the Fortran 90, Fortran 77, and C compilers. See Chapter 4, “Basic shared-memory programming,” Chapter 5, “Memory classes,” and Chapter 6, “Advanced shared-memory programming,” for information on using the various features of the Exemplar programming model. (The HP Exemplar aC++ compiler does not support the pragmas of the Exemplar programming model.)

The +O3 optimizations produce faster runtime code than +O2 on code that frequently calls small functions within a file. Linking with +O3 optimizations is faster than linking with +O4 optimizations.

Inlining within a single source file

Inlining substitutes selected function calls with copies of the function's object code. Only functions that meet the optimizer's criteria are inlined. Inlining may result in slightly larger executable files. However, this increase in size is offset by the elimination of time-consuming procedure calls and procedure returns.

The following is an example of inlining at the source code level. Before inlining, the source file looks like this:

```
/* Return the greatest common divisor of two positive integers, */
/* int1 and int2, computed using Euclid's algorithm. (Return 0  */
/* if either is not positive.)                                  */

int gcd(int1,int2)
  int int1;
  int int2;
{
  int inttemp;

  if ( (int1 <= 0) || (int2 <= 0) ) {
    return(0);
  }
  do {
      if (int1 < int2) {
        inttemp = int1;
        int1    = int2;
        int2    = inttemp;
      }
      int1 = int1 - int2;
  } while (int1 > 0);
  return(int2);
}

main()
{
  int xval,yval,gcdxy;
  .
  .       /* statements before call to gcd */
  .
  gcdxy = gcd(xval,yval);
  .
  .       /* statements after call to gcd */
  .
}
```

After inlining, `main` looks like this:

```
main()
{
  int xval,yval,gcdxy;
  .
  .       /* statements before inlined version of gcd */
  .
  {
    int int1;
    int int2;

      int1 = xval;
      int2 = yval;
      {
        int inttemp;

        if ( (int1 <= 0) || (int2 <= 0) ){
           gcdxy = (0);
           goto AA003;
        }
        do {
            if (int1 < int2){
              inttemp = int1;
              int1    = int2;
              int2    = inttemp;
            }
            int1 = int1 - int2;
        } while (int1 > 0);
        gcdxy = (int2);
      }
  }
AA003 : ;
  .
  .       /* statements after inlined version of gcd */
  .
}
```

At `+O3`, inlining is performed within a file; at `+O4`, it is performed across files. Inlining is affected by the `+O[no]inline[=`*namelist*`]` and `+Oinline_budget=`*n* command-line options. See Appendix D, “Optimization options,” for more information.

Cloning within a single source file

Cloning is the replacement of a call to a routine by a call to a clone of that routine. The clone is optimized differently than the original routine. Cloning can expose additional opportunities for interprocedural optimization. At `+O3`, cloning is performed within a file; at `+O4`, it is performed across files. Cloning is enabled by default; it can be disabled by specifying the `+Onoinline` command-line option.

Test promotion

Test promotion involves promoting a test out of the loop that encloses it by replicating the containing loop(s) for each branch of the test. The replicated loops contain fewer tests than the originals, or no tests at all, so the loops execute much faster. Multiple tests can be promoted, and copies of the loop are made for each test.

Consider the following Fortran loop:

```
DO I=1, 100
  DO J=1, 100
    IF(FOO .EQ. BAR) THEN
      A(I,J) = I + J
    ELSE
      A(I,J) = 0
    ENDIF
  ENDDO
ENDDO
```

Test promotion (and loop interchange) produces the following code:

```
IF(FOO .EQ. BAR) THEN
  DO J=1, 100
    DO I=1, 100
      A(I,J) = I + J
    ENDDO
  ENDDO
ELSE
  DO J=1, 100
    DO I=1, 100
      A(I,J) = 0
    ENDDO
  ENDDO
ENDIF
```

For loops containing large numbers of tests, loop replication can greatly increase the size of the code.

Each `DO` loop in Fortran and `for` loop in C and C++ whose bounds are not known at compile-time is implicitly tested to check that the loop will iterate at least once. This test may be promoted, with the promotion noted in the Optimization Report. If you see unexpected promotions in the report, this implicit testing may be the cause. For more information on the Optimization Report, see Appendix E, “Optimization Report.”

Data localization

Data localization occurs by means of various loop transformations that take place at +O2 or +O3. Because optimizations are cumulative, however, specifying +O3 or +O4 takes advantage of the transformations that happen at +O2.

Table 2 Loop transformations affecting data localization

Loop transformation	Options required for behavior to occur
Loop unrolling	`+O2 +Oloop_unroll` (`+Oloop_unroll` is on by default at +O2 and above)
Loop distribution	`+O3 +Oloop_transform` (`+Oloop_transform` is on by default at +O3 and above)
Loop interchange	`+O3 +Oloop_transform`
Loop blocking	`+O3 +Oloop_transform +Oloop_block` (`+Oloop_block` is off by default)
Loop fusion	`+O3 +Oloop_transform`
Loop unroll and jam	`+O3 +Oloop_transform +Oloop_unroll_jam` (`+Oloop_unroll_jam` is on by default at +O3 and above)

Data localization keeps heavily used data in the processor data cache, thus eliminating the need for more costly CTIcache (on multinode, scalable SMPs) or memory accesses.

Loops that manipulate arrays are the main candidates for localization optimizations. Most of these loops are eligible for the various transformations the compiler performs at +O3 to achieve localization. These transformations are explained in detail in this section.

Some loop transformations cause loops to be fully or partially replicated. Because unlimited loop replication can significantly increase compile times, loop replication is limited by default. You can increase this limit (and possibly increase your program's compile time and code size) by specifying the `+Onosize` and `+Onolimit` compiler options.

NOTE Most of the following code examples demonstrate the optimization in question by showing the original code first and optimized code second. While the optimized code is shown in the same language as the original code, this is for illustrative purposes only.

Inhibitors of localization

Any of the following conditions can inhibit or prevent data localization:

- Loop-carried dependences
- Aliased scalar or array variables
- Multiple loop entries or exits
- `RETURN` or `STOP` statements in Fortran
- `return` or `exit` statements in C
- `throw` statements in C++
- Computed or assigned `GOTO` statements in Fortran
- Procedure calls
- I/O statements

The sections below discuss these conditions and their effects on data localization.

Loop-carried dependences

A loop-carried dependence (LCD) exists when one iteration of a loop assigns a value to an address that is referenced or assigned on another iteration. In some cases, LCDs can inhibit loop interchange, thereby inhibiting localization. Typically, these cases involve array indexes that are offset in opposite directions. The Fortran loop below contains an interchange-inhibiting LCD:

```
DO I = 2, M
  DO J = 2, N
    A(I,J) = A(I-1,J-1) + A(I-1,J+1)
  ENDDO
ENDDO
```

C and C++ loops can contain similar constructs, but to simplify illustration, we will only consider this Fortran example.

As written, this loop uses `A(I-1,J-1)` and `A(I-1,J+1)` to compute `A(I,J)`. Table 3 shows the sequence in which values of A are computed for this loop.

Table 3 Computation sequence of `A(I,J)`: original loop

I	J	A(I,J)	A(I-1,J-1)	A(I-1,J+1)
2	2	A(2,2)	A(1,1)	A(1,3)
2	3	A(2,3)	A(1,2)	A(1,4)
2	4	A(2,4)	A(1,3)	A(1,5)
...	...	...	...	...
3	2	A(3,2)	A(2,1)	A(2,3)
3	3	A(3,3)	A(2,2)	A(2,4)
3	4	A(3,4)	A(2,3)	A(2,5)
...	...	...	...	...

As enumerated in Table 3, the original loop computes the elements of the current row of A using the elements of the previous row of A. For all rows except the first (which is never written), the values contained in the previous row must be written before the current row is computed. This dependence must be honored for the loop to yield its intended results. If a row element of A is computed before the previous row elements (that it depends on) are computed, the result will be incorrect.

Interchanging the `I` and `J` loops yields the following code:

```
DO J = 2, N
  DO I = 2, M
    A(I,J) = A(I-1,J+1) + A(I-1,J-1)
  ENDDO
ENDDO
```

After interchange, the loop computes values of A in the sequence shown in Table 4 below.

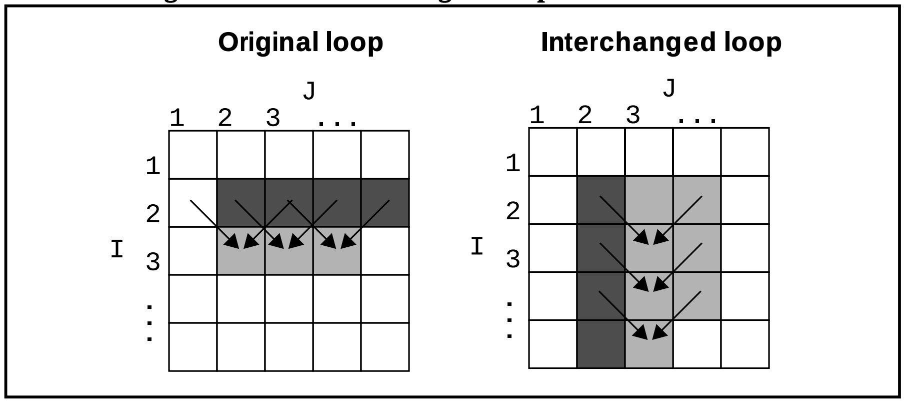
Table 4 **Computation sequence of `A(I,J)`: interchanged loop**

I	J	A(I,J)	A(I-1,J-1)	A(I-1,J+1)
2	2	A(2,2)	A(1,1)	A(1,3)
3	2	A(3,2)	A(2,1)	A(2,3)
4	2	A(4,2)	A(3,1)	A(3,3)
...	...	...	...	...
2	3	A(2,3)	A(1,2)	A(1,4)
3	3	A(3,3)	A(2,2)	A(2,4)
4	3	A(4,3)	A(3,2)	A(3,4)
...	...	...	...	...

Here, the elements of the current column of A are computed using the elements of the previous column and the *next* column of A.

The problem here is that columns of A are being computed using elements from the next column, which have not been written yet. This computation violates the dependence illustrated in Table 3. The element-to-element dependences in both the original and interchanged loop are illustrated in Figure 12.

Figure 12 **LCDs in original and interchanged loops**



The arrows in Figure 12 represent dependences from one element to another; the arrows point at elements that depend on the elements at the arrows' bases. Shaded elements indicate a typical row or column computed in the inner loop:

- Darkly shaded elements have already been computed.
- Lightly shaded elements have not yet been computed.

This figure helps to illustrate the sequence in which the array elements are cycled through by the respective loops: the original loop cycles across all the columns in a row, then moves on to the next row; the interchanged loop cycles down all the rows in a column first, then moves on to the next column.

Interchange is only inhibited by loops that contain dependences that change when the loop is interchanged. Most LCDs do not fall into this category and thus do not inhibit data localization.

Occasionally the compiler encounters an apparent LCD. If it cannot determine whether the LCD actually inhibits interchange, it conservatively avoids interchanging the loop.

The following Fortran example illustrates this situation:

```
DO I = 1, N
  DO J = 2, M
    A(I,J) = A(I+IADD,J+JADD) + B(I,J)
  ENDDO
ENDDO
```

An analogous C example follows:

```
for(j=0;j<n;j++)
  for(i=1;i<m;i++)
    a[i][j] = a[i+IADD][j+JADD] + b[i][j];
```

In these examples, if `IADD` and `JADD` are either both positive or both negative, the loop contains no interchange-inhibiting dependence. However, if one and only one of the variables is negative, interchange is inhibited. The compiler has no way of knowing the runtime values of `IADD` and `JADD`, so it will avoid interchanging the loop. If you are sure the `IADD` and `JADD` will either both be negative or both be positive, you can indicate to the compiler that the loop is free of dependences using the `no_loop_dependence` compiler directive or pragma.

In Fortran, this directive has the form:

`C$DIR NO_LOOP_DEPENDENCE(`*namelist*`)`

The `no_loop_dependence` C pragma has the form:

`#pragma _CNX no_loop_dependence(`*namelist*`)`

where

namelist	is a comma-separated list of variables and/or arrays that have no dependences for the immediately following loop.

The previous Fortran loop can be interchanged when the `NO_LOOP_DEPENDENCE` directive is specified for `A` on the `J` loop as shown in the following code:

```
      DO I = 1, N
C$DIR   NO_LOOP_DEPENDENCE(A)
        DO J = 2, M
          A(I,J) = A(I+IADD,J+JADD) + B(I,J)
        ENDDO
      ENDDO
```

The `no_loop_dependence` pragma can similarly be used on the `C` loop:

```
for(i=0;i<n;i++)
#pragma _CNX no_loop_dependence(a)
  for(j=1;j<m;j++)
    a[i][j] = a[i+IADD][j+JADD] + b[i][j];
```

If `IADD` and `JADD` acquire opposite-signed values at runtime, these loops may result in incorrect answers.

Dependences and loop fusion

In some cases, loop fusion is also inhibited by simpler dependences than those that inhibit interchange. Consider the following Fortran example:

```
DO I = 1, N-1
  A(I) = B(I+1) + C(I)
ENDDO
DO J = 1, N-1
  D(J) = A(J+1) + E(J)
ENDDO
```

It would appear that this loop would profit from fusion. Fusing it would yield the following incorrect code:

```
DO ITEMP = 1, N-1
  A(ITEMP) = B(ITEMP+1) + C(ITEMP)
  D(ITEMP) = A(ITEMP+1) + E(ITEMP)
ENDDO
```

This loop produces different answers than the original loops, because the reference to `A(ITEMP+1)` in the fused loop accesses a value that has not been assigned yet, while the analogous reference to `A(J+1)` in the original `J` loop accesses a value that was assigned in the original `I` loop.

An analogous C/C++ example follows:

```
for(i=0; i<n-1; i++)
  a[i] = b[i+1] + c[i];
for(j=0; j<n-1; j++)
  d[j] = a[j+1] + e[j];
```

After fusion:

```
for(itemp=0; itemp<n-1; itemp++) {
  a[itemp] = b[itemp+1] + c[itemp];
  d[itemp] = a[itemp+1] + e[itemp];
}
```

Aliasing

An *alias* is an alternate name for some object. Aliasing occurs in a program when two or more names are attached to the same memory location. Aliasing is typically caused in Fortran by use of the `EQUIVALENCE` statement and in C and C++ by use of pointers. Passing identical actual arguments into different dummy arguments in a Fortran subprogram can also cause aliasing, as can passing the same address into different pointers in a C or C++ function.

Aliasing interferes with data localization because it can mask LCDs, as shown in the following Fortran example, where the arrays A and B have been equivalenced:

```
INTEGER A(100,100), B(100,100), C(100,100)
EQUIVALENCE(A,B)
.
.
.
DO I = 1, N
  DO J = 2, M
    A(I,J) = B(I-1,J+1) + C(I,J)
  ENDDO
ENDDO
```

This loop has the same problem as the loop used to demonstrate LCDs in the previous section; because A and B refer to the same array, the loop contains an LCD on A, which prevents interchange and thus interferes with localization.

The C/C++ equivalent of this loop follows. Keep in mind that C and C++ store arrays in row-major order, which requires different subscripting to access the same elements.

```
int a[100][100], c[100][100], i, j;
int (*b)[100];
b = a;
.
.
.
for(i=1;i<n;i++){
  for(j=0;j<m;j++){
    a[j][i] = b[j+1][i-1] + c[j][i];
  }
}
```

Fortran’s EQUIVALENCE statement can be imitated in C and C++; through the use of pointers, arrays can be effectively equivalenced, as shown.

Passing the same address into different dummy procedure arguments can yield the same result. Fortran passes arguments by reference while C and C++ pass them by value, but pass-by-reference can be simulated in C and C++ by passing the argument’s address into a pointer in the receiving procedure or in C++ by using references.

The following Fortran code exhibits the same aliasing problem as the previous example, but the alias is created by passing the same actual argument into different dummy arguments.

NOTE The code below violates the Fortran standard.

```
.
.
.
CALL ALI(A,A,C)
.
.
.
SUBROUTINE ALI(A,B,C)
INTEGER A(100,100), B(100,100), C(100,100)
DO J = 1, N
  DO I = 2, M
    A(I,J) = B(I-1,J+1) + C(I,J)
  ENDDO
ENDDO
.
.
.
```

The following (legal ANSI C) code shows the same argument-passing problem in C:

```
.
.
.
ali(&a,&a,&c);
.
.
.
void ali(a,b,c)
int a[100][100], b[100][100], c[100][100];
{
  int i,j;
  for(j=0;j<n;j++){
    for(i=1;i<m;i++){
      a[j][i] = b[j+1][i-1] + c[j][i];
    }
  }
}
```

Multiple loop entries or exits

Loops that contain multiple entries or exits inhibit data localization because they cannot safely be interchanged. Extra loop entries are usually created when a loop contains a branch destination. Extra exits are more common; they are often created in C and C++ using the `break` statement and in Fortran using the `GOTO` statement.

Consider the following C/C++ code:

```
for(j=0;j<n;j++){
  for(i=0;i<m;i++){
    a[i][j] = b[i][j] + c[i][j];
    if(a[i][j] == 0) break;
    .
    .
    .
  }
}
```

Interchanging this loop would change the order in which the values of `a` are computed; the original loop computes `a` column-by-column, whereas the interchanged loop would compute it row-by-row. This means that the interchanged loop may hit the `break` statement and exit after computing a different set of elements than the original loop computes. Interchange therefore may cause the results of the loop to differ and must be avoided.

A similar loop construct written in Fortran follows:

```
      DO J = 1, M
        DO I = 1, N
          A(I,J) = B(I,J) + C(I,J)
          IF(A(I,J) .EQ. 0) GOTO 50
           .
           .
           .
        ENDDO
      ENDDO
       .
       .
       .
50    CONTINUE
```

Again, the order of computation changes if the loops are interchanged.

`RETURN` or `STOP` statements in Fortran

Like loops with multiple exits, `RETURN` and `STOP` statements in Fortran inhibit localization because they inhibit interchange. If a loop containing a `RETURN` or `STOP` is interchanged, its order of computation may change, giving wrong answers.

`return` or `exit` statements in C or C++

Similar to Fortran’s `RETURN` and `STOP` statements (discussed in the previous section), `return` and `exit` statements in C and C++ inhibit localization because they inhibit interchange.

`throw` statements in C++

In C++, `throw` statements, like loops containing multiple exits, inhibit localization because they inhibit interchange.

Computed or assigned `GOTO` statements in Fortran

When the Fortran compiler encounters a computed or assigned `GOTO` statement in an otherwise interchangeable loop, it cannot always determine whether the branch destination is within the loop. Because an out-of-loop destination would be a loop exit, these statements often prevent loop interchange and therefore data localization.

Procedure calls

The Exemplar compilers are unaware of the side effects of most procedures, and therefore cannot determine whether they might interfere with loop interchange; consequently, the compilers will not perform loop interchange. These side effects may include data dependences involving loop arrays, aliasing (as described in the section "Aliasing" on page 73), and processor data cache usage that conflicts with the loop's usage of the cache, rendering useless any data localization optimizations performed on the loop.

I/O statements

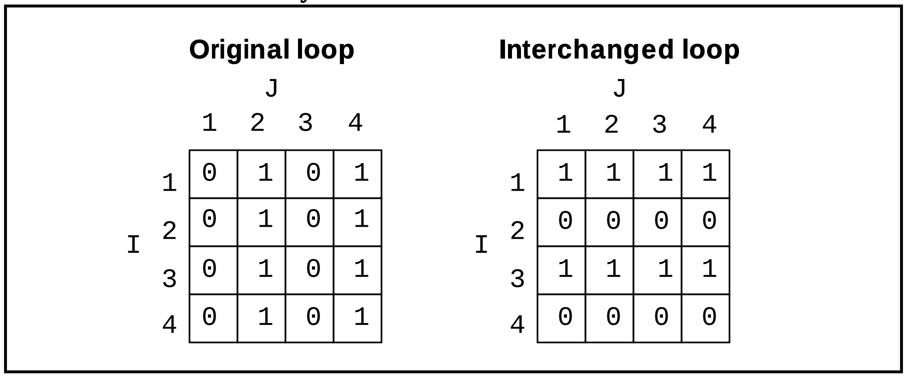
The order in which values are read into or written from a loop may change if the loop is interchanged, so I/O statements inhibit interchange and therefore data localization.

For example, consider the following Fortran code:

```
DO I = 1, 4
  DO J = 1, 4
    READ *, IA(I,J)
  ENDDO
ENDDO
```

Given a data stream consisting of alternating zeros and ones (0,1,0,1,0,1...), the contents for `A(I,J)` for both the original loop and the interchanged loop are shown in Figure 13.

Figure 13 **Values read into array A**



Original loop

I \ J	1	2	3	4
1	0	1	0	1
2	0	1	0	1
3	0	1	0	1
4	0	1	0	1

Interchanged loop

I \ J	1	2	3	4
1	1	1	1	1
2	0	0	0	0
3	1	1	1	1
4	0	0	0	0

C and C++ loops exhibit the same limitations. A C/C++ example that produces the data patterns shown in Figure 13 follows:

```
for(i=1;i<5;i++)
  for(j=1;j<5;j++)
    scanf("%d",&ia[i][j]);
```

Preventing loop reordering

The `no_loop_transform` directive or pragma allows you to prevent all loop-reordering transformations on the immediately following loop. In Fortran, it has the form:

```
C$DIR NO_LOOP_TRANSFORM
```

In C it has the form:

```
#pragma _CNX no_loop_transform
```

You can use the command-line option `+Onoloop_transform` (at `+O3` and above) to disable loop distribution, loop blocking, loop fusion, loop interchange, loop unroll, and loop unroll and jam on a file basis.

Strip mining

Strip mining is a fundamental +O3 transformation. In and of itself, strip mining is not profitable. However, it is used by loop blocking, loop unroll and jam, and, in a sense, by parallelization.

Strip mining involves splitting a single loop into a nested loop. The resulting inner loop iterates over a section or *strip* of the original loop, and the new outer loop runs the inner loop enough times to cover all the strips, achieving the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's *strip length*.

Consider the following Fortran code:

```
DO I = 1, 10000
  A(I) = A(I) * B(I)
ENDDO
```

Strip mining this loop using a strip length of 1000 yields the following loop nest:

```
DO IOUTER = 1, 10000, 1000
  DO ISTRIP = IOUTER, IOUTER+999
    A(ISTRIP) = A(ISTRIP) * B(ISTRIP)
  ENDDO
ENDDO
```

In this loop, the strip length integrally divides the number of iterations, so the loop is evenly split up. If the iteration count was not an integral multiple of the strip length, for example, if I went from 1 to 10500 rather than 1 to 10000, the final iteration of the strip loop would execute 500 iterations instead of 1000.

Loop distribution

Loop distribution takes place at +O3 and above and is enabled by default. Specifying +Onoloop_transform disables loop distribution (as well as loop interchange, loop blocking, loop fusion, loop unroll, and loop unroll and jam).

Loop distribution is another fundamental +O3 transformation that is necessary for some more advanced transformations. These advanced transformations require that all calculations in a nested loop be performed inside the innermost loop. To facilitate this, loop distribution transforms complicated nested loops into several simple loops (or nests) that contain all computations inside the body of the innermost loop.

Consider the following Fortran code:

```
DO I = 1, N
  C(I) = 0
  DO J = 1, M
    A(I,J) = A(I,J) + B(I,J) * C(I)
  ENDDO
ENDDO
```

Loop distribution creates two copies of the I loop, separating the nested J loop from the assignments to array C. In this way, all assignments are moved to innermost loops. Interchange is then performed on the I and J loops. The distribution and interchange is shown in the following transformed code:

```
DO I = 1, N
  C(I) = 0
ENDDO
DO J = 1, M
  DO I = 1, N
    A(I,J) = A(I,J) + B(I,J) * C(I)
  ENDDO
ENDDO
```

An analogous C/C++ example follows:

```
for(j=0;j<n;j++) {
  c[j] = 0;
  for(i=0;i<m;i++)
    a[i][j] = a[i][j] + b[i][j] * c[j];
}
```

This loop is distributed and interchanged as shown below:

```
for(j=0;j<n;j++)
  c[j] = 0;
for(i=0;i<m;i++)
  for(j=0;j<n;j++)
    a[i][j] = a[i][j] + b[i][j] * c[j];
```

Distribution can improve efficiency by reducing the number of memory references per loop iteration, and can reduce cache thrashing. It also creates more opportunities for interchange.

Loop distribution can be disabled for specific loops by specifying the `no_distribute` directive or pragma immediately before the loop.

In Fortran, it has the form:

```
C$DIR NO_DISTRIBUTE
```

In C:

```
#pragma _CNX no_distribute
```

Loop interchange

Loop interchange takes place at `+O3` and above and is enabled by default. Specifying `+Onoloop_transform` disables loop interchange (as well as loop distribution, loop blocking, loop fusion, loop unroll, and loop unroll and jam).

The compiler may interchange (or reorder) nested loops for the following reasons:

- To facilitate other transformations
- To relocate the loop that is the most profitable to parallelize so that it is outermost
- To optimize inner-loop memory accesses

Consider the Fortran matrix addition algorithm that follows:

```
DO I = 1, N
  DO J = 1, M
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

This loop accesses the arrays A, B and C row by row, which, in Fortran, is very inefficient. Interchanging the I and J loops, as shown in the following example, will facilitate column by column access.

```
DO J = 1, M
  DO I = 1, N
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

Unlike Fortran, C and C++ access arrays in row-major order. An analogous example in C/C++, then, employs an opposite nest ordering, as shown below.

```
for(j=0;j<m;j++)
  for(i=0;i<n;i++)
    a[i][j] = b[i][j] + c[i][j];
```

Interchange facilitates row-by-row access. The interchanged loop is shown below.

```
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
    a[i][j] = b[i][j] + c[i][j];
```

Loop blocking

The loop-blocking optimization is only available at +O3 (and above) in the Fortran 90 and aC++ compilers, and in the C Version 2.0 compiler. Loop blocking is disabled by default. To enable loop blocking, use the +Oloop_block option. Specifying +Onoloop_block (the default) disables both automatic and directive-specified loop blocking. Specifying +Onoloop_transform also disables loop blocking (as well as loop distribution, loop interchange, loop fusion, loop unroll, and loop unroll and jam).

Loop blocking is a combination of strip mining and interchange that maximizes data localization. It is provided primarily to deal with nested loops that manipulate arrays that are too large to fit into the cache. Under certain circumstances, loop blocking allows reuse of these arrays by transforming the loops that manipulate them so that they manipulate strips of the arrays that fit into the cache. Effectively, a blocked loop accesses array elements in sections that are optimally sized to fit in the cache.

Data reuse

Data reuse is important to understand when discussing blocking. There are two types of data reuse associated with loop blocking:

- *Spatial reuse*
- *Temporal reuse*

Spatial reuse is using data that was encached as a result of fetching another piece of data from memory. Remember that data is fetched by cache lines; 32 bytes of data is encached on every fetch on V2200 and X2000 servers. (Cache line sizes may be different on other HP SMPs.) On the initial fetch of array data from memory within a stride-one loop, the requested item can be located anywhere in the 32 bytes, unless the array is aligned on cache line boundaries (refer to the section “Data alignment” on page 28). Starting with the second memory fetch, the requested data is at the beginning of the cache line, and the rest of the cache line will contain subsequent array elements. For a `REAL*4` array, this means the requested element and the seven following elements are encached on each fetch after the first. If any of these seven elements could then be used, say on any subsequent iterations of the loop, the loop would be exploiting spatial reuse. For loops with strides greater than one, spatial reuse can still occur; however, the cache lines will contain fewer usable elements.

Temporal reuse is using the same data item on more than one iteration of the loop. An array element whose subscript does not change as a function of the iterations of a surrounding loop exhibits temporal reuse in the context of the loop.

Loops that stride through arrays are candidates for blocking when there is an outermore loop that carries spatial or temporal reuse. Blocking the innermore loop allows data referenced by the outermore loop to remain in the cache across multiple iterations. Blocking exploits spatial reuse by ensuring that once fetched, cache lines are not overwritten until their spatial reuse is exhausted. Temporal reuse is similarly exploited.

Reuse example

The following Fortran loop nest exhibits both spatial and temporal reuse:

```
REAL*4 A(100,100), B(100,100), C(100)
COMMON /BLK1/ A, B, C
.
.
.
DO J = 1, 100
  DO I= 1, 100
    A(I,J) = B(J,I) + C(I)
  ENDDO
ENDDO
```

As written, this loop nest contains spatial reuse on the A and B arrays, and both spatial and temporal reuse on the C array. Because the arrays are in a COMMON block and each array is of a total length that is an integral multiple of the X2000 CTIcache line length (32 bytes), we know that each array will begin on a CTIcache line boundary. All cache lines fetched will be full of reusable data. Spatial reuse is achieved on the A array because every 8th iteration of the I loop fetches a cache line containing 8 of its elements; the 7 iterations between main memory accesses can proceed with virtually no load delays. This continues throughout the entire range of the J loop.

Similar spatial reuse is achieved on the B array. During the first iteration of J, every referenced element of B, along with its containing cache line, will be fetched from memory. All the elements contained in this cache line will be reusable on some subsequent iteration of one of the loops. On subsequent iterations of J, a cache line will be fetched from memory only if the required element was not previously encached, and all the elements it contains will be usable. However, keep in mind that fetches are a function of I and may occur for different J values (after J = 1) based on the value of I. Because B's row index is J, any unused encached elements are used on the subsequent iterations of J for a given value of I. Though the data is not used as immediately as it is for A, spatial reuse is still fully exploited.

Spatial reuse is similarly achieved on the C array, but only for J = 1. Assuming the loop is compiled as written, when the first iteration of J finishes, C is completely contained in the processor data cache and will remain there for the duration of J. C can then be temporally reused for every subsequent iteration of J.

This loop does not require blocking to achieve this reuse because the arrays occupy less than 80 kbytes altogether, so they fit easily into the cache. Spatial reuse is graphically illustrated for a similar but more realistic example in the following section.

Blocking example: simple loop

In order to exploit reuse in more realistic examples that manipulate arrays that will not all fit in the cache, the compiler can apply the blocking transformation.

Consider the following Fortran example:

```
REAL*8 A(1000,1000),B(1000,1000)
REAL*8 C(1000),D(1000)
COMMON /BLK2/ A, B, C
.
.
.
DO J = 1, 1000
  DO I = 1, 1000
    A(I,J) = B(J,I) + C(I) + D(J)
  ENDDO
ENDDO
```

Here the array elements occupy nearly 16 Mbytes of memory—much larger than the 2 Mbyte cache on PA-8200 processors and the 1 Mbyte cache on PA-8000 processors. Thus, blocking becomes quite profitable.

First the compiler strip mines the `I` loop:

```
DO J = 1, 1000
  DO IOUT = 1, 1000, IBLOCK
    DO I = IOUT, IOUT+IBLOCK-1
      A(I,J) = B(J,I) + C(I) + D(J)
    ENDDO
  ENDDO
ENDDO
```

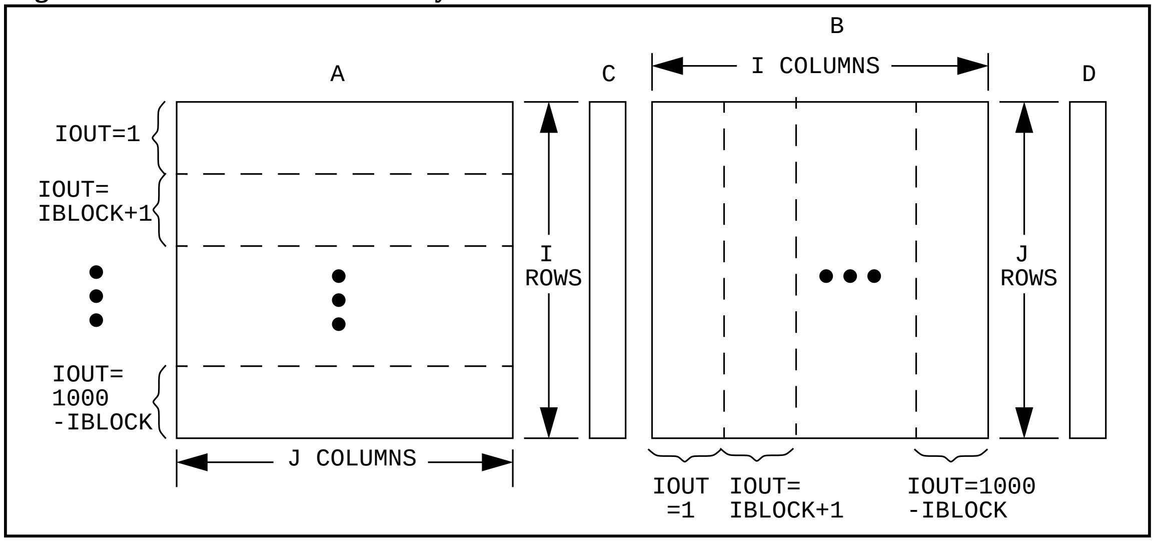
`IBLOCK` is the block factor (also referred to as the strip mine length) the compiler chooses based on the size of the arrays and size of the cache. Note that this example assumes the chosen `IBLOCK` divides 1000 evenly.

Now the compiler moves the outer strip loop (IOUT) outward as far as possible.

```
DO IOUT = 1, 1000, IBLOCK
  DO J = 1, 1000
    DO I = IOUT, IOUT+IBLOCK-1
      A(I,J) = B(J,I) + C(I) + D(J)
    ENDDO
  ENDDO
ENDDO
```

This new nest accesses IBLOCK rows of A and IBLOCK columns of B for every iteration of J. At every iteration of IOUT, the nest accesses 1000 IBLOCK-length columns of A (or an IBLOCK × 1000 chunk of A) and 1000 IBLOCK-width rows of B are accessed. This is illustrated in Figure 14.

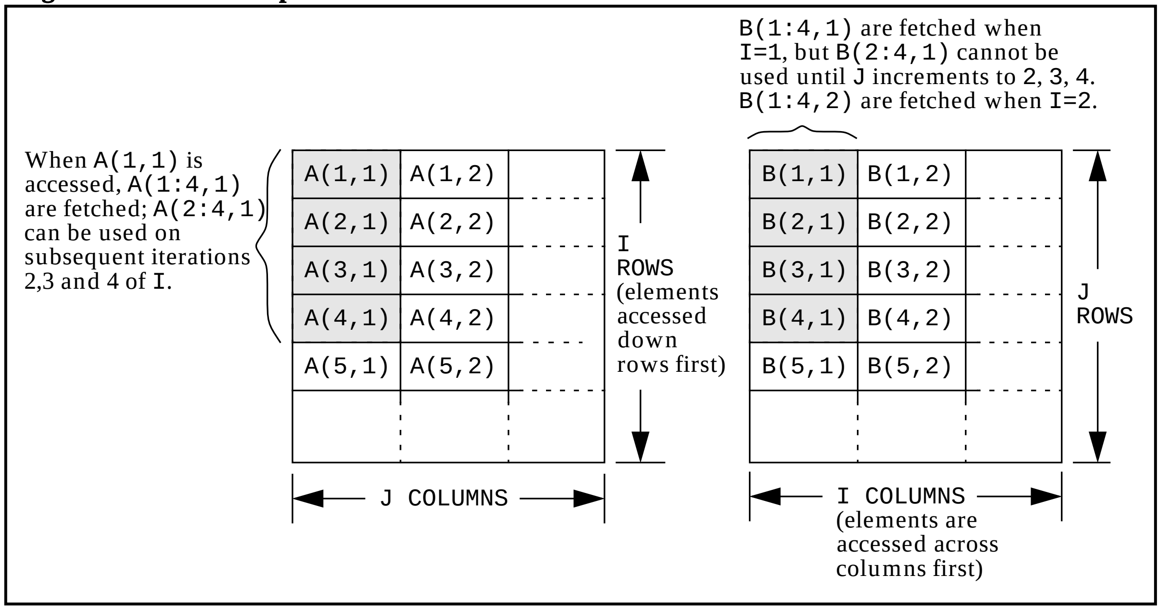
Figure 14 Blocked array access



Fetches of A encache the needed element and the three elements that are used in the three subsequent iterations, giving spatial reuse on A. Since the I loop traverses columns of B, fetches of B encache extra elements that will not be spatially reused until J increments. IBLOCK is chosen by the compiler to efficiently exploit spatial reuse of both A and B.

Figure 15 illustrates how cache lines of each array are fetched (A and B both start on cache line boundaries because they are in COMMON and are of lengths integrally divisible by the X2000 CTIcache line length). The shaded area represents the initial cache line fetched.

Figure 15 **Spatial reuse of A and B**


B(1:4,1) are fetched when I=1, but B(2:4,1) cannot be used until J increments to 2, 3, 4. B(1:4,2) are fetched when I=2.
When A(1,1) is accessed, A(1:4,1) are fetched; A(2:4,1) can be used on subsequent iterations 2,3 and 4 of I.
A(1,1)
A(1,2)
A(2,1)
A(2,2)
A(3,1)
A(3,2)
A(4,1)
A(4,2)
A(5,1)
A(5,2)
I ROWS (elements accessed down rows first)
B(1,1)
B(1,2)
B(2,1)
B(2,2)
B(3,1)
B(3,2)
B(4,1)
B(4,2)
B(5,1)
B(5,2)
J ROWS
J COLUMNS
I COLUMNS (elements are accessed across columns first)

Typically, IBLOCK elements of C will remain in the cache for several iterations of J before being overwritten, giving temporal reuse on C for those iterations. By the time any of the arrays are overwritten, all spatial reuse has been exhausted. The load of D is hoisted out of the I loop so that it remains in a register for all iterations of I.

Blocking example: matrix multiply

The more complicated matrix multiply algorithm, which follows, is a prime candidate for blocking:

```
REAL*8 A(1000,1000),B(1000,1000),C(1000,1000)
COMMON /BLK3/ A, B, C
.
.
.
DO I = 1, 1000
  DO J = 1, 1000
    DO K = 1, 1000
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

This loop is blocked as shown below:

```
DO IOUT = 1, 1000, IBLOCK
  DO KOUT = 1, 1000, KBLOCK
    DO J = 1, 1000
      DO I = IOUT, IOUT+IBLOCK-1
        DO K = KOUT, KOUT+KBLOCK-1
          C(I,J) = C(I,J) + A(I,K) * B(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Here, we get:

- Spatial reuse of B with respect to the K loop
- Temporal reuse of B with respect to the I loop
- Spatial reuse of A with respect to the I loop
- Temporal reuse of A with respect to the J loop
- Spatial reuse of C with respect to the I loop
- Temporal reuse of C with respect to the K loop

An analogous C/C++ example follows:

```
static double a[1000][1000], b[1000][1000];
static double c[1000][1000];
.
.
.
for(i=0;i<1000;i++)
    for(j=0;j<1000;j++)
      for(k=0;k<1000;k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Exemplar C and C++ interchange and block the loop in this example to provide optimal access efficiency for the row-major C/C++ arrays. The blocked loop is shown below:

```
for(jout=0;jout<1000;jout+=jblk)
  for(kout=0;kout<1000;kout+=kblk)
    for(i=0;i<1000;i++)
      for(j=jout;j<jout+jblk;j++)
        for(k=kout;k<kout+kblk;k++)
          c[i][j]=c[i][j]+a[i][k]*b[k][j];
```

As you can see, the interchange was done differently because of C/C++'s different array storage. This code yields:

- Spatial reuse of `b` with respect to the `j` loop
- Temporal reuse of `b` with respect to the `i` loop
- Spatial reuse of `a` with respect to the `k` loop
- Temporal reuse of `a` with respect to the `j` loop
- Spatial reuse on `c` with respect to the `j` loop
- Temporal reuse on `c` with respect to the `k` loop

Blocking is inhibited when loop interchange is inhibited. If a candidate loop nest contains loops that cannot be interchanged, blocking will not be performed.

Blocking directives and pragmas

The `+Onoloop_block` command-line option (the default), disables both automatic and directive-specified loop blocking. Loop blocking can be enabled for specific loops using the `block_loop` compiler directive and pragma. You can also advise the compiler to use a specific block factor via the `block_loop` directive and pragma. You can use the `no_block_loop` directive and pragma to disable loop blocking for a particular loop.

In Fortran, these directives have the following form:

```
C$DIR BLOCK_LOOP[(BLOCK_FACTOR = n)]
C$DIR NO_BLOCK_LOOP
```

In C, these pragmas have the form:

```
#pragma _CNX block_loop[(block_factor = n)]
#pragma _CNX no_block_loop
```

In the block_loop directive and pragma, *n* is the requested block factor, which must be a compile-time integer constant. The compiler will use this value as stated, so, for best performance, the block factor multiplied by the data type size of the data in the loop should be an integral multiple of the cache line size. In the absence of the block_factor argument, this directive is useful for indicating which loop in a nest to block. In this case, the compiler uses a heuristic to determine the block factor.

These directives and pragmas affect the loop that immediately follows them.

Reconsider the matrix multiply example, this time with a block_loop directive:

```
      REAL*8 A(1000,1000),B(1000,1000)
      REAL*8 C(1000,1000)
      COMMON /BLK3/ A, B, C
      .
      .
      .
      DO I = 1,1000
        DO J = 1, 1000
C$DIR     BLOCK_LOOP(BLOCK_FACTOR = 112)
          DO K = 1,1000
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
          ENDDO
        ENDDO
      ENDDO
```

We know from the original example involving this code that the compiler blocks the I and K loops. In this example, the BLOCK_LOOP directive instructs the compiler to use a block factor of 112 for the K loop. This is an efficient blocking factor for this example because 112 $\times$ 8 bytes = 896 bytes, and 896/32 bytes (the cache line size) = 28, which is an integer, so partial cache lines will not be needed. The compiler-chosen value is still used on the I loop.

Loop fusion

Loop fusion takes place at `+O3` and above and is enabled by default. Specifying `+Onoloop_transform` disables loop fusion (as well as loop distribution, loop interchange, loop blocking, loop unroll, and loop unroll and jam).

Loop fusion involves creating one loop out of two or more neighboring loops that have identical loop bounds and trip counts. This reduces loop overhead, memory accesses, and register usage, and can lead to other optimizations. By potentially reducing the number of parallelizable loops in a program and increasing the amount of work in each of those loops, loop fusion can greatly reduce parallelization overhead, because fewer spawns and joins will be necessary.

Consider the following Fortran code:

```
DO I = 1, N
  A(I) = B(I) + C(I)
ENDDO
DO J = 1, N
  IF(A(J) .LT. 0) A(J) = B(J)*B(J)
ENDDO
```

These two loops can be fused into the following loop:

```
DO I = 1, N
  A(I) = B(I) + C(I)
  IF(A(I) .LT. 0) A(I) = B(I)*B(I)
ENDDO
```

Occasionally loops that do not appear to be fusible become fusible as a result of compiler transformations that precede fusion. For instance, interchanging a loop may make it suitable for fusing with another loop.

Loop fusion is especially beneficial when applied to Fortran 90 array assignments or to Fortran 90-style array assignments used in HP Fortran 77 programs. The compiler translates these statements into loops; when such loops do not contain code that would inhibit fusion, they can be fused.

Consider the following Fortran example:

```
REAL A(100,100), B(100,100), C(100,100)
.
.
.
C = 2.0 * B
A = A + B
```

The compiler would first transform these Fortran 90 array assignments into loops, generating code similar to that shown below.

```
DO TEMP1 = 1, 100
  DO TEMP2 = 1, 100
    C(TEMP2, TEMP1) = 2.0 * B(TEMP2, TEMP1)
  ENDDO
ENDDO
DO TEMP3 = 1, 100
  DO TEMP4 = 1, 100
    A(TEMP4,TEMP3)=A(TEMP4,TEMP3)+B(TEMP4,TEMP3)
  ENDDO
ENDDO
```

These two loops would then be fused as shown in the following loop nest:

```
DO TEMP1 = 1, 100
  DO TEMP2 = 1, 100
    C(TEMP2,TEMP1) = 2.0 * B(TEMP2, TEMP1)
    A(TEMP2,TEMP1)=A(TEMP2,TEMP1)+B(TEMP2,TEMP1)
  ENDDO
ENDDO
```

And the compiler can apply further optimizations to this new nest.

Loop peeling to enable fusion

When trip counts of adjacent loops differ by only a single iteration (+1 or -1), the compiler may peel an iteration from one of the two loops so that the loops may then be fused. The peeled iteration is performed separately from the original loop.

Consider the following example:

```
DO I = 1, N-1
  A(I) = I
ENDDO

DO J = 1, N
  A(J) = A(J) + 1
ENDDO
```

Here, the `N`th iteration of the `J` loop is peeled, resulting in a trip count of `N - 1`. The `N`th iteration is performed outside the `J` loop. The example now looks like the following:

```
DO I = 1, N-1
  A(I) = I
ENDDO

DO J = 1, N-1
  A(J) = A(J) + 1
ENDDO

A(N) = A(N) + 1
```

The `I` and `J` loops now have the same trip count and can be fused, as shown below:

```
DO I = 1, N-1
  A(I) = I
  A(I) = A(I) + 1
ENDDO

A(N) = A(N) + 1
```

Loop unroll and jam

Loop unroll and jam takes place at `+O3` and above and is enabled by default in the Fortran 90 and aC++ compilers, and in the C Version 2.0 compiler. To disable loop unroll and jam on the command line, use the `+Onoloop_unroll_jam` option, which prevents both automatic and directive-specified unroll and jam. The `no_unroll_and_jam` directive and pragma (available in Fortran 90 and C) can be used to disable loop unroll and jam for an individual loop. Also, specifying `+Onoloop_transform` disables loop unroll and jam (as well as loop distribution, loop interchange, loop blocking, loop fusion, and loop unroll).

The loop unroll and jam transformation is primarily intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop. Improved register usage decreases the need for main memory accesses and sometimes allows better exploitation of certain machine instructions.

Unroll and jam involves partially unrolling one or more loops higher in the nest than the innermost loop, and fusing (“jamming”) the resulting loops back together. For unroll and jam to be effective, a loop must be nested and must contain data references that can be temporally reused with respect to some loop other than the innermost. (See the section “Data reuse” on page 84 for information on temporal reuse.) The unroll and jam optimization is automatically applied only to those loops that consist strictly of a basic block.

Consider the following matrix multiply loop:

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

Here, the compiler can exploit a maximum of 3 registers: one for `A(I,J)`, one for `B(I,K)`, and one for `C(K,J)`.

Register exploitation can be vastly increased on this loop by unrolling and jamming the `I` and `J` loops. First, the compiler unrolls the `I` loop. To simplify the illustration, we will use an unrolling factor of 2 for `I`. This is the number of times the contents of the loop will be replicated.

The following Fortran example shows this replication:

```
DO I = 1, N, 2
  DO J = 1, N
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
    ENDDO
  ENDDO
  DO J = 1, N
    DO K = 1, N
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

The “jam” part of unroll and jam occurs when the loops are fused back together, to create the following:

```
DO I = 1, N, 2
  DO J = 1, N
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

This new loop can exploit registers for two additional references: `A(I,J)` and `A(I+1,J)`. However, the compiler still has the `J` loop to unroll and jam. We will use an unroll factor of 4 for the `J` loop, in which case unrolling gives the following:

```
DO I = 1, N, 2
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
    ENDDO
    DO K = 1, N
      A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
      A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
    ENDDO
    DO K = 1, N
      A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
      A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
    ENDDO
    DO K = 1, N
      A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
      A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
    ENDDO
  ENDDO
ENDDO
```

Jamming the unrolled loop, we get:

```
DO I = 1, N, 2
  DO J = 1, N, 4
    DO K = 1, N
      A(I,J) = A(I,J) + B(I,K) * C(K,J)
      A(I+1,J) = A(I+1,J) + B(I+1,K) * C(K,J)
      A(I,J+1) = A(I,J+1) + B(I,K) * C(K,J+1)
      A(I+1,J+1) = A(I+1,J+1) + B(I+1,K) * C(K,J+1)
      A(I,J+2) = A(I,J+2) + B(I,K) * C(K,J+2)
      A(I+1,J+2) = A(I+1,J+2) + B(I+1,K) * C(K,J+2)
      A(I,J+3) = A(I,J+3) + B(I,K) * C(K,J+3)
      A(I+1,J+3) = A(I+1,J+3) + B(I+1,K) * C(K,J+3)
    ENDDO
  ENDDO
ENDDO
```

This new loop exploits more registers and requires fewer loads and stores than the original. Recall that the original loop could use no more than 3 registers. This unrolled-and-jammed loop can use 14, one for each of the following references:

```
A(I,J)        B(I,K)        C(K,J)        A(I+1,J)
B(I+1,K)      A(I,J+1)      C(K,J+1)      A(I+1,J+1)
A(I,J+2)      C(K,J+2)      A(I,J+3)      A(I+1,J+2)
A(I+1,J+3)    C(K,J+3)
```

Fewer loads and stores per operation are required because all of the registers containing these elements are referenced at least twice. This particular example can also benefit from the PA-RISC `FMPYFADD` instruction, which is available with PA-8*x*00 processors. This instruction doubles the speed of the operations in the body of the loop by simultaneously performing related adds and multiplies. With PA-7*x*00 processors, the `FMPYADD` instruction (notice that this instruction does not have the second `F` like the PA-8*x*00 instruction above) would be used to double the speed of the operations; however, with this instruction, the adds and multiplies are unrelated.

Remember, this is a very simplified example. In reality, the compiler attempts to exploit as many of the PA-RISC processor's registers as possible. For the matrix multiply algorithm used here, the compiler would pick a larger unrolling factor, creating a much larger `K` loop body. This would result in increased register exploitation and fewer loads and stores per operation.

However, excessive unrolling may introduce extra register spills if the unrolled and jammed loop body becomes too large. These register spills may have negative effects on performance.

Try to choose unroll factors that align data references in the innermost loop on cache boundaries. When this is achieved, references to the consecutive memory regions in the innermost loop can have very high cache hit ratios. Therefore, unroll factors of 5 or 7 may not be good choices, because most array element sizes are either 4 bytes or 8 bytes and the cache line size (on V2200 and X2000 servers) is 32 bytes. Thus, an unroll factor of 2 or 4 is more likely to effectively exploit cache line reuse for the references that access consecutive memory regions.

Unroll and jam is enabled by default at optimization levels `+O3` and higher. Unroll and jam can be disabled on the command line using the `+Onoloop_transform` option.

You can specify unroll and jam for individual loops by using the `unroll_and_jam` compiler directive and pragma. In Fortran, it has the following form:

```
C$DIR UNROLL_AND_JAM[(UNROLL_FACTOR=n)]
```

The C pragma has the following form:

```
#pragma _CNX unroll_and_jam[(unroll_factor=n)]
```

Where the optional `unroll_factor=`*n* argument allows you to specify an unroll factor for the loop in question.

You can disable unroll and jam for individual loops using the `no_unroll_and_jam` directive and pragma. In Fortran, it has the following form:

```
C$DIR NO_UNROLL_AND_JAM
```

The C pragma has the following form:

```
#pragma _CNX no_unroll_and_jam
```

NOTE Because unroll and jam is only performed on nested loops, you must ensure that the directive or pragma is specified on a loop that, after any compiler-initiated interchanges, is not the innermost loop. You can determine which loop in a nest will be innermost by compiling the nest without any directives and examining the Optimization Report. (See Appendix E, “Optimization Report” for more information.)

As with all optimizations that replicate code, the number of new loops created when the compiler performs the unroll and jam optimization is limited by default to ensure reasonable compile times. To increase the replication limit (and possibly increase your compile time and code size), specify the `+Onosize` `+Onolimit` compiler options.

Parallelization

Using the `+Oparallel` option at `+O3` and above allows the compiler to automatically parallelize loops that are profitable to parallelize. Also, the compiler recognizes the parallelism-related directives and pragmas of the Exemplar programming model. In Fortran 77 Version 1.2.3 and C Version 1.2.3, `+Oexemplar_model` (the default) must also be in effect for these directives and pragmas to be enabled. (The HP Exemplar aC++ compiler does not support the Exemplar programming model pragmas.)

Parallelization divides a program into threads. A *thread* is a single flow of control within a process. It can be a unique flow of control that performs a specific function, or one of several instances of a flow of control, each of which is operating on a unique data set.

The Exemplar compilers find parallelism at the loop level and generate parallel code that will automatically run on as many processors as are available at runtime. Normally, these are all the processors of the system on which your program is running. You can specify a smaller number of processors via the:

- `loop_parallel(max_threads=`*m*`)` directive and pragma—available in Fortran 90, Fortran 77, and C
- `prefer_parallel(max_threads=`*m*`)`directive and pragma—available in Fortran 90, Fortran 77, and C

  For more information on the `loop_parallel` and `prefer_parallel` directives and pragmas see the chapters “Basic shared-memory programming” and “Advanced shared-memory programming.”
- `MP_NUMBER_OF_THREADS` environment variable

  `MP_NUMBER_OF_THREADS` is read at runtime by your program. If this variable is set to some positive integer *n*, your program executes on *n* processors; *n* must be less than or equal to the number of processors in the system where the program is executing.

Automatic parallelization is useful for programs containing loops. You can use compiler directives or pragmas to improve on the automatic optimizations and to assist the compiler in locating additional opportunities for parallelization.

If you are writing your program entirely under the message-passing paradigm, you must explicitly handle parallelism yourself as discussed in the manual *HP MPI User's Guide*.

Basic operation

Parallelism can exist at the loop level, region level, and task level. Exemplar compilers automatically exploit loop-level parallelism. You can easily specify task-level parallelism using the `begin_tasks`, `next_task` and `end_tasks` directives and pragmas, as discussed in the section “Task parallelization” on page 138. You can also specify parallel regions using the `parallel` and `end_parallel` directives and pragmas, as discussed in the section “Region parallelization” on page 144. These directives and pragmas allow the compiler to run identified sections of code in parallel.

Loop-level parallelism involves dividing a loop into several smaller iteration spaces and parceling these out to run simultaneously on the available processors.

NOTE Only loops with iteration counts that can be determined prior to loop invocation at runtime are candidates for parallelization. Loops with iteration counts that depend on values or conditions calculated within the loop cannot be parallelized by any means.

Consider the following Fortran code:

```
PROGRAM PARAXPL
.
.
.
DO I = 1, 1024
  A(I) = B(I) + C(I)
  .
  .
  .
ENDDO
```

Assuming the `I` loop does not contain any parallelization-inhibiting code, this program can be parallelized to run on 8 processors by running 128 iterations per processor (1024 iterations divided by 8 processors = 128 iterations each). One processor would run the loop for `I` = 1 to 128; the next would run `I` = 129 to 256, and so on. The loop could similarly be parallelized to run on any number of processors, with each one taking its appropriate share of iterations. At a certain point, however, adding more processors will not improve performance. The compiler generates code that will run on as many processors as are available, but the dynamic selection optimization (described in the section “Dynamic selection” on page 109) ensures that parallel code is generated only if it is profitable to do so. If the number of available processors does not evenly divide the number of iterations, some processors will perform fewer iterations than others.

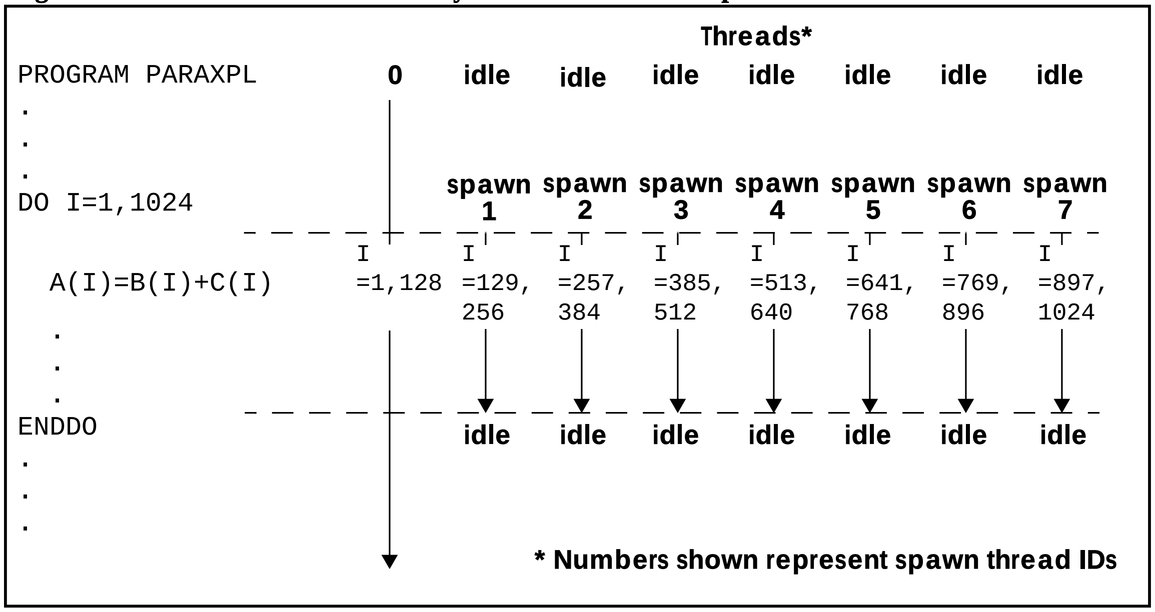
On an HP SMP server, shared-memory programs run as a collection of threads on multiple processors. When a program starts, a separate execution thread is created on each of the processors of the system on which the program is running. Using the CPSlib model of thread management, HP-UX identifies each of these threads by a unique *kernel* thread ID. (See Appendix F, “Compiler Parallel Support Library” for more information on CPSlib.) All but one of these threads is then idle; the nonidle thread is known as thread 0, and this thread runs all of the serial code in the program.

Spawn thread IDs are assigned only to nonidle threads when they are spawned—that is, when thread 0 encounters parallelism and “wakes up” other idle threads to execute the parallel code. Spawn thread IDs are consecutive. They range from 0 to *nt*-1, where *nt* is the number of threads spawned as a result of the spawn operation; this operation defines the current *spawn context*. The spawn context is the loop, task list, or region that initiates the spawning of the threads. Spawn thread IDs are valid only within a given spawn context.

Therefore, the idle threads are not assigned spawn thread IDs at the time of their creation. When thread 0 encounters a parallel loop, task, or region, it spawns the other threads, signaling them to begin execution. The threads then become active, acquire spawn thread IDs, run until their portion of the parallel code is finished, and go idle once again, as shown in Figure 16.

Machine loading does not affect the number of threads spawned, but it may affect the order in which the threads in a given spawn context complete. Again using the CPSlib thread model, HP-UX places an implicit barrier at the end of each parallelized loop, task, and region to ensure that, regardless of the machine load, all child threads of a given process execute to the completion of their spawn context before that process’s thread 0 continues.

Figure 16 Thread activity: one-dimensional parallelism


Threads*
PROGRAM PARAXPL
.
.
.
DO I=1,1024
A(I)=B(I)+C(I)
.
.
.
ENDDO
.
.
.
0
idle idle idle idle idle idle idle
spawn 1 spawn 2 spawn 3 spawn 4 spawn 5 spawn 6 spawn 7
I =1,128
I =129, 256
I =257, 384
I =385, 512
I =513, 640
I =641, 768
I =769, 896
I =897, 1024
idle idle idle idle idle idle idle
* Numbers shown represent spawn thread IDs

This example is greatly simplified for illustrative purposes. Various loop transformations can affect the manner in which a loop is parallelized.

To implement this, the compiler transforms the loop in a manner similar to strip mining. However, unlike the strip mining described in the section “Strip mining” on page 80, the outer loop is conceptual. Because the strips will be executing on different processors, there is no processor to run an outer loop like the one created in traditional strip mining.

Instead, the loop is transformed such that the starting and stopping iteration values are variables that are determined at runtime based on how many threads are available and which thread is running the strip in question.

Consider our previous Fortran example written for an unspecified number of iterations:

```
DO I = 1, N
  A(I) = B(I) + C(I)
ENDDO
```

The code shown in Figure 17 is a conceptual representation of the transformation the compiler performs on this example when it is compiled for parallelization, assuming that `N >= NumThreads`.

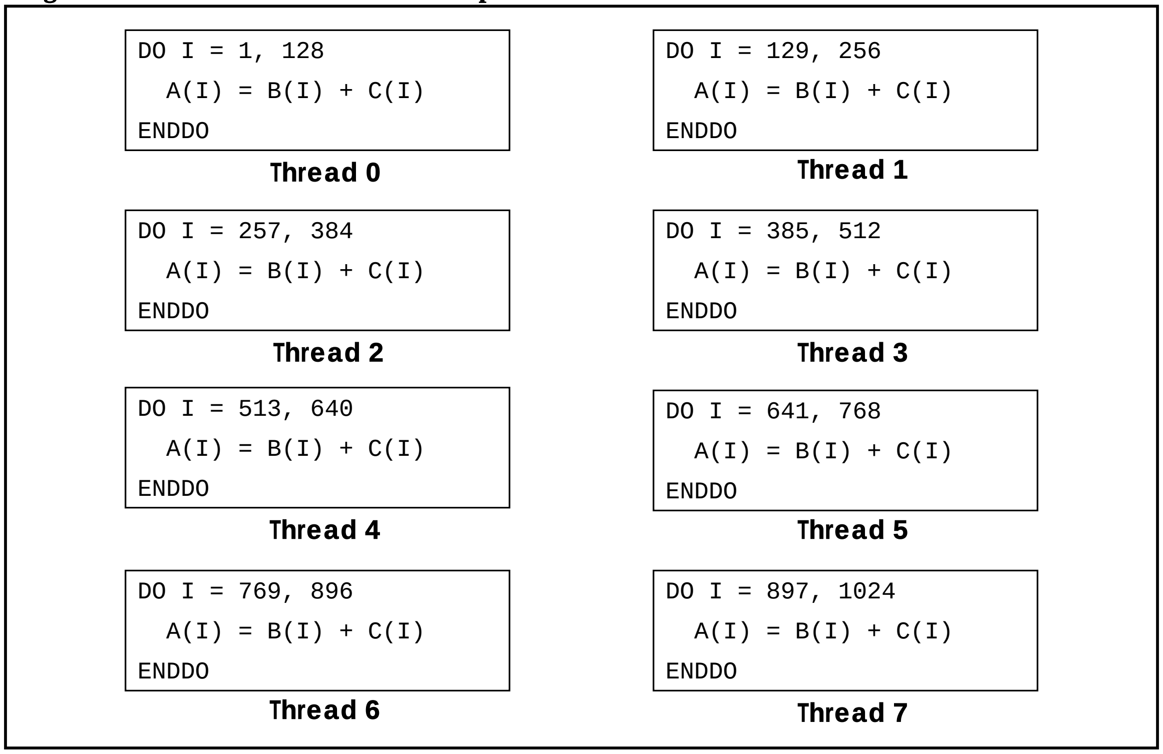
For `N < NumThreads`, the compiler uses `N` threads, assuming there is enough work in the loop to justify the overhead of parallelizing it. If `NumThreads` is not an integral divisor of `N`, some threads will do fewer iterations than others.

Figure 17 Conceptual strip mine for parallelization

```
For each available thread do:
  DO I = ThrdID*(N/NumThreads)+1,ThrdID*(N/NumThreads)+N/NumThreads
    A(I) = B(I) + C(I)
  ENDDO
```

`NumThreads` is the number of available threads. `ThrdID` is the ID number of the thread this particular loop will run on, which is between 0 and `NumThreads-1`. A unique `ThrdID` is assigned to each thread, and the `ThrdIDs` are consecutive. So, for `NumThreads = 8`, as in Figure 16, 8 loops would be spawned, with `ThrdIDs` = 0 through 7. These 8 loops are illustrated in Figure 18.

Figure 18 Parallelized loop



```
DO I = 1, 128
  A(I) = B(I) + C(I)
ENDDO
```

```
DO I = 129, 256
  A(I) = B(I) + C(I)
ENDDO
```

```
DO I = 257, 384
  A(I) = B(I) + C(I)
ENDDO
```

```
DO I = 385, 512
  A(I) = B(I) + C(I)
ENDDO
```

```
DO I = 513, 640
  A(I) = B(I) + C(I)
ENDDO
```

```
DO I = 641, 768
  A(I) = B(I) + C(I)
ENDDO
```

```
DO I = 769, 896
  A(I) = B(I) + C(I)
ENDDO
```

```
DO I = 897, 1024
  A(I) = B(I) + C(I)
ENDDO
```

The strip-based parallelism described here is the default. Stride-based parallelism is possible through use of the `prefer_parallel` and `loop_parallel` compiler directives and pragmas, which are described in Chapter 4, “Basic shared-memory programming.”

In these examples, the data being manipulated within the loop is disjoint; that is, no two threads attempt to write the same data item. If two parallel threads attempt to update the same storage location, their actions must be synchronized. This is discussed further in Chapter 4 “Basic shared-memory programming.”

Idle thread states

Idle threads can be suspended or spin-waiting. Suspended threads release control of the processor; spin-waiting threads repeatedly check an encached global semaphore that indicates whether or not they have code to execute. Obviously, this prevents any other process from gaining control of the CPU and can severely degrade multiprocess performance. On the other hand, waking a suspended thread takes substantially longer than activating a spin-waiting thread. Therefore, by default, idle threads spin-wait briefly after creation or a join, then suspend themselves if no work is received. This spin-wait time can be changed by using the `cps_wait_attr()` function, which is described in the "Compiler Parallel Support Library" appendix.

When threads are suspended, HP-UX may schedule threads of another process on their processors in order to balance the machine load. However, threads have an *affinity* for their original processors; HP-UX attempts to schedule unsuspended threads to their original processors in order to exploit the presence of any data encached during the thread's last timeslice. This affinity is realized only if the original processor is available; otherwise, the thread is assigned to the first processor to become available within the hypernode on which it was spawned.

Node-parallelism vs. thread-parallelism

Exemplar compilers support two dimensions of parallelism: thread-parallelism and, on multihypernode systems, node-parallelism. The compilers (excluding the HP aC++ compiler) support node-parallelism through the specification of the `nodes` attribute in the `loop_parallel`, `prefer_parallel`, `parallel`, and `begin_tasks` directives and pragmas. Unlike thread-parallelism, node-parallelism is not performed automatically; you must use directives or pragmas. By supporting two dimensions of parallelism, the compilers allow you to exploit parallelism within parallelism when it occurs in your program.

If two-dimensional parallelism is not present in a particular loop nest, or if exploiting it would be inefficient, you can also run single-dimensional (thread) parallelism across two or more hypernodes of the system. In such cases, you can use the `prefer_parallel`, `loop_parallel`, or `parallel` compiler directives or pragmas with the `threads` attribute to parallelize across all available threads on multiple hypernodes, or to limit parallelization to a subset of available threads or hypernodes. Refer to Chapter 4, “Basic shared-memory programming,” for more information on these and other parallelization directives and pragmas.

Parallel threads are started at program startup, as described in the section “Basic operation” on page 101, without regard to the presence or absence of two-dimensional parallelism. The level of parallelism affects only the physical location of the threads that are activated when the parallel construct is encountered; it does not affect the total number of threads available to the program.

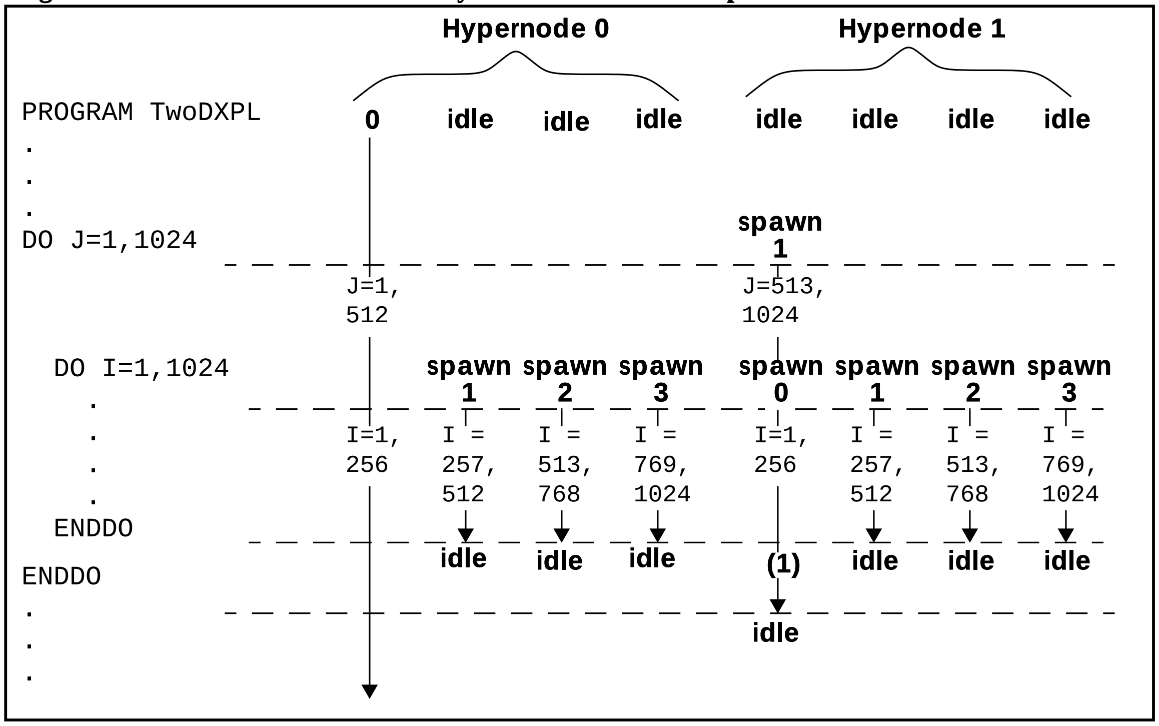
When node-parallelism is encountered, assuming the program is running on a multihypernode system, a single thread on each hypernode is activated. These threads are given spawn thread IDs ranging from 0 (the thread that encountered the node-parallelism) to one less than the number of hypernodes. If thread-parallelism is then encountered within the node-parallelism, a new spawn context is established, and the available threads on each hypernode (assuming the parallel construct does not limit the number of spawned threads) are activated. In this new spawn context, the threads on each hypernode are given spawn thread IDs ranging from 0 to one less than the number of threads on the hypernode. This means that spawn thread IDs are duplicated on each hypernode in the context of thread-parallel code within node-parallel code. On a given hypernode, however, they are unique.

Consider the following Fortran example:

```
      PROGRAM TwoDXPL
      .
      .
      .
C$DIR PREFER_PARALLEL(NODES)
      DO J = 1, 1024
C$DIR   PREFER_PARALLEL(THREADS)
        DO I = 1, 1024
          A(I,J) = B(I,J) + C(I,J)
          .
          .
          .
        ENDDO
      ENDDO
```

Here, assuming the loop nest does not contain any parallelization inhibitors, the compiler parallelizes the J loop across hypernodes and the I loop across threads within those hypernodes. Assuming this program is running on an 8-processor system consisting of four processors from two hypernodes, the parallelization is illustrated in Figure 19.

Figure 19 Thread activity: two-dimensional parallelism



This example is greatly simplified for illustrative purposes. Various loop transformations can affect the manner in which a loop is parallelized.

As shown in Figure 19, the node-parallel J loop spawns two threads, one on each hypernode. Thread ID 0 was already running the serial code in the program; ID 1 is activated when the node-parallel loop begins execution.

The I loop then spawns thread-parallelism on each hypernode. The original thread 0 maintains its spawn thread ID, and IDs 1-3 are also spawned on hypernode 0. The thread that is spawn thread 1 in the context of the J loop becomes spawn thread 0 in the context of the I loop, and IDs 1-3 are also spawned on hypernode 1 in the context of the I loop.

Note that when the `I` loop terminates, the spawn context returns to that of the `J` loop, and thread 0 on hypernode 1 becomes thread 1 again, as it had been before the `I` loop began.

Node-parallelism is disabled by default. This prevents the compiler from exploiting node-parallelism, but allows the exploitation of both automatic and directive-specified thread-parallelism. Node-parallelism can be enabled by specifying the `+Onodepar` command-line option.

Parallel optimizations

Simple loops can be parallelized without the need for extensive transformations, as shown in the section “Basic operation” on page 101. However, most loop transformations, if they are applicable to the loop in question, can aid parallelization in some way. For instance, loop interchange orders loops so that the innermost loop best exploits the processor data cache, and the outermost loop is the most efficient loop to parallelize. Loop blocking similarly aids parallelization by maximizing cache data reuse on each of the processors that the loop runs on, and by ensuring that each processor is working on nonoverlapping array data.

Dynamic selection

The compiler has no way of determining how many processors will be available to run compiled code; therefore, it must generate both serial and parallel code for loops that can be parallelized. Replicating the loop in this manner is called *cloning*, and the resulting versions of the loop are called *clones*. Cloning is also performed when the loop-iteration count is unknown at compile-time.

It is not always profitable, however, to run the parallel clone when multiple processors are available. Some overhead is involved in executing parallel code. This overhead includes the time it takes to spawn parallel threads, to privatize any variables used in the loop that must be privatized, and to join the parallel threads when they complete their work.

Exemplar compilers use a powerful form of dynamic selection known as *workload-based dynamic selection*. When a loop’s iteration count is available at compile time, workload-based dynamic selection determines the profitability of parallelizing the loop and only writes a parallel version to the executable if it is profitable to do so. Omitting the parallel version from the executable when it will never be needed enhances performance further by eliminating the runtime decision as to which version to use.

The power of dynamic selection becomes more apparent when the loop's iteration count is unknown at compile time. In this case, the compiler generates code that, at runtime, compares the amount of work performed in the loop nest (given the actual iteration counts) to the parallelization overhead for the available number of processors and runs the parallel version of the loop only if it is profitable to do so.

Workload-based dynamic selection is enabled by default at optimization level `+O3`, when `+Oparallel` is specified. The `+Onodynsel` compiler option can be used to disable dynamic selection. When dynamic selection is disabled, the compiler assumes that it is profitable to parallelize all parallelizable loops, and generates both serial and parallel clones for them. In this case the parallel version is run if there are multiple processors at runtime, regardless of the profitability of doing so.

You can enable workload-based dynamic selection for selected loops by using the `dynsel` compiler directive or pragma. In Fortran, this directive has the following form:

```
C$DIR DYNSEL[(THREAD_TRIP_COUNT = n | NODE_TRIP_COUNT = m)]
```

The C pragma has the form:

```
#pragma _CNX dynsel[(thread_trip_count = n | node_trip_count = m)]
```

where

`thread_trip_count` and `node_trip_count`
: are optional attributes used to specify threshold iteration counts.

When `thread_trip_count = ` *n* is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the thread-parallel version is run. When `node_trip_count = ` *m* is specified, the serial version of the loop is run if the iteration count is less than *m*; otherwise, the node-parallel version is run, assuming `+Onodepar` is specified. *n* and *m* must be compile-time constants.

If a trip count is not specified for a `dynsel` directive or pragma, the compiler uses a heuristic to estimate the actual execution costs. This estimate is then used to determine if it is profitable to execute the loop in parallel.

The `thread_trip_count` attribute cannot be used on loops that also specify the `loop_parallel(nodes)` directive or pragma. Similarly, the `node_trip_count` attribute cannot be used on loops that also specify the `loop_parallel(threads)` directive or pragma. These combinations are contradictory.

These directives can be used to specify dynamic selection for specific loops in programs compiled using the `+Onodynsel` option, or to provide trip count information for specific loops in programs compiled with dynamic selection enabled. You can disable dynamic selection for selected loops by using the `no_dynsel` compiler directive or pragma. In Fortran, this directive has the following form:

```
C$DIR NO_DYNSEL
```

The C pragma has the form:

```
#pragma _CNX no_dynsel
```

This directive or pragma can be used to disable dynamic selection on specific loops in programs compiled with dynamic selection enabled.

As with all optimizations that replicate loops, the number of new loops created when the compiler performs dynamic selection is limited by default to ensure reasonable compile times. To increase the replication limit (and possibly increase your compile time and code size), specify the `+Onosize` `+Onolimit` compiler options.

Inhibitors of parallelization

Most constructs that inhibit data localization also inhibit parallelization. Specifically, these are:

- Loop-carried dependences
- Aliased scalar or array variables
- Multiple loop entries or exits
- Procedure calls
- I/O statements

Most of these items inhibit parallelization for the same reasons they inhibit localization. An exception to this is that more categories of loop-carried dependences can inhibit parallelization than data localization, as described in the following sections.

Loop-carried dependences

The specific loop-carried dependences (LCDs) that inhibit data localization represent a very small portion of all loop-carried dependences. A much broader set of LCDs, including those that inhibit data localization, can inhibit parallelization.

LCDs fall into three categories:

- *Forward LCDs*
- *Backward LCDs*
- *Output LCDs*

The LCD that inhibits localization is a combination of these. Each of these LCDs inhibit parallelization.

Forward LCDs

A forward LCD exists when one iteration references a variable whose value is assigned on a later iteration. The Fortran loop below contains a forward LCD on the array `A`.

```
DO I = 1, N - 1
  A(I) = A(I + 1) + B(I)
ENDDO
```

In this example, the first iteration assigns a value to `A(1)` and references `A(2)`. The second iteration assigns a value to `A(2)` and references `A(3)`. The reference to `A(I)` depends on the fact that the `I+1`th iteration, which assigns a new value to `A(I)`, has not yet executed. Forward LCDs inhibit parallelization because if the loop is broken up to run on several processors, when `I` reaches its terminal value on one processor, `A(I+1)` will usually have already been computed by another processor (it is, in fact, the first value computed by another processor). Because the calculation depends on `A(I+1)` being uncomputed, this would produce wrong answers.

An analogous C/C++ loop follows:

```
for(i=0;i<n-1;i++)
  a[i] = a[i+1] + b[i];
```

Backward LCDs

A backward LCD exists when one iteration references a variable whose value was assigned on an earlier iteration. The Fortran loop below contains a backward LCD on the array A.

```
DO I = 2, N
  A(I) = A(I-1) + B(I)
ENDDO
```

Here, each iteration assigns a value to A based on the value assigned to A in the previous iteration. If `A(I-1)` has not been computed before `A(I)` is assigned, wrong answers will result. Backward LCDs inhibit parallelism because if the loop is broken up to run on several processors, `A(I-1)` will not have been computed for the first iteration of the loop on every processor except the processor running the chunk of the loop containing `I = 1`.

An analogous C/C++ loop follows:

```
for(i=1;i<n;i++)
  a[i] = a[i-1] + b[i];
```

Output LCDs

An output LCD exists when the same memory location is assigned values on two or more iterations. A potential output LCD exists when the compiler cannot determine whether an array subscript contains the same values between loop iterations. The Fortran loop below contains a potential output LCD on the array A:

```
DO I = 1, N
  A(J(I)) = B(I)
ENDDO
```

Here, if any referenced elements of J contain the same value, the same element of A will be assigned several different elements of B. In this case, as this loop is written, any A elements that are assigned more than once should contain the final assignment at the end of the loop. If the loop is run in parallel, however, this cannot be guaranteed.

An analogous C/C++ loop follows:

```
for(i=0;i<n;i++)
  a[j[i]] = b[i];
```

Apparent LCDs

The compiler will not parallelize loops containing apparent LCDs rather than risk wrong answers by doing so.

If you are sure that a loop with an apparent LCD is safe to parallelize, you can indicate this to the compiler using the `no_loop_dependence` directive or pragma, which is explained in the section “Loop-carried dependences” on page 68.

The following Fortran example illustrates a `NO_LOOP_DEPENDENCE` directive being used on the output LCD example presented previously:

```
C$DIR NO_LOOP_DEPENDENCE(A)
      DO I = 1, N
        A(J(I)) = B(I)
      ENDDO
```

This effectively tells the compiler that no two elements of `J` are identical, so there is no output LCD and the loop is safe to parallelize. If any of the `J` values are identical, wrong answers could result.

Use of the `no_loop_dependence` pragma is illustrated in the following C example:

```
#pragma _CNX no_loop_dependence(a)
for(i=0;i<n;i++)
  a[j[i]] = b[i];
```

Reductions

In many cases, the compiler can recognize and parallelize loops containing a special class of dependence known as a reduction. In general, a reduction has the form:

`X = X` *operator* `Y`

where `X` is a variable not assigned or used elsewhere in the loop, `Y` is a loop constant expression not involving `X`, and *operator* is `+`, `*`, `.AND.`, `.OR.`, or `.XOR.`.

The compiler also recognizes reductions of the form:

`X =` *function*`(X,Y)`

where `X` is a variable not assigned or referenced elsewhere in the loop, `Y` is a loop constant expression not involving `X`, and *function* is the intrinsic `MAX` function or intrinsic `MIN` function.

Reductions commonly appear in the form of sum operations, as shown in the following Fortran example:

```
DO I = 1, N
  A(I) = B(I) + C(I)
  .
  .
  .
  ASUM = ASUM + A(I)
ENDDO
```

Assuming this loop does not contain any parallelization-inhibiting code, the compiler would automatically parallelize it. The code generated to accomplish this creates temporary, thread-specific copies of `ASUM` for each thread that will be running the loop. When each parallel thread completes its portion of the loop, thread 0 for the current spawn context accumulates the thread-specific values into the global `ASUM`.

Generally, the compiler automatically recognizes reductions in a loop and is able to parallelize the loop. If the loop is under the influence of the `prefer_parallel` directive or pragma, the compiler still recognizes reductions. However, in a loop under the influence of the `loop_parallel` directive or pragma, reduction analysis is not performed. Consequently, the loop may not be correctly parallelized unless the reduction is pointed out by using the `reduction` directive or pragma.

The Fortran 90 directive has the form:

`C$DIR REDUCTION(`*namelist*`)`

The C pragma has the form:

`#pragma _CNX reduction(`*namelist*`)`

where

namelist	is a comma-separated list of scalar variables

The following example shows the use of this directive:

```
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(FUNCTEMP), REDUCTION(SUM)
      DO I = 1, N
        .
        .
        .
        FUNCTEMP = FUNC(X(I))
        SUM = SUM + FUNCTEMP
        .
        .
        .
      ENDDO
```

Preventing parallelization

You can prevent parallelization on a loop basis by using the `no_parallel` directive or pragma. The Fortran directive has the form:

```
C$DIR NO_PARALLEL
```

The C pragma has the form:

```
#pragma _CNX no_parallel
```

You can use these directives to prevent parallelization of the loop that immediately follows them. Only parallelization is inhibited; all other loop optimizations will still be applied.

The following Fortran example illustrates the use of this directive:

```
      DO I = 1, 1000
C$DIR   NO_PARALLEL
        DO J = 1, 1000
          A(I,J) = B(I,J)
        ENDDO
      ENDDO
```

In this example, parallelization of the `J` loop is prevented. The `I` loop can still be parallelized.

An analogous C example follows:

```
for(i=0;i<1000;i++)
#pragma _CNX no_parallel
  for(j=0;j<1000;j++)
    a[i][j] = b[i][j];
```

The `+Onoautopar` compiler option is available to disable automatic parallelization but will allow parallelization of directive-specified loops. Refer to Chapter 4, “Basic shared-memory programming,” and Appendix D, “Optimization options,” for more information on `+Onoautopar`.

The `+Ononodepar` compiler option is available to disable directive-specified node-parallelism. Thread-parallelism—both automatic and directive-specified—is still implemented. Refer to Chapter 4, “Basic shared-memory programming,” and Appendix D, “Optimization options,” for more information on `+Ononodepar`.

Other parallelization directives and pragmas

Several directives and pragmas are available to allow you to manually control certain aspects of loop parallelization and to parallelize tasks outside of loops. These directives (which are not supported by the HP aC++ compiler) are:

`prefer_parallel`

Requests parallelization of the immediately following loop; accepts attributes for node- and thread-parallelism, strip-length adjustment, maximum number of threads, and ordered execution. The compiler handles data privatization and does not parallelize the loop if it is not safe to do so.

`loop_parallel`

Forces parallelization of the immediately following loop. Accepts the same attributes as `prefer_parallel`, but requires you to manually privatize loop data and synchronize data dependences.

`begin_tasks`, `next_task` and `end_tasks`

Allow you to parallelize consecutive blocks of serial code. Accepts attributes for node- and thread-parallelism, ordered execution, and maximum number of threads.

`parallel` and `end_parallel`

Allow you to parallelize a single code region to run on multiple threads. Unlike the tasking directives, which run discrete sections of code in parallel, `parallel`/`end_parallel` run multiple copies of a single section. Accepts attributes for node- and thread-parallelism, and maximum number of threads.

Within a parallel region, loop directives (`prefer_parallel`, `loop_parallel`) and tasking directives (`begin_tasks`) may appear with the `dist` attribute. The `dist` attribute causes the compiler to use existing threads rather than spawning new threads.

`critical_section` and `end_critical_section`

Allow you to isolate nonordered manipulations of a shared variable within the loop. Only one parallel thread can execute the code contained in the critical section at a time, eliminating possible contention.

`ordered_section` and `end_ordered_section`

Allow you to isolate dependences within a loop so that code contained within the ordered section executes in iteration order. Only useful when used with the `loop_parallel(ordered)` directive or pragma.

These directives and pragmas are discussed in detail in Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming.”

Parallelism in HP aC++

Parallelism in the HP aC++ compiler is available through the following command-line options or libraries:

- `+O3 +Oparallel` or `+O4 +Oparallel`—Some automatic parallelization is available from the compiler; see the section "Parallelization" on page 100 for more information
- HP MPI—HP's implementation of the message-passing interface; see Chapter 7, "Message-passing programming" and the manual *HP MPI User's Guide* for more information
- CPSlib—the Compiler Parallel Support Library provides thread management and synchronization routines; see the cps(3) man page or Appendix F, "Compiler Parallel Support Library," for additional information
- Pthreads—POSIX threads; see the pthread(3t) man page or the manual *Programming with Threads on HP-UX* (B2355-90060) for more information

Almost none of the pragmas in this book, regardless of whether they are related to parallelism, are available in the HP aC++ compiler. However, C++ does support the memory classes that are fully described in Chapter 5, "Memory classes" and briefly explained in Appendix B, "Exemplar compiler directives and pragmas." These classes are implemented through storage class specifiers (`far_shared`, `near_shared`, `node_private`, and `thread_private`).

+O4 level optimization

At +O4, optimizations are performed across all files in the application that have been compiled with +O4. All optimizations of the previous levels are performed, and three additional optimizations are performed:

- Inlining across multiple source files
- Cloning across multiple source files
- Global and static variable optimizations

During +O4 optimizations, the compiler optimizes across the function boundaries (of all files that have been compiled with +O4) to produce better and faster code sequences. Normally, global optimizations are performed within individual functions or source code files. Interprocedural optimizations look at function interactions within a program and transform particular code sequences into faster code. Because information about every function within a program is required, this level of optimization must be performed at link time. Because analysis is done at link time, the compile time is generally shorter (than at lower optimization levels), but linking takes more time.

Inlining across multiple source files

Inlining substitutes function calls with copies of the function's object code. Only functions that meet the optimizer's criteria are inlined. This may result in slightly larger executable files. However, this increase in size is offset by the elimination of time-consuming procedure calls and procedure returns. See the section "Inlining within a single source file" on page 64 for an example of inlining.

Inlining at +O4 is performed across all procedures within the program. Inlining at +O3 is done within one file.

Inlining is affected by the `+O[no]inline[=`*namelist*`]` and `+Oinline_budget=`*n* command-line options. See Appendix D, "Optimization options," for more information.

Cloning across multiple source files

Cloning is the replacement of a call to a routine by a call to a clone of that routine. The clone is optimized differently than the original routine. Cloning can expose additional opportunities for interprocedural optimization.

Cloning at +O4 is performed across all procedures within the program. Cloning at +O3 is done within one file. Cloning is enabled by default; it can be disabled by specifying the +Onoinline command-line option.

Global and static variable optimizations

Global and static variable optimizations look for ways to reduce the number of instructions required for accessing global and static variables (COMMON and SAVE variables in Fortran; extern and static variables in C and C++). The compiler normally generates two machine instructions when referencing global variables. Depending on the locality of the global variables, single machine instructions may sometimes be used to access these variables. The linker rearranges the storage location of global and static data to increase the number of variables that can be referenced by single instructions.

Global variable optimization coding standards

Because this optimization rearranges the location and data alignment of global variables, follow the programming practices given below:

- Do not make assumptions about the relative storage location of variables, such as generating a pointer by adding an offset to the address of another variable.
- Do not rely on pointer or address comparisons between two different variables.
- Do not make assumptions about the alignment of variables, such as assuming that a short integer is aligned the same as an integer.

4 Basic shared-memory programming

This chapter discusses programming techniques that allow you to increase code efficiency with minimal effort. The pragmas discussed in this chapter are not available in the HP aC++ compiler.

Simple manual loop, task, and region parallelization

The Exemplar compilers automatically exploit strip-based loop parallelism in loops that are clearly dependence-free, as described in Chapter 3, “Compiler optimizations.” The `prefer_parallel`, `loop_parallel`, and `parallel` directives and pragmas allow you to increase parallelization opportunities and to manually control many aspects of parallelization.

The compiler cannot automatically locate task parallelism, but the tasking directives and pragmas mentioned in Chapter 3, “Compiler optimizations,” (and discussed here) allow you to specify consecutive blocks of code that can be run in parallel. Similarly, the `parallel` and `end_parallel` directives and pragmas allow you to specify a code region that can be run in its entirety on several processors.

The subsections that follow discuss specifying simple, unordered loop, task, and region parallelism using the `prefer_parallel`, `loop_parallel`, `begin_tasks`/`next_task`/`end_tasks`, and `parallel`/`end_parallel` directives and pragmas. These directives and pragmas can be nested in any order as long as node-parallelism is outside thread-parallelism.

Critical sections that do not rely on ordered execution are also covered here. Any necessary variable privatization is provided by the `loop_private`, `task_private` and `parallel_private` directives and pragmas, which are described in detail in the “Loop-specific, task-specific, and region-specific data privatization” section of this chapter.

For a detailed discussion of ordered parallelism, parallel synchronization, and the effective use of memory classes, refer to Chapter 5, “Memory classes,” and Chapter 6, “Advanced shared-memory programming.”

Loop parallelization

This section discusses simple uses of the `prefer_parallel` and `loop_parallel` directives and pragmas, which, when specified, apply to the immediately following loop. The data privatization necessary when using `loop_parallel` is illustrated in this chapter’s examples using the `loop_private` directive, which is discussed in the section “`loop_private`” on page 154. Manual data privatization using memory classes is discussed in Chapter 5, “Memory classes,” and Chapter 6, “Advanced shared-memory programming.”

NOTE Use these directives and pragmas only on Fortran `DO` and C `for` loops that have iteration counts that can be determined prior to loop invocation at runtime.

`prefer_parallel` and `loop_parallel` generally take the same attributes. The `threads` attribute is the default attribute for both `prefer_parallel` and `loop_parallel`. In Fortran, these directives have the following form:

`C$DIR PREFER_PARALLEL[(`*attribute-list*`)]`

and

`C$DIR LOOP_PARALLEL[(`*attribute-list*`)]`

In C, they have the form:

`#pragma _CNX prefer_parallel[(`*attribute-list*`)]`

and

`#pragma _CNX loop_parallel(ivar = `*indvar*`[, `*attribute-list*`])`

where

`ivar = `*indvar*

specifies that the primary loop induction variable is *indvar*. `ivar = `*indvar* is optional in Fortran, but required in C. Use it only with `loop_parallel`.

the optional *attribute-list*

can contain one of the following case-insensitive attributes.

NOTE The values of *n* and *m* must be compile-time constants for all of the below attributes in which they appear.

`threads`

Causes thread-parallelism. This is the default.

`nodes`

Causes thread-based node-parallelism. See the section “`nodes` attribute” on page 128 for additional information.

`dist`

Causes the compiler to distribute the iterations of a loop across active threads instead of spawning new threads. Use `dist` with `prefer_parallel` or `loop_parallel` inside a `parallel`/`end_parallel` region. The level of parallelism is determined by using either the `threads` or `nodes` attribute in the `parallel` directive or pragma. See “Region parallelization” on page 144 for more information.

`ordered`

Causes ordered invocation of each loop iteration; provides no automatic synchronization. Designed for use with the `loop_parallel` directive and pragma on loops containing ordered sections.

`max_threads =` *m*

Allows no more than *m* threads to be allocated to the execution of the loop. *m* must be an integer constant.

`chunk_size =` *n*

Divides the loop into chunks of *n* or fewer iterations, and distributes the chunks round-robin to the processors as shown in Table 5 and Table 6 on page 131. *n* must be an integer constant.

`threads, ordered`

Causes ordered invocation of each iteration across threads.

`nodes, ordered`
: Causes ordered invocation of each iteration across hypernodes.

`dist, ordered`
: Causes ordered invocation of each iteration across threads or nodes, as specified in the attribute list to the `parallel` directive.

`threads, max_threads =` *m*
: Causes thread-parallelism on no more than *m* threads.

`nodes, max_threads =` *m*
: Causes node-parallelism on no more than *m* nodes; this starts one thread per node on no more than *m* hypernodes.

`dist, max_threads =` *m*
: Causes thread-parallelism or node-parallelism (as determined by the attribute list to the `parallel` directive) on no more than *m* threads (if thread-parallelism) or nodes (if node-parallelism).

`ordered, max_threads =` *m*
: Causes ordered parallelism on no more than *m* threads.

`threads, chunk_size =` *n*
: Causes thread-parallelism by chunks.

`nodes, chunk_size =` *n*
: Causes node-parallelism by chunks.

`dist, chunk_size =` *n*
: Causes thread-parallelism or node-parallelism (as determined by the attribute list to the `parallel` directive) by chunks.

`threads, ordered, max_threads =` *m*
: Causes ordered thread-parallelism on no more than *m* threads.

`nodes, ordered, max_threads =` *m*
: Causes ordered node-parallelism on no more than *m* hypernodes.

`dist, ordered, max_threads =` *m*

Causes ordered thread-parallelism on no more than *m* threads, or ordered node-parallelism on no more than *m* hypernodes—depending on the attribute list used with the `parallel` directive.

`chunk_size =` *n*`, max_threads =` *m*

Causes chunk parallelism on no more than *m* threads.

`threads, chunk_size =` *n*`, max_threads =` *m*

Causes thread-parallelism by chunks of size *n* on no more than *m* threads.

`nodes, chunk_size =` *n*`, max_threads =` *m*

Causes node-parallelism by chunks of size *n* on no more than *m* hypernodes.

`dist, chunk_size =` *n*`, max_threads =` *m*

Causes thread-parallelism by chunks on no more than *m* threads, or node-parallelism by chunks on no more than *m* hypernodes—depending on the attribute list used with the `parallel` directive.

Combining the attributes

The allowed combinations of attributes are those combinations listed in the preceding section. In such combinations the attributes can be listed in any order.

The `loop_parallel` C pragma requires the `ivar =` *indvar* attribute, which specifies the primary loop induction variable. If this is not present, the compiler will issue a warning and ignore the pragma. `ivar` should specify only the primary induction variable; any other loop induction variables should be a function of this variable and should be declared `loop_private`.

In Fortran, `ivar` is optional for `DO` loops; if not provided, the compiler will pick the primary induction variable for the loop. `ivar` is required for `DO WHILE` and hand-rolled loops in Fortran.

`prefer_parallel` does not require `ivar`, and the compiler will issue an error if it encounters this combination.

Using the attributes

The attributes associated with the `prefer_parallel` and `loop_parallel` directives and pragmas are explained in the following sections.

`threads`

The optional `threads` attribute causes parallelization across threads; this is the default for `loop_parallel` and `prefer_parallel`. If the `threads` attribute appears in a parallelization directive on the outermost loop in a nest, the loop will go parallel on all the threads available to the process. If the `threads` attribute appears in a parallelization directive nested within a node-parallel construct, the specified loop will go thread-parallel on the threads available within each parallel hypernode.

`nodes`

The `nodes` attribute causes parallelization across hypernodes in a multinode, scalable SMP system. In this case, a single thread on each available hypernode will execute a portion of the specified loop. A node-parallel construct cannot exist inside a thread-parallel construct. See the section “Node-parallelism vs. thread-parallelism” on page 106 for a comparison of the two levels of parallelism.

`dist`

The `dist` attribute tells the compiler to distribute the iterations of a loop across the currently active threads—instead of spawning new threads. Using currently active threads significantly reduces the parallelization overhead. `loop_parallel(dist)` and `prefer_parallel(dist)` should be used inside a `parallel`/`end_parallel` region. The level of parallelism is determined by using either the `threads` or `nodes` attribute to the `parallel` directive (or pragma). See “Region parallelization” on page 144 for information on the attributes available to the `parallel` directive and pragma.

The `dist` attribute can be used with any `prefer_parallel` or `loop_parallel` attribute, except the `nodes` or `threads` attributes.

NOTE

Any loop under the influence of `loop_parallel(dist)` or of `prefer_parallel(dist)` will appear in the Optimization Report (generated by specifying the `+Oreport` option) as being serial, because it is already inside a parallel region. For more information on the Optimization Report, see Appendix E, “Optimization Report.”

In the following example, threads are spawned when the `parallel` directive is used. No additional threads are spawned until the `loop_parallel` directive is used without the `dist` attribute:

```
C$DIR PARALLEL (NODES, MAX_THREADS = 4), PARALLEL_PRIVATE(A, C)
C SPAWN ONE THREAD PER NODE, UP TO A MAXIMUM OF 4

      A = B  ! THIS STATEMENT WILL BE EXECUTED BY ALL 4 NODE-WAY THREADS

C$DIR LOOP_PARALLEL(DIST, MAX_THREADS = 3)
      DO I = 1, 10000
                     ! THIS LOOP WILL BE DISTRIBUTED TO AT MOST 3 OF
        X(I) = Y(I)  ! THE 4 ACTIVE NODE-WAY THREADS; THIS MEANS THAT
                     ! EACH NODE-WAY THREAD EXECUTES
                     ! ABOUT 10000/3 ITERATIONS
      ENDDO

      C = X(1)      ! THIS STATEMENT WILL BE EXECUTED BY ALL
                    ! NODE-WAY THREADS

C$DIR LOOP_PARALLEL(DIST)
      DO J = 1, 10000
                    ! THIS LOOP WILL BE DISTRIBUTED TO THE 4 ACTIVE
        Y(J) = X(J) ! NODE-WAY THREADS, MEANING THAT EACH THREAD
                    ! EXECUTES 10000/4 ITERATIONS
      ENDDO

C$DIR LOOP_PARALLEL
      DO K = 1, 10000
                    ! SPAWN ADDITIONAL THREADS ON EACH NODE, UP TO THE
        W(K, MY_NODE()) = X(K) ! MAXIMUM AVAILABLE (TYPICALLY 16) AND
                    ! ON EACH NODE, DISTRIBUTE THE WORK ACROSS
                    ! ALL THE THREADS SO THAT
                    ! EACH THREAD EXECUTES 10000/16 ITERATIONS
      ENDDO
C$DIR END_PARALLEL
```

`loop_parallel` and `loop_parallel(dist)` directives can be nested as long as node-parallel loops are outside all thread-parallel loops. The compiler will pick the loop that is most appropriate for the directive or pragma being processed (the loop picked is usually the outermost parallel loop).

`ordered`

The `ordered` attribute causes the iterations of the loop to be initiated in loop order across the processors. It is useful only in loops with manually-synchronized dependences, so it is only useful with the `loop_parallel` directive. To achieve ordered parallelism, dependences must be synchronized within ordered sections, such as those constructed using the `ordered_section` and `end_ordered_section` directives. Using `loop_parallel(ordered)` and its associated synchronization directives is covered in Chapter 6, “Advanced shared-memory programming.”

`max_threads =` *m*

The `max_threads =` *m* attribute restricts execution of the specified loop to no more than *m* threads if specified alone or with the `threads` attribute; if specified with the `nodes` attribute, execution is restricted to *m* `nodes` running one thread each. If specified with the `chunk_size =` *n* attribute, the chunks are parallelized across no more than *m* threads. `max_threads =` *m* is useful when you know the maximum number of threads your loop will run on efficiently.

`chunk_size =` *n*

The `chunk_size =` *n* attribute specifies a number of iterations by which to strip mine the loop for parallelization. If this attribute is present alone or with the `threads` attribute, *n* or fewer loop iterations are distributed round-robin (as shown in Table 5 and Table 6 on page 131) to each available thread until there are no remaining iterations. If `chunk_size =` *n* is combined with the `nodes` attribute, the chunks are distributed round-robin to each available hypernode until there are no remaining chunks. If the number of threads does not evenly divide the number of iterations, some threads will perform one less chunk than others. *n* must be a compile-time integer constant.

This stride-based parallelism differs from the default strip-based parallelism described in Chapter 3, “Compiler optimizations,” that divides the loop’s iterations into a number of contiguous chunks equal to the number of available threads, and each thread computes one chunk. The `chunk_size =` *n* attribute allows each thread to do several noncontiguous chunks.

Specifying `chunk_size =` ((*number of iterations* - 1) / *number of threads*) + 1 is similar to default strip mining for parallelization.

Using chunk_size = 1 distributes individual iterations cyclically across the processors. For example, if a loop has 1000 iterations to be distributed among 4 processors, specifying chunk_size = 1 causes the distribution shown in Table 5.

Table 5 Iteration distribution using chunk_size = 1

	CPU0	CPU1	CPU2	CPU3
Iterations	1	2	3	4
	5	6	7	. . .

For chunk_size = *n*, with *n* > 1, the distribution is round-robin, however it is not the same as specifying the ordered attribute. For example, using the same loop as above, specifying chunk_size = 5 produces the distribution shown in Table 6.

Table 6 Iteration distribution using chunk_size = 5

	CPU0	CPU1	CPU2	CPU3
Iterations	1, 2, 3, 4, 5	6, 7, 8, 9, 10	11, 12, 13, 14, 15	16, 17, 18, 19, 20
	21, 22, 23, 24, 25	26, 27, 28, 29, 30	31, 32, 33, 34, 35,	. . .

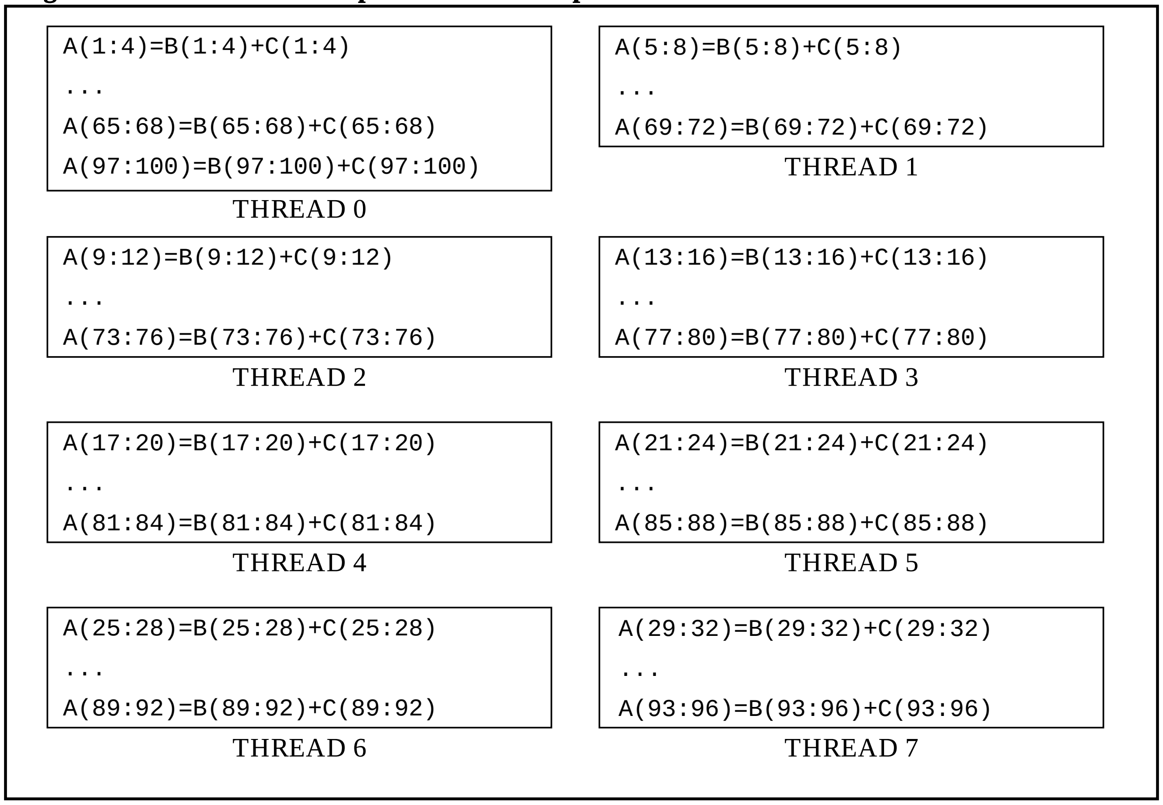
Consider the following Fortran example, which uses the PREFER_PARALLEL directive, but applies to LOOP_PARALLEL as well:

```
C$DIR PREFER_PARALLEL(CHUNK_SIZE = 4)
      DO I = 1, 100
        A(I) = B(I) + C(I)
      ENDDO
```

In this example, the loop is parallelized by parcelling out chunks of 4 iterations to each available thread. Figure 20 uses Fortran 90 array syntax to illustrate the iterations performed by each thread, assuming 8 available threads.

Figure 20 shows that the 100 iterations of I are parcelled out in chunks of 4 iterations to each of the 8 available threads; after the chunks are distributed evenly to all threads, there is one chunk left over (iterations 97:100), which executes on thread 0.

Figure 20 Stride-parallelized loop


A(1:4)=B(1:4)+C(1:4)
...
A(65:68)=B(65:68)+C(65:68)
A(97:100)=B(97:100)+C(97:100)
THREAD 0
A(5:8)=B(5:8)+C(5:8)
...
A(69:72)=B(69:72)+C(69:72)
THREAD 1
A(9:12)=B(9:12)+C(9:12)
...
A(73:76)=B(73:76)+C(73:76)
THREAD 2
A(13:16)=B(13:16)+C(13:16)
...
A(77:80)=B(77:80)+C(77:80)
THREAD 3
A(17:20)=B(17:20)+C(17:20)
...
A(81:84)=B(81:84)+C(81:84)
THREAD 4
A(21:24)=B(21:24)+C(21:24)
...
A(85:88)=B(85:88)+C(85:88)
THREAD 5
A(25:28)=B(25:28)+C(25:28)
...
A(89:92)=B(89:92)+C(89:92)
THREAD 6
A(29:32)=B(29:32)+C(29:32)
...
A(93:96)=B(93:96)+C(93:96)
THREAD 7

An analogous C example follows:

```
#pragma _CNX prefer_parallel(chunk_size = 4)
for(i=0;i<100;i++)
  a[i] = b[i] + c[i];
```

The chunk_size = *n* attribute is most useful on loops in which the amount of work increases or decreases as a function of the iteration count. (These loops are also known as *triangular loops*.) The following Fortran example shows such a loop. Again, PREFER_PARALLEL is used here, but the concept applies to LOOP_PARALLEL also.

```
C$DIR PREFER_PARALLEL(CHUNK_SIZE = 4)
      DO J = 1,N
        DO I = J, N
          A(I,J) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

Here, the work of the I loop decreases as J increases. By specifying a chunk_size for the J loop, we more evenly balance the load across the threads executing the loop. If this loop was strip mined in the traditional manner, the amount of work contained in the strips would decrease with each successive strip; the threads performing early iterations of J would do substantially more work than those performing later iterations.

An analogous C example follows:

```
#pragma _CNX prefer_parallel(chunk_size = 4)
for(j=0;j<n;j++)
  for(i=j;i<n;i++) {
    a[i][j] = ...
    .
    .
    .
  }
```

For more information and examples on using the chunk_size = *n* attribute, see the sections “Distributing iterations on cache line boundaries” on page 280 and “Triangular loops” on page 308.

prefer_parallel

The prefer_parallel directive and pragma cause the compiler to parallelize the immediately following loop if it is free of dependences and other parallelization inhibitors. The compiler automatically privatizes any loop variables that must be privatized. prefer_parallel requires less manual intervention and is less forceful than the loop_parallel directive and pragma.

On multihypernode systems, when prefer_parallel is specified without a nodes or threads attribute, the compiler will determine if opportunities for parallelism exist within the loop and, if possible, parallelize the loop across threads. If the threads attribute is specified, the compiler attempts to find and exploit thread-parallelism within the loop. If the nodes attribute is specified, the compiler trys to locate and exploit node-parallelism within the loop.

prefer_parallel can also be used to indicate the preferred loop in a nest to parallelize, as shown in the following Fortran example:

```
      DO J = 1, 100
C$DIR   PREFER_PARALLEL
        DO I = 1, 100
          .
          .
          .
        ENDDO
      ENDDO
```

In this example, PREFER_PARALLEL causes the compiler to choose the innermost loop for parallelization, provided it is free of dependences. PREFER_PARALLEL does not inhibit loop interchange.

An analogous C example follows:

```
for(j=0;j<100;j++)
  #pragma _CNX prefer_parallel
  for(i=0;i<100;i++) {
    .
    .
    .
  }
```

Do not use the `ordered` attribute in a `prefer_parallel` directive, as it is only useful if the loop contains synchronized dependences, and `prefer_parallel` will not parallelize a loop containing any loop-carried dependences. The `ordered` attribute is useful in the `loop_parallel` directive, as described in Chapter 6, “Advanced shared-memory programming.”

loop_parallel

The `loop_parallel` directive forces parallelization of the immediately following loop. The compiler does not check for data dependences, perform variable privatization, or perform parallelization analysis. You must synchronize any dependences manually and manually privatize loop data as necessary. In absence of a `nodes` or `threads` attribute, `loop_parallel` defaults to thread parallelization.

The section “Critical sections” on page 148 contains an example of using `loop_parallel` to parallelize a loop with a dependence; the dependence is manually handled in a critical section.

The `threads`, `nodes`, `chunk_size` = *n* and `max_threads` = *m* attributes and combinations of these attributes have exactly the same effect as explained for `prefer_parallel`. `loop_parallel(ordered)` is useful for manually parallelizing loops containing manually-ordered dependences as described in Chapter 6, “Advanced shared-memory programming.”

Parallelizing loops with calls

`loop_parallel` can be useful for manually parallelizing loops containing procedure calls.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL
      DO I = 1, N
        X(I) = FUNC(I)
      ENDDO
```

The call to `FUNC` in this loop would normally prevent it from parallelizing. However, if you are sure that `FUNC` has no side effects and is compiled for reentrancy (the default on Exemplar compilers), this loop can be safely parallelized as shown. (A function does not have side effects if it does not modify its arguments, it does not modify the same memory location from one call to the next, it performs no I/O, and it does not call any procedures that have side effects. If `FUNC` does have side effects or is not reentrant, this loop may yield wrong answers.

An analogous C example follows:

```
#pragma _CNX loop_parallel(ivar=i)
  for(i=0;i<n;i++)
    x[i] = func(i);
```

NOTE In some cases, global register allocation can interfere with `loop_parallel` loops that contain procedure calls. Refer to the “Global register allocation” section of Chapter 3, “Compiler optimizations,” for more information.

Unparallelizable loops

The compiler will not parallelize any loop that does not have a number of iterations that can be determined prior to loop invocation at execution time, even when `loop_parallel` is specified.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL
      DO WHILE(A(I) .GT. 0)!WILL NOT PARALLELIZE
        .
        .
        A(I) = ...
        .
        .
      ENDDO
```

There is no way the compiler can determine the loop’s iteration count prior to loop invocation here, so the loop cannot be parallelized.

Comparing prefer_parallel and loop_parallel

The `prefer_parallel` and `loop_parallel` directives (and pragmas) are both used in parallelizing loops. Table 7 gives an overview of the differences between the two directives (pragmas). See the sections “`prefer_parallel`” on page 134 and “`loop_parallel`” on page 135 for more information.

Table 7 Comparison of prefer_parallel and loop_parallel

Directive/pragma	Advantages	Disadvantages
`prefer_parallel` Requests compiler to perform parallelization analysis on the following loop then parallelize the loop if it is safe to do so. When used with the `+Oautopar` option (the default), `prefer_parallel` overrides the compiler heuristic for picking the loop in a loop nest to parallelize. When used with `+Onoautopar`, the compiler only performs directive-specified parallelization (no heuristic is used to pick the loop in a nest to parallelize); in such cases, `prefer_parallel` requests loop parallelization.	Compiler performs parallelization analysis and variable privatization for you.	Loop may or may not execute in parallel.
`loop_parallel` Forces compiler to parallelize the following loop—assuming the iteration count can be determined prior to loop invocation.	Allows you to parallelize loops that the compiler is not able to automatically parallelize because it cannot determine dependences or side effects.	You are responsible for: • Checking for and synchronizing data dependences • Performing variable privatization

Task parallelization

The compiler does not automatically parallelize code outside a loop, but you can use tasking directives and pragmas to instruct the compiler to parallelize such code. The `begin_tasks` directive and pragma tells the compiler to begin parallelizing a series of tasks. The `next_task` directive and pragma marks the end of a task and the start of the next task. The `end_tasks` directive and pragma marks the end of a series of tasks to be parallelized and prevents execution from continuing until all tasks have completed. The sections of code delimited by these directives are referred to as a *task list*.

Within a task list, the compiler does not check for data dependences, perform variable privatization, or perform parallelization analysis. You must manually synchronize any dependences between tasks and manually privatize data as necessary. In absence of a `nodes` or `threads` attribute, `begin_tasks` defaults to thread parallelization.

The Fortran tasking directives have the following forms:

```
C$DIR BEGIN_TASKS[(attribute-list)]
C$DIR NEXT_TASK
C$DIR END_TASKS
```

The C tasking pragmas have the forms:

```
#pragma _CNX begin_tasks[(attribute-list)]
#pragma _CNX next_task
#pragma _CNX end_tasks
```

The optional *attribute-list* can contain one of the following attribute combinations (*m* is an integer constant):

- `threads`
- `nodes`
- `dist`
- `ordered`
- `max_threads =` *m*
- `threads, ordered`
- `nodes, ordered`
- `dist, ordered`
- `threads, max_threads =` *m*
- `nodes, max_threads =` *m*
- `dist, max_threads =` *m*
- `ordered, max_threads =` *m*
- `threads, ordered, max_threads =` *m*
- `nodes, ordered, max_threads =` *m*
- `dist, ordered, max_threads =` *m*

The `threads` attribute causes the tasks to run thread-parallel, and is the default. As with parallel loops, node-parallelism cannot be nested within thread-parallelism in task lists.

The `nodes` attribute causes the tasks to run node-parallel, on one thread per available hypernode.

The `dist` attribute tells the compiler to distribute the tasks across the currently active threads—instead of spawning new threads. Use the `dist` attribute (along with other valid attributes) to `begin_tasks` inside a `parallel/end_parallel` region. `begin_tasks` and `parallel/end_parallel` must appear inside the same function. The attribute list to the `parallel` directive (or pragma) determines the level of parallelism. See “Region parallelization” on page 144 for information on the attributes available to the `parallel` directive and pragma.

The `ordered` attribute causes the tasks to be initiated in their lexical order; that is, the first task in the sequence begins to run on its respective thread before the second and so on. In the absence of the `ordered` argument, the starting order will be indeterminate. While this argument ensures an ordered starting sequence, it does not provide any synchronization between tasks, and does not guarantee any particular ending order. You can manually synchronize the tasks as described in Chapter 6, “Advanced shared-memory programming,” if necessary.

The attributes specifying `max_threads =` *m* will run on no more than *m* threads, where *m* is an integer constant of known value at compile time. As shown, these attributes can include any combination of thread- or node-parallel, ordered or unordered execution.

The `ordered`, `nodes` and `ordered`, `threads` attributes cause the tasks to run ordered node-parallel and ordered thread-parallel, respectively.

NOTE Do not use tasking directives or pragmas unless you ensure that dependences do not exist or you insert your own synchronization code, if necessary, in the code delimited by the tasking directives or pragmas. The compiler performs no dependence checking or synchronization on the code in these regions. Synchronization is discussed in Chapter 6, “Advanced shared-memory programming.”

The following Fortran example shows how to insert tasking directives into a section of code containing three tasks that can be run in parallel:

```
C$DIR BEGIN_TASKS
          parallel task 1
C$DIR NEXT_TASK
          parallel task 2
C$DIR NEXT_TASK
          parallel task 3
C$DIR END_TASKS
```

The example above specifies thread-parallelism by default. The compiler transforms the code into a parallel loop and creates machine code equivalent to the following Fortran code:

```
C$DIR LOOP_PARALLEL(THREADS)
      DO 40 I = 1,3
        GOTO (10,20,30)I
10      parallel task 1
        GOTO 40
20      parallel task 2
        GOTO 40
30      parallel task 3
        GOTO 40
40    CONTINUE
```

If there are more tasks than available threads, some threads will execute multiple tasks; if there are more threads than tasks, some threads will not execute tasks.

The END_TASKS directive and pragma acts as a barrier; all parallel tasks must complete before the code following END_TASKS can execute.

Examples

The following Fortran example illustrates how to use these directives to specify simple task-parallelism:

```
C$DIR BEGIN_TASKS
      DO I = 1, N - 1
        A(I) = A(I+1) + B(I)
      ENDDO
C$DIR NEXT_TASK
      CALL TSUB(X,Y)
C$DIR NEXT_TASK
      C(1:1000:2) = D(1:500)
C$DIR END_TASKS
```

In this example, one thread executes the `DO I` loop, another thread executes the `CALL TSUB(X,Y)`, and a third thread assigns the elements of the array `D` to every other element of `C`. These threads execute in parallel, but their starting and ending orders are indeterminate.

Unless the `nodes` attribute is supplied with the `BEGIN_TASKS` directive, the tasks are thread-parallelized. This means that there is no room for nested parallelization within the individual parallel tasks of this example, so the forward LCD on the `DO I` loop is inconsequential; there is no way for the loop to run but serially. The Fortran 90 array assignment in the last task will not parallelize either, even though it is technically parallelizable.

An analogous C example follows:

```
#pragma _CNX begin_tasks, task_private(i)
for(i=0;i<n-1;i++)
  a[i] = a[i+1] + b[i];
#pragma _CNX next_task
tsub(x,y);
#pragma _CNX next_task
for(i=0;i<500;i++)
  c[i*2] = d[i];
#pragma _CNX end_tasks
```

The loop induction variable `i` must be manually privatized here because it is used to control loops in two different tasks. If `i` was not private, both tasks would modify it, causing wrong answers. This is not necessary in the Fortran example because the second loop is implemented as a Fortran 90 array assignment, for which the compiler generates an independent induction variable. The `task_private` directive and pragma is described in detail in the section “`task_private`” on page 162.

Nested task parallelism is also possible. In order to nest any parallelism on an X2000 server, thread-parallelism must be nested within node-parallelism; when nesting tasking directives or pragmas, `begin_tasks(nodes)` must enclose `begin_tasks(threads)`. Also, if a node-parallel task contains a parallel loop, the loop cannot go node-parallel. Thread-parallelism nested within node-parallelism can only run on the threads of the hypernode it is contained within.

The following Fortran example is more involved and exploits two-dimensional parallelism:

```
C$DIR BEGIN_TASKS(NODES)
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1,N
        IF(B(I) .NE. 0) THEN
          A(I) = B(I)*C(I)
        ELSE
          A(I) = C(I)*D(I)
        ENDIF
      ENDDO
C$DIR NEXT_TASK
C$DIR   BEGIN_TASKS(THREADS)
        CALL T1SUB()
C$DIR   NEXT_TASK
        CALL T2SUB()
C$DIR   NEXT_TASK
        CALL T3SUB()
C$DIR   END_TASKS            !(THREADS)
C$DIR NEXT_TASK
      X(1:1000) = Y(1:1000)
C$DIR END_TASKS              !(NODES)
```

Here, the first node-parallel task contains a `LOOP_PARALLEL(THREADS)` loop that goes parallel on the threads of the hypernode on which this task is running. The second node-parallel task contains a task list of three subroutine calls, each of which runs on a separate thread within the hypernode. The third node-parallel task contains a Fortran 90 array section assignment which is a candidate for parallelization.

An analogous C example follows:

```
#pragma _CNX begin_tasks(nodes)
#pragma _CNX loop_parallel(threads, ivar=i)
for(i=0;i<n;i++)
  if(b[i] != 0)
    a[i] = b[i]*c[i];
  else
    a[i] = c[i]*d[i];
#pragma _CNX next_task
  #pragma _CNX begin_tasks(threads)
  t1sub();
  #pragma _CNX next_task
  t2sub();
  #pragma _CNX next_task
  t3sub();
  #pragma _CNX end_tasks   /* (threads) */
#pragma _CNX next_task
for(j=0;j<1000;j++)
  x[j] = y[j];
#pragma _CNX end_tasks     /* (nodes) */
```

Task parallelism can become even more involved, as described in Chapter 6, "Advanced shared-memory programming."

Region parallelization

A parallel region is a single block of code that is written to run replicated on several (or many) threads. The idea is that any scalar code within the parallel region is run by each thread in preparation for work-sharing parallel constructs such as `prefer_parallel(dist)`, `loop_parallel(dist)`, or `begin_tasks(dist)`. The scalar code typically assigns data into `parallel_private` variables so that subsequent references to the data have a high cache hit rate. Within a parallel region, code execution can be restricted to subsets of threads by using conditional blocks that test the thread ID. For an example of how to use the `dist` attribute, see the section "Using the attributes" on page 128.

Region parallelism differs from task parallelism in that parallel tasks are separate, contiguous blocks of code; when parallelized using the tasking directives and pragmas, each block generally runs on a separate thread, whereas a single parallel region runs on several threads.

Specifying parallel tasks is also typically less time consuming because each thread's work is implicitly defined by the task boundaries; in region parallelism, you must manually modify the region to identify thread-specific code. However, region parallelism can reduce parallelization overhead as discussed in the section explaining the `dist` attribute on page 128.

The beginning of a parallel region is denoted by the `parallel` directive or pragma; the end is denoted by the `end_parallel` directive or pragma. `end_parallel` also prevents execution from continuing until all copies of the parallel region have completed.

Within a parallel region, the compiler does not check for data dependences, perform variable privatization, or perform parallelization analysis; you must manually synchronize any dependences between copies of the region and manually privatize data as necessary. In absence of a `nodes` or `threads` attribute, `parallel` defaults to thread parallelization.

The `parallel`/`end_parallel` Fortran directives have the following form:

```
C$DIR PARALLEL[(attribute-list)]
C$DIR END_PARALLEL
```

The C pragmas have the form:

```
#pragma _CNX parallel(attribute-list)
#pragma _CNX end_parallel
```

The optional *attribute-list* can contain one of the following attributes (*m* is an integer constant):

- `threads`
- `nodes`
- `max_threads = ` *m*
- `threads`, `max_threads = ` *m*
- `nodes`, `max_threads = ` *m*

The `threads` attribute causes the region to run thread-parallel and is the default. As with parallel loops, node-parallelism cannot be nested within thread-parallelism in regions.

The `nodes` attribute causes the region to run node-parallel, on one thread per available hypernode.

The `max_threads = ` *m* attribute will cause the region to run on no more than *m* threads, where *m* is an integer constant. As shown, these attributes can include any combination of thread- or node-parallel execution.

NOTE Do not use the parallel region directives or pragmas unless you ensure that dependences do not exist or you insert your own synchronization code, if necessary, in the region. The compiler performs no dependence checking or synchronization on the code delimited by the parallel region directives and pragmas. Synchronization is discussed in Chapter 6, “Advanced shared-memory programming.”

Consider the following Fortran example:

```
      REAL A(1000,8), B(1000,8), C(1000,8), RDONLY(1000), SUM(8)
      INTEGER MYTID
      .
      .
      .
C     FIRST INITIALIZATION OF RDONLY IN SERIAL CODE:
      CALL INIT1(RDONLY)
      IF(NUM_THREADS() .LT. 8) STOP "NOT ENOUGH THREADS; EXITING"
C$DIR PARALLEL(MAX_THREADS = 8), PARALLEL_PRIVATE(I, J, K, MYTID)
      MYTID = MY_THREAD() + 1 !ADD 1 FOR PROPER SUBSCRIPTING
      DO I = 1, 1000
        A(I, MYTID) = B(I, MYTID) * RDONLY(I)
      ENDDO
      IF(MYTID .EQ. 1) THEN ! ONLY THREAD 0 EXECUTES SECOND
        CALL INIT2(RDONLY)  ! INITIALIZATION
      ENDIF
      DO J = 1, 1000
        B(J, MYTID) = B(J, MYTID) * RDONLY(J)
        C(J, MYTID) = A(J, MYTID) * B(J, MYTID)
      ENDDO
      DO K = 1, 1000
        SUM(MYTID) = SUM(MYTID) + A(K,MYTID) + B(K,MYTID) + C(K,MYTID)
      ENDDO
C$DIR END_PARALLEL
```

In this example, all arrays that are written to in the parallel code have one dimension for each of the anticipated number of parallel threads; each thread can work on disjoint data, there is no chance of two threads attempting to update the same element, and, therefore, there is no need for explicit synchronization. The RDONLY array is one-dimensional, but it is never written to by parallel threads. Before the parallel region, RDONLY is initialized in serial code.

The PARALLEL_PRIVATE directive is used to privatize the induction variables used in the parallel region. This must be done so that the various threads processing the region do not attempt to write to the same shared induction variables. PARALLEL_PRIVATE is covered in more detail in the section “parallel_private” on page 165.

At the beginning of the parallel region, the NUM_THREADS() intrinsic, which is described in detail in Chapter 6, “Advanced shared-memory programming,” is called to ensure that the expected number of threads are available. Then the MY_THREAD() intrinsic, which is also described in Chapter 6, is called by each thread to determine its thread ID; all subsequent code in the region is executed based on this ID. In the I loop, each thread computes one row of A using RDONLY and the corresponding row of B.

RDONLY is reinitialized in a subroutine call that is only executed by thread 0 before it is used again in the computation of B in the J loop, where again each thread computes a row. The J loop similarly computes C.

Finally, the K loop sums each dimension of A, B, and C into the SUM array. No synchronization is necessary here because each thread is running the entire loop serially and assigning into a discrete element of SUM.

An analogous C example follows:

```
float a[8][1000], b[8][1000], c[8][1000], rdonly[1000], sum[8];
int i, j, k, mytid;
  .
  .
  .
/* first initialization of rdonly in serial code: */
init1(rdonly);
if(num_threads() < 8) {
  fprintf(stderr, "not enough threads; exiting\n");
  exit(2);
}
#pragma _CNX parallel(max_threads = 8), parallel_private(i,j,k,mytid)
mytid = my_thread();
for(i=0; i<1000; i++)
  a[mytid][i] = b[mytid][i] * rdonly[i];
if(mytid == 0) init2(rdonly);
for(j=0; j<1000; j++) {
  b[mytid][j] = b[mytid][j] * rdonly[j];
  c[mytid][j] = a[mytid][j] * b[mytid][j];
}
for(k=0; k<1000; k++)
  sum[mytid] = sum[mytid] + a[mytid][k] + b[mytid][k] + c[mytid][k];
#pragma _CNX end_parallel
```

Critical sections

The `critical_section` and `end_critical_section` directives and pragmas allow you to specify sections of code in parallel loops or tasks that must be executed by only one thread at a time. These directives cannot be used for ordered synchronization within a `loop_parallel(ordered)` loop, but are suitable for simple synchronization in any other `loop_parallel` loops. (Use the `ordered_section` and `end_ordered_section` directives or pragmas for ordered synchronization within a `loop_parallel(ordered)` loop.)

A `critical_section` directive or pragma and its associated `end_critical_section` must appear in the same procedure and under the same control flow. They do not have to appear in the same procedure as the parallel construct in which they are used. In other words, the pair can appear in a procedure called from a parallel loop.

As discussed in this chapter, these directives have the following form in Fortran:

```
C$DIR CRITICAL_SECTION
C$DIR END_CRITICAL_SECTION
```

The C pragmas have the form:

```
#pragma _CNX critical_section
#pragma _CNX end_critical_section
```

The `critical_section` directive and pragma can take an optional gate attribute that allows the declaration of multiple critical sections as described in Chapter 6, “Advanced shared-memory programming;” however, we will only discuss simple critical sections here.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(FUNCTEMP)
      DO I = 1, N   ! LOOP IS PARALLELIZABLE
        .
        .
        .
        FUNCTEMP = FUNC(X(I))
C$DIR   CRITICAL_SECTION
        SUM = SUM + FUNCTEMP
C$DIR   END_CRITICAL_SECTION
        .
        .
        .
      ENDDO
```

Because `FUNC` has no side effects and can be called in parallel, the `I` loop can be parallelized as long as the `SUM` variable is only updated by one thread at a time. The critical section created around `SUM` ensures this behavior.

The `LOOP_PARALLEL` directive and the critical section are required to parallelize this loop because the call to `FUNC` would normally inhibit parallelization. If this call were not present, and if the loop did not contain other parallelization inhibitors, the compiler would automatically parallelize the reduction of `SUM` as described in the section “Reductions” on page 114. However, the presence of the call necessitates the `LOOP_PARALLEL` directive, which prevents the compiler from automatically handling the reduction; this, in turn, requires using either a critical section or the `reduction` directive. We will use a critical section for this example. Placing the call to FUNC outside of the critical section allows FUNC to be called in parallel, decreasing the amount of serial work within the critical section.

An analogous C example follows:

```
#pragma _CNX loop_parallel(ivar=i)
#pragma _CNX loop_private(functemp)
for(i=0;i<n;i++) {  /* loop is parallelizable */
  .
  .
  .
  functemp = func(x(i));
  #pragma _CNX critical_section
  sum = sum + functemp;
  #pragma _CNX end_critical_section
  .
  .
  .
}
```

In order to justify the cost of the compiler-generated synchronization code associated with the use of critical sections, loops that contain them must also contain a large amount of parallelizable (non-critical section) code. If you are unsure of the profitability of using a critical section to help parallelize a certain loop, time the loop with and without the critical section to see if parallelization justifies the overhead of the critical section.

Again, for this particular example, the `reduction` directive or pragma could have been used in place of the `critical_section`, `end_critical_section` combination. For more information, see the section “Reductions” on page 114.

+Onoautopar compiler option

You can disable *automatic* loop thread-parallelism by specifying the +Onoautopar option on the compiler command line. +Onoautopar is only meaningful when specified with the +Oparallel option at +O3 or +O4.

This option causes the compiler to parallelize only those loops that are immediately preceded by a loop_parallel or prefer_parallel directive or pragma; all other loops, even if they could normally be automatically parallelized, are not analyzed for parallelization. Because the compiler does not automatically find parallel tasks or regions, user-specified task and region parallelization is not affected by this option.

+O[no]nodepar compiler option

By default, loop, task, and region node-parallelism is disabled. In other words, +Ononodepar is the default. The +O[no]nodepar option is only meaningful when specified with the +Oparallel option at +O3 or +O4.

The +Ononodepar option causes the compiler to generate code for a single-node machine. When this option is used, serial code is generated for node-parallel constructs; thus, node-parallelism is not implemented. Thread-parallelism—both automatic and directive-specified—is still implemented.

Use the +Onodepar option to enable directive-specified node-parallelism when compiling with +Oparallel at +O3 or +O4 on a multinode, scalable SMP.

Reentrant compilation

Exemplar compilers compile for reentrancy by default in that the compiler itself does not introduce static or global references beyond what exist in the original code. Reentrant compilation causes procedures to store uninitialized local variables on the stack; no locals can carry values from one invocation of the procedure to the next (unless the variables appear in Fortran COMMON blocks or DATA or SAVE statements or in C/C++ static statements). This allows loops containing procedure calls to be manually parallelized, assuming no other inhibitors of parallelization exist.

When procedures are called in parallel, each thread receives a private stack on which to allocate local variables. This allows each parallel copy of the procedure to manipulate its local variables without interfering with any other copy's locals of the same name. When the procedure returns and the parallel threads join, all values on the stack are lost.

Default stack size

Thread 0's stack can grow to the size specified in the `maxssiz` configurable kernel parameter. Refer to the *Managing Systems and Workgroups* manual for more information on configurable kernel parameters.

Any threads your program spawns (as the result of `loop_parallel` or tasking directives or pragmas, for example) receive a default stack size of 80 Mbytes. This means that if:

- A parallel construct declares more than 80 Mbytes of `loop_private`, `task_private`, or `parallel_private` data, or
- A subprogram with more than 80 Mbytes of local data is called in parallel, or
- The cumulative size of all local variables in a chain of subprograms called in parallel exceeds 80 Mbytes,

you must modify the stack size of the spawned threads via the `CPS_STACK_SIZE` environment variable. Under `csh`, this can be done with the following command:

`setenv CPS_STACK_SIZE` *size_in_kbytes*

where

size_in_kbytes	is the desired stack size in kbytes. This value is read at program startup; it cannot be changed during execution.

For example, the following command sets the thread stack size to 100 Mbytes:

```
setenv CPS_STACK_SIZE 102400
```

Loop-specific, task-specific, and region-specific data privatization

Once assigned, the memory classes discussed in detail in Chapter 5, “Memory classes,” are in effect throughout your entire program. Any loops that manipulate variables that have been explicitly assigned a memory class must be manually parallelized, and once a variable is assigned a class, its class cannot change. While very efficient programs can be written using these memory classes, they also require a great deal of manual intervention.

To get around these problems, the Exemplar Fortran 90, Exemplar Fortran 77, and Exemplar C compilers support the `loop_private`, `task_private`, and `parallel_private` directives and pragmas. The `save_last` directive and pragma is provided to save the value of `loop_private` data objects assigned in the last iteration of the loop. These directives and pragmas allow you to easily privatize parallel loop, task, or region data temporarily; when used with `prefer_parallel`, they do so without inhibiting any automatic compiler optimizations. They can help you further increase the performance of your shared-memory program with less extra work than is required when using the standard memory classes accompanying manual parallelization and synchronization.

You can use the above directives on local variables and arrays of any type, but they should not be used on data assigned one of the static or dynamic memory classes (`thread_private`, `node_private`, `near_shared`, `far_shared` or `block_shared`).

In some cases, data declared `loop_private`, `task_private`, or `parallel_private` is stored on the stacks of the spawned threads. Spawned thread stacks default to 80 Mbytes in size; if the amount of `loop_private`, `task_private` or `parallel_private` data declared exceeds this, you can use the `CPS_STACK_SIZE` environment variable to increase the default. Refer to the section “Default stack size” on page 152 for more information.

`loop_private`

The `loop_private` directive and pragma declares a list of variables and/or arrays private to the immediately following Fortran `DO` or C `for` loop. The compiler assumes that data objects declared to be `loop_private` have no loop-carried dependences with respect to the parallel loops in which they are used. If dependences exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6, “Advanced shared-memory programming.”

`loop_private` array dimensions must be determinable at compile-time.

Each parallel thread of execution receives a private copy of the `loop_private` data object for the duration of the loop; no starting values can be assumed for the data, and unless a `save_last` directive or pragma is specified (as described in a following section), no ending value can be assumed. If a `loop_private` data object is referenced within an iteration of the loop, it must have been assigned a value previously on that same iteration.

In Fortran, the `LOOP_PRIVATE` directive has the following form:

`C$DIR LOOP_PRIVATE(`*namelist*`)`

In C, the pragma has the form:

`#pragma _CNX loop_private(`*namelist*`)`

where

namelist	is a comma-separated list of variables and/or arrays that are to be private to the immediately following loop. *namelist* cannot contain structures, dynamic arrays, allocatable arrays, or automatic arrays.

Consider the following Fortran example:

```
C$DIR LOOP_PRIVATE(S)
      DO I = 1, N
C       S IS ONLY CORRECTLY PRIVATE IF AT LEAST
C       ONE IF TEST PASSES ON EACH ITERATION:
        IF(A(I) .GT. 0) S = A(I)
        IF(U(I) .LT. V(I)) S = V(I)
        IF(X(I) .LE. Y(I)) S = Z(I)
        B(I) = S * C(I) + D(I)
      ENDDO
```

An apparent LCD on `S` exists in this example; if none of the `IF` tests are true on a given iteration, the value of `S` must wrap around from the previous iteration. The `LOOP_PRIVATE(S)` directive indicates to the compiler that `S` does, in fact, get assigned on every iteration, and therefore it is safe to parallelize this loop.

If on any iteration none of the `IF` tests pass, an actual LCD exists and privatizing `S` will result in wrong answers.

An analogous C example follows:

```
#pragma _CNX loop_private(s)
for(i=0;i<=n;i++) {
/* s is only private if at least one if
   test passes: */
  if(a[i] > 0) s = a[i];
  if(u[i] < v[i]) s = v[i];
  if(x[i] < y[i]) s = z[i];
  b[i] = s * c[i] + d[i];
}
```

Using loop_private with loop_parallel

Because the compiler does not automatically perform variable privatization in `loop_parallel` loops, you must manually privatize loop data requiring privatization. This can be easily done using the `loop_private` directive or pragma.

Consider the following Fortran example:

```
      SUBROUTINE PRIV(X,Y,Z)
      REAL X(1000), Y(4,1000), Z(1000)
      REAL XMFIED(1000)
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(XMFIED, J)
      DO I = 1, 4
C   INITIALIZE XMFIED; MFY MUST NOT WRITE TO X:
        CALL MFY(X, XMFIED)
        DO J = 1, 999
          IF (XMFIED(J) .GE. Y(I,J)) THEN
            Y(I,J) = XMFIED(J) * Z(J)
          ELSE
            XMFIED(J+1) = XMFIED(J)
          ENDIF
        ENDDO
      ENDDO
      END
```

Here, the `LOOP_PARALLEL` directive is required to parallelize the `I` loop because of the call to `MFY`. The `X` and `Y` arrays are in shared memory by default. `X` and `Z` are not written to, and the portions of `Y` written to in the `J` loop's `IF` statement are disjoint, so these shared arrays require no special attention. The local array `XMFIED`, however, is written to. But because `XMFIED` carries no values into or out of the `I` loop, it can be privatized using `LOOP_PRIVATE`. This gives each thread running the `I` loop its own private copy of `XMFIED`, eliminating the expensive necessity of synchronized access to `XMFIED`. Note that a loop-carried dependence exists for `XMFIED` in the `J` loop, but because this loop runs serially on each processor, this dependence is safe.

`J` is privatized as discussed in the section “Privatizing induction variables in nested loops” on page 160.

An analogous C example follows:

```
void priv(float x[1000], float y[4][1000], float z[1000]) {
  float xmfied[1000];
  int i,j;
#pragma _CNX loop_parallel(ivar=i), loop_private(xmfied,j)
  for(i=0;i<4;i++) {
    mfy(x,xmfied);
    for(j=0;j<999;j++) {
      if(xmfied[j] >= y[i][j]) y[i][j] = xmfied[j]*z[j];
      else xmfied[j+1] = xmfied[j];
    }
  }
}
```

Denoting induction variables in parallel loops

To safely parallelize a loop with the `loop_parallel` directive or pragma, the compiler must be able to correctly determine the loop's primary induction variable.

The compiler can find primary Fortran `DO` loop induction variables; it may, however, have trouble with `DO WHILE` or hand-rolled Fortran loops, and with all `loop_parallel` loops in C. Therefore, when you use the `loop_parallel` directive or pragma to manually parallelize a loop other than an explicit Fortran `DO` loop, you should indicate the loop's primary induction variable using the `IVAR=`*indvar* attribute to `loop_parallel`.

Consider the following Fortran example:

```
      I = 1
C$DIR LOOP_PARALLEL(IVAR = I)
10    A(I) = ...
      .
      .          ! ASSUME NO DEPENDENCES
      .
      I = I + 1
      IF(I .LE. N) GOTO 10
```

This is a hand-rolled loop that uses `I` as its primary induction variable. To ensure parallelization, the `LOOP_PARALLEL` directive has been placed immediately before the start of the loop, and the induction variable, `I`, has been specified.

Primary induction variables in C loops can be difficult for the compiler to find, so `ivar` is required in all `loop_parallel` C loops. Its use is shown in the following example:

```
#pragma _CNX loop_parallel(ivar=i)
  for(i=0; i<n; i++) {
    a[i] = ...;
    .
    . /* assume no dependences */
    .
  }
}
```

Secondary induction variables are variables used to track loop iterations even though they do not appear in the Fortran `DO` statement. They cannot appear in addition to the primary induction variable in the C `for` statement. Such variables *must* be a function of the primary loop induction variable; they cannot be independent. Secondary induction variables must also either be assigned a memory class manually (as described in Chapter 5, “Memory classes”) or declared `loop_private`.

The following Fortran example contains an incorrectly incremented secondary induction variable:

```
C WARNING: INCORRECT EXAMPLE!!!!
      J = 1
C$DIR LOOP_PARALLEL
      DO I = 1, N
        J = J + 2 ! WRONG!!!
```

Here, `J` will not produce expected values in each iteration because multiple threads are overwriting its value with no synchronization. The compiler cannot privatize `J` because it is a loop-carried dependence (LCD). This example can be corrected by privatizing `J` and making it a function of `I`, as shown below.

```
C CORRECT EXAMPLE:
      J = 1
C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(J) ! J IS PRIVATE
      DO I = 1, N
        J = (2*I)+1 ! J IS PRIVATE
```

Here, `J` will be assigned correct values on each iteration because it is a function of `I`, and can be safely privatized.

In C, secondary induction variables are sometimes included in `for` statements, as shown in the following example:

```
/* warning: unparallelizable code follows */
#pragma _CNX loop_parallel(ivar=i)
  for(i=j=0; i<n;i++,j+=2) {
    a[i] = ...;
    .
    .
    .
  }
}
```

Because secondary induction variables must be private to the loop and must be a function of the primary induction variable, this example cannot be safely parallelized using `loop_parallel(ivar=i)`. In the presence of this directive, the secondary induction variable will not be recognized. To manually parallelize this loop, you must remove `j` from the `for` statement and either privatize it and make it a function of `i`, or declare `j` to be shared (which is the default storage class), specify the `ordered` attribute on the `loop_parallel` directive, and increment it within an ordered critical section inside the loop. This latter method is costly in terms of synchronization overhead and may degrade the performance of the loop.

The following example demonstrates how to restructure the loop so that `j` is a valid secondary induction variable:

```
#pragma _CNX loop_parallel(ivar=i)
#pragma _CNX loop_private(j)
  for(i=0; i<n; i++) {
    j = 2*i;
    a[i] = ...;
    .
    .
    .
  }
}
```

This method runs faster than placing `j` in a critical section because it requires no synchronization overhead, and the private copy of `j` used here can typically be more quickly accessed than a shared variable.

Privatizing induction variables in nested loops

The induction variables of nonparallel loops that are contained within parallel loops must be declared `loop_private` with respect to their closest enclosing parallel loop.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL(THREADS)
C$DIR LOOP_PRIVATE(J)
      DO I = 1, N      ! I LOOP GOES PARALLEL
        DO J = 1, M    ! J LOOP IS SERIAL
          .
          .
          .
        ENDDO
      ENDDO
```

Here, `LOOP_PARALLEL` causes the `I` loop to be parallelized across threads. The `J` loop, then, runs serially. `J` must be private with respect to the `I` loop so that the threads that run the `I` loop do not attempt to update the same copy of `J`. If the loop is automatically parallelized by the compiler, or parallelized due to the presence of a `PREFER_PARALLEL` directive, this privatization will be automatic. But the presence of the `LOOP_PARALLEL` directive requires manual privatization.

An analogous C example follows:

```
#pragma _CNX loop_parallel(threads, ivar=i)
#pragma _CNX loop_private(j)
for(i=0;i<50;i++) {
  for(j=0;j<50;j++) {
    .
    .
    .
  }
}
```

This also applies to nested parallel outer loops. In this case, loop variables contained within a parallel construct—even if they are used in a parallel loop themselves—must be declared private with respect to the innermost enclosing parallel loop.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J)
      DO I = 1, N      ! I LOOP GOES NODE PAR
C$DIR LOOP_PARALLEL(THREADS)
C$DIR LOOP_PRIVATE(K)
        DO J = 1, M      ! J LOOP GOES THREAD PAR
          DO K = 1, L    ! K LOOP IS SERIAL
          .
          .
          .
          ENDDO
        ENDDO
      ENDDO
```

Here, `LOOP_PARALLEL` is used to parallelize the `I` loop across hypernodes, and the `J` loop across processors on each hypernode. `K` must be declared private to the `J` loop to ensure that the thread-parallel threads do not interfere with each other in updating it. `J` must be declared private to the `I` loop to ensure that each node-parallel thread gets its own copy.

An analogous C example follows:

```
#pragma _CNX loop_parallel(nodes, ivar=i), loop_private(j)
for(i=0;i<n;i++) {
#pragma _CNX loop_parallel(threads, ivar=j)
#pragma _CNX loop_private(k)
  for(j=0;j<m;j++) {
    for(k=0;k<l;k++) {
    .
    .
    .
    }
  }
}
```

task_private

The `task_private` directive declares a list of variables and/or arrays private to the immediately following tasks; it serves the same purpose for parallel tasks that `loop_private` serves for loops.

The `task_private` directive must immediately precede or appear on the same line as its corresponding `begin_tasks` directive. The compiler assumes that data objects declared to be `task_private` have no dependences between the tasks in which they are used. If dependences exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6, “Advanced shared-memory programming.”

Each parallel thread of execution receives a private copy of the `task_private` data object for the duration of the tasks; no starting or ending values can be assumed for the data. If a `task_private` data object is referenced within a task, it must have been assigned a value previously in that task.

In Fortran, the `TASK_PRIVATE` directive has the following form:

`C$DIR TASK_PRIVATE(`*namelist*`)`

In C, the pragma has the form:

`#pragma _CNX task_private(`*namelist*`)`

where

namelist	is a comma-separated list of variables and/or arrays that are to be private to the immediately following tasks. *namelist* cannot contain dynamic, allocatable, or automatic arrays.

Consider the following Fortran example:

```
      REAL*8 A(1000), B(1000), WRK(1000)
      .
      .
      .
C$DIR BEGIN_TASKS, TASK_PRIVATE(WRK)
      DO I = 1, N
        WRK(I) = A(I)
      ENDDO
      DO I = 1, N
        A(I) = WRK(N+1-I)
        .
        .
        .
      ENDDO
C$DIR NEXT_TASK
      DO J = 1, M
        WRK(J) = B(J)
      ENDDO
      DO J = 1, M
        B(J) = WRK(M+1-J)
        .
        .
        .
      ENDDO
C$DIR END_TASKS
```

Here, the WRK array is used in the first task to temporarily hold the A array so that its order can be reversed. It serves the same purpose for the B array in the second task. WRK is assigned before it is used in each task.

An analogous C example follows:

```
float a[1000], b[1000], wrk[1000];
.
.
.
#pragma _CNX task_private(wrk)
#pragma _CNX begin_tasks
for(i=0;i<n;i++)
  wrk[i] = a[i];
for(i=0;i<n;i++) {
  a[i] = wrk[n-1-i];
  .
  .
  .
}
#pragma _CNX next_task
for(j=0;j<m;j++)
  wrk[j] = b[j];
for(j=0;j<m;j++) {
  b[j] = wrk[m-1-j];
  .
  .
  .
}
#pragma _CNX end_tasks
```

parallel_private

The `parallel_private` directive declares a list of variables and/or arrays private to the immediately following parallel region; it serves the same purpose for parallel regions that `task_private` serves for tasks.

The `parallel_private` directive must immediately precede or appear on the same line as its corresponding `parallel` directive. Using `parallel_private` asserts that there are no dependences in the parallel region; do not use this directive if there are dependences.

Each parallel thread of execution receives a private copy of the `parallel_private` data object for the duration of the region; no starting or ending values can be assumed for the data. If a `parallel_private` data object is referenced within a region, it must have been assigned a value previously in the region.

In Fortran, the `PARALLEL_PRIVATE` directive has the form:

`C$DIR PARALLEL_PRIVATE(`*namelist*`)`

In C, the pragma has the form:

`#pragma _CNX parallel_private(`*namelist*`)`

where

namelist	is a comma-separated list of variables and/or arrays that are to be private to the immediately following parallel region. *namelist* cannot contain dynamic, allocatable, or automatic arrays.

Consider the following Fortran example:

```
      REAL A(1000,8), B(1000,8), C(1000,8), AWORK(1000), SUM(8)
      INTEGER MYTID
      .
      .
      .
C$DIR PARALLEL(MAX_THREADS = 8), PARALLEL_PRIVATE(I,J,K,L,M,AWORK,MYTID)
      IF(NUM_THREADS() .LT. 8) STOP "NOT ENOUGH THREADS; EXITING"
      MYTID = MY_THREAD() + 1 !ADD 1 FOR PROPER SUBSCRIPTING
      DO I = 1, 1000
        AWORK(I) = A(I, MYTID)
      ENDDO
      DO J = 1, 1000
        A(J, MYTID) = AWORK(J) + B(J, MYTID)
      ENDDO
      DO K = 1, 1000
        B(K, MYTID) = B(K, MYTID) * AWORK(K)
        C(K, MYTID) = A(K, MYTID) * B(K, MYTID)
      ENDDO
      DO L = 1, 1000
        SUM(MYTID) = SUM(MYTID) + A(L,MYTID) + B(L,MYTID) + C(L,MYTID)
      ENDDO
      DO M = 1, 1000
        A(M, MYTID) = AWORK(M)
      ENDDO
C$DIR END_PARALLEL
```

This example is similar to the one presented in the section "Region parallelization" on page 144 in the way it checks for a certain number of threads and divides up the work among those threads. However, the `parallel_private` variable AWORK is introduced.

Each thread initializes its private copy of AWORK to the values contained in a dimension of the array A at the beginning of the parallel region; this allows the threads to reference AWORK without regard to thread ID, because no thread can access any other thread's copy of AWORK. Note that AWORK cannot carry values into or out of the region, so it must be initialized within the region.

All induction variables contained in a parallel region must be privatized. Remember that the code contained in the region runs on all available threads, so failing to privatize an induction variable would allow each thread to update the same shared variable, creating indeterminate loop counts on every thread.

In the J loop after AWORK is initialized, AWORK is effectively used in a reduction on A (since at this point its contents are identical to the MYTID dimension of A). After A is modified here and used in the K and L loops, each thread restores a dimension of A's original values from its private copy of AWORK, which carried the appropriate dimension through the region unaltered.

An analogous C example follows:

```
float a[8][1000], b[8][1000], c[8][1000], awork[1000];
int i, mytid;
  .
  .
  .
#pragma _CNX parallel(max_threads = 8)
#pragma _CNX parallel_private(i,j,k,l,m,awork,mytid)
if(num_threads() < 8) {
  fprintf(stderr, "not enough threads; exiting\n");
  exit(2);
}
mytid = my_thread();
for(i=0; i<1000; i++)
  awork[i] = a[mytid][i];
for(j=0; j<1000; j++)
  a[mytid][j] = awork[j] + b[mytid][j];
for(k=0; k<1000; k++) {
  b[mytid][k] = b[mytid][k] * awork[k];
  c[mytid][k] = a[mytid][k] * b[mytid][k];
}
for(l=0; l<1000; l++)
  sum[mytid] = sum[mytid] + a[mytid][l] + b[mytid][l] + c[mytid][l];
for(m=0; m<1000; m++)
  a[mytid][m] = awork[m];
#pragma _CNX end_parallel
```

save_last[(*list*)]

The save_last directive and pragma allow you to save the final value of loop_private data objects assigned in the last iteration of the immediately following loop. If *list* (the optional, comma-separated list of loop_private data objects) is specified, only the final values of those data objects in *list* are saved. If *list* is not specified, the final values of all loop_private data objects assigned in the last loop iteration are saved.

The values must be assigned in the last iteration; if the assignment is executed conditionally, it is your responsibility to ensure that the condition is met and the assignment executes. Incorrect answers can result if the assignment does not execute on the last iteration. For loop_private arrays, only those elements of the array assigned on the last iteration will be saved.

In Fortran, the SAVE_LAST directive has the form:

C$DIR SAVE_LAST[(*list*)]

In C, the pragma has the form:

#pragma _CNX save_last[(*list*)]

save_last must appear immediately before or after the associated loop_private directive or pragma, or on the same line.

A save_last directive or pragma causes the thread that executes the last iteration of the loop to write back the private (or local) copy of the variable into the global reference.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(ATEMP, X, Y)
C$DIR SAVE_LAST(ATEMP, X)
      DO I = 1, N
        IF (I .EQ. D(I)) ATEMP = A(I)
        IF (I .EQ. E(I)) ATEMP = B(I)
        IF (I .EQ. F(I)) ATEMP = C(I)
        A(I) = B(I) + C(I)
        B(I) = ATEMP
        X = ATEMP * A(I)
        Y = ATEMP * C(I)
      ENDDO
      .
      .
      .
      IF(ATEMP .GT. AMAX) THEN
        .
        .
        .
```

Here, the LOOP_PRIVATE variable ATEMP is conditionally assigned in the loop; in order for ATEMP to be truly private, you must be sure that at least one of the conditions is met so that ATEMP is assigned on every iteration. When the loop terminates, the SAVE_LAST directive ensures that ATEMP and X contain the values they are assigned on the last iteration. These values can then be used later in the program. The value of Y however is not available once the loop finishes because Y is not specified as an argument to SAVE_LAST.

An analogous C example follows:

```
#pragma _CNX loop_parallel(ivar=i)
#pragma _CNX loop_private(atemp, x, y)
#pragma _CNX save_last(atemp, x)
for(i=0;i<n;i++) {
  if(i==d[i]) atemp = a[i];
  if(i==e[i]) atemp = b[i];
  if(i==f[i]) atemp = c[i];
  a[i] = b[i] + c[i];
  b[i] = atemp;
  x = atemp * a[i];
  y = atemp * c[i];
}
.
.
.
if(atemp > amax) {
.
.
.
```

Note that the `save_last` directive can be misleading in certain loop contexts.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(S)
C$DIR SAVE_LAST
      DO I = 1, N
        IF(G(I) .GT. 0) THEN
          S = G(I) * G(I)
        ENDIF
      ENDDO
```

While it may appear that the last value of S assigned (on whatever iteration) is saved in this example, you must remember that the SAVE_LAST directive applies only to the last (Nth) iteration, without regard for any conditionals contained in the loop. For SAVE_LAST to be valid here, `G(N)` must be greater than 0 so that the assignment to S takes place on the final iteration. Obviously, if this condition can be predicted, the loop can be more efficiently written to exclude the `IF` test, so the presence of a SAVE_LAST in such a loop is suspect.

5 Memory classes

Chapter 2 discusses the partitions of physical memory available on Hewlett-Packard SMP servers.

For nonscalable SMP systems, there is only one partition, hypernode-local memory, which is accessed using the `thread_private` and `node_private` virtual memory classes. All other memory classes are automatically mapped to the `node_private` memory class. For applications that will be ported to HP scalable SMPs, all five memory classes can be useful.

For multinode, scalable SMP systems, there are three partitions:

- Hypernode-local memory, which is accessed via the `thread_private` and `node_private` virtual memory classes.
- System-global memory, which is accessed via the `near_shared`, `far_shared` and `block_shared` virtual memory classes.
- CTIcache physical memory, which holds copies of shared-memory data that is not resident in the hypernode's physical memory, but is accessed by threads running on the hypernode.

NOTE The memory classes discussed here are of interest to programmers who wish to manually optimize their shared-memory programs by using compiler directives or pragmas to partition memory and otherwise control compiler optimizations as discussed in Chapter 6, "Advanced shared-memory programming."

Private versus shared memory

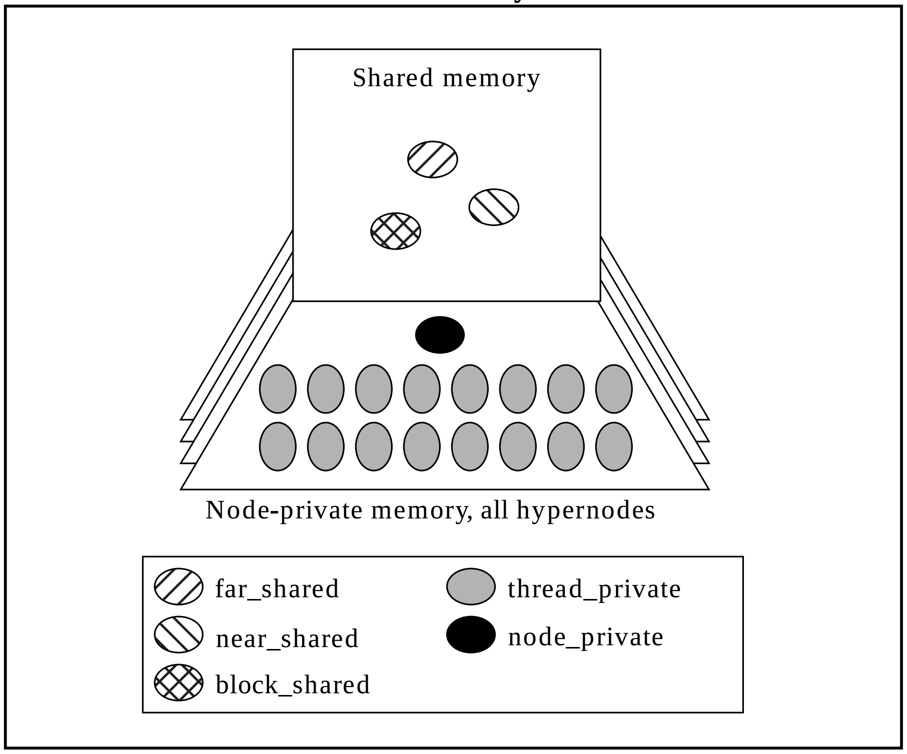
Private and shared data are differentiated by their accessibility, and, as noted above, by the physical memory classes in which they are stored.

Both `node_private` and `thread_private` data are stored in hypernode-local memory, and are therefore inaccessible to any hypernode other than the one on which they reside. In the case of `thread_private`, access is further restricted to the declaring thread. Latency is identical for private data items that must be fetched from memory. `near_shared`, `far_shared` and `block_shared` data, on the other hand, are stored in system-global physical memory and are therefore accessible from any hypernode in the system on which the process is running. Memory latency can vary for the shared-memory classes depending on whether or not the data is resident on the requesting hypernode.

Memory class addressing

Figure 21 shows the virtual addresses associated with a data item stored in each memory class by a single process running on a conceptual 4-hypernode, 16-processors-per-hypernode X2000 server. Each oval represents a unique virtual address for the same data item within a class; the memory class is indicated by the oval's fill pattern as explained in the illustration.

Figure 21 **Virtual addresses for various memory classes**



As shown, the shared data items are accessible from any hypernode using the same virtual address; the `node_private` data item has a single unique virtual address; the `thread_private` item has up to 16 unique virtual addresses, one for each thread within a hypernode.

Figure 22 shows the physical addresses associated with the data items shown in Figure 21, for an identical server. For illustrative purposes, all shared data in Figure 22 is assumed to be array data, and is represented by circles and portions of circles rather than ovals.

Figure 22 Physical addresses for various memory classes

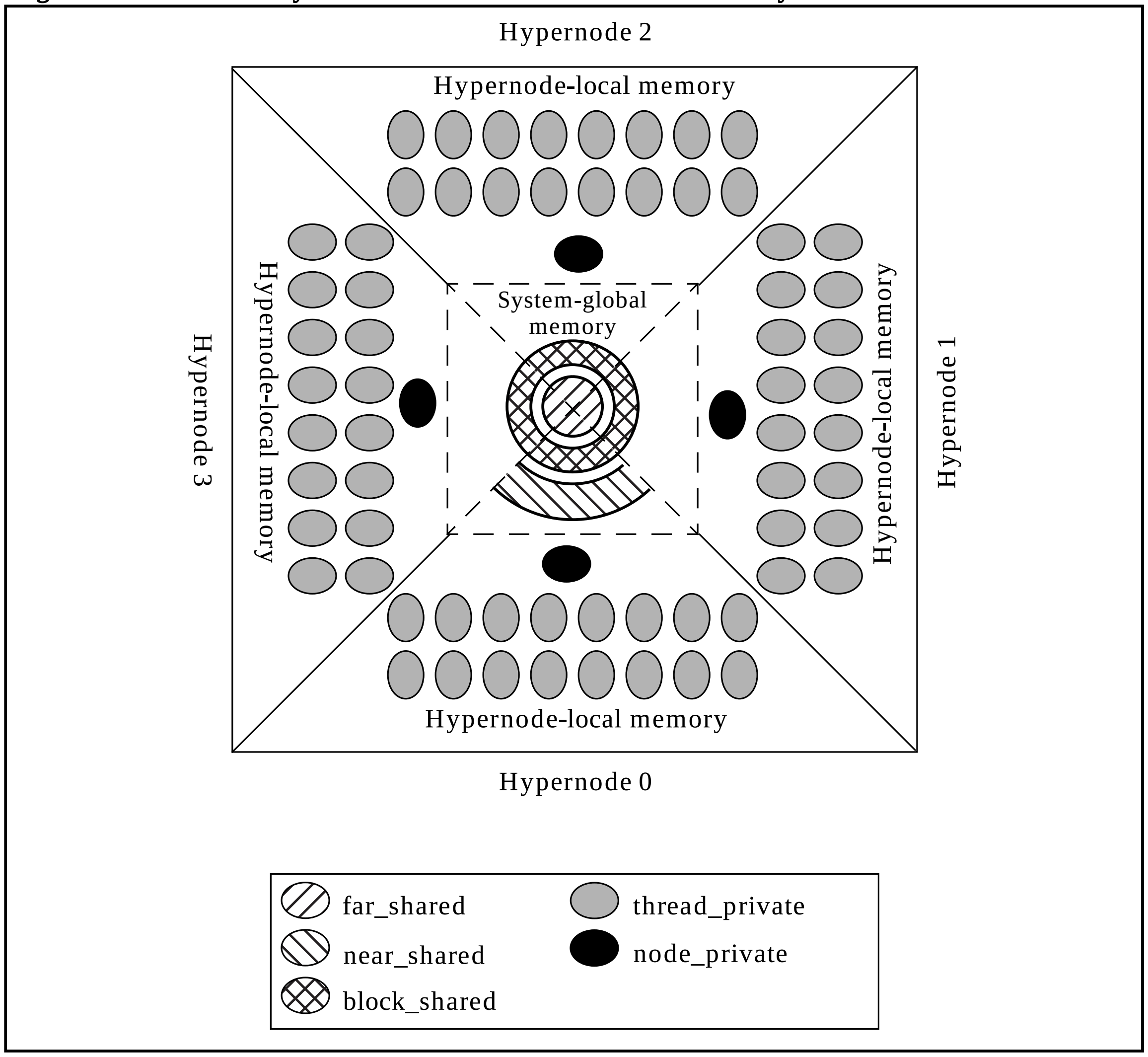


Figure 22 shows the physical memory replication of the private memory classes, as well as the distributed nature of the `far_shared` and `block_shared` classes. You can control hypernode placement of the near_shared class, shown here residing physically on logical hypernode 0. All hypernodes can access this data, because it resides in system-global physical memory.

The following subsections explain virtual and physical addressing for each memory class in detail.

thread_private

thread_private data is private to each thread of a process. Each thread_private data object has its own unique virtual address within a hypernode. For statically-declared thread_private data on multihypernode systems, these unique virtual addresses are replicated on each hypernode. If the data is dynamically allocated, no virtual address replication is done; all virtual addresses are unique.

These virtual addresses map to unique physical addresses in hypernode-local physical memory on each hypernode; therefore, while a thread_private data item can have identical virtual addresses on different hypernodes, these virtual addresses map to unique physical addresses. For example, on a 4-hypernode, 16-processors-per-hypernode system, a single, statically allocated thread_private data item is accessed by 16 virtual addresses that map to 64 physical addresses.

Obviously, this physical address replication can cause a single data item to occupy a large amount of memory; similarly, virtual address replication subtracts from the total 16-Tbyte virtual address space available to the process.

Any sharing of thread_private data items between threads (regardless of whether they are running on the same hypernode) must be done by synchronized copying of the item into a shared variable, or by message passing.

thread_private data cannot be initialized in Fortran DATA statements.

node_private

node_private data is private to the threads running on a given hypernode. node_private data items have one virtual address, and any thread on a hypernode can access that hypernode’s node_private data using the same virtual address. This virtual address maps to a unique physical address in hypernode-local memory. On a multihypernode system, a physical copy of the data item is contained in each hypernode’s hypernode-local memory, and this copy is accessed by the same virtual address on any hypernode.

This physical replication will multiply the amount of physical memory a node_private data item takes up by the number of hypernodes in the system.

Any sharing of node_private data items between hypernodes must be done by synchronized copying into a shared variable, or by message passing.

near_shared

A near_shared data item is accessible by any thread running on any hypernode of a system. However, it is physically stored entirely within the system-global memory of a particular hypernode. This allows faster access to the threads running on that hypernode. near_shared data has a single virtual address through which it is accessed by every thread. The data’s physical placement in hypernode-global memory eliminates the need for replication. Threads running on the hypernode on which the data is stored can access it via the crossbar (or bus, depending on the system architecture); threads running on other hypernodes must access the data via their CTIcache (on multinode systems) if it is encached there; if not, they must fetch it over the CTI rings.

As explained in the section “Static assignments” on page 181, the near_shared class is typically used for data that is accessed heavily by threads running on the hypernode on which it resides, but which must also be easily accessible to threads running on other hypernodes.

far_shared

far_shared data is accessible by any thread running on any hypernode. Because it is distributed evenly across the system-global memory of all hypernodes in a system, it provides the best average access time when it is used equally by threads running on all available hypernodes. A far_shared data item has a single virtual address and a single physical address, but the virtual pages of far_shared data are mapped, in an approximately round-robin manner, to physical memory pages on all the hypernodes in the system. The default page size is 4 kbytes. (For information on using different page sizes, see the section "Variable-sized pages" on page 37.)

As explained in the "Static assignments" section later in this chapter, the far_shared class is typically used for data that is accessed equally by all the hypernodes in a system.

The far_shared memory class is the default for any data not specifically classified by the programmer.

block_shared

A block_shared data item has a unique virtual and a unique physical address, and can be accessed by any thread running on any hypernode in the system. This memory class is used to store arrays that are dynamically allocated at runtime, when the number of hypernodes on which the process is running is known. The virtual pages of the arrays are then divided into a number of chunks equal to the number of available hypernodes, and these chunks (which likely contain multiple contiguous pages each) are distributed to the system-global physical pages of the available hypernodes, 1 chunk per hypernode. If the number of pages of a block_shared array is not integrally divisible by the number of hypernodes, the array size is increased to allow integral division.

block_shared allocation is only useful when combined with manual parallelization of loops that use the arrays. In this case it allows you to parallelize the loops such that a parallel section running on a given hypernode can access the portion of the array residing on that hypernode with low latency, but interhypernode access of the remaining elements is also possible.

Using the block_shared class is explained in detail in the "Dynamic assignments" section later in this chapter.

Memory class assignments

In Fortran, compiler directives are used to assign memory classes to data items. In C and C++, memory classes are assigned through the use of syntax extensions, which are defined in the header file `/usr/include/spp_prog_model.h`. This file must be included in any C or C++ program that uses memory classes. In C++, you can also use operator `new` to assign memory classes.

The following general form for Fortran memory class directives is:

`C$DIR` *memory_class_name*`(`*namelist*`)`

where

memory_class_name
: can be `THREAD_PRIVATE`, `NODE_PRIVATE`, `NEAR_SHARED`, `FAR_SHARED`, or `BLOCK_SHARED`. For `BLOCK_SHARED`, *namelist* must include only allocatable arrays.

namelist
: is a comma-separated list of variables, arrays, and/or `COMMON` block names to be assigned the class *memory_class_name*.

: `COMMON` block names must be enclosed in slashes (/), and only entire `COMMON` blocks can be assigned a class. This means arrays and variables in *namelist* must not also appear in a `COMMON` block, and must not be equivalenced to data objects in `COMMON` blocks.

These Fortran memory class declarations must appear with other specification statements; they cannot appear within executable statements.

In C and C++, Exemplar type-qualifier extensions are used, so memory classes are assigned in variable declarations. The general form for assigning memory classes in C and C++ is:

```
#include <spp_prog_model.h>
.
.
.
```

[*storage_class_specifier*] *memory_class_name type_specifier namelist*

where

storage_class_specifier

specifies a nonautomatic storage class

memory_class_name

is the desired memory class (`thread_private`, `node_private`, `near_shared`, or `far_shared`; the `block_shared` class must be allocated dynamically, as described in the “`block_shared`” on page 209)

type_specifier

is a C or C++ data type (`int`, `float`, etc.)

namelist

is a comma-separated list of variables and/or arrays of type *type_specifier*

In C and C++, data objects that are assigned a memory class must have static storage duration. This means that if the object is declared within a function, it must have the storage class `extern` or `static`. If such an object is not given one of these storage classes, its storage class defaults to `automatic` and it is allocated on the stack. Stack-based objects cannot be assigned a memory class, and attempting to do so will result in a compile-time error.

Data objects declared at file scope and assigned a memory class need not specify a storage class. For more information on C scoping rules, refer to the manual, *The C Programming Language*. For information on C++ scoping rules, refer to the manual *The C++ Programming Language*.

NOTE

All C and C++ code examples presented in this chapter assume that the following line appears above the code presented:

```
#include <spp_prog_model.h>
```

This header file maps user symbols to the implementation reserved space.

If operator `new` is used, it is also assumed that the line below appears above the code:

```
#include <new.h>
```

If you assign a memory class to a C or C++ structure, all structure members must be of the same class.

Once a data item is assigned a memory class, the class cannot be changed.

The Exemplar compilers provide mechanisms for assigning memory classes statically and dynamically. Static assignments make sense for the private classes and for `far_shared` memory; dynamic assignments make most sense for `near_shared` and `block_shared` memory. The sections that follow explain both static and dynamic memory class assignments in detail.

Static assignments

Static memory class assignments are physically located with variable type declarations in the source. Static memory classes are typically used with data objects that are accessed with equal frequency by all threads; these include objects of the `thread_private`, `node_private`, and `far_shared` classes. Static assignments for all classes are explained in the subsections that follow.

thread_private

Because `thread_private` variables are replicated for every thread on every hypernode, static declarations make the most sense for them.

In Fortran, the `thread_private` memory class is assigned using the `THREAD_PRIVATE` compiler directive, as shown in the following example:

```
      REAL*8 TPX(1000)
      REAL*8 TPY(1000)
      REAL*8 TPZ(1000), X, Y
      COMMON /BLK1/ TPZ, X, Y
C$DIR THREAD_PRIVATE(TPX, TPY, /BLK1/)
```

Each array declared here is 8000 bytes in size, and each variable is 8 bytes, for a total of 24,016 bytes of data. The entire `COMMON` block `BLK1` is placed in `thread_private` memory along with `TPX` and `TPY`. All memory space is replicated for each thread in hypernode-local physical memory on every hypernode.

`thread_private` variables and arrays cannot be initialized in Fortran `DATA` statements.

The following C/C++ example demonstrates several ways to declare `thread_private` storage. Note that the data objects declared here are not scoped analogously to those declared in the Fortran example:

```
/* tpa is global: */
thread_private double tpa[1000];
func() {
  /* tpb is local to func: */
  static thread_private double tpb[1000];
  /* tpc, a and b are declared elsewhere: */
  extern thread_private double tpc[1000],a,b;
  .
  .
  .
```

The C/C++ `double` data type provides the same precision as Fortran's `REAL*8`. The `thread_private` data declared here occupies the same amount of memory as that declared in the Fortran example. `tpa` is available to all functions lexically following it in the file. `tpb` is local to `func` and inaccessible to other functions. `tpc`, `a`, and `b` are declared at filescope in another file that is linked with this one.

`thread_private` `COMMON` blocks in parallel subroutines

Data local to a procedure that is called in parallel is effectively private because storage for it is allocated on the thread's private stack. However, if the data is in a Fortran `COMMON` block (or if it appears in a `DATA` or `SAVE` statement), it will not be stored on the stack. Parallel accesses to such nonprivate data must be synchronized if it is assigned a shared class; or, if the parallel copies of the procedure do not need to share the data, it can be assigned a private class.

Consider the following Fortran example:

```
      INTEGER A(1000,1000)
      .
      .
      .
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, N
        CALL PARCOM(A(1,I))
        .
        .
        .
      ENDDO
      SUBROUTINE PARCOM(A)
      INTEGER A(*)
      INTEGER C(1000), D(1000)
      COMMON /BLK1/ C, D
C$DIR THREAD_PRIVATE(/BLK1/)
      INTEGER TEMP1, TEMP2
      D(1:1000) = ...
      .
      .
      .
      CALL PARCOM2(A, JTA)
      .
      .
      .
      END

      SUBROUTINE PARCOM2(B,JTA)
      INTEGER B(*), JTA
      INTEGER C(1000), D(1000)
      COMMON /BLK1/ C, D
C$DIR THREAD_PRIVATE(/BLK1/)
      DO J = 1, 1000
        C(J) = D(J) * B(J)
      ENDDO
      END
      .
      .
      .
```

Here, COMMON block BLK1 is declared THREAD_PRIVATE, so every parallel instance of PARCOM gets its own copy of the arrays C and D.

Because this code is already thread-parallel when the COMMON block is defined, no further parallelism is possible, and BLK1 is therefore suitable for use anywhere in PARCOM. The local variables TEMP1 and TEMP2 are allocated on the stack, so each thread effectively has private copies of them.

A similar concept applies to node_private COMMON blocks in node-parallel loops, as described in the following “node_private” section.

node_private

Because the space for node_private variables is physically replicated on every hypernode, static declarations make the most sense for them.

In Fortran, the node_private memory class is assigned using the NODE_PRIVATE compiler directive, as shown in the following example:

```
      REAL*8 XNP(1000)
      REAL*8 YNP(1000)
      REAL*8 ZNP(1000), X, Y
      COMMON /BLK1/ ZNP, X, Y
C$DIR NODE_PRIVATE(XNP, YNP, /BLK1/)
```

Again, the data requires 24,016 bytes. The contents of BLK1 are placed in node_private memory along with XNP and YNP. Space for each data item is replicated once per hypernode in hypernode-local physical memory. The same virtual address is used by each thread to access its hypernode’s copy of a data item.

node_private variables and arrays can be initialized in Fortran DATA statements.

The following example shows several ways to declare `node_private` data objects in C and C++:

```
/* npa is global: */
node_private double npa[1000];
func() {
  /* npb is local to func: */
  static node_private double npb[1000];
  /* npc, a and b are declared elsewhere: */
  extern node_private double npc[1000],a,b;
  .
  .
  .
```

The `node_private` data declared here occupies the same amount of memory as that declared in the Fortran example. Scoping rules for this data are similar to those given for the `thread_private` C/C++ example.

For either language example, assuming a 16-processors-per-hypernode, 4-hypernode system, each data item would require a single virtual address, for a total of 24,016 bytes of virtual space. These virtual addresses map to 4 physical addresses each, one per hypernode, for a total of 96,064 bytes of physical memory.

Because `node_private` data is physically replicated across hypernodes but not replicated in virtual memory, on multihypernode systems it can effectively expand the physical space available to a process. For example, if a process declares 2 Gbytes of data `node_private`, the virtual addresses it uses to access this data map to a total of 32 Gbytes of physical memory on a 16-hypernode system (2 Gbytes per hypernode). Assuming this `node_private` data is made up of an array, you could manually split up a loop that manipulates the array to run on several hypernodes. Each hypernode would then compute its entire 2 Gbytes of the array; to the hypernode, this private copy appears to be the entire array, when in fact other hypernodes are working on private 2-Gbyte chunks with identical array names (and thus identical virtual addresses) that also appear to be the entire array. Through careful manual synchronization, the results of these hypernode-private computations can be shared through the use of “communication” arrays of the `near_shared`, `far_shared`, or `block_shared` memory class.

Such an approach approximates message passing using shared-memory constructs, and can be beneficial when arrays contain quantities of data that surpass available virtual memory.

`node_private` `COMMON` blocks in parallel subroutines

Fortran `COMMON` blocks created in subroutines called from node-parallel loops must be handled in the same ways as those appearing in thread-parallel loops, as discussed in the preceding “`thread_private`” section. One way to deal with them is to assign them the `node_private` memory class, as shown in the following Fortran example:

```
      INTEGER A(1000,1000)
      .
      .
      .
C$DIR LOOP_PARALLEL(NODES)
      DO I = 1, N
        CALL PARCOM(A(1,I))
        .
        .
        .
      ENDDO

      SUBROUTINE PARCOM(A)
      INTEGER A(*)
      INTEGER C(1000), D(1000)
      COMMON /BLK1/ C, D
C$DIR NODE_PRIVATE(/BLK1/)
      INTEGER TEMP1, TEMP2
      D(1:1000) = ...
      .
      .
      .
      CALL PARCOM2(A, JTA)
      .
      .
      .
      END
      SUBROUTINE PARCOM2(B,JTA)
      INTEGER B(*), JTA
      INTEGER C(1000), D(1000)
      COMMON /BLK1/ C, D
C$DIR NODE_PRIVATE(/BLK1/)
      DO J = 1, 1000
        C(J) = B(J) * D(J)
      ENDDO
      END
      .
      .
      .
```

Here, PARCOM is run on one thread per available hypernode. Each hypernode gets its own copy of C and D in NODE_PRIVATE memory. All threads within a hypernode can access that node's copy of BLK1, which contains C and D, but cannot access any other node's copy of the COMMON block.

If BLK1 is declared in the calling procedure (PARCOM in the example above), it must be assigned the NODE_PRIVATE memory class there. If BLK1 is not declared in the calling procedure, but is declared in the called procedure, the COMMON block becomes inaccessible when the called procedure exits.

If PARCOM initiates a thread-parallel loop or task that modifies anything in BLK1, care must be taken to ensure that the data being modified is either thread-privatized, modified by no more than one thread at a time, or properly synchronized for shared access. Local variables TEMP1 and TEMP2 are still allocated on each thread's stack, but because each hypernode is only running one thread as the result of the I loop (and therefore only one copy of PARCOM), each hypernode only gets one copy of each of TEMP1 and TEMP2.

Placing BLK1 in shared memory (the default if no class is specified) in this example would likely cause wrong answers because each copy of PARCOM could potentially modify the same data items in BLK1. Additionally, storing a copy of BLK1 on each hypernode decreases access time compared to storing it in any shared-memory class.

near_shared

Static assignments of the near_shared memory class are of limited usefulness, because they place the declared near_shared memory on logical hypernode 0. (Logical hypernode IDs are assigned in the order in which your program occupies the system.) The purpose of declaring the near_shared memory is defeated unless the code that most frequently accesses this data happens to be running on logical hypernode 0, which is unlikely on a multihypernode system.

However, a mechanism is provided for statically declaring near_shared memory. In Fortran, the near_shared memory class is statically assigned using the NEAR_SHARED compiler directive, as in the following example:

```
      SUBROUTINE FUNC()
      REAL*8 XNS(1000, 1000)
C$DIR NEAR_SHARED(XNS)
      .
      .
      .
```

Here, XNS is local to FUNC().

near_shared variables and arrays can be initialized in Fortran DATA statements.

A similar example in C/C++ follows:

```
func() {
  static near_shared double xns[1000][1000];
  .
  .
  .
```

Here, xns is local to func(). The near_shared storage class specifier is also legal for global and local declarations of the extern storage class.

Both language examples allocate 8,000,000 bytes of virtual address space, which maps to 8,000,000 bytes of physical memory on logical hypernode 0. Any thread running on any other hypernode will experience interhypernode access latency when accessing this data. Therefore, this code only makes sense for single-hypernode systems or for code that will be executed primarily on logical hypernode 0.

The true power of near_shared memory is realized only when it is resident on the hypernode that most frequently accesses it. The near_shared class is therefore most efficiently assigned dynamically at runtime, by the thread that will access the near_shared data object most often, on the hypernode on which that particular thread is running. Such dynamic allocation is discussed in the “Dynamic assignments” section later in this chapter.

far_shared

Because far_shared memory is physically distributed among all hypernodes and is best used when all hypernodes will be accessing it with similar frequency, static declarations make the most sense.

In Fortran, the far_shared memory class is assigned as shown in the following example:

```
      SUBROUTINE FUNC()
      REAL*8 XFS(1000,1000)
C$DIR FAR_SHARED(XFS)
      .
      .
      .
```

Here, XFS is local to FUNC().

far_shared variables and arrays can be initialized in Fortran DATA statements.

A similar C/C++ example follows:

```
void func() {
static far_shared double xfs[1000][1000];
.
.
.
```

Here, xfs is local to func(). The far_shared storage class specifier is also legal for global and local declarations of the extern storage class.

These declarations allocate 8,000,000 bytes of virtual address space, which is mapped round-robin to physical memory by pages to each hypernode on a multihypernode system. When all hypernodes are accessing the far_shared data with relatively equal frequency, this provides the best average access times. However, if your program only spawns threads on a subset of the hypernodes in a multihypernode system, `far_shared` memory is a poor choice because it will still be distributed across all hypernodes in the system. If you know that your program will only spawn threads on one hypernode, use the `near_shared` memory class to allocate memory on that hypernode; if your program will spawn threads on multiple hypernodes but not every hypernode in the system, use the `block_shared` memory class as described in the “Dynamic assignments” section.

`block_shared`

The `block_shared` memory class can only be allocated dynamically, as described in the following section.

Dynamic assignments

Dynamic memory class assignments are used with:

- Exemplar Fortran `ALLOCATABLE` arrays
- The `memory_class_malloc` function in Exemplar C and C++
- Operator `new` in C++

In Fortran, the class assignments are located with variable declarations. As with static assignments in Fortran, compiler directives are used to specify the desired memory class for a previously-declared data object. In C and C++, the memory class is specified when memory is allocated for the object using a type-qualifier extension. The allocation is done at the specific point in the program where the memory is needed, using the Fortran `ALLOCATE` statement, the C/C++ `memory_class_malloc` function, or operator `new` in C++. At this point, virtual memory is allocated, and the program’s available virtual space is decreased by the amount of memory allocated. This virtual memory does not map to physical memory until the allocated data objects are referenced.

While any memory class can be dynamically assigned, the `block_shared` class can only be assigned dynamically, and the `near_shared` class is most useful when dynamically assigned.

Memory class pointers

All shared memory classes are accessible to all threads when dynamically allocated in serial code, regardless of the allocating thread, because all threads access these classes from the same physical memory using the same virtual addresses. However, in Fortran, if more than one thread in a parallel construct attempts to allocate a shared class array, only the last allocation will exist. This is because there can only be one (internal) pointer to the allocated array; by default, this pointer is of the same class as the allocated memory, and each allocating thread resets this shared pointer.

This problem is overcome by adding class-specification directives of the following form:

`C$DIR` *memory_class_name*`_POINTER(`*allocatable-namelist*`)`

where

memory_class_name

is one of `THREAD_PRIVATE`, `NODE_PRIVATE`, `NEAR_SHARED`, or `FAR_SHARED`. *memory_class_name* cannot contain `BLOCK_SHARED` because the `BLOCK_SHARED` class is specifically designed to hold array objects, and a pointer is a scalar object.

allocatable-namelist

is a comma-separated list of arrays previously declared to be `ALLOCATABLE` in the same procedure. The private classes are included in *allocatable-namelist* because it is often only necessary to access a particular shared array from the particular thread or hypernode on which the array is being manipulated.

Allocatable private arrays are only accessible from the thread that allocates them; threads executing `ALLOCATE` statements in parallel will each be able to access the private array they allocate. Private arrays allocated outside of parallel constructs will only be accessible by thread 0.

The C/C++ `memory_class_malloc` function is very similar to standard `malloc` and has the following form:

memcls_ptr `= memory_class_malloc(size_t` *bytes*`, int` *memory_class_name*`);`

where

memcls_ptr
: is a previously-declared pointer to a variable of the desired memory class. Note that *memcls_ptr* need not be of the class indicated in *memory_class_name*, allowing you to allocate one class of memory which is accessed by a pointer of a different class. This is analogous to using the `C$DIR` *memory_class_name*`_POINTER` Fortran directive to allocate a pointer of one class to an array of a different class.

bytes
: is the requested number of bytes.

memory_class_name
: is one of `THREAD_PRIVATE_MEM`, `NODE_PRIVATE_MEM`, `NEAR_SHARED_MEM`, `FAR_SHARED_MEM`, or `BLOCK_SHARED_MEM`; these symbolic constants are defined in `spp_prog_model.h`.

In addition to its standard form, operator `new` has the following additional forms:

```
#include <new.h>
#include <spp_prog_model.h>
.
.
.
```

memcls_ptr `= new (`*memory_class_specifier*`)` *type_specifier*`;`
memcls_ptr `= new (`*memory_class_specifier*`)`
type_specifier`[`*array_size*`];`

where

memcls_ptr

is a previously-declared pointer to a variable of the desired memory class. Note that *memcls_ptr* need not be of the class indicated in *memory_class_specifier*, allowing you to allocate one class of memory which is accessed by a pointer of a different class.

memory_ class_specifier

is the desired memory class specifier (`__thread_private`, `__node_private`, `__near_shared`, `__far_shared`, or `__block_shared`)

type_specifier

is a C++ data type (`int`, `float`, etc.)

array_size

specifies the number of elements in the array if *memcls_ptr* is a pointer to an array

Not all combinations of pointer classes with data classes are supported, and not all make sense. Observe the following general rules:

- `thread_private` memory must be referenced by `thread_private` pointers.
- `node_private` memory should be referenced by `near_shared` or `far_shared` pointers.

- When shared data objects are referenced by private pointers, direct access to the object is restricted by the scope of the pointer. For example, if a `thread_private` pointer is used, access is restricted to the thread from which the pointer was allocated. If a `node_private` pointer is used, access is restricted to the threads on the hypernode from which the pointer was allocated. To access shared data using a `thread_private` pointer from a thread other than the thread that allocated the pointer, you must pass the pointer between threads. `node_private` pointers must similarly be passed between hypernodes when they are used.
- Allocatable `block_shared` arrays are never explicitly assigned a pointer in Fortran.
- Using shared-class pointers to point to shared-class memory provides pointer access to all threads, but, as with any shared-memory data, appropriate latency rules apply to the pointers.
- In Fortran, when one of the *memory_class_name*_`POINTER` directives is not used, dynamically allocated memory is accessed via a pointer of the same class as the allocated data.

Table 8 shows proper and improper pointer/data class combinations.

Table 8 **Pointer class/data class combinations**

Pointer class	**Data class**				
	`thread_private`	`node_private`	`near_shared`	`far_shared`	`block_shared`
`thread_private_pointer`	OK	NR	OK	OK	OK
`node_private_pointer`	NR	OK	OK	OK	OK
`near_shared_pointer`	NR	OK	OK	OK	OK
`far_shared_pointer`	NR	OK	OK	OK	OK

In the table above, OK means the pointer/data class combination is acceptable; NR means the combination is not recommended.

While pointer classes are provided for private variables, they are typically only needed when allocating shared memory.

Default classes for dynamic memory

NOTE For applications that run on nonscalable, single-node systems, `far_shared` memory, `near_shared` memory, and `block_shared` memory are automatically mapped to `node_private` memory.

Using standard `malloc` in a shared-memory C or C++ program allocates, by default, `far_shared` memory. Memory allocated by either standard `malloc` or `memory_class_malloc` can be deallocated using the `free` function. In C++, operator `new` allocates `far_shared` memory by default. Operator `delete` deallocates memory.

In Fortran, memory allocated using the `ALLOCATE` statement and not specifically assigned a class will be assigned the `far_shared` class by default. Such memory is deallocated using the `DEALLOCATE` statement.

The stack can exist in only one physical space, so it is allocated in `near_shared` memory by default. This means that all local data is `near_shared` and resides on logical hypernode 0 by default. (In Fortran, local data is all data not declared in `COMMON` blocks, is a dummy argument, or `SAVE`d. Within a C/C++ function, local data is all data not assigned a storage class of `static` or `extern`.)

Using each dynamic memory class is explained in detail in the sections that follow.

thread_private

Because only the allocating thread can access dynamically allocated thread_private memory, it should be allocated (and deallocated) inside thread-parallel constructs, where each thread can allocate its own copy.

Consider the following Fortran example:

```
      REAL*8 XTP(:)
C$DIR THREAD_PRIVATE(XTP)
      ALLOCATABLE XTP
      .
      .
      .
C$DIR LOOP_PARALLEL(THREADS), LOOP_PRIVATE(J)
      DO I = 1, NUM_THREADS()
C       THE FOLLOWING CODE MUST OCCUR INSIDE A
C       THREAD-PARALLEL CONSTRUCT
        ALLOCATE(XTP(N))
        DO J = 1, N
          . !COMPUTATIONS USING XTP
          .
          .
        ENDDO
        DEALLOCATE(XTP)
      ENDDO
```

This example assumes that the ALLOCATE statement is contained within a manually-created parallel loop or task. Then each parallel thread would get a private copy of the XTP array, of size N elements. Note the nested construct, in which the outer loop allocates and deallocates the thread_private array, and the inner loop uses it in computation. The outer loop calls the NUM_THREADS() intrinsic, which returns the number of threads on which the process is running, as described in Chapter 6, “Advanced shared-memory programming.”

If this code was executed inside a node-parallel construct, each thread on each hypernode would allocate XTP.

If the ALLOCATE in this example was executed in a nonparallel section of code, only thread 0 would be able to reference XTP.

A similar C example follows:

```
static thread_private double *xtp;
int nt = num_threads();
.
.
.
#pragma _CNX loop_parallel(threads, ivar=i), loop_private(j)
for(i=0;i<nt;i++) {
/* the following statement must occur inside a
   parallel construct                          */
  xtp=(double *)memory_class_malloc(sizeof(double)*n,
                                     THREAD_PRIVATE_MEM);
  for(j=0;j<n;j++) {
    . /*computations using xtp*/
    .
    .
  }

  free(xtp);
}
```

This example allocates memory in the same manner as the Fortran example.

node_private

Recall that all threads access `node_private` memory via the same virtual addresses, and that these virtual addresses map to different physical addresses on each hypernode. Dynamic allocations of `node_private` memory should be executed in serial code. If you require physical memory on every hypernode and wish to reference it using the same virtual addresses from every hypernode, you can allocate the memory from serial code using a shared pointer. Allocation examples are given later in this section.

In order to access dynamically allocated `node_private` memory from node-parallel code, a shared pointer is needed. This is because when serial Fortran code running on logical hypernode 0 executes the `ALLOCATE` statement, the default `node_private` array pointer is assigned on that hypernode only. It is true that the virtual address maps to physical memory on every hypernode, but only the pointer of the assigning hypernode gets a value; the pointers on other hypernodes are unassigned. If logical hypernode 0 is the only hypernode requiring access to the array, all the threads running on it will be able to access the array via this node_private pointer. However, because the physical memory associated with the other hypernodes' pointers has not been assigned, attempting to use the values it contains will cause an error. Therefore, arrays dynamically allocated in serial code that are to be accessed in parallel code must be accessed with explicitly-declared near_shared or far_shared pointers. Because the allocation takes place in serial code, there is no advantage to using one shared class over the other; both will be stored in physical memory on logical hypernode 0, causing pointer accesses from other hypernodes to take longer.

In Fortran, pointers default to the same memory class as the objects to which they point, so compiler directives are used to assign different memory classes to pointers. In C/C++, the pointers must be explicitly declared separately, so the memory class assignment is handled in the pointer declaration.

Consider the following Fortran example:

```
      REAL*8 XNP(:)
C$DIR NODE_PRIVATE(XNP)
C$DIR FAR_SHARED_POINTER(XNP)
      ALLOCATABLE XNP
      .
      .
      .
C THE FOLLOWING CODE SHOULD OCCUR OUTSIDE
C ANY PARALLEL CONSTRUCT:
      ALLOCATE(XNP(1000))
```

This example assumes that the ALLOCATE statement is contained within a nonparallel section of code. Assuming this code is running on a 4-hypernode system, when the ALLOCATE statement executes, 8,000 bytes (1000 elements $\times$ 8 bytes per element) of virtual space are allocated for XNP. If the array is then accessed from a node-parallel construct, a unique physical copy of XNP will be created on any accessing hypernodes, and each accessing thread will access its hypernode's copy of XNP. If every hypernode on the system accesses the array, it will occupy a total of 32,000 bytes of physical memory.

A similar C/C++ example follows:

```
static far_shared double *xnp;
.
.
.
/* the following statement should occur outside any
   parallel construct:                             */
xnp = (double *)memory_class_malloc(sizeof(double)*1000,
                                    NODE_PRIVATE_MEM);
```

The same example using operator `new`:

```
static far_shared double *xnp;
.
.
.
// the following statement should occur outside any
// parallel construct:
xnp = new (__node_private) double[1000];
```

These examples allocate memory in the same manner as the Fortran example. Note that the pointer to `xnp` is explicitly assigned the `far_shared` class in its declaration.

The preceding examples allow you to use the same names to access physically unique `node_private` arrays on each hypernode from node-parallel code. This can effectively increase your program's virtual memory space, because the same amount of virtual space is used for the arrays no matter how many hypernodes hold physical copies or run code that accesses them.

Parallel allocations of `node_private` memory should normally be done outside parallel code. This memory should then be accessed via shared pointers. When memory is allocated in such a fashion, threads will only be able to access the data that is stored physically on their hypernode. To access `node_private` data that is stored on another hypernode, the data must be passed via a shared variable.

These allocations are useful when private work arrays are needed by each hypernode in a node-parallel construct.

Consider the following Fortran example:

```
      REAL*8 NODE_V(:)
      ALLOCATABLE NODE_V
C$DIR NODE_PRIVATE(NODE_V)
C$DIR FAR_SHARED_POINTER(NODE_V)
      .
      .
      .
      NN = NUM_NODES()
      JLMT = (JTOT/NN) + 1
      ALLOCATE(NODE_V(JLMT))
C$DIR LOOP_PARALLEL(NODES, CHUNK_SIZE = 1)
C$DIR LOOP_PRIVATE(J)
      DO I = 1, M
C$DIR   LOOP_PARALLEL(THREADS)
        DO J = 1, JLMT
         NODE_V(J) = PI*(RAD(J)**2)*L(I,J)
         P(I,J) = (N(I,J)*R*T(I,J))/NODE_V(J)
        ENDDO
C$DIR   LOOP_PARALLEL(THREADS)
        DO J = 2, JLMT
          IF((NODE_V(J) .GT. NODE_V(J-1)).AND.
     >      (NODE_V(J) .GT. NODE_V(J+1)))
     >        LMAX_V(I,J) = NODE_V(J)
        ENDDO
        .
        .
        .
      ENDDO
      DEALLOCATE(NODE_V)
```

Here, the NODE_V array is used privately by each hypernode in the computation of the array P and to find values for LMAX_V. NODE_V is not needed outside of the I loop. Note that the NUM_NODES() intrinsic (described in Chapter 6, “Advanced shared-memory programming”) is used to determine the number of hypernodes on which the process is running.

An analogous C example follows:

```
static far_shared double *node_v;
.
.
.
nn = num_nodes();
jlmt = (jtot/nn) + 1;
node_v = (double*)memory_class_malloc(jlmt*sizeof(double),
                                      NODE_PRIVATE_MEM);
#pragma _CNX loop_parallel(nodes, chunk_size = 1, ivar=i)
#pragma _CNX loop_private(j)
for(i=0; i<m; i++) {
#pragma _CNX loop_parallel(threads, ivar=j)
  for(j=0; j<jlmt; j++) {
    node_v[j] = pi*rad[j]*rad[j]*l[i,j];
    p[i,j] = (n[i,j]*r*t[i,j])/node_v[j];
  }
#pragma _CNX loop_parallel(threads, ivar=j)
  for(j=1; j<jlmt; j++) {
    if((node_v[j] > node_v[j-1]) && (node_v[j] > node_v[j+1]))
      lmax_v[i,j] = node_v[j];
  }
}
free(node_v);
```

near_shared

To be most useful, `near_shared` memory must be dynamically allocated on the hypernode that will most heavily access it. `near_shared` memory should be allocated from within explicitly-defined node-parallel structures to ensure that it is allocated on the hypernodes that will use it most. Node-parallel structures are manually defined by the programmer using the `loop_parallel(nodes)`, `parallel(nodes)`, or `begin_tasks(nodes)` directives and pragmas, which are discussed in detail in Chapter 4, “Basic shared-memory programming.”

Unconditionally allocating `near_shared` arrays in node-parallel structures is possible and is discussed later. However, to take full advantage of `near_shared` arrays, they should be allocated conditionally, based on the allocating hypernode, within node-parallel code.

A Fortran example of this follows. For simplicity, we assume this example will always run on a 2-hypernode system.

```
      REAL*8 XNS(:), YNS(:)
      ALLOCATABLE XNS, YNS
C$DIR NEAR_SHARED(XNS, YNS)
      .
      .
      .
C THE FOLLOWING CODE MUST RUN NODE-PARALLEL
C$DIR LOOP_PARALLEL(NODES) ! 2-NODE SUBCMPLX
C$DIR LOOP_PRIVATE(NODE_ID)! IS ASSUMED
      DO I = 1, NUM_NODES()
        NODE_ID = MY_NODE()
        IF(NODE_ID .EQ. 0) THEN
          ALLOCATE(XNS(1000))
        ELSE
          ALLOCATE(YNS(1000))
        ENDIF
        .
        .
        .
      ENDDO
```

The LOOP_PARALLEL(NODES) directive is used to achieve node-parallelism; the MY_NODE() function returns the hypernode ID of the hypernode on which the current thread is executing. These are both discussed in more detail in Chapter 6, “Advanced shared-memory programming.” This example allocates a near_shared XNS(1000) array for logical hypernode 0, and a near_shared YNS(1000) array for logical hypernode 1. When accessed, XNS is stored in hypernode-global physical memory on logical hypernode 0; it is therefore accessible from that hypernode with the least latency. Similarly, YNS is physically stored on and quickly accessible from logical hypernode 1. The arrays’ unique virtual addresses allow either to be accessed by the hypernode it does not occupy, and more latency is involved in such accesses. This example presumes that code that follows in the program conditionally manipulates these arrays on their respective hypernodes, but that some sharing of the array data between hypernodes also occurs.

A C example that allocates memory in the same fashion follows. Again we assume that this code will always run on a two-hypernode system.

```
static near_shared double *xns, *yns;int node_id;
int nn = num_nodes();
.
.
.
/* the following code must run node-parallel on a 2-node
   system:                                          */
#pragma _CNX loop_parallel(nodes, ivar=i), loop_private(node_id)
for(i=0;i<nn;i++) {
  node_id = my_node();
  if(node_id == 0)
    xns = (double *)memory_class_malloc(sizeof(double)*1000,
                                       NEAR_SHARED_MEM);
  else
    yns = (double *)memory_class_malloc(sizeof(double)*1000,
                                       NEAR_SHARED_MEM);
.
.
.
}
```

Note that in C and C++, the pointer class is assigned with the type declaration, and the data class is assigned by the `memory_class_malloc` function. In C++, the data class can also be assigned by operator `new`.

Recall that `near_shared` data objects have a single virtual address used by all hypernodes, and, when accessed, a single physical address (on the allocating hypernode). In the above examples, the pointers used to access the `near_shared` arrays were of the same memory class as the arrays (by default in Fortran and by explicit typing in C and C++). The allocations are not thread-parallel, so a single thread on each hypernode will allocate its respective array. If, at some point later in the program, the code goes thread-parallel, all threads on a given hypernode will be able to access the arrays via their `near_shared` pointers.

However, if you wish to allocate `near_shared` memory from within a thread-parallel structure, the `near_shared` pointer can present a problem when the `near_shared` data space is actually allocated: all threads will be allocating the space using the same shared pointer, so each thread’s ALLOCATE will reset the pointer. In C and C++ the pointer class is directly controllable, but in Fortran the *memory_class*_POINTER directive must be used to assign a pointer of a different class to the array.

Table 8 on page 194 covers acceptable pointer classes to use for near_shared data. While a certain combination may be allowed, it may not make sense for the task at hand.

In the following Fortran example, the pointer to ZNS is assigned the THREAD_PRIVATE memory class, because ZNS is being allocated in a thread-parallel structure. This causes each thread to allocate its own near_shared copy of ZNS on its hypernode.

```
      REAL*8 ZNS(:)
      ALLOCATABLE ZNS
C$DIR NEAR_SHARED(ZNS)
C$DIR THREAD_PRIVATE_POINTER(ZNS)
      .
      .
      .
C THE FOLLOWING CODE MUST RUN THREAD-PARALLEL
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, NUM_THREADS()
        ALLOCATE(ZNS(1000))
        .
        .
        .
      ENDDO
```

Here, thread-parallelism is achieved using the LOOP_PARALLEL(THREADS) directive, which is discussed further in Chapter 4, “Basic shared-memory programming.” The hypernode on which the ALLOCATE statement executes allocates a near_shared copy of ZNS for each thread. Each thread running on that hypernode can then reference its copy of ZNS via the thread_private pointers provided by the THREAD_PRIVATE_POINTER directive on ZNS. If the thread-parallel section of code shown is also running node-parallel, each hypernode running it will allocate as many near_shared ZNS arrays as it has threads. For example, if 8 threads are running on each of 2 hypernodes, 16 physical and virtual copies of ZNS will be created. However, because of the thread-private pointers, these near_shared copies will lose their direct-accessibility; accesses by any thread other than the allocating thread is only possible in C and C++ and will require sharing of the thread_private pointers.

A similar C example follows:

```
static thread_private double *zns;
int nt = num_threads();
.
.
.
/* the following code must run thread-parallel   */
#pragma _CNX loop_parallel(threads, ivar=i)
for(i=0;i<nt;i++)
  zns = (double *)memory_class_malloc(sizeof(double)*1000,
                                    NEAR_SHARED_MEM);
.
.
.
```

This example allocates memory in the same manner as the Fortran example.

When only one parallel hypernode in a group of hypernode-parallel tasks needs a private array, the `near_shared` class can be allocated dynamically from within the task in question, a `near_shared` pointer can be used to provide low-latency access to the array.

The following Fortran code shows a parallel allocation for a single parallel hypernode:

```
      REAL*8 NODE_SCRATCH(:)
      ALLOCATABLE NODE_SCRATCH
C$DIR NEAR_SHARED(NODE_SCRATCH)
      .
      .
      .
C$DIR BEGIN_TASKS(NODES), TASK_PRIVATE(I)
      DO I = 1, N
        A(I) = B(I) + C(I)
      ENDDO
C$DIR NEXT_TASK
      CALL TSUB(X,Y)
      .
      .
      .
```

```
C$DIR NEXT_TASK
      ALLOCATE(NODE_SCRATCH(1000))

C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, 1000
        NODE_SCRATCH(I) = Z(I)
      ENDDO
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, 999
        Z(I) = NODE_SCRATCH(I+1)
        .
        .
        .
      ENDDO
      DEALLOCATE(NODE_SCRATCH)
C$DIR END_TASKS
      .
      .
      .
```

Here, the final task in the list allocates the temporary `near_shared` work array `NODE_SCRATCH`, uses it, and deallocates it. The compiler will thread-parallelize both loops within this task because of the `LOOP_PARALLEL` directives on them, and they will both be able to access the `NODE_SCRATCH` array with minimal latency. Because this array is `near_shared`, it can also be accessed by any other hypernode, though that does not happen in this example. Using `LOOP_PARALLEL` to manually parallelize loops is further discussed in Chapter 6, “Advanced shared-memory programming.”

An analogous C example follows:

```
static near_shared double *node_scratch;
.
.
.
#pragma _CNX begin_tasks(nodes), task_private(i)
for (i=0; i<n; i++) {
  a[i] = b[i] + c[i];
}
#pragma _CNX next_task
tsub(x,y);
.
.
.

#pragma _CNX next_task
node_scratch = (double *)memory_class_malloc(1000*sizeof(double),
                                             NEAR_SHARED_MEM);
#pragma _CNX loop_parallel(threads, ivar=i)
for(i=0; i<1000; i++) {
  node_scratch[i] = z[i];
}
#pragma _CNX loop_parallel(threads, ivar=i)
for(i=0; i<999; i++) {
  z[i] = node_scratch[i+1];
  .
  .
  .
}
free(node_scratch);
#pragma _CNX end_tasks
.
.
.
```

far_shared

Because far_shared memory is distributed across all hypernodes in the system, it is best dynamically allocated in nonparallel structures. Allocating far_shared memory in a thread- or node-parallel structure would create multiple copies of the requested far_shared array, one for each allocating thread or hypernode in the structure. This replication increases the amount of memory, both physical and virtual, used by the process, and is of questionable utility.

Consider the following Fortran example:

```
      REAL*8 XFS(:)
      ALLOCATABLE XFS
C$DIR FAR_SHARED(XFS)
      .
      .
      .
C THE FOLLOWING CODE SHOULD BE EXECUTED IN
C A NONPARALLEL STRUCTURE:
      ALLOCATE(XFS(10000))
```

Because the far_shared data in this example is allocated outside of parallel code, the default far_shared pointer is suitable; the array is allocated once, and there is no danger of parallel allocations resetting the pointer. When the ALLOCATE statement executes, 80,000 bytes of virtual space is allocated. When the array is accessed, this maps to 80,000 bytes of physical memory, which is distributed, in an approximately round-robin manner, by pages to each hypernode in the system.

A similar C/C++ example follows:

```
static far_shared double *xfs;
.
.
.
/* the following code should run in a nonparallel
   structure:                                        */
xfs = (double *)memory_class_malloc(sizeof(double)*10000,
                                    FAR_SHARED_MEM);
```

This example allocates memory in the same manner as the Fortran example.

The same example using operator new:

```
static far_shared double *xfs;
.
.
.
// the following code should run in a nonparallel
// structure:
xfs = new (__far_shared) double[10000];
```

Keep in mind that even dynamically allocated far_shared memory is distributed across all hypernodes in the system, not just across the hypernodes on which your program can spawn threads. Therefore, if your program is constrained to a subset of the hypernodes available in its system, you should use the block_shared or near_shared memory classes to avoid placing data on unused hypernodes.

far_shared is the default memory class for all data not otherwise assigned a class. (On nonscalable systems, far_shared is the same as node_private.)

block_shared

block_shared memory is specifically provided for dynamic allocation by a process running on multiple hypernodes. It is ideal for array-manipulating loops that will parallelize across hypernodes, with each hypernode computing chunks of contiguous elements of the arrays. Using block_shared memory for such arrays distributes the chunks across the hypernodes on which the program is running; work can then be distributed so that each hypernode accesses the elements that reside on it most frequently. If necessary, hypernodes can still directly access each other’s block_shared data.

As with far_shared data, this block_shared data allocation should take place outside of parallel structures; in this case, the compiler will automatically distribute the data evenly across the hypernodes of the system. Allocating block_shared memory from parallel structures will cause multiple copies (one per parallel thread) of the block_shared data to be created, a situation that wastes memory and is of questionable utility.

Consider the following Fortran example:

```
      REAL*8 XBS(:), YBS(:)
      ALLOCATABLE(XBS, YBS)
C$DIR BLOCK_SHARED (XBS,YBS)
      .
      .
      .
C THE FOLLOWING CODE SHOULD BE EXECUTED IN
C A NONPARALLEL STRUCTURE:
      ALLOCATE(XBS(10240), YBS(7680))
```

Here, 81,920 bytes of virtual space is requested for XBS, and 61,440 bytes is requested for YBS. Assuming this code is running on a 4-hypernode system, these `block_shared` arrays will have their physical pages divided equally, in contiguous chunks, among the 4 hypernodes. Recall that the default page size is 4,096 bytes, so XBS occupies 20 pages (81,920/4,096 = 20). The number of hypernodes, 4, is an integral divisor of 20, giving 5 pages per hypernode. So the first 5 pages of XBS are physically mapped to logical hypernode 0 (yet still accessible via their virtual addresses from any other hypernode), the second 5 pages go to logical hypernode 1, the third 5 pages to logical hypernode 2, and the last 5 pages to logical hypernode 3.

The 61,440 bytes of YBS, on the other hand, occupy 15 pages. 4 does not integrally divide 15, so the size of YBS is automatically increased just enough to allow the number of hypernodes to integrally divide its pages. YBS becomes an 8,192 element array; it now occupies 65,536 bytes or 16 pages of memory. These 16 pages are divided up so that the first 4 pages map to logical hypernode 0 and so on, with the last 4 mapping to logical hypernode 3.

Any hypernode-parallel code that follows the above allocation should be written such that the portion running on a particular hypernode accesses the array elements resident on that hypernode. For example, the code running on logical hypernode 2 should make most frequent use of `XBS(5121:7680)` and `YBS(3841:5760)`. The “Accessing hypernode-local block_shared elements” section, which follows, discusses how to determine which elements of a `block_shared` array are on a given hypernode.

A similar C/C++ example follows:

```
static near_shared double *xbs, *ybs;
.
.
.
/* the following code should be run in a nonparallel
   section of code:                                    */
xbs = (double *)memory_class_malloc(sizeof(double)*10240,
                                     BLOCK_SHARED_MEM);
ybs = (double *)memory_class_malloc(sizeof(double)*7680,
                                     BLOCK_SHARED_MEM);
```

The same example using operator `new`:

```
static near_shared double *xbs, *ybs;
.
.
.
// the following code should be run in a nonparallel
// section of code:
xbs = new (__block_shared) double[10240];
ybs = new (__block_shared) double[7680];
```

These examples allocate and distribute `xbs` and `ybs` as `block_shared` arrays exactly as the Fortran example did, including the automatic resizing of `ybs`. `near_shared` pointers are used to access the arrays.

When allocated, `block_shared` arrays are distributed across all hypernodes available to your program; you cannot constrain the number of hypernodes that the arrays occupy. This makes the `block_shared` class especially suitable for programs whose data sets scale with the number of available hypernodes. If there is any question as to whether your `block_shared` arrays will occupy enough pages to efficiently use the number of hypernodes available to your program, you should roughly compute the number of pages the array in question is likely to occupy. If this number is not at least equal to the number of hypernodes that will typically be available to the program, you can still use the `block_shared` class, but your program might waste some memory by expanding the array to occupy at least one page per hypernode.

While you cannot constrain the allocation of `block_shared` memory from within your program, you can constrain the number of hypernodes on which your program runs by using the `max_threads` attribute to `loop_parallel`. `block_shared` memory is only allocated on the hypernodes on which your program can spawn threads, so if your program is constrained in this way, `block_shared` memory is preferable to `far_shared` memory, which is distributed across all hypernodes in the system.

Accessing hypernode-local `block_shared` elements

Determining the range of `block_shared` array element indexes located on a given hypernode is easily computable given the page size, number of hypernodes, element size, and number of elements. The following Fortran function takes these parameters along with the current hypernode number as arguments and returns the base index for the elements on the current hypernode:

```
      INTEGER FUNCTION MIN_NODE_ELT(PGSZ,NN,ELTSZ,NELTS,CUR_NODE)
      INTEGER PGSZ, NN, ELTSZ, NELTS, CUR_NODE
      INTEGER NUM_PGS, PGS_PER_NODE
      NUM_PGS = 1 + (NELTS*ELTSZ - 1)/PGSZ
      PGS_PER_NODE = 1 + (NUM_PGS - 1)/NN
C     ADJUST MIN_NODE_ELT BY +1 TO COMPENSATE FOR ARRAY INDEXING
C     FROM 1:
      MIN_NODE_ELT = (CUR_NODE*PGS_PER_NODE*PGSZ/ELTSZ) + 1
      RETURN
      END
```

This Fortran example assumes the array is indexed from 1.

The analogous C/C++ function, which assumes indexing begins at 0, is shown below.

```
int min_node_elt(int pgsz,int nn,int eltsz,int nelts,int cur_node)
{
  int num_pgs, pgs_per_node;
  num_pgs = 1 + (nelts*eltsz - 1)/pgsz;
  pgs_per_node = 1 + (num_pgs - 1)/nn;
  return (cur_node*pgs_per_node*pgsz/eltsz);
}
```

The following Fortran example shows one way in which MIN_NODE_ELT can be used in a hypernode-parallel loop so that each hypernode accesses only its local array elements. This example assumes that the number of pages occupied by XBS is large enough to efficiently exploit all available hypernodes.

```
      REAL*8 XBS(:)
      INTEGER ARSZ
      ALLOCATABLE XBS
C$DIR BLOCK_SHARED(XBS)
      .
      .
      .
C **NO PARALLELISM**
      ALLOCATE(XBS(ARSZ))
      NN = NUM_NODES()
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J, MN, MIN_ELT, MAX_ELT)
      DO I = 1, NN        ! DO ON EACH NODE:
        MN = MY_NODE()    ! GET CURRENT NODE NUMBER
C       GET MIN AND MAX ELEMENT NUMBERS FOR CURRENT NODE:
        MIN_ELT = MIN_NODE_ELT(PGSZ,NN,8,ARSZ,MN)
        MAX_ELT = MIN_NODE_ELT(PGSZ,NN,8,ARSZ,MN+1) - 1
C       GO THREAD PARALLEL ON CURRENT NODE:
C$DIR   LOOP_PARALLEL(THREADS)
        DO J = MIN_ELT, MAX_ELT ! LOOP OVER LOCAL ELEMENTS
          XBS(J) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

Here, the array XBS is allocated in serial code. When the program goes node-parallel, each iteration calls MY_NODE to get its hypernode ID (which ranges from 0..NUM_NODES), then uses this ID in the following calls to MIN_NODE_ELT to determine the minimum and maximum indexes of the hypernode-local elements of XBS. Each hypernode then computes its elements of XBS in a thread-parallel loop that iterates over only the resident elements of XBS.

The analogous C code follows:

```
static near_shared double *xbs;
.
.
.
/********************** no parallelism *********************/
xbs = (double *)memory_class_malloc(sizeof(double)*arsz,
                                    BLOCK_SHARED_MEM);
nn = num_nodes();
#pragma _CNX loop_parallel(nodes, ivar=i)
#pragma _CNX loop_private(j, mn, min_elt, max_elt)
for(i=0; i<nn; i++) { /* do on each node:                   */
  mn = my_node();     /* get current node number            */
  /* get min and max element numbers for current node:      */
  min_elt = min_node_elt(pgsz,nn,sizeof(double),arsz,mn);
  max_elt = min_node_elt(pgsz,nn,sizeof(double),arsz,mn+1) - 1;
  /* go thread parallel on current node:                    */
#pragma _CNX loop_parallel(threads, ivar=j)
  for(j=min_elt; j<=max_elt; j++) { /* loop over local elts */
    xbs[j] = ...
    .
    .
    .
  }
}
```

Another easy way to access hypernode-local elements is to add a dimension of size 1..`num_nodes()` to your `block_shared` array when you allocate it.

The following Fortran example shows such an allocation:

```
      REAL*8 ABS(:,:)
      ALLOCATABLE ABS
C$DIR BLOCK_SHARED(ABS)
      .
      .
      .
C **NO PARALLELISM**
      ALLOCATE(ABS(N,NUM_NODES()))
      .
      .
      .
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J)
      DO I = 1, NUM_NODES()
C$DIR   LOOP_PARALLEL(THREADS)
        DO J = 1, N
          ABS(J,I) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

Here, N—which must be an integral multiple of the page size (the default page size is 4 kbytes)—is chosen so that `N*NUM_NODES()` is equal to or greater than the total number of ABS elements required. We assume that the original problem (before it is rewritten for parallelization) does not require a two-dimensional array, and that the second dimension is provided only to allow each parallel hypernode to easily index its local elements. Inside the `J` loop, the `I` index into ABS ensures that each parallel hypernode accesses its local elements automatically.

6 Advanced shared-memory programming

Most of the manual parallelization techniques discussed in Chapter 4, “Basic shared-memory programming,” allow you to take advantage of the compilers’ automatic dependence checking and data privatization. The examples that used the LOOP_PRIVATE and TASK_PRIVATE directives and pragmas are exceptions to this; in these cases, manual privatization was required, but it was done on a loop-by-loop basis. Only the simplest data dependences were handled in Chapter 4.

This chapter is concerned with manual parallelizations that use the program-wide memory classes discussed in Chapter 5, “Memory classes,” and that handle multiple and ordered data dependences.

Before we can discuss specific examples of such parallelization, however, we must introduce the remaining underlying concepts and available functions.

Parallel information functions

Several intrinsics are available to provide information regarding the parallelism or potential parallelism of your program. These are all integer functions, available in both 4- and 8-byte lengths; they can appear in executable statements anywhere an integer expression is legal. The 8-byte versions, which are suffixed with _8, are typically only used in Fortran programs in which the default data lengths have been changed using the `-I8` or similar compiler options. When default integer lengths are modified via compiler options in Fortran, the correct intrinsic is automatically chosen regardless of which is specified. These versions expect 8-byte input arguments and return 8-byte values.

NOTE All C/C++ code examples presented in this chapter assume that the line

```
#include <spp_prog_model.h>
```

appears above the C/C++ code presented. This header file contains the necessary type and function definitions.

The subsections that follow describe these functions.

Number of processors

These functions return the total number of processors on which the process has initiated threads. These threads are not necessarily active.

In Fortran, these functions have the forms:

```
INTEGER NUM_PROCS()
INTEGER*8 NUM_PROCS_8()
```

In C, they have the forms:

```
int num_procs(void);
long long num_procs_8(void);
```

`num_procs` can be used to dimension automatic and adjustable arrays in Fortran, and may be used in Fortran, C, and C++ to dynamically specify array dimensions and allocate storage.

Number of threads

These functions return the total number of threads the process creates at initiation, regardless of how many hypernodes the threads occupy, and regardless of how many are idle or active. They are typically used to manually define thread-parallel loops which may span hypernodes.

In Fortran, these functions have the forms:

```
INTEGER NUM_THREADS()
INTEGER*8 NUM_THREADS_8()
```

In C, they have the forms:

```
int num_threads(void);
long long num_threads_8(void);
```

The return value will only differ from `num_procs` if threads are oversubscribed.

Number of hypernodes

These functions return the number of hypernodes on which the process is running. They can be used to dimension automatic and adjustable arrays in Fortran and can be used in Fortran, C, and C++ to dynamically specify array dimensions and allocate storage.

In Fortran, these functions have the forms:

```
INTEGER NUM_NODES()
INTEGER*8 NUM_NODES_8()
```

In C, they have the forms:

```
int num_nodes(void);
long long num_nodes_8(void);
```

Number of threads on current hypernode

These functions return the number of the calling process's threads running on the hypernode from which the function is called. This number can vary from one hypernode to another depending on system configurations, usage of manual parallelization directives, and the number of processors installed on each hypernode.

In Fortran, these functions have the forms:

```
INTEGER NUM_NODE_THREADS()
INTEGER*8 NUM_NODE_THREADS_8()
```

In C, they have the forms:

```
int num_node_threads(void);
long long num_node_threads_8(void);
```

Thread ID

When called from parallel code these functions return the spawn thread ID of the calling thread, in the range 0..*nst*-1, where *nst* is the number of threads in the current spawn context (the number of threads spawned by the last parallel construct). Use them when you wish to direct specific tasks to specific threads inside parallel constructs.

In Fortran, these functions have the forms:

```
INTEGER MY_THREAD()
INTEGER*8 MY_THREAD_8()
```

In C, they have the forms:

```
int my_thread(void);
long long my_thread_8(void);
```

When called from serial code, these functions return 0.

Hypernode ID

These functions return the logical hypernode ID of the hypernode on which the calling thread is running, in the range 0..`num_nodes()`-1. Use them when you wish to direct specific tasks to specific hypernodes inside parallel constructs.

In Fortran, these functions have the forms:

```
INTEGER MY_NODE()
INTEGER*8 MY_NODE_8()
```

In C, they have the forms:

```
int my_node(void);
long long my_node_8(void);
```

Logical hypernode IDs range from 0..*n*-1, where *n* is the number of available hypernodes in the system. Logical IDs are assigned in the order in which your program occupies the system. The hypernode that your program’s thread 0 runs on is considered logical hypernode 0; any hypernodes it expands to later are assigned increasing logical ID numbers. Because the operating system starts a program on the least-loaded hypernode, mapping of logical hypernode IDs to physical hypernodes can differ between programs due to load balancing; thus two programs running on the same system are unlikely to address identical hypernodes with identical logical IDs.

Logical hypernode IDs are mapped to physical hypernode IDs, which are unique for each hypernode at the machine level.

Level of parallelism

These functions return a value representing the level of parallelism of the calling process.

In Fortran, these functions have the forms:

```
INTEGER LEVEL_OF_PARALLELISM()
INTEGER*8 LEVEL_OF_PARALLELISM_8()
```

In C and C++, they have the forms:

```
int level_of_parallelism(void);
long long level_of_parallelism_8(void);
```

The return value is one or a sum (bit-wise OR) of the values shown in Table 9. In C and C++, these values are `#defined` as symbolic constants in `spp_prog_model.h`.

Table 9 **Levels of parallelism**

Function return value	C/C++ symbolic constant name	Meaning
0	CPS_PL_NONE	Not parallel
1	CPS_PL_PARALLEL	Asymmetric thread active
2	CPS_PL_NODE	Node-parallelism
4	CPS_PL_NTHREAD	Thread-parallelism within a hypernode
8	CPS_PL_THREAD	Single-dimensional thread-parallelism

As an example of how these can be summed, assume the return value is 6. This means the process is two-dimensionally parallel; it first went parallel across hypernodes, and, within the current hypernode, it went parallel again on the threads of the hypernode. This differs from a return value of 8, which means the process went one-dimensionally thread-parallel and occupies all available threads on all available hypernodes with no nested parallelism.

The valid sum values are: 3,5,6,7, and 9.

A return value of 1, or a sum including 1, means an asymmetric thread is active in the calling program. Asymmetric parallelism is currently only supported by the Compiler Parallel Support Library. Refer to Appendix F, “Compiler Parallel Support Library,” for more information.

Stack memory type

These functions return a value representing the memory class that the current thread stack is allocated from. The thread stack holds all the procedure-local arrays and variables not manually assigned a class. The thread stack is created in `near_shared` memory by default. (For nonscalable SMP systems, `near_shared` memory is automatically mapped to `node_private` memory.)

In Fortran, these functions have the forms:

```
INTEGER MEMORY_TYPE_OF_STACK()
INTEGER*8 MEMORY_TYPE_OF_STACK_8()
```

In C and C++, they have the forms:

```
int memory_type_of_stack(void);
long long memory_type_of_stack_8(void);
```

These functions return one of the values described in Table 10.

Table 10 **Stack type return values**

Function return value	C/C++ symbolic constant name	Stack memory type
4	FAR_SHARED_MEM	far_shared
3	NEAR_SHARED_MEM	near_shared
2	NODE_PRIVATE_MEM	node_private

Thread IDs and nested parallelism

As discussed in Chapter 4, “Basic shared-memory programming,” you can manually parallelize nested loops and tasks to exploit up to two dimensions of parallelism. If you choose to do this, the first dimension must be node-parallel and the second must be thread-parallel. If thread-parallelism is exploited first, no dimensions are left; it is a programming error to attempt to spawn node-parallelism from within a thread-parallel construct. However, single-dimensional thread-parallel code can exploit all the threads on a system, even if they span hypernodes.

If you attempt to spawn thread-parallelism from within a thread-parallel construct and the two constructs are in the same routine, the compiler will ignore your directives on the inner thread-parallel construct. Consequently, the inner parallel construct will simply run serially. Calling a thread-parallel routine from another thread-parallel routine is considered an error but is not caught at compile-time.

Thread ID assignments

Chapter 3, “Compiler optimizations,” discusses how programs are initiated as a collection of threads, one per available processor, and how all but thread 0 are idle until parallelism is encountered. We will now discuss the details of how threads are spawned and assigned IDs.

When a process begins, the threads created to run it have unique *kernel* thread IDs. Thread 0, which runs all the serial code in the program, has kernel thread ID 0; the rest of the threads have unique but unspecified kernel thread IDs at this point. The `num_threads()` intrinsic will return the number of threads created, regardless of how many are active when it is called.

When thread 0 encounters parallelism, it *spawns* some or all of the threads created at program start. This means it causes these threads to go from idle to active, at which point they begin working on their share of the parallel code. All available threads are spawned by default, but this can be changed using various compiler directives.

If the parallel structure is thread-parallel, then `num_threads()` threads will be spawned, subject to user-specified limits. At this point, kernel thread 0 becomes *spawn* thread 0, and the spawned threads are assigned spawn thread IDs ranging from 0..`num_threads()`-1 (this range begins at what used to be kernel thread 0). If you manually limit the number of spawned threads, these IDs will range from 0 to one less than your limit. If you attempt to spawn thread-parallelism within an already thread-parallel structure, the thread attempting to spawn will acquire spawn thread ID 0. If all threads attempt to spawn thread-parallelism in this manner, they will all become spawn thread 0, each in a unique context.

If the parallel structure is node-parallel, then `num_nodes()` threads will be spawned, one per available hypernode, subject to user-specified limits. Again, kernel thread 0 becomes spawn thread 0, and in this case, the spawn thread IDs range from 0..`num_nodes()`-1, subject to user limits as described above.

If thread-parallelism is then encountered within this node-parallelism, `num_node_threads()` threads will be spawned on the hypernode or hypernodes encountering the thread-parallelism. These spawned threads will have spawn thread IDs, which are specific to the hypernode they are running on, ranging from 0..`num_node_threads()`-1, with spawn thread ID 0 belonging to the initial thread that executes the spawn. `num_node_threads()` may return a different value on each hypernode when called from node-parallel code.

Note that, with nested parallelism, a node-parallel thread that encounters a thread-parallel construct becomes spawn thread 0 on that hypernode regardless of its previous spawn thread ID. When this thread exits the thread-parallel construct, it returns to its previous spawn thread ID. The `my_thread()` intrinsic function returns the caller’s spawn thread ID, which depends on the level of parallelism.

Synchronization tools

The compiler cannot automatically parallelize loops containing complex dependences. However, a rich set of directives, pragmas and data types is available to help you manually parallelize such loops by synchronizing (and, if necessary, ordering) access to the code containing the dependence. These directives can also be used to synchronize dependences in parallel tasks. They allow you to efficiently exploit parallelism in structures that would otherwise be unparallelizable.

Gates and barriers

Gates allow you to restrict execution of a block of code to a single thread. They can be allocated, locked, unlocked and deallocated via the functions described in the section “Synchronization functions” on page 227, or they can be used with the ordered or critical section directives, which automate the locking and unlocking functions.

Barriers block further execution until all executing threads reach the barrier.

Gates and barriers use dynamically allocatable variables, declared using compiler directives in Fortran and using data type statements in C and C++. They may be initialized and referenced only by passing them as arguments to the functions discussed in the following “Synchronization functions” section.

In C and C++, gates and barriers are declared using the `gate_t`, `gate8_t`, `barrier_t` and `barrier8_t` variable declarations, which have the following forms:

```
gate_t namelist;
gate8_t namelist;
barrier_t namelist;
barrier8_t namelist;
```

where

namelist	is a comma-separated list of one or more gate or barrier names, as appropriate.

`gate8_t` and `barrier8_t` are used to declare 8-byte gate and barrier variables. The other declarations declare default-size variables.

In C and C++, gates and barriers should appear only in definition and declaration statements, and as formal and actual arguments.

In Fortran, gates and barriers are declared using the `GATE` and `BARRIER` compiler directives, which have the forms:

```
C$DIR GATE(namelist)
C$DIR BARRIER(namelist)
```

where

namelist	is a comma-separated list of one or more gate or barrier names, as appropriate.

These declare variables of the appropriate size; separate 4- and 8-byte versions are not needed in Fortran. No other type declarations are necessary for these variables; the compiler directives alone are sufficient.

In Fortran, gates and barriers can only appear:

- In `COMMON` statements (statement must precede `GATE` directive/`BARRIER` directive)
- In `DIMENSION` statements (statement must precede `GATE` directive/`BARRIER` directive)
- In preceding type statements
- As dummy arguments
- As actual arguments

Gate and barrier types override other types declared using the same names prior to the gate/barrier declaration. Once a variable is declared as a gate or barrier, it cannot be redeclared as another type. Gates and barriers cannot be equivalenced. If you place gates or barriers in `COMMON`, the `COMMON` block declaration must precede the `GATE` directive/`BARRIER` directive, and the `COMMON` block should contain only gates or only barriers. Arrays of gates or barriers must be dimensioned using `DIMENSION` statements. The `DIMENSION` statement must precede the `GATE` directive/`BARRIER` directive.

Synchronization functions

The Fortran, C, and C++ allocation, deallocation, lock and unlock functions provided for use with gates and barriers are listed here. 4- and 8-byte versions are provided; the 8-byte Fortran functions are primarily for use with compiler options that change the default data size to 8 bytes (for example, `-I8`). You must be consistent in your choice of versions—memory allocated using an 8-byte function must be deallocated using an 8-byte function.

Examples of using these functions are presented and explained in the “Synchronizing code” section, which follows.

Allocation functions

These functions allocate memory for a gate or barrier. When first allocated, gate variables are unlocked.

The Fortran gate and barrier allocation functions have the following declarations:

```
INTEGER FUNCTION ALLOC_GATE(gate)
INTEGER*8 FUNCTION ALLOC_GATE_8(gate)
INTEGER FUNCTION ALLOC_BARRIER(barrier)
INTEGER*8 FUNCTION ALLOC_BARRIER_8(barrier)
```

Where *gate* and *barrier* are the gate or barrier variables, as appropriate. These variables must be declared as described in the “Gates and barriers” section of this chapter.

In C and C++, the functions have the declarations:

```
int alloc_gate(gate_t *gate_p);
long long alloc_gate_8(gate8_t *gate_p);
int alloc_barrier(barrier_t *barrier_p);
long long alloc_barrier_8(barrier8_t *barrier_p);
```

Where *gate_p* and *barrier_p* are pointers of the indicated type, which have been previously declared as described in the “Gates and barriers” section of this chapter.

Deallocation functions

These functions free the memory assigned to the specified gate or barrier variable.

The Fortran gate and barrier deallocation functions have the following declarations:

```
INTEGER FUNCTION FREE_GATE(gate)
INTEGER*8 FUNCTION FREE_GATE_8(gate)
INTEGER FUNCTION FREE_BARRIER(barrier)
INTEGER*8 FUNCTION FREE_BARRIER_8(barrier)
```

Where *gate* and *barrier* are the previously-declared gate or barrier variables, as appropriate.

In C and C++, the functions have the declarations:

```
int free_gate(gate_t *gate_p);
long long free_gate_8(gate8_t *gate_p);
int free_barrier(barrier_t *barrier_p);
long long free_barrier_8(barrier8_t *barrier_p);
```

Always free gates and barriers when you are done using them.

Locking functions

These functions acquire a gate for exclusive access. If the gate cannot be immediately acquired, the calling thread waits for it. The conditional locking functions, which are prefixed with `COND_` or `cond_`, acquire a gate if doing so does not require a wait. If the gate is acquired, the functions return 0; if not, they return -1.

The Fortran gate locking functions have the declarations:

```
INTEGER FUNCTION LOCK_GATE(gate)
INTEGER*8 FUNCTION LOCK_GATE_8(gate)
INTEGER FUNCTION COND_LOCK_GATE(gate)
INTEGER*8 FUNCTION COND_LOCK_GATE_8(gate)
```

Where *gate* is a gate variable.

In C and C++, the functions have the following declarations:

```
int lock_gate(gate_t *gate_p);
long long lock_gate_8(gate8_t *gate_p);
int cond_lock_gate(gate_t *gate_p);
long long cond_lock_gate_8(gate8_t *gate_p);
```

Where *gate_p* is a pointer of the indicated type.

Unlocking functions

These functions release a gate from exclusive access. Gates are typically released by the thread that locks them, unless a gate was locked by thread 0 in serial code. In that case it might be unlocked by a single different thread in a parallel construct.

The Fortran gate unlocking functions have the following declarations:

```
INTEGER FUNCTION UNLOCK_GATE(gate)
INTEGER*8 FUNCTION UNLOCK_GATE_8(gate)
```

Where *gate* is a gate variable.

In C and C++, the functions have the declarations:

```
int unlock_gate(gate_t *gate_p);
long long unlock_gate_8(gate8_t *gate_p);
```

Where *gate_p* is a pointer of the indicated type.

Wait functions

These functions use a barrier to cause the calling thread to wait until the specified number of threads call the function, at which point all threads are released from the function simultaneously.

The Fortran barrier wait functions have the following declarations:

```
INTEGER FUNCTION WAIT_BARRIER(barrier,nthr)
INTEGER*8 FUNCTION WAIT_BARRIER_8(barrier,nthr)
```

Where *barrier* is a barrier variable of the indicated type and *nthr* is the number of threads calling the routine.

In C and C++, the functions have the declarations:

```
int wait_barrier(barrier_t *barrier_p,const int *nthr);
long long wait_barrier_8(barrier8_t *barrier_p,const long long *nthr);
```

Where *barrier_p* is a pointer of the indicated type and *nthr* is a pointer referencing the number of threads calling the routine.

A barrier variable can be used in multiple calls to the `wait` function, as long as the programmer ensures that two such barriers are not simultaneously active. It is also the programmer's responsibility to ensure that *nthr* reflects the correct number of threads.

sync_routine directive and pragma

Among the most basic optimizations performed by the Exemplar compiler is code motion, which is described in Chapter 3, "Compiler optimizations." This optimization can move some code across routine calls. If the routine call is to a synchronization function that the compiler cannot identify as such, and the code moved must execute on a certain side of it, this movement can cause wrong answers.

The compiler is aware of all synchronization functions presented in this chapter and in Chapter 4, "Basic shared-memory programming," and will not move code across them when they appear directly in code. However, if the synchronization function is hidden in a user-defined routine, the compiler has no way of knowing about it and may move code across it.

Anytime you call synchronization functions indirectly via your own routines (or directly via CPSlib), you must identify your routines with a `sync_routine` directive or pragma.

In Fortran, `sync_routine` has the form:

```
C$DIR SYNC_ROUTINE (routinelist)
```

In C, it has the form:

```
#pragma _CNX sync_routine (routinelist)
```

where

routinelist	is a comma-separated list of synchronization routines.

`sync_routine` is only effective for the listed routines that lexically follow it in the file in which it appears.

Consider the following Fortran example:

```
      INTEGER MY_LOCK, MY_UNLOCK
C$DIR GATE(LOCK)
C$DIR SYNC_ROUTINE(MY_LOCK, MY_UNLOCK)
      .
      .
      .
      LCK = ALLOC_GATE(LOCK)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        LCK = MY_LOCK(LOCK)
        .
        .
        .
        SUM = SUM + A(I)
        LCK = MY_UNLOCK(LOCK)
      ENDDO
      .
      .
      .
      INTEGER FUNCTION MY_LOCK(LOCK)
C$DIR GATE(LOCK)
      LCK = LOCK_GATE(LOCK)
      MY_LOCK = LCK
      RETURN
      END

      INTEGER FUNCTION MY_UNLOCK(LOCK)
C$DIR GATE(LOCK)
      LCK = UNLOCK_GATE(LOCK)
      MY_UNLOCK = LCK
      RETURN
      END
```

Here, MY_LOCK and MY_UNLOCK are user functions that call the LOCK_GATE and UNLOCK_GATE intrinsics. The SYNC_ROUTINE directive prevents the compiler from moving code across the calls to MY_LOCK and MY_UNLOCK.

Such a programming technique might be used to implement code that is portable across several parallel architectures that support critical sections using different syntax; MY_LOCK and MY_UNLOCK could simply be modified to call the correct locking and unlocking functions.

An analogous C example follows:

```
#include <spp_prog_model.h>
main() {
  int i, n, lck, sum, a[1000];
  gate_t lock;
#pragma _CNX sync_routine(mylock, myunlock)
  .
  .
  .
  lck = alloc_gate(&lock);
#pragma _CNX loop_parallel(ivar=i)
  for(i=0; i<n; i++) {
    lck = mylock(&lock);
    .
    .
    .
    sum = sum+a[i];
    lck = myunlock(&lock);
  }
}

int mylock(gate_t *lock) {
  int lck;
  lck = lock_gate(lock);  return lck;
}
int myunlock(gate_t *lock) {
  int lck;
  lck = unlock_gate(lock);
  return lck;
}
```

`sync_routine` is also useful when CPSlib routines are used for synchronization. Refer to Appendix F, “Compiler Parallel Support Library,” for more information.

`loop_parallel(ordered)`

The `loop_parallel(ordered)` directive and pragma was briefly introduced in Chapter 4, “Basic shared-memory programming.” It is designed to be used with ordered sections (which are discussed in the next section) to execute loops with ordered dependences in loop order. It accomplishes this by parallelizing the loop so that consecutive iterations are initiated on separate processors, in loop order. While `loop_parallel(ordered)` guarantees starting order, it does not guarantee ending order, and it provides no automatic synchronization. To avoid wrong answers, you *must* manually synchronize dependences using the ordered section directives, pragmas, or the synchronization intrinsics.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, 100
        .
        . !CODE CONTAINING ORDERED SECTION
        .
      ENDDO
```

Or the analogous C code:

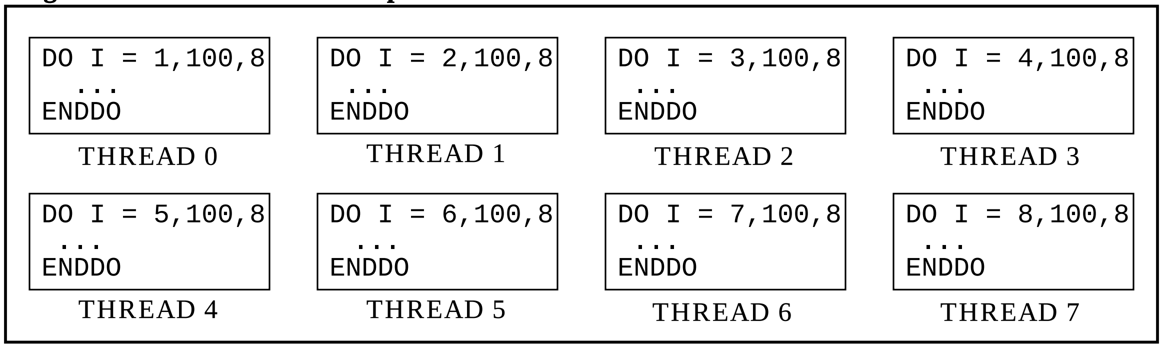
```
#pragma _CNX loop_parallel(ordered, ivar=i)
for(i=0;i<100;i++) {
  .
  . /* code containing ordered section */
  .
}
```

Assume that the body of this loop contains code that is parallelizable except for an ordered data dependence (otherwise there is no need to order the parallelization). This dependence is isolated using directives described in the next section. Also assume that 8 threads, numbered 0..7, are available to run the loop in parallel. Each thread would then execute code equivalent to the following:

```
DO I = (my_thread()+1), 100, num_threads()
  ...
ENDDO
```

Figure 23 illustrates the idea.

Figure 23 Ordered parallelization



Here, thread 0 executes first, followed by thread 1, and so on; each thread starts its iteration after the preceding iteration has started. A manually-defined ordered section will prevent one thread from executing the code in the ordered section until the previous thread exits the section, so thread 0 cannot enter the section for iteration 9 until thread 7 exits it for iteration 8. Obviously, this is only efficient if the loop body contains enough code to keep a thread busy until all other threads start their consecutive iterations, thus taking advantage of parallelism. You may find the `max_threads` attribute helpful when fine-tuning `loop_parallel(ordered)` loops to fully exploit their parallel code.

Examples of synchronizing `loop_parallel(ordered)` loops are given in the “Synchronizing code” section.

Critical and ordered sections

As discussed in Chapter 4, “Basic shared-memory programming,” critical sections allow you to synchronize simple, nonordered dependences.

The `critical_section` and `end_critical_section` directives and pragmas are used to specify critical sections. In Fortran, these directives have the following form:

```
C$DIR CRITICAL_SECTION[(gate)]
      ...
C$DIR END_CRITICAL_SECTION
```

In C, these pragmas have the form:

```
#pragma _CNX critical_section[(gate)]
      ...
#pragma _CNX end_critical_section
```

where

gate	is an optional gate variable used for access to the critical section. *gate* must be appropriately declared as described in the “Gates and barriers” section of this chapter.

The gate variable is required when synchronizing access to a shared variable from multiple parallel tasks. When a gate variable is specified, it must be allocated (using the `alloc_gate` intrinsic) outside of parallel code prior to use. If no gate is specified, the compiler creates a unique gate for the critical section. When a gate is no longer needed, it should be deallocated using the `free_gate` function.

Critical sections must be entered through the `critical_section` and exited through the `end_critical_section` directive or pragma. They must not contain branches to outside the section. The two directives must appear in the same procedure, but they do not have to be in the same procedure as the parallel construct in which they are used; that is, the directives can exist in a procedure which is called in parallel.

Ordered sections, discussed in detail here for the first time, allow you to synchronize dependences that must execute in iteration order.

The `ordered_section` and `end_ordered_section` directives and pragmas are used to specify critical sections within manually-defined, ordered `loop_parallel` loops only. In Fortran, these directives have the following form:

```
C$DIR ORDERED_SECTION(gate)
      ...
C$DIR END_ORDERED_SECTION
```

In C, these pragmas have the form:

```
#pragma _CNX ordered_section(gate)
      ...
#pragma _CNX end_ordered_section
```

where

gate	is a required gate variable that must be allocated and, if necessary, unlocked prior to invocation of the parallel loop containing the ordered section. *gate* must be appropriately declared as described in the “Gates and barriers” section of this chapter.

Ordered sections must be entered through the `ordered_section` and exited through the `end_ordered_section` directive or pragma; they cannot contain branches to outside the section. Ordered sections are subject to the same control flow rules as critical sections.

Use critical and ordered sections with care, as they add synchronization overhead to your program. They should only be used when the amount of parallel code is significantly larger than the amount of code containing the dependence.

Synchronizing code

Code containing dependences can be parallelized by synchronizing the way the parallel tasks access the dependence. This can be done manually using the gates, barriers and synchronization functions, or semiautomatically using critical and ordered sections.

Critical sections

The critical section example shown in “Critical sections” on page 148 isolates a single critical section in a loop, so the `critical_section` directive does not require a gate. In this case, the critical section directives automate allocation, locking, unlocking and deallocation of the needed gate. Multiple dependences and dependences in manually-defined parallel tasks can be handled when user-defined gates are used with the directives.

Consider the following Fortran example:

```
      REAL GLOBAL_SUM
C$DIR FAR_SHARED(GLOBAL_SUM)
C$DIR GATE(SUM_GATE)
      .
      .
      .
      LOCK = ALLOC_GATE(SUM_GATE)
C$DIR BEGIN_TASKS
      CONTRIB1 = 0.0
      DO J = 1, M
        CONTRIB1 = CONTRIB1 + FUNC1(J)
      ENDDO
      .
      .
      .
C$DIR CRITICAL_SECTION (SUM_GATE)
      GLOBAL_SUM = GLOBAL_SUM + CONTRIB1
C$DIR END_CRITICAL_SECTION
      .
      .
      .
```

```
C$DIR NEXT_TASK
      CONTRIB2 = 0.0
      DO I = 1, N
        CONTRIB2 = CONTRIB2 + FUNC2(J)
      ENDDO
      .
      .
      .
C$DIR CRITICAL_SECTION (SUM_GATE)
      GLOBAL_SUM = GLOBAL_SUM + CONTRIB2
C$DIR END_CRITICAL_SECTION
      .
      .
      .
C$DIR END_TASKS
      LOCK = FREE_GATE(SUM_GATE)
```

Here, both parallel tasks must access the shared GLOBAL_SUM variable, which is assigned a function of itself. To ensure that GLOBAL_SUM is only updated by one task at a time, it is placed in a critical section. The critical sections both reference the SUM_GATE variable; this variable is unlocked on entry into the parallel code (gates are always unlocked when they are allocated). When one task reaches the critical section, the CRITICAL_SECTION directive automatically locks SUM_GATE. The END_CRITICAL_SECTION directive unlocks SUM_GATE on exit from the section. Because access to both critical sections is controlled by a single gate, the sections must execute one at a time.

An analogous C example follows:

```
static far_shared float global_sum;
static gate_t sum_gate;
.
.
.
lock = alloc_gate(&sum_gate);
#pragma _CNX begin_tasks
contrib1 = 0.0;
for(j=0;j<m;j++)
  contrib1 = contrib1 + func1(j);
.
.
.
#pragma _CNX critical_section(sum_gate)
global_sum = global_sum + contrib1;
#pragma _CNX end_critical_section
.
.
.
#pragma _CNX next_task
contrib2 = 0.0;
for(i=0;i<n;i++)
  contrib2 = contrib2 + func2(j);
.
.
.

#pragma _CNX critical_section(sum_gate)
global_sum = global_sum + contrib2;
#pragma _CNX end_critical_section
.
.
.
#pragma _CNX end_tasks
lock = free_gate(&sum_gate);
```

Gated critical sections are also useful in loops containing multiple critical sections when there are dependences between the critical sections. If no dependences exist between the sections, gates are not needed, as the compiler will automatically supply a unique gate for every critical section lacking a gate.

Consider the following Fortran example:

```
      REAL ABSUM
C$DIR FAR_SHARED(ABSUM)
C$DIR GATE(GATE1)
      LOGICAL ADJB(100)
      .
      .
      .
      LOCK = ALLOC_GATE(GATE1)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        A(I) = B(I) + C(I)
C$DIR   CRITICAL_SECTION(GATE1)
        ABSUM = ABSUM + A(I)
C$DIR   END_CRITICAL_SECTION
        IF(ADJB(I)) THEN
          B(I) = C(I) + D(I)
C$DIR     CRITICAL_SECTION(GATE1)
          ABSUM = ABSUM + B(I)
C$DIR     END_CRITICAL_SECTION
        ENDIF
        .
        .
        .
      ENDDO
      LOCK = FREE_GATE(GATE1)
```

Here, the shared variable ABSUM must be updated after A(I) is assigned and again if B(I) is assigned. Access to ABSUM must be guarded by the same gate to ensure that two threads do not attempt to update it at once. The critical sections protecting the assignment to ABSUM must explicitly name this gate, or the compiler will choose unique gates for each section, potentially resulting in incorrect answers. Note that there must be a substantial amount of parallelizable code outside of these critical sections to make parallelizing this loop cost-effective.

An analogous C example follows:

```
static far_shared float absum;
static gate_t gate1;
int adjb[...];
.
.
.
lock = alloc_gate(&gate1);
#pragma _CNX loop_parallel(ivar=i)
for(i=0;i<n;i++) {
  a[i] = b[i] + c[i];
#pragma _CNX critical_section(gate1)
  absum = absum + a[i];
#pragma _CNX end_critical_section
  if(adjb[i]) {
    b[i] = c[i] + d[i];
#pragma _CNX critical_section(gate1)
    absum = absum + b[i];
#pragma _CNX end_critical_section
  }
  .
  .
  .
}
lock = free_gate(&gate1);
```

Ordered sections

Like critical sections, ordered sections lock and unlock a specified gate to isolate a section of code in a loop. However, they also ensure that the enclosed section of code executes in the same order as the iterations of the ordered parallel loop that contains it. Once a given thread passes through an ordered section, it cannot enter again until all other threads have passed through in order. This ordering is difficult to implement without using the ordered section directives or pragmas.

NOTE You must use a `loop_parallel(ordered)` directive or pragma to parallelize any loop containing an ordered section.

Consider the following Fortran code, which contains a backward loop-carried dependence on the array A that would normally inhibit parallelization.

```
DO I = 2, N
  . ! PARALLELIZABLE CODE...
  .
  .
  A(I) = A(I-1) + B(I)
  . ! MORE PARALLELIZABLE CODE...
  .
  .
ENDDO
```

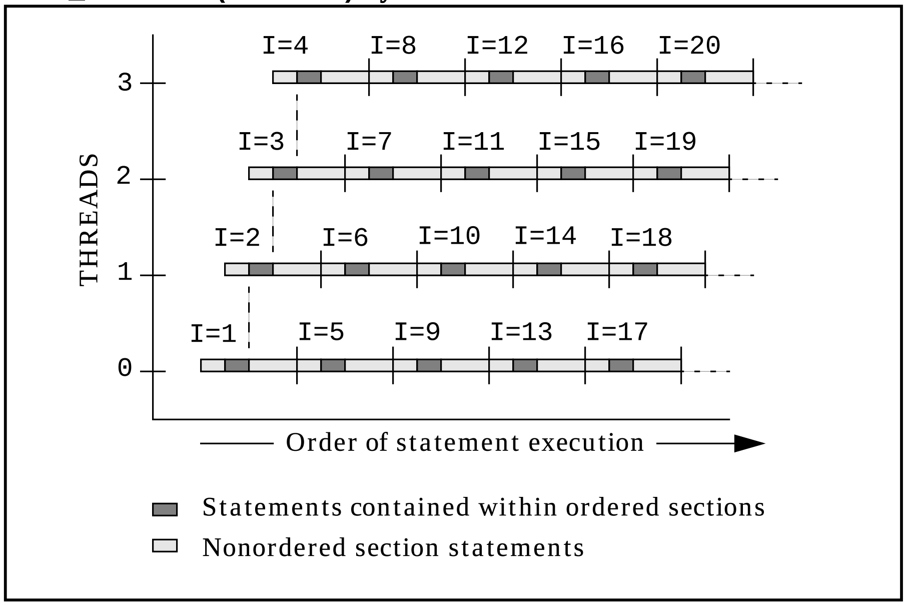
To simplify illustration, we will use only Fortran, but an analogous C example could similarly be parallelized.

Assuming that the dependence shown is the only one in the loop, and that a significant amount of parallel code exists elsewhere in the loop, we can isolate the dependence and parallelize the loop as shown in the following example:

```
C$DIR GATE(LCD)
      LOCK = ALLOC_GATE(LCD)
      .
      .
      .
      LOCK = UNLOCK_GATE(LCD)
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 2, N
        . ! PARALLELIZABLE CODE...
        .
        .
C$DIR   ORDERED_SECTION(LCD)
        A(I) = A(I-1) + B(I)
C$DIR   END_ORDERED_SECTION
        . ! MORE PARALLELIZABLE CODE...
        .
        .
      ENDDO
      LOCK = FREE_GATE(LCD)
```

The loop is now parallelized in the manner described in the section “`loop_parallel(ordered)`” on page 233 and the ordered section containing the `A(I)` assignment will execute in iteration order, ensuring that the value of `A(I-1)` used in the assignment is always valid. Assuming this loop runs on 4 threads, the synchronization of statement execution between threads is illustrated in Figure 24.

Figure 24 **`LOOP_PARALLEL(ORDERED)` synchronization**


I=4
I=8
I=12
I=16
I=20
I=3
I=7
I=11
I=15
I=19
I=2
I=6
I=10
I=14
I=18
I=1
I=5
I=9
I=13
I=17
THREADS
3
2
1
0
Order of statement execution
Statements contained within ordered sections
Nonordered section statements

As shown by the dashed lines between initial iterations for each thread, one ordered section must be done before the next is allowed to begin execution. Once a thread exits an ordered section, it cannot reenter it until all other threads have passed through in sequence. Overlap of nonordered statements, represented as lightly shaded boxes, allows all threads to proceed fully loaded, with only brief idle periods on 1, 2, and 3 at the beginning of the loop, and on 0, 1, and 2 at the end.

Limitations

Each thread in a parallel loop containing an ordered section must pass through the ordered section exactly once on every iteration of the loop. If you execute an ordered section conditionally, you must execute it in all possible branches of the condition; if the code contained in the section is not valid for some branches, you can insert a blank ordered section, as shown in the following Fortran example:

```
C$DIR GATE (LCD)
      .
      .
      .
      LOCK = ALLOC_GATE(LCD)
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, N
        .
        .
        .
        IF (Z(I) .GT. 0.0) THEN
C$DIR     ORDERED_SECTION(LCD)
C         HERE’S THE BACKWARD LCD:
          A(I) = A(I-1) + B(I)
C$DIR     END_ORDERED_SECTION
        ELSE
C         HERE IS THE BLANK ORDERED SECTION:
C$DIR     ORDERED_SECTION(LCD)
C$DIR     END_ORDERED_SECTION
        ENDIF
        .
        .
        .
      ENDDO
      LOCK = FREE_GATE(LCD)
```

Here, no matter which path through the IF statement the loop takes, it must pass through the ordered section, even though the ELSE section is empty. This allows the compiler to properly synchronize the ordered loop. Note that again, we assume a substantial amount of parallel code exists outside the ordered sections, to offset the synchronization overhead.

An analogous C example follows:

```
static gate_t lcd;
.
.
.
lock = alloc_gate(&lcd);
#pragma _CNX loop_parallel(ordered,ivar=i)
for(i=0;i<n;i++) {
  .
  .
  .
  if(z[i] > 0.0) {
#pragma _CNX ordered_section(lcd)
    a[i] = a[i-1] + b[i]; /* backward lcd */
#pragma _CNX end_ordered_section
  } else {
#pragma _CNX ordered_section(lcd)
    /* here is the blank ordered section */
#pragma _CNX end_ordered_section
  }
  .
  .
  .
}
lock = free_gate(&lcd);
```

Ordered sections within nested loops can create similar, but more difficult to recognize, problems. Consider the following Fortran example (gate manipulation is omitted for brevity):

```
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, 99
        DO J = 1,M
          .
          .
          .
C$DIR     ORDERED_SECTION(ORDGATE)
          A(I,J) = A(I+1,J)
C$DIR     END_ORDERED_SECTION
          .
          .
          .
        ENDDO
      ENDDO
```

Recall that once a given thread has passed through an ordered section, it cannot reenter it until all other threads have passed through in order. This is only possible in the given example if the number of available threads integrally divides 99 (the `I` loop limit). If not, deadlock results.

To see why, assume 6 threads, numbered 0 through 5, are running the parallel `I` loop. For `I` = 1, `J` = 1, thread 0 passes through the ordered section and loops back through `J`, stopping when it reaches the ordered section again for `I` = 1, `J` = 2. It cannot enter until threads 1 through 5 (which are executing `I` = 2 through 6, `J` = 1 respectively) pass through in sequence. This is not a problem, and the loop proceeds through `I` = 96 in this fashion in parallel. However, for `I` > 96, all 6 threads are no longer needed. In a single loop nest this would not pose a problem; the leftover 3 iterations would be handled by threads 0 through 2; when thread 2 exited the ordered section it would hit the `ENDDO` and the `I` loop would terminate normally. But in this example, the `J` loop isolates the ordered section from the `I` loop, so thread 0 executes `J` = 1 for `I` = 97, loops through `J` and waits during `J` = 2 at the ordered section for thread 5, which has gone idle, to complete. Threads 1 and 2 similarly execute `J` = 1 for `I` = 98 and `I` = 99, and similarly wait after incrementing `J` to 2. The entire `J` loop must terminate before the `I` loop can terminate, but the `J` loop can never terminate because the idle threads 3, 4, and 5 never pass through the ordered section. Deadlock results.

The analogous C code looks like this:

```
#pragma _CNX loop_parallel(ordered,ivar=i)
for(i=0;i<99;i++) {
  for(j=0;j<m;j++) {
    .
    .
    .
#pragma _CNX ordered_section(ordgate)
    a[i][j] = a[i+1][j];
#pragma _CNX end_ordered_section
    .
    .
    .
  }
}
```

To handle this problem, you can expand the ordered section to include the entire `J` loop, as shown in the following Fortran example:

```
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, 99
C$DIR   ORDERED_SECTION(ORDGATE)
        DO J = 1,M
          .
          .
          .
          A(I,J) = A(I+1,J)
          .
          .
          .
        ENDDO
C$DIR   END_ORDERED_SECTION
      ENDDO
```

In this approach, each thread executes the entire `J` loop each time it enters the ordered section, allowing the `I` loop to terminate normally regardless of the number of threads available.

The analogous C code follows:

```
#pragma _CNX loop_parallel(ordered,ivar=i)
for(i=0;i<99;i++) {
#pragma _CNX ordered_section(ordgate)
  for(j=0;j<m;j++) {
    .
    .
    .
    a[i][j] = a[i+1][j];
    .
    .
    .
  }
#pragma _CNX end_ordered_section
}
```

Another approach is to manually interchange the I and J loops, as shown in the following example:

```
      DO J = 1, M
        LOCK = UNLOCK_GATE(ORDGATE)
C$DIR   LOOP_PARALLEL(ORDERED)
        DO I = 1, 99
          .
          .
          .
C$DIR     ORDERED_SECTION(ORDGATE)
          A(I,J) = A(I+1,J)
C$DIR     END_ORDERED_SECTION
          .
          .
          .
        ENDDO
      ENDDO
```

Here, the I loop is parallelized on every iteration of the J loop. Again, the ordered section is not isolated from its parent loop, so the loop can terminate normally. This example has added benefits; elements of A are accessed more efficiently, and this nest can be further optimized by parallelizing the J loop across hypernodes by using a LOOP_PARALLEL(NODES) directive.

The analogous C code follows:

```
for(j=0;j<99;j++) {
lock = unlock_gate(ordgate);
#pragma _CNX loop_parallel(ordered,ivar=i)
  for(i=0;i<m;i++) {
    .
    .
    .
#pragma _CNX ordered_section(ordgate)
    a[i][j] = a[i+1][j];
#pragma _CNX end_ordered_section
    .
    .
    .
  }
}
```

Manual synchronization

Ordered and critical sections allow you to isolate dependences in a structured, semiautomatic manner. The same isolation can be accomplished manually using the functions discussed in the "Synchronization functions" section of this chapter.

Recall our simple critical section example:

```
C$DIR LOOP_PARALLEL
      DO I = 1, N  ! LOOP IS PARALLELIZABLE
        .
        .
        .
C$DIR   CRITICAL_SECTION
        SUM = SUM + X(I)
C$DIR   END_CRITICAL_SECTION
        .
        .
        .
      ENDDO
```

As shown, this example is easily implemented using critical sections. It can be manually implemented in Fortran as shown below.

```
C$DIR GATE(CRITSEC)
      .
      .
      .
      LOCK = ALLOC_GATE(CRITSEC)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        .
        .
        .
        LOCK = LOCK_GATE(CRITSEC)
        SUM = SUM + X(I)
        LOCK = UNLOCK_GATE(CRITSEC)
        .
        .
        .
      ENDDO
      LOCK = FREE_GATE(CRITSEC)
```

As shown, the manual implementation requires declaring, allocating and deallocating a gate, which must be locked on entry into the critical section using the `LOCK_GATE` function and unlocked on exit using `UNLOCK_GATE`.

An analogous manual C implementation follows:

```
static gate_t critsec;
.
.
.
lock = alloc_gate(&critsec);
#pragma _CNX loop_parallel(ivar=i)
for(i=0;i<n;i++) {
  .
  .
  .
  lock = lock_gate(&critsec);
  sum = sum + x[i];
  lock = unlock_gate(&critsec);
  .
  .
  .
}
lock = free_gate(&critsec);
```

Using synchronization functions to implement critical and ordered sections generally requires more work, and the compiler will not check such constructs for errors as thoroughly as it will check directive-delimited sections. However, because the functions are unstructured, they can be used to create more complex critical regions than are supported by the directives.

Consider the following Fortran example:

```
C$DIR GATE(TASK1, TASK2)
      .
      .
      .
      LOK1 = ALLOC_GATE(TASK1)
      LOK2 = ALLOC_GATE(TASK2)
      LOK1 = LOCK_GATE (TASK1)  ! LOCKING HERE PREVENTS 3RD TASK
      LOK2 = LOCK_GATE (TASK2)  ! FROM ACCESSING A OR B BEFORE
                                ! FIRST TWO TASKS ASSIGN THEM
C$DIR BEGIN_TASKS(ORDERED)  ! TASK ONE:
      DO I = 1, N
        A(I) = 2.0 * A(I) + C(I) ! A GETS ASSIGNED IN THIS TASK
      ENDDO
      LOK1 = UNLOCK_GATE (TASK1) ! A CAN NOW BE ACCESSED
C$DIR NEXT_TASK   ! TASK TWO:
      DO J = 1, N
        B(J) = C(J) * SIN (X(J)) ! B GETS ASSIGNED IN THIS TASK
      ENDDO
      LOK2 = UNLOCK_GATE (TASK2) ! B CAN NOW BE ACCESSED
C$DIR NEXT_TASK    ! TASK THREE:
      DO K = 1, N  ! COMPUTE P IN PARALLEL WITH A AND B:
        P(K) = EXP (3.0 * SQRT (Q(K))) / ATAN (R(K))
      ENDDO
      LOK1 = LOCK_GATE (TASK1)    ! WAIT FOR UNLOCK IN TASK 1
      LOK1 = UNLOCK_GATE (TASK1)  ! WHEN LOCK IS ATTAINED,UNLOCK
      LOK2 = LOCK_GATE (TASK2)    ! WAIT FOR UNLOCK IN TASK 2
      LOK2 = UNLOCK_GATE (TASK2)  ! WHEN ATTAINED, UNLOCK
      DO L = 1, N                 ! WHEN WE HAVE BOTH LOCKS,
        D(L) = P(L) * B(L) / A(L) ! COMPUTE D
      ENDDO
C$DIR NEXT_TASK   ! TASK FOUR:
      DO M = 1, N                 ! NO DEPENDENCES IN THIS TASK
        Y(M) = X(M) * Y(M) / Z(M) ! Y CAN BE COMPUTED WITH
      ENDDO                       ! A, B, P, D
C$DIR END_TASKS
      LOK1 = FREE_GATE(TASK1)
      LOK2 = FREE_GATE(TASK2)
```

Here, the `BEGIN_TASKS(ORDERED)` directive guarantees that the following parallel tasks begin in lexical order (ending order is indeterminate). Ordered sections, however, cannot be used with parallel tasks. To order the dependence in task 3 of this code, where the computation of D assumes that A, B and P are fully computed, we must use manually locked and unlocked gates. A task that is waiting on another task must lexically follow the task it is waiting on.

Tasks 1, 2, 4 and the K loop of task 3 can all run in parallel. To allow this while postponing execution of the L loop in task 3 until A, B and P are computed, we allocate and lock two gates, named TASK1 and TASK2, before the parallel tasks begin. TASK1 remains locked until A is computed, at which point it is unlocked; TASK2 remains locked until B is computed, and is then unlocked. The I and J loops are free to run in parallel, along with the K loop in task 3 and the M loop in task 4, because none of these loops depend on each other's gates. The L loop in task 3, however, cannot begin execution until task 3 can lock both TASK1 and TASK2 gates. Task 3 immediately unlocks these gates because it does not need them; their purpose was to force it to wait until A, B and P were computed. Once task 3 has acquired and relinquished both locks, the L loop is free to run.

A similar C example follows:

```
static gate_t task1, task2;
.
.
.
lok1 = alloc_gate(&task1);
lok2 = alloc_gate(&task2);
lok1 = lock_gate(&task1); /* locking here prevents 3rd task   */
lok2 = lock_gate(&task2); /* from accessing a or b before     */
                          /* first two tasks assign them      */
#pragma _CNX begin_tasks(ordered) /* task 1: */
for(i=0;i<n;i++)
  a[i] = 2.0 * a[i] + c[i];  /* a gets assigned in this task */
lok1 = unlock_gate(&task1);  /* a can now be accessed        */
#pragma _CNX next_task            /* task 2: */
for(j=0;j<n;j++)
  b[j] = c[j] * sin(x[j]);   /* b gets assigned in this task */
lok2 = unlock_gate(&task2);   /* b can now be accessed        */
#pragma _CNX next_task            /* task 3: */
for(k=0;k<n;k++)        /* compute p in parallel with a and b */
  p[k] = exp(3.0*sqrt(q[k]))/atan(r[k]);
lok1 = lock_gate(&task1);   /* wait for unlock in task 1      */
lok1 = unlock_gate(&task1); /* when lock is attained, unlock  */
lok2 = lock_gate(&task2);   /* wait for unlock in task 2      */
lok2 = unlock_gate(&task2); /* when attained, unlock          */
for(l=0;l<n;l++)            /* when we have both locks,       */
  d[l] = p[l] * b[l]/a[l];  /* compute d                      */
#pragma _CNX next_task            /* task 4: */
for(m=0;m<n;m++)            /* no dependences in this task   */
  y[m] = x[m] * y[m]/z[m];  /* y can be computed with a,b,p,d */
#pragma _CNX end_tasks
lok1 = free_gate(&task1);
lok2 = free_gate(&task2);
```

Another advantage of manually defined critical sections is the ability to conditionally lock them. This allows the task that wishes to execute the section to proceed with other work if the lock cannot be acquired. This construct is useful, for example, in situations where one thread is performing I/O for several other parallel threads; while a processing thread is reading from the input queue, the queue is locked, and the I/O thread can move on to do output. While a processing thread is writing to the output queue, the I/O thread can do input. This allows the I/O thread to keep as busy as possible while the parallel computational threads execute their (presumably large) computational code. This situation is illustrated in the following Fortran 90 example. Task 1 performs I/O for the 7 other tasks, which perform parallel computations by calling the THREAD_WRK subroutine.

```
      COMMON INGATE,OUTGATE,COMPBAR
C$DIR GATE (INGATE, OUTGATE)
C$DIR BARRIER (COMPBAR)
      REAL DIN(:), DOUT(:)      ! I/O BUFFERS FOR TASK 1
      ALLOCATABLE DIN, DOUT     ! THREAD 0 WILL ALLOCATE
      REAL QIN(1000,1000), QOUT(1000,1000) ! SHARED I/O QUEUES
      INTEGER NIN/0/,NOUT/0/ ! QUEUE ENTRY COUNTERS
C     CIRCULAR BUFFER POINTERS:
      INTEGER IN_QIN/1/,OUT_QIN/1/,IN_QOUT/1/,OUT_QOUT/1/
        COMMON /DONE/ DONEIN, DONECOMP
      LOGICAL DONECOMP, DONEIN
C                           SIGNALS FOR COMPUTATION DONE AND INPUT DONE
      LOGICAL COMPDONE, INDONE
C                           FUNCTIONS TO RETURN DONECOMP AND DONEIN
      LOGICAL INFLAG, OUTFLAG  ! INPUT READ AND OUTPUT WRITE FLAGS
C$DIR THREAD_PRIVATE (INFLAG,OUTFLAG) ! ONLY NEEDED BY TASK 1
C                                        (WHICH RUNS ON THREAD 0)
        IF (NUM_THREADS() .LT. 8) STOP 1
        IN = 10
        OUT = 11
      LOCK = ALLOC_GATE(INGATE)
      LOCK = ALLOC_GATE(OUTGATE)
      IBAR = ALLOC_BARRIER(COMPBAR)
      DONECOMP = .FALSE.
```

```
C$DIR BEGIN_TASKS             ! TASK 1 STARTS HERE
      INFLAG = .TRUE.
      DONEIN = .FALSE.
      ALLOCATE(DIN(1000),DOUT(1000)) ! ALLOCATE LOCAL BUFFERS
      DO WHILE(.NOT. INDONE() .OR. .NOT. COMPDONE() .OR. NOUT .GT. 0)
C                       DO TILL EOF AND COMPUTATION DONE AND OUTPUT DONE
        IF(NIN.LT.1000.AND.(.NOT.COMPDONE()) .AND.(.NOT. INDONE())) THEN
C                       FILL QUEUE
          IF (INFLAG) THEN  ! FILL BUFFER FIRST:
            READ(IN, IOSTAT = IOS) DIN  ! READ A RECORD; QUIT ON EOF
            IF(IOS .EQ. -1) THEN
              DONEIN = .TRUE. ! SIGNAL THAT INPUT IS DONE
              INFLAG = .TRUE.
            ELSE
              INFLAG = .FALSE.
            ENDIF
          ENDIF
C SYNCHRONOUSLY ENTER INTO INPUT QUEUE:
C         BLOCK QUEUE ACCESS WITH INGATE:
          IF (COND_LOCK_GATE(INGATE) .EQ. 0 .AND. .NOT. INDONE()) THEN
            QIN(:,IN_QIN) = DIN(:)  ! COPY INPUT BUFFER INTO QIN
            IN_QIN=1+MOD(IN_QIN,1000)  ! INCREMENT INPUT BUFFER PTR
            NIN = NIN + 1  ! INCREMENT INPUT QUEUE ENTRY COUNTER
            INFLAG = .TRUE.
            LOCK = UNLOCK_GATE(INGATE)  ! ALLOW INPUT QUEUE ACCESS
          ENDIF
        ENDIF
C SYNCHRONOUSLY REMOVE FROM OUTPUT QUEUE:
C         BLOCK QUEUE ACCESS WITH OUTGATE:
          IF (COND_LOCK_GATE(OUTGATE) .EQ. 0) THEN
           IF (NOUT .GT. 0) THEN
            DOUT(:)=QOUT(:,OUT_QOUT)  ! COPY OUTPUT QUE INTO BUFFR
            OUT_QOUT=1+MOD(OUT_QOUT,1000)
C                                      INCREMENT OUTPUT BUFR PTR
            NOUT = NOUT - 1  ! DECREMENT OUTPUT QUEUE ENTRY COUNTR
            OUTFLAG = .TRUE.
           ELSE
            OUTFLAG = .FALSE.
           ENDIF
            LOCK = UNLOCK_GATE(OUTGATE)
C                                ALLOW OUTPUT QUEUE ACCESS
            IF (OUTFLAG) WRITE(OUT) DOUT  ! WRITE A RECORD
          ENDIF
      ENDDO
C                               TASK 1 ENDS HERE
```

```
C$DIR NEXT_TASK                ! TASK 2:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                ! TASK 3:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                ! TASK 4:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                ! TASK 5:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                ! TASK 6:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                ! TASK 7:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                ! TASK 8:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
      DONECOMP = .TRUE.
C$DIR END_TASKS
      END
```

Before looking at the `THREAD_WRK` subroutine we will examine these parallel tasks, particularly task 1, the I/O server.

Task 1 does all the I/O required by all the tasks; the other 7 tasks do computational work, receiving their input from and sending their output to task 1’s queues. Conditionally locked gates control task 1’s access to one section of code that fills the input queue and one that empties the output queue. Task 1 works by first filling an input buffer; the code that does this does not require gate protection because no other tasks attempt to access the input buffer array. The section of code where the input buffer is copied into the input queue, however, must be protected by gates to prevent any threads from trying to read the input queue while it is being filled.

Similarly, if a task acquires a lock on the input queue, task 1 cannot fill it until the task is done reading from it. When task 1 cannot get a lock to access the input queue code, it goes on and tries to lock the output queue code. If it gets a lock here, it can copy the output queue into the output buffer array and relinquish the lock; it can then proceed to empty the output buffer. If another task is writing to the output queue, task 1 loops back and begins the entire process over again. When the end of the input file is reached, all computation is complete, and the output queue is empty, task 1 is finished. Note that the task loops on `DONEIN` (via `INDONE()`), which is initially false. When input is exhausted, `DONEIN` is set to true, signalling all tasks that there is no more input.

The `INDONE()` function references `DONEIN`, forcing a memory reference. If `DONEIN` were referenced directly, the compiler might optimize it into a register and consequently not detect a change in its value.

So task 1 has four main jobs to do:

1. Read input into input buffer—no other tasks access the input buffer, so this can be done in parallel regardless of what other tasks are doing, as long as the buffer needs filling.

2. Copy input buffer into input queue—the other tasks read their input from the input queue; therefore it can only be filled when no computational task is reading it. This section of code is protected by the `INGATE` gate. It can run in parallel with the computational portions of other tasks, but only one task can access the input queue at a time.

3. Copy output queue into output buffer—the output queue is where other tasks write their output, so it can only be emptied when no computational task is writing to it. This section of code is protected by the `OUTGATE` gate. It can run in parallel with the computational portions of other tasks, but only one task can access the output queue at a time.

4. Write out output buffer—no other tasks access the output buffer, so this can be done in parallel regardless of what the other tasks are doing.

Now we will look at the subroutine THREAD_WRK, which tasks 2-7 call to perform computations.

```
      SUBROUTINE
     > THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      INTEGER NIN,NOUT
      REAL QIN(1000,1000), QOUT(1000,1000) ! SHARED I/O QUEUES
      INTEGER OUT_QIN, OUT_QOUT
      COMMON INGATE,OUTGATE,COMPBAR
C$DIR GATE(INGATE, OUTGATE)
      REAL WORK(1000)     ! LOCAL THREAD PRIVATE WORK ARRAY
      LOGICAL OUTFLAG, INDONE
      OUTFLAG = .FALSE.
C$DIR THREAD_PRIVATE (WORK) ! EVERY THREAD WILL CREATE A COPY

      DO WHILE(.NOT. INDONE() .OR. NIN.GT.0 .OR. OUTFLAG)
C                                WORK/QOUT EMPTYING LOOP
        IF (.NOT. OUTFLAG) THEN  ! IF NO PENDING OUTPUT
C$DIR CRITICAL_SECTION (INGATE)  ! BLOCK ACCESS TO INPUT QUE
          IF (NIN .GT. 0) THEN  ! MORE WORK TO DO
            WORK(:) = QIN(:,OUT_QIN)
            OUT_QIN = 1 + MOD(OUT_QIN, 1000)
            NIN = NIN - 1
            OUTFLAG = .TRUE.
C                         INDICATE THAT INPUT DATA HAS BEEN RECEIVED
          ENDIF
C$DIR END_CRITICAL_SECTION
         .
         .  ! SIGNIFICANT PARALLEL CODE HERE USING WORK ARRAY
         .
        ENDIF
        IF (OUTFLAG) THEN  ! IF PENDING OUTPUT, MOVE TO OUTPUT QUEUE
C AFTER INPUT QUEUE IS USED IN COMPUTATION, FILL OUTPUT QUEUE:
C$DIR CRITICAL_SECTION (OUTGATE) ! BLOCK ACCESS TO OUTPUT QUEUE
          IF(NOUT.LT.1000) THEN
C                           IF THERE IS ROOM IN THE OUTPUT QUEUE
            QOUT(:,IN_QOUT) = WORK(:) ! COPY WORK INTO OUTPUT QUEUE
            IN_QOUT =1+MOD(IN_QOUT,1000) ! INCREMENT BUFFER PTR
            NOUT = NOUT + 1 ! INCREMENT OUTPUT QUEUE ENTRY COUNTER
            OUTFLAG = .FALSE.  ! INDICATE NO OUTPUT PENDING
          ENDIF
C$DIR END_CRITICAL_SECTION
        ENDIF
      ENDDO ! END WORK/QOUT EMPTYING LOOP
      END   ! END THREAD_WRK

      LOGICAL FUNCTION INDONE()
C THIS FUNCTION FORCES A MEMORY REFERENCE TO GET THE DONEIN VALUE
      LOGICAL DONEIN
      COMMON /DONE/ DONEIN, DONECOMP
      INDONE = DONEIN
      END

      LOGICAL FUNCTION COMPDONE()
C THIS FUNCTION FORCES A MEMORY REFERENCE TO GET THE DONECOMP VALUE
      LOGICAL DONECOMP
      COMMON /DONE/ DONEIN, DONECOMP
      COMPDONE= DONECOMP
      END
```

Notice that the gates are accessed through `COMMON` blocks, and that each thread that calls this subroutine will allocate a `thread_private WORK` array.

This subroutine contains a loop that tests `INDONE()`. The loop copies the input queue into the local `WORK` array, then does a presumably significant amount of computational work that has been omitted for simplicity (because the computational work is the main code that executes in parallel, if there is not a large amount of it, the overhead of setting up these parallel tasks and critical sections cannot be justified). The loop encompasses this computation, and also the section of code that copies the `WORK` array to the output queue. This construct allows final output to be written after all input has been used in computation.

To avoid accessing the input queue while it is being filled or accessed by another thread, the section of code that copies it into the local `WORK` array is protected by a critical section. This section must be unconditionally locked; the computational threads cannot do something else until they receive their input. Once the input queue has been copied, `THREAD_WRK` can perform its large section of computational code in parallel with whatever the other tasks are doing. After the computational section is finished, another unconditional critical section must be entered so that the results can be written to the output queue. This prevents two threads from accessing the output queue at once.

Problems like this require performance testing and tuning to achieve optimal parallel efficiency. Variables such as the number of computational threads and the size of the I/O queues can be adjusted with experience to yield the best processor utilization.

7 Message-passing programming

This chapter presents a high-level overview of message passing using MPI and lists sources for more information.

A brief overview of the message-passing and shared-memory paradigms is given in Chapter 1, “Introduction.”

Overview of message passing

Message passing is an approach to writing portable parallel programs. An application that uses message passing consists of several concurrent tasks, each with its own data and memory, using messages to communicate with one another. Message passing requires the programmer to explicitly handle all parallelism and data distribution.

Message-passing programs are inherently parallel, and unless explicitly coordinated by waiting for messages, all processes execute independently. In a conventionally coded message-passing program, all variables are private to each process. So, regardless of whether variables have been declared to be in any of the memory classes, no process can access the variables of any other process. Synchronization among the processes occurs explicitly through message passing.

Approaches to parallelism

Message-passing programs generally take one of two approaches to parallelism: the multiple-program multiple-data (MPMD) approach (also known as the manager/worker approach) or the single-program multiple-data (SPMD) approach.

With MPMD, a set of computational worker processes perform work for one or more manager processes. This method is generally used when little synchronization is required between worker processes.

With SPMD, the program spawns several identical processes that perform the same work independently on different data sets. With this method, synchronization is often required between processes. Exemplar systems are especially suited to this model because of their fast shared-memory communication, which minimizes synchronization delays.

Message passing on Exemplar systems

A Hewlett-Packard implementation of the MPI message-passing library, known as HP MPI, has been developed specifically for HP-UX systems. Special compilation utilities are provided with HP MPI for compiling applications written in Fortran 90, Fortran 77, C, or C++.

HP MPI

HP MPI is an optional product. It is a native implementation of version 1.2 of the Message-Passing Interface (MPI) standard. HP MPI is optimized for HP-UX workstations, servers, and Exemplar systems. It allows you to create and run applications composed of one or more processes that interact using the MPI communication model.

The default location for HP MPI is the /opt/mpi directory. Example programs are available in the /opt/mpi/help directory. For information on specific utilities, refer to the man page for the utility in question. Man pages are stored in the directory /opt/mpi/share/man. For more information on using HP MPI, refer to the *HP MPI User's Guide* (B6011-90001).

Message-passing programming vs. shared-memory programming

The message-passing and shared-memory programming paradigms each have advantages and disadvantages. For example, programs that use message passing can be easily ported between architectures. In the Exemplar V2200 and X2000 architectures, these programs benefit from low-latency interconnects that guarantee minimal overhead in parallel synchronization and data distribution. However, message-passing code generally requires more software overhead than parallel shared-memory code.

The shared-memory style of programming is convenient because the compiler automatically optimizes computations that can be safely parallelized. In addition, the compiler allows you to manually parallelize (by using high-level directives, pragmas, CPSlib functions, or Pthreads functions) those computations that the compiler cannot parallelize automatically. Also, the compiler's command-line options give you the ability to enable (or disable) many of the parallel optimizations on an individual basis.

However, shared-memory programming requires you to handle the synchronization in some cases. In addition, although the directives, pragmas, and CPSlib functions allow you to get better performance from your programs, their functionality is not directly portable to other vendors' platforms. Other vendors' compilers will, however, ignore the Exemplar programming model directives and pragmas. Also, the memory classes provided in the shared-memory programming paradigm are unique to the Exemplar programming model.

Given the benefits and drawbacks of the two paradigms, determining whether to use one over the other, or a combination of the two, should be done on a case-by-case basis.

8 Programming conventions for optimal code

This chapter discusses coding tips and common optimization problems you may encounter when developing programs for SMP servers and presents some possible solutions.

Optimization can remove instructions, replace them, and change the order in which they execute. In some cases, improper optimizations can cause unexpected or incorrect results or code that slows down at higher optimization levels. In some cases, user error can cause similar problems in code that contains syntactically-correct constructs or directives that are used improperly. If you encounter any of these problems, look for the following possible causes:

- Aliasing
- False cache line sharing
- Floating-point imprecision (roundoff error)
- Invalid subscripts
- Misused directives and pragmas
- Misused memory classes
- Triangular loops
- Compiler assumptions

NOTE Compilers perform optimizations assuming that the source code being compiled is valid. Optimizations done on source that violates certain ANSI standard rules can cause the compilers to generate incorrect code.

Aliasing in Fortran

As described in the section "Inhibitors of localization" on page 68, an alias is an alternate name for some object. Fortran `EQUIVALENCE` statements, C pointers, and procedure calls in both languages can potentially cause aliasing problems. The examples presented in Chapter 3 concern aliasing problems that occur at optimization levels `+O3` and above. However, code motion can also cause aliasing problems at optimization levels `+O1` and above.

Aliasing in C

Because they frequently use pointers, C programs are especially susceptible to aliasing problems. By default, the optimizer assumes that a pointer can point to any object in the entire application. Thus, any two pointers are potential aliases. The C compiler has two algorithms you can specify in place of the default: an ANSI-C aliasing algorithm and a type-safe algorithm. The ANSI-C algorithm is enabled [disabled] through the `+O[no]ptrs_ansi` option. The type-safe algorithm is enabled [disabled] by specifying the command-line option `+O[no]ptrs_strongly_typed`. The defaults for these options are `+Onoptrs_ansi` and `+Onoptrs_strongly_typed`.

ANSI algorithm

ANSI C provides strict type-checking. Pointers and variables cannot alias with pointers or variables of a different base type. The ANSI C aliasing algorithm may not be safe if your program is not ANSI compliant.

Type-safe algorithm

The type-safe algorithm provides stricter type-checking. This allows the C compiler to use a stricter algorithm that eliminates many potential aliases found by the ANSI algorithm.

Specifying aliasing modes

To specify an aliasing mode, use one of the following options on the C compiler command line:

- `+Optrs_ansi`
- `+Optrs_strongly_typed`

These and other C aliasing options are further discussed in Appendix D, “Optimization options.”

Iteration and stop values

Aliasing a variable in an array subscript can make it unsafe for the compiler to parallelize a loop. Below are several situations that can prevent parallelization.

Using potential aliases as addresses of variables

In the following example, the code passes `&j` to `getval`; `getval` can use that address in any number of ways, including possibly assigning it to `iptr`. (Even though `iptr` is not passed to `getval`, `getval` might still access it as a global variable or through another alias.) This situation makes `j` a potential alias for `*iptr`.

```
void subex(iptr, n, j)
int *iptr, n, j;
{
   n = getval(&j,n);

   for (j--; j<n; j++)
      iptr[j] += 1;
}
```

This potential alias means that `j` and `iptr[j]` might occupy the same memory space for some value of `j`. The assignment to `iptr[j]` on that iteration would also change the value of `j` itself. The possible alteration of `j` prevents the compiler from safely parallelizing the loop. In this case, the Optimization Report says that no induction variable could be found for the loop, and the compiler does not parallelize the loop. (For information on Optimization Reports, see Appendix D, “Optimization Report.”

Avoid taking the address of any variable that will be used as the iteration variable for a loop. To parallelize the loop in subex, use a temporary variable i as shown in the following example:

```
void subex(iptr, n, j)
int *iptr, n, j;
{
   int i;
   n = getval(&j,n);
   i=j;
   for (i--; i<n; i++)
      iptr[i] += 1;
}
```

Using hidden aliases as pointers

In the next example, ialex takes the address of j and assigns it to *ip. Thus, j becomes an alias for *ip and, potentially, for *iptr. Assigned values to iptr[j] within the loop could alter the value of j. As a result, the compiler cannot use j as an induction variable and, without an induction variable, it cannot count the iterations of the loop. When the compiler cannot find the loop's iteration count, the compiler cannot parallelize the loop.

```
int *ip;
void ialex(iptr)
int *iptr;{
  int j;
  *ip = &j;
  for (j=0; j<2048; j++)
     iptr[j] = 107;
}
```

To parallelize this loop, remove the line of code that takes the address of j or introduce a temporary variable.

Using a pointer as a loop counter

Compiling the following function, the compiler finds that `*j` is not an induction variable (because an assignment to `iptr[*j]` could alter the value of `*j` within the loop.) The compiler does not parallelize the loop.

```
void ialex2(iptr, j, n)
int *iptr;
int *j, n;
{
   for (*j=0; *j<n; (*j)++)
      iptr[*j] = 107;
}
```

Again, this problem can be solved by introducing a temporary iteration variable.

Aliasing stop variables

In the following code, the stop variable `n` becomes a possible alias for `*iptr` when `&n` is passed to `foo`. This means that `n` can be altered during the execution of the loop. As a result, the compiler cannot count the number of iterations and cannot parallelize the loop.

```
void salex(int *iptr, int n)
{
   int i;
   foo(&n);
   for (i=0; i < n; i++)
      iptr[i] += iptr[i];
   return;
}
```

To parallelize the affected loop, eliminate the call to `foo`, move the call below the loop (in which case flow-sensitive analysis takes care of the aliasing), or create a temporary variable as shown below:

```
void salex(int *iptr, int n)
{
   int i, tmp;
   foo(&n);
   tmp = n;
   for (i=0; i < tmp; i++)
      iptr[i] += iptr[i];
   return;
}
```

Because `tmp` is not aliased to `iptr`, the loop has a fixed stop value and the compiler parallelizes it.

Global variables

Potential aliases involving global variables cause optimization problems in many programs. The compiler cannot tell whether another function causes a global variable to become aliased.

The following code uses a global variable, `n`, as a stop value. Because `n` may have its address taken and assigned to `ik` outside the scope of the function, `n` must be considered a potential alias for `*ik`. The value of `n`, therefore, can be altered on any iteration of the loop. The compiler cannot determine the stop value and cannot parallelize the loop.

```
int n, *ik;
void foo(int *ik)
{
   int i;

   for (i=0; i<n; i++)
      ik[i]=i;
}
```

Using a temporary local variable solves the problem.

```
int n;
void foo(int *ik)
{
   int i,stop = n;

   for (i=0; i<stop; ++i)
      ik[i]=i;
}
```

If `ik` is a global variable instead of a pointer, the problem does not occur. Global variables do not cause aliasing problems except when pointers are involved. The following code will parallelize:

```
int n, ik[1000];
void foo()
{
   int i;

   for (i=0; i<n; i++)
      ik[i] = i;
}
```

False cache line sharing

False cache line sharing is a form of cache thrashing. It occurs whenever two or more threads in a parallel program are assigning different data items in the same cache line. This section discusses how to avoid false cache line sharing by restructuring the data layout and controlling the distribution of loop iterations among threads. To simplify explanations, only Fortran examples are given.

Consider the following example:

```
REAL*4 A(8)
DO I = 1, 8
  A(I) = ...
  .
  .
  .
ENDDO
```

Imagine eight threads, each executing one of the above iterations. Assume that `A(1)` is on a processor cache line boundary (32-byte boundary for V2200 and X2000 servers) so that all eight elements are in the same cache line. Only one thread at a time can “own” the cache line, so not only is the above loop, in effect, run serially, but every assignment by a thread requires an invalidation of the line in the cache of its previous “owner.” These problems would likely eliminate any benefit of parallelization.

Now consider this example:

```
REAL*4 B(100,100)
DO I = 1, 100
  DO J = 1, 100
    B(I,J) = ...B(I,J-1)...
  ENDDO
ENDDO
```

Imagine 8 threads working on the `I` loop in parallel (the `J` loop cannot be parallelized because of the dependence). Table 11 shows how the array maps to cache lines, assuming that `B(1,1)` is on a cache line boundary. Array entries that fall on cache line boundaries are in shaded cells.

Table 11 Initial mapping of array to cache lines

1, 1	1, 2	**1, 3**	1, 4	. . .	**1, 99**	1,100
2, 1	2, 2	2, 3	2, 4	. . .	2, 99	2,100
3, 1	3, 2	3, 3	3, 4	. . .	3, 99	3,100
4, 1	4, 2	4, 3	4, 4	. . .	4, 99	4,100
5, 1	**5, 2**	5, 3	**5, 4**	. . .	5, 99	**5,100**
6, 1	6, 2	6, 3	6, 4	. . .	6, 99	6,100
7, 1	7, 2	7, 3	7, 4	. . .	7, 99	7,100
8, 1	8, 2	8, 3	8, 4	. . .	8, 99	8,100
9, 1	9, 2	**9, 3**	9, 4	. . .	**9, 99**	9,100
10, 1	10, 2	10, 3	10, 4	. . .	10, 99	10,100
11, 1	11, 2	11, 3	11, 4	. . .	11, 99	11,100
12, 1	12, 2	12, 3	12, 4	. . .	12, 99	12,100
13, 1	**13, 2**	13, 3	**13, 4**	. . .	13, 99	**13, 100**
. . .	. . .	. . .	. . .	. . .	. . .	. . .
97, 1	97, 2	**97, 3**	97, 4	. . .	**97, 99**	97,100
98, 1	98, 2	98, 3	98, 4	. . .	98, 99	98,100
99, 1	99, 2	99, 3	99, 4	. . .	99, 99	99,100
100, 1	100, 2	100, 3	100, 4	. . .	100, 99	100,100

Array entries in shaded cells are on cache line boundaries.

The Exemplar compilers, by default, give each thread about the same number of iterations, assigning (if necessary) one extra iteration to some threads until all iterations are assigned to a thread. Table 12 shows the default distribution of the `I` loop across 8 threads.

Table 12 **Default distribution of the `I` loop**

Thread ID	Iteration range	Number of iterations
0	1-12	12
1	13-25	13
2	26-37	12
3	38-50	13
4	51-62	12
5	63-75	13
6	76-87	12
7	88-100	13

This distribution of iterations causes threads to share cache lines. For example, thread 0 assigns the elements `B(9:12,1)` and thread 1 assigns elements `B(13:16,1)` in the *same* cache line. In fact, every thread shares cache lines with at least one other thread; most share cache lines with two other threads. This type of sharing is called *false* because it is a result of the data layout and the compiler's distribution of iterations; it is *not* inherent in the algorithm itself. Therefore, it can be reduced or even removed by

1. Restructuring the data layout by aligning data on cache line boundaries
2. Controlling the iteration distribution

Aligning data to avoid false sharing

Because false cache line sharing is partially due to the layout of the data, one step in avoiding it is to adjust the layout. Typically, these adjustments are made by aligning data on cache line boundaries. (Aligning arrays generally improves performance; however, it can occasionally decrease performance.) The second step in avoiding false cache line sharing (which is covered in the section, "Distributing iterations on cache line boundaries" on page 280) is to adjust the distribution of loop iterations.

Aligning arrays on cache line boundaries

Note the assumption that in the previous example, array B starts on a cache line boundary. The methods below force arrays in Fortran to start on cache line boundaries:

- Using uninitialized COMMON blocks (blocks with no DATA statements). These blocks start on 64-byte boundaries.
- Using ALLOCATE statements. These statements return addresses on 64-byte boundaries. (Applies only to parallel executables.)
- Using the directive ALIGN_CTI on an X2000 server. This directive forces arrays of any size to start on CTIcache boundaries (32 bytes on X2000 servers).

The methods below force arrays in C to start on cache line boundaries:

- Using the functions malloc or memory_class_malloc. These functions return pointers on 64-byte boundaries. (Applies only to parallel executables.)
- Using uninitialized global arrays or structs that are at least 32 bytes. Such arrays and structs are aligned on 64-byte boundaries.
- Using the pragma align_cti on an X2000 server. This pragma forces arrays of any size to start on CTIcache boundaries (32 bytes on X2000 servers).
- Using uninitialized data of the external storage class in C that is at least 32 bytes. Data is aligned on 64-byte boundaries.

Aligning multidimensional arrays on cache line boundaries

Multidimensional arrays can also be aligned on cache line boundaries. Recall that in the example from the beginning of the section:

```
REAL*4 B(100,100)
DO I = 1, 100
  DO J = 1, 100
    B(I,J) = ...B(I,J-1)...
  ENDDO
ENDDO
```

we assumed B(1,1) starts on a cache line boundary. However, because the iteration distribution caused alignment on cache line boundaries to vary from dimension to dimension, performance suffered. Choose a value *x* for the leftmost dimension (rightmost in C and C++) in arrays so that *x* times the data size (in bytes) is an integral multiple of the CTIcache line size. (The code that follows gives an example of this idea.) Using such a value aligns data on CTIcache line boundaries at the same index points in all dimensions.

On X2000 servers, both the processor cache lines and the CTIcache lines are 32 bytes. (V2200 servers and nonscalable SMPs do not have CTIcaches.)

For general use, try to align everything on 32-byte boundaries for V2200 and X2000 servers. This alignment will help to eliminate false cache line sharing between processor caches and also between CTIcaches where the penalty for false sharing is even higher. Where possible, try to parameterize cache line size in anticipation of future systems that might have different cache line sizes.

Changing the example so that array `B` is aligned and padded (to 64 bytes), the leftmost dimension is now 112 instead of 100. (See the modified example below.) The number 112 is the smallest value *x* greater than 100 such that *x* times the data size (4 bytes) is an integral multiple of 64.

```
REAL*4 B(112,100)
COMMON /ALIGNED/ B
  DO I = 1, 100
    DO J = 1, 100
      B(I,J) = ...B(I,J-1)...
    ENDDO
  ENDDO
```

Note that loop limits have not changed. Placing `B` in a `COMMON` block has forced `B(1,1)` onto a cache line boundary (64 bytes), and changing the first dimension to 112 assures that `B(1,2)`, `B(1,3)`, ..., `B(1,100)` all start on cache line boundaries. Table 13 shows how the restructured array maps to (processor) cache lines.

The next section, “Distributing iterations on cache line boundaries,” explains how to make the compiler distribute iterations so that threads work on whole cache lines.

Table 13 Restructured mapping of array to cache lines

1, 1	1, 2	1, 3	1, 4	. . .	1, 99	1,100
2, 1	2, 2	2, 3	2, 4	. . .	2, 99	2,100
3, 1	3, 2	3, 3	3, 4	. . .	3, 99	3,100
4, 1	4, 2	4, 3	4, 4	. . .	4, 99	4,100
5, 1	5, 2	5, 3	5, 4	. . .	5, 99	5,100
6, 1	6, 2	6, 3	6, 4	. . .	6, 99	6,100
7, 1	7, 2	7, 3	7, 4	. . .	7, 99	7,100
8, 1	8, 2	8, 3	8, 4	. . .	8, 99	8,100
9, 1	9, 2	9, 3	9, 4	. . .	9, 99	9,100
10, 1	10, 2	10, 3	10, 4	. . .	10, 99	10,100
11, 1	11, 2	11, 3	11, 4	. . .	11, 99	11,100
12, 1	12, 2	12, 3	12, 4	. . .	12, 99	12,100
. . .	. . .	. . .	. . .	. . .	. . .	. . .
97, 1	97, 2	97, 3	97, 4	. . .	97, 99	97,100
98, 1	98, 2	98, 3	98, 4	. . .	98, 99	98,100
99, 1	99, 2	99, 3	99, 4	. . .	99, 99	99,100
100, 1	100, 2	100, 3	100, 4	. . .	100, 99	100,100
101, 1	101, 2	101, 3	101, 4	. . .	101, 99	101,101
102, 1	102, 2	102, 3	102, 4	. . .	102, 99	102,102
103, 1	103, 2	103, 3	103, 4	. . .	103, 99	103,103
104, 1	104, 2	104, 3	104, 4	. . .	104, 99	104,104
105, 1	105, 2	105, 3	105, 4	. . .	105, 99	105,105
106, 1	106, 2	106, 3	106, 4	. . .	106, 99	106,106
107, 1	107, 2	107, 3	107, 4	. . .	107, 99	107,107
108, 1	108, 2	108, 3	108, 4	. . .	108, 99	108,108
109, 1	109, 2	109, 3	109, 4	. . .	109, 99	109,109
110, 1	110, 2	110, 3	110, 4	. . .	110, 99	110,110
111, 1	111, 2	111, 3	111, 4	. . .	111, 99	111,112
112, 1	112, 2	112, 3	112, 4	. . .	112, 99	112,112

Array entries in shaded cells are on cache line boundaries.

Distributing iterations on cache line boundaries

Recall that the default iteration distribution causes thread 0 to work on iterations 1-12 and thread 1 to work on iterations 13-25, and so on. Even though the cache lines are aligned across the columns of the array (see Table 13), we still need to change the iteration distribution. Use CHUNK_SIZE to change the distribution:

```
      REAL*4 B(112,100)
      COMMON /ALIGNED/ B
C$DIR PREFER_PARALLEL (CHUNK_SIZE=16)
      DO I = 1, 100
        DO J = 1, 100
          B(I,J) = ...B(I,J-1)...
        ENDDO
      ENDDO
```

You must specify a constant CHUNK_SIZE. However, the ideal would be to distribute work so that all but one thread works on the same number of whole cache lines, and the remaining thread works on any partial cache line. For example, given:

```
NITS   = number of iterations
NTHDS  = number of threads
LSIZE  = CTIcache line size in words (8 for 4-byte data,
          4 for 8-byte data, 2 for 16-byte data on X2000 servers)
```

the ideal CHUNK_SIZE would be:

```
CHUNK_SIZE = LSIZE * (1 + ( (1 + (NITS - 1) / LSIZE ) - 1 )/NTHDS)
```

For the code above, these numbers are:

```
        NITS   = 100
        NTHDS  = 8
        LSIZE  = 8 (aligns on X2000 CTIcache boundaries for
                    4-byte data)
```

```
CHUNK_SIZE = 8 * (1 + ( (1 + (100 - 1) / 8 ) - 1) / 8)
           = 8 * (1 + ( (1 +  12          ) - 1) / 8)
           = 8 * (1 + ( 12                     ) / 8)
           = 8 * (1 + 1                            )
           = 16
```

`CHUNK_SIZE = 16` causes threads 0, 1, ..., 6 to execute iterations 1-16, 17-32, ..., 81-96, respectively. Thread 7 executes iterations 97-100. As a result there is no false cache line sharing, and parallel performance is greatly improved.

While you cannot specify the ideal `CHUNK_SIZE` for every loop, using

`CHUNK_SIZE` = *x*

where *x* times the data size (in bytes) is an integral multiple of 32 will eliminate false cache line sharing if the two conditions below are met:

- The arrays are already properly aligned (as discussed earlier in this section).
- The first iteration accesses the first element of each array being assigned. (For example, in a loop `DO I = 2, N`, because the loop starts at `I = 2`, the first iteration does not access the first element of the array; consequently, the iteration distribution does not match the cache line alignment.)

The number 32 is used because the CTIcache line size is 32 bytes for X2000 servers.

Thread-specific array elements

Sometimes a parallel loop has each thread update a unique element of a shared array which is further processed by thread 0 outside the loop.

Consider the following Fortran example:

```
      REAL*4 S(8)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        .
        .
        .
        S(MY_THREAD()+1) = ... ! EACH THREAD ASSIGNS ONE ELEMENT OF S
        .
        .
        .
      ENDDO
C$DIR NO_PARALLEL
      DO J = 1, NUM_THREADS()
        = ...S(J)              ! THREAD 0 POST-PROCESSES S
      ENDDO
```

The problem here is that potentially all the elements of S are in a single cache line, so the assignments cause false sharing. One approach is to change the code to force the unique elements into different cache lines, as indicated below:

```
      REAL*4 S(8,8)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        .
        .
        .
        S(1,MY_THREAD()+1) = ... ! EACH THREAD ASSIGNS ONE ELEMENT OF S
        .
        .
        .
      ENDDO
C$DIR NO_PARALLEL
      DO J = 1, NUM_THREADS()
        = ...S(1,J)             ! THREAD 0 POST-PROCESSES S
      ENDDO
```

For multihypernode applications on X2000 servers, the dimensions should be `S(8,`*number_of_threads*`)` because the CTIcache line size is 32 bytes, and the data size is 4 bytes.

Scalars sharing a cache line

Sometimes parallel tasks will assign unique scalar variables that are in the same cache line, as in the following example:

```
      COMMON /RESULTS/ SUM, PRODUCT
C$DIR BEGIN_TASKS
      DO I = 1, N
        .
        .
        .
        SUM = SUM + ...
        .
        .
        .
      ENDDO
C$DIR NEXT_TASK
      DO J = 1, M
        .
        .
        .
        PRODUCT = PRODUCT * ...
        .
        .
        .
      ENDDO
C$DIR END_TASKS
```

This problem is similar to the example in the previous section (“Thread-specific array elements”), and can be avoided by padding enough space between the two scalar variables so that the variables are in separate CTIcache lines:

```
COMMON /RESULTS/ SUM, PAD(7), PRODUCT
```

where `PAD(7)` represents 28 bytes—`SUM` and `PAD(7)` together take up 32 bytes, forcing `PRODUCT` into the next CTIcache line on multinode X2000 applications because the CTIcache lines are 32 bytes.

Working with unaligned arrays

The most common cache-thrashing complication using arrays and loops will be that arrays assigned within a loop are unaligned (and possibly unalignable) with each other. There are several possible causes for this:

- Arrays that are local to a routine are allocated on the stack.
- Array dummy arguments might be passed an element other than the first in the actual argument.
- Array elements might be assigned with different offset indexes.

Consider the following Fortran example:

```
COMMON /OKAY/ X(112,100)
        ...
CALL UNALIGNED (X(I,J))
        ...
SUBROUTINE UNALIGNED (Y)
REAL*4 Y(*)
    ! Y(1) PROBABLY NOT ON A CACHE LINE BOUNDARY
```

The address of `Y(1)` is unknown. However, if elements of `Y` are heavily assigned in this routine, it may be worthwhile to compute an alignment, given by the following formula:

`LREM = LSIZE - ( ( MOD ( LOC(Y(1))-4, LSIZE*`*x*`) + 4) /`*x*`)`

where

`LSIZE`	is the appropriate cache line size in words
x	is the data size for elements of `Y`

For this case, assume `LSIZE` is CTIcache line size (32 bytes on X2000 servers) in single precision words (8 words). Note that

`( ( MOD ( LOC(Y(1))-4, LSIZE*4) + 4) /4)`

returns a value in the set 1, 2, 3, ..., `LSIZE`, so `LREM` is in the range 0 to 7.

Then a loop such as:

```
DO I = 1, N
  Y(I) = ...
ENDDO
```

can be transformed to:

```
C$DIR NO_PARALLEL
      DO I = 1, MIN (LREM, N) ! 0 <= LREM < 8
        Y(I) = ...
      ENDDO
C$DIR PREFER_PARALLEL (CHUNK_SIZE = 16)
      DO I = LREM+1, N
         ! Y(LREM+1) IS ON A CACHE LINE BOUNDARY
        Y(I) = ...
      ENDDO
```

The first loop takes care of elements from the first (if any) partial cache line of data. The second loop begins on a cache line boundary, and can be controlled with CHUNK_SIZE to avoid false sharing among the threads.

Working with dependences

Data dependences in loops may prevent parallelization and prevent the elimination of false cache line sharing. If certain conditions are met, some performance gains can be achieved.

For example, consider the following code:

```
COMMON /ALIGNED / P(128,128), Q(128,128), R(128,128)
REAL*4 P, Q, R
DO J =  2, 128
  DO I = 2, 127
    P(I-1,J) = SQRT (P(I-1,J-1) + 1./3.)
    Q(I  ,J) = SQRT (Q(I  ,J-1) + 1./3.)
    R(I+1,J) = SQRT (R(I+1,J-1) + 1./3.)
  ENDDO
ENDDO
```

Only the I loop can be parallelized. (Because of the loop-carried dependences in the J loop, it cannot be parallelized.) It is impossible to distribute the iterations such that there will be no false cache line sharing in the above loop. If all loops that refer to these arrays always use the same offsets (which is unlikely) then you could make dimension adjustments that would allow a better iteration distribution. For example, the following would work well for 8 threads:

```
      COMMON /ADJUSTED/ P(128,128), PAD1(15), Q(128,128),
     > PAD2(15), R(128,128)

      DO J =  2, 128
C$DIR   PREFER_PARALLEL (CHUNK_SIZE=16)
        DO I = 2, 127
          P(I-1,J) = SQRT (P(I-1,J-1) + 1./3.)
          Q(I  ,J) = SQRT (Q(I  ,J-1) + 1./3.)
          R(I+1,J) = SQRT (R(I+1,J-1) + 1./3.)
        ENDDO
      ENDDO
```

Padding 60 bytes before the declarations of both `Q` and `R` causes the `P(1,J)`, `Q(2,J)`, and `R(3,J)` to be aligned on 64-byte boundaries for all `J`. Combined with a `CHUNK_SIZE` of 16, this causes threads to assign data to unique whole cache lines.

Often in real-world code you will find a mix of all the above problems in some CPU-intensive loops. You will not be able to avoid all false cache line sharing, but by careful inspection of the problems and careful application of some of the workarounds shown here, you will usually be able to significantly enhance performance of your parallel loops.

Floating-point imprecision

The compiler applies normal arithmetic rules to real numbers. It assumes that two arithmetically equivalent expressions produce the same numerical result.

Most real numbers cannot be represented exactly in digital computers. Instead, these numbers are rounded to a floating-point value that can be represented. When optimization changes the evaluation order of a floating-point expression, the results can change. Possible consequences of floating-point roundoff include program aborts, division by zero, address errors, and incorrect results.

In any parallel program, the execution order of the instructions will differ from the serial version of the same program. This can cause noticeable roundoff differences between the two versions. Running a parallel code under different machine configurations or conditions can also yield roundoff differences, because the execution order can differ under differing machine conditions, causing roundoff errors to propagate in different orders between executions. Accumulator variables (reductions) are especially susceptible to these problems.

Consider the following Fortran example:

```
C$DIR GATE(ACCUM_LOCK)
      LK = ALLOC_GATE(ACCUM_LOCK)
      .
      .
      .
      LK = UNLOCK_GATE(ACCUM_LOCK)
C$DIR BEGIN_TASKS, TASK_PRIVATE(I)
      CALL COMPUTE(A)
C$DIR CRITICAL_SECTION(ACCUM_LOCK)
      ACCUM = ACCUM + A
C$DIR END_CRITICAL_SECTION
C$DIR NEXT_TASK

      DO I = 1, 10000
        B(I) = FUNC(I)
C$DIR   CRITICAL_SECTION(ACCUM_LOCK)
        ACCUM = ACCUM + B(I)
C$DIR   END_CRITICAL_SECTION
        .
        .
        .
      ENDDO

C$DIR NEXT_TASK
      DO I = 1, 10000
        X = X + C(I) + D(I)
      ENDDO
C$DIR CRITICAL_SECTION(ACCUM_LOCK)
      ACCUM = ACCUM/X
C$DIR END_CRITICAL_SECTION
C$DIR END_TASKS
```

Here, three parallel tasks are all manipulating the real variable ACCUM, using real variables which have themselves been manipulated. Each manipulation is subject to roundoff error, so the total roundoff error here might be substantial. When the program runs in serial, the tasks execute in their written order, and the roundoff errors accumulate in that order. However, if the tasks run in parallel, there is no guarantee as to what order the tasks will run in, meaning the roundoff error will accumulate in a different order than it does during the serial run. Depending on machine conditions, the tasks may run in different orders during different parallel runs also, potentially accumulating roundoff errors differently and yielding different answers.

An analogous C example follows:

```
static gate_t accum_lock;
lk = alloc_gate(&accum_lock);
.
.
.
lk = unlock_gate(&accum_lock);
#pragma _CNX begin_tasks, task_private(i)
compute(a);
#pragma _CNX critical_section(accum_lock)
accum = accum + a;
#pragma _CNX end_critical_section
#pragma _CNX next_task
for(i=0;i<10000;i++) {
  b[i] = func[i];
#pragma _CNX critical_section(accum_lock)
  accum = accum + b[i];
#pragma _CNX end_critical_section
.
.
.
}
#pragma _CNX next_task
for(i=0;i<10000;i++)
  x = x + c[i] + d[i];
#pragma _CNX critical_section(accum_lock)
accum = accum/x;
#pragma _CNX end_critical_section
#pragma _CNX end_tasks
```

Problems with floating-point precision can also occur when a program tests the value of a variable without allowing enough tolerance for roundoff errors. To solve the problem, adjust the tolerances to allow for greater roundoff errors or declare the variables to be of a higher precision (use the `double` type instead of `float` in C and C++, or `REAL*8` rather than `REAL*4` in Fortran). It is always poor practice to test floating point numbers for exact equality.

Enabling sudden underflow

By default, PA-RISC processor hardware represents a floating point number in denormalized format when the number is *tiny*. A floating point number is considered tiny if its exponent field is zero but its mantissa is nonzero (for more information, refer to the *HP-UX Floating-Point Guide*). This practice is extremely costly in terms of execution time and seldom provides any benefit. You can enable sudden underflow (flush to zero) of denormalized values by passing the `+FPD` flag to the linker. This is done using the `-W` compiler option.

The following example shows such an `f90` command line:

```
%f90 -Wl,+FPD prog.f
```

This command line compiles the program `prog.f` and instructs the linker to enable sudden underflow.

Invalid subscripts

An array reference in which *any* subscript falls outside declared bounds for that dimension is called an invalid subscript. Invalid subscripts are a common cause of answers that vary between optimization levels and programs that abort and dump core. Use the command-line option `-C` (check subscripts) with `f90` or `f77` to check that each subscript is within its array bounds. See the f90(1) or f77(1) man page for more information. The C and aC++ compilers do not have an option corresponding to the Fortran compilers’ `-C` option.

Misused directives and pragmas

Misused directives and pragmas are a common cause of wrong answers. Some of the more common misuses of directives and pragmas involve the following:

- Loop-carried dependences
- Reductions
- Nondeterminism of parallel execution
- Hidden ordered sections

Descriptions of, and methods for, avoiding the items listed above are in the sections below.

Loop-carried dependences

For example, forcing parallelization of a loop containing a call is safe only if the called routine contains no dependences.

Do not assume that it is always safe to parallelize a loop whose data is safe to localize. You can safely localize loop data in loops that do not contain a loop-carried dependence (LCD) of the form shown in the following Fortran loop:

```
DO I = 2, M
  DO J = 1, N
    A(I,J) = A(I+IADD,J+JADD) + B(I,J)
  ENDDO
ENDDO
```

where one of `IADD` and `JADD` is negative and the other is positive. This is explained in detail in the section “Inhibitors of localization” on page 68.

You cannot safely parallelize a loop that contains any kind of LCD, except by using ordered sections around the LCDs as described in the section “Ordered sections” on page 242. Also see the section “Inhibitors of parallelization” on page 111.

The `MAIN` section of the Fortran program that follows initializes `A`, calls `CALC`, and outputs the new array values. In subroutine `CALC`, the indirect index used in `A(IN(I))` introduces a potential dependence that prevents the compiler from parallelizing `CALC`’s `I` loop.

```
PROGRAM MAIN
REAL A(1025)
INTEGER IN(1025)
COMMON /DATA/ A
DO I = 1, 1025
  IN(I) = I
ENDDO
CALL CALC(IN)
CALL OUTPUT(A)
END


SUBROUTINE CALC(IN)
INTEGER IN(1025)
REAL A(1025)
COMMON /DATA/ A
DO I = 1, 1025
  A(I) = A(IN(I))
ENDDO
RETURN
END
```

An analogous C example follows:

```
float arra[1025];

void calc(int in[])
{
  int i,j;

    for(i = 0; i < 1025; i++)
      arra[i] = arra[in[i]];
}
main()
{
  int i,j,in[1025];

  for(i = 0; i < 1025; i++)
    in[i] = i;
  calc(in);
  output(arra);
}
```

Because you know that `IN(I) = I`, you can use the `NO_LOOP_DEPENDENCE` directive, as shown below. This directive allows the compiler to ignore the apparent dependence and parallelize the loop, when compiling with `+O3 +Oparallel`.

```
      SUBROUTINE CALC(IN)
      INTEGER IN(1025)
      REAL A(1025)
      COMMON /DATA/ A
C$DIR NO_LOOP_DEPENDENCE(A)
      DO I = 1, 1025
         A(I) = A(IN(I))
      ENDDO
      RETURN
      END
```

In C:

```
void calc(int in[])
{
  int i,j;
  #pragma _CNX no_loop_dependence(arra)
  for(i = 0; i < 1025; i++)
    arra[i] = arra[in[i]];
}
```

Reductions

Reductions are a special class of dependence that the compiler can parallelize. An apparent LCD can prevent the compiler from parallelizing a loop containing a reduction. The loop in the following Fortran example is not parallelized because of an apparent dependence between the references to A(I) on line 6 and the assignment to A(JA(J)) on line 7. The compiler does not realize that the values of the elements of JA never coincide with the values of I, and so, assuming that they might, conservatively avoids parallelizing the loop.

```
DO I = 1,100
  JA(I) = I + 10
ENDDO
DO I = 1, 100
  DO J = I, 100
      A(I) = A(I) + B(J) * C(J)   !LINE 6
      A(JA(J)) = B(J) + C(J)      !LINE 7
  ENDDO
ENDDO
```

NOTE

In this example as well as the examples that follow, the apparent dependence becomes real if any of the values of the elements of JA are equal to the values iterated over by I.

A no_loop_dependence directive or pragma placed before the J loop tells the compiler that the indirect subscript does not cause a true dependence. Because reductions are a form of dependence, this directive also tells the compiler to ignore the reduction on A(I), which it would normally handle. Ignoring this reduction causes the compiler to generate incorrect code for the assignment on line 6; the apparent dependence on line 7 is properly handled because of the directive. The resulting code runs fast but produces incorrect answers.

In the following analogous C example, the apparent dependence is between the reference to `a[i]` on line 5 and `a[ja[j]]` on line 6:

```
for (i=0;i<100;i++)
  ja[i] = i + 10;
for (i=0; i<100; i++)
  for (j=0; j<100; j++) {
    a[i] += b[j] * c[j];    /* line 5 */
    a[ja[j]] = b[j] + c[j]; /* line 6 */
  }
```

To solve this problem, distribute the J loop, isolating the reduction from the other statements, as shown in the following Fortran example:

```
      DO I = 1, 100
        DO J = I, 100
          A(I) = A(I) + B(J) * C(J)
        ENDDO
      ENDDO
C$DIR NO_LOOP_DEPENDENCE(A)
      DO I = 1, 100
        DO J = I, 100
          A(JA(J)) = B(J) + C(J)
        ENDDO
      ENDDO
```

And in C:

```
for (i=0; i<100; i++)
  for (j=i; j<100; j++)
    a[i] += b[j] * c[j];
#pragma _CNX no_loop_dependence(a)
for (i=0; i<100; i++)
  for (j=i; j<100; j++)
    a[ja[j]] = b[j] + c[j];
```

The apparent dependence is removed, and both loops can be optimized.

Nondeterminism of parallel execution

In a parallel program, threads do not execute in a predictable or determined order. If you force the compiler to parallelize a loop when a dependence exists, the results are unpredictable and can vary from one execution to the next.

Consider the following Fortran example:

```
DO I = 1, N-1
  A(I) = A(I+1) * B(I)
  .
  .
  .
ENDDO
```

The compiler will not parallelize this code as written because of the dependence on `A(I)`. This dependence requires that the original value of `A(I+1)` is available for the computation of `A(I)`. If this code was parallelized, some values of A would be assigned by some processors before they were used by others, resulting in incorrect assignments. Because the results depend on the order in which statements execute, the errors are nondeterministic. The loop must therefore execute in iteration order to ensure that all values of A are computed correctly.

The analogous C code follows:

```
for(i=0;i<n-1;i++) {
  a[i] = a[i+1] * b[i];
  .
  .
  .
}
```

Loops containing dependences can sometimes be manually parallelized using the `LOOP_PARALLEL(ORDERED)` directive as described in Chapter 6, “Advanced shared-memory programming.” Otherwise, unless you are sure that no loop-carried dependence exists, it is safest to let the compiler choose which loops to parallelize.

Hidden ordered sections

While it is legal and sometimes useful to place ordered sections in separate routines from their parent ordered loops, this practice can cause runtime deadlock in some situations. Consider the following Fortran example:

```
      PROGRAM SEPMAIN
      REAL A(100)
      .
      .
      .
C$DIR BEGIN_TASKS(NODES)
      CALL SUB1(A)
      .
      .
      .
C$DIR NEXT_TASK
      CALL SUBN
      .
      .
      .
C$DIR END_TASKS
      .
      .
      .
      END

      SUBROUTINE SUB1(A)
      COMMON LOCK
      REAL A(100)
C$DIR GATE(LOCK)
      LK = ALLOC_GATE(LOCK)
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 2, 100
        CALL SUB2(LOCK,A,I)
      ENDDO
      LK = FREE_GATE(LOCK)
      END

      SUBROUTINE SUB2(LOCK,A,I)
C$DIR GATE(LOCK)
      REAL A(100)
      INTEGER I
C$DIR ORDERED_SECTION(LOCK)
      A(I) = A(I-1)
C$DIR END_ORDERED_SECTION
      END
```

The ordered section in SUB2 should be associated with the loop_parallel construct. However, if for any reason that construct does not execute in parallel, the ordered section is then associated with the closest parallel construct; in this case, the begin_tasks construct.

The ordered section in SUB2 expects that it will be executed by all parallel threads. Only thread 0 from the node-level spawn is executing SUB1, because only the first task calls it; thread 0 therefore runs the DO loop in SUB1 and passes through the ordered section in SUB2. After this, the ordered section will wait for thread 1 to enter before allowing thread 0 back in on the next iteration of the I loop. Thread 1 never calls SUB1, so it never has the opportunity to enter the ordered section. This can cause the program to deadlock in the ordered section.

If you encounter this kind of problem, try moving the ordered section into the same routine as its parent loop. Also, you should apply the NO_DYNSEL directive to the loop_parallel loop or use the +Onodynsel command-line option so that it does not run serially.

The analogous C code follows:

```
void sub2(gate_t lock, float *a, int i) {
#pragma _CNX ordered_section(lock)
  a[i] = a[i-1];
#pragma _CNX end_ordered_section
}

void sub1(float *a) {
  static gate_t lock;
  int i, lk;
  lk = alloc_gate(&lock);
#pragma _CNX loop_parallel(ordered, ivar=i)
  for(i=1;i<100;i++)
    sub2(lock,a,i);
  lk = free_gate(&lock);
}

main() {
  float a[100];
  .
  .
  .
#pragma _CNX begin_tasks(nodes)
  sub1(a);
  .
  .
  .
#pragma _CNX next_task
  subn();
  .
  .
  .
#pragma _CNX end_tasks
  .
  .
  .
}
```

Misused memory classes

While manually assigned memory classes can substantially boost performance when coupled with manual parallelization, assigning the wrong memory class to data can cause wrong answers and in some cases degrade performance. This section discusses some common misuses of memory classes.

Improper dynamic allocations

Dynamically allocating `thread_private` memory from serial code can give unexpected results if the memory is later accessed from parallel code.

Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!!
      REAL*8 WRONGTP(:)
C$DIR THREAD_PRIVATE(WRONGTP)
      ALLOCATABLE WRONGTP
      .
      .
      .
C THE FOLLOWING ALLOCATE ONLY ALLOCATES
C WRONGTP(N) FOR THREAD 0:
      ALLOCATE(WRONGTP(N))
C$DIR LOOP_PARALLEL(THREADS, IVAR = I)
C$DIR LOOP_PRIVATE(J)
      DO I = 1, NUM_THREADS()
        DO J = 1, N
          WRONGTP(J) = ... ! ONLY EXISTS FOR
          .                ! THREAD 0
          .
          .
        ENDDO
      ENDDO
```

Here, the array `WRONGTP` is allocated, but because the allocation takes place in serial code, which is run by thread 0, only thread 0 allocates the array. When other threads attempt to access the array in the `J` loop, it does not exist. To fix this, allocate the array inside the thread-parallel `I` loop, as discussed in Chapter 5, “Memory classes.”

An analogous C example follows:

```
/* INCORRECT EXAMPLE FOLLOWS!!!                           */
static thread_private double *wrongtp;
.
.
.
/* the following memory_class_malloc only allocates wrongtp
   for thread 0                                           */
wrongtp=(double *)memory_class_malloc(sizeof(double)*n,
                                        THREAD_PRIVATE_MEM);
#pragma _CNX loop_parallel(threads, ivar=i)
#pragma _CNX loop_private(j)
for(i=0;i<num_threads();i++) {
  for(j=0;j<n;j++) {
    wrongtp[j] = ... /* only exists for thread 0 */
    .
    .
    .
  }
}
```

In general, memory of classes other than thread_private should be dynamically allocated in serial code. Allocating node_private, near_shared, far_shared and block_shared memory from within parallel code will create wasteful redundant copies.

Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!
      REAL*8 WRONGNP(:)
C$DIR NODE_PRIVATE(WRONGNP)
C$DIR FAR_SHARED_POINTER(WRONGNP)
      ALLOCATABLE WRONGNP
      .
      .
      .
      N = NUM_NODES
C$DIR LOOP_PARALLEL(NODES, IVAR = I)
      DO I = 1, N
        ALLOCATE(WRONGNP(M))
        .
        .
        .
      ENDDO
```

Recall from Chapter 5, “Memory classes,” that when a node_private array is allocated, a physical copy is created on each hypernode on which the program is running. Here, each loop iteration executes the ALLOCATE statement (or memory_class_malloc function in C), thus allocating N copies of the array. This is N-1 times more copies than are actually needed. To further complicate things, node_private arrays manipulated in parallel code must be accessed by shared pointers, which is why the Fortran example includes a far_shared_pointer statement. In the code above, this pointer would be overwritten every time the I loop executed the ALLOCATE statement (or memory_class_malloc function in C), meaning that only the final copy allocated would be accessible. Since the hypernodes’ execution of the loop code is not perfectly synchronized, the actual memory accessed by WRONGNP(I) would vary depending on which hypernode was last to perform the allocation.

An analogous C example follows:

```
/* INCORRECT EXAMPLE FOLLOWS!!!                            */
static far_shared double *wrongnp;
.
.
.
n = numnodes();
#pragma _CNX loop_parallel(nodes, ivar=i)
for(i=0;i<n;i++) {
  wrongnp = (double *)memory_class_malloc(sizeof(double)*m,
                                          NODE_PRIVATE_MEM);
  .
  .
  .
}
```

While dynamically allocated near_shared, far_shared and block_shared arrays do not normally require special pointer types, they suffer from the same redundant-copy problem. Allocating any shared-memory arrays from within parallel code will create as many copies of the data as there are hypernodes (or threads) executing the ALLOCATE (or memory_class_malloc) statement. As with the node_private example above, the actual memory accessed will depend on which hypernode most recently executed the ALLOCATE statement. After all hypernodes have executed the ALLOCATE, the memory allocated by all but the last will be lost. Such lost arrays are not only unusable, they cannot be deallocated.

To avoid such redundancy problems, follow the allocation examples discussed in Chapter 5, “Memory classes,” and only allocate memory from within parallel constructs as described there.

Incorrect array pointers

As mentioned in the previous section, sometimes it is necessary to access dynamically allocated arrays using pointers of different memory classes. For example, when accessing `node_private` arrays from node-parallel code, `far_shared` pointers must be used (refer to Chapter 5, “Memory classes”). Failing to do this will render the copies of the arrays on all but logical hypernode 0 inaccessible.

Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!!
      REAL*8 N0NP(:)
C$DIR NODE_PRIVATE(N0NP)
      ALLOCATABLE N0NP
      .
      .
      .
      ALLOCATE(N0NP(M))
      N = NUM_NODES
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J)
      DO I = 1, N
C$DIR LOOP_PARALLEL(THREADS)
        DO J = 1, M
          N0NP(J) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

While the `N0NP` array is correctly allocated in serial code here, it is not explicitly given a shared pointer, so the arrays created will be accessed by the default `node_private` pointer. A physical copy of `N0NP` will be created on every hypernode, but the `node_private` pointer by which these copies are accessed will only be initialized on logical hypernode 0, because it is the only hypernode executing the `ALLOCATE` statement (or `memory_class_malloc` in C). The contents of the (`node_private`) pointers on other hypernodes are uninitialized and therefore indeterminate. When, in the hypernode-parallel `J` loop, these other hypernodes attempt to access `N0NP`, they will do so using the garbage contents of their uninitialized pointers, typically causing a runtime error.

An analogous C example follows:

```
/* INCORRECT EXAMPLE FOLLOWS!!!                           */
static node_private double *n0np;
.
.
.
n0np = (double *)memory_class_malloc(sizeof(double)*m,
                                     NODE_PRIVATE_MEM);
n = numnodes();
#pragma _CNX loop_parallel(nodes, ivar=i), loop_private(j)
for(i=0;i<n;i++) {
#pragma _CNX loop_parallel(threads, ivar=j)
  for(j=0;j<m;j++) {
    n0np[j] = ...
    .
    .
    .
  }
}
```

Chapter 5, “Memory classes,” covers correct pointer/data combinations and explains the situations in which nondefault pointers should be used. To avoid uninitialized pointer problems such as the one described above, follow the recommendations of Chapter 5 carefully.

Hidden dependences

Improperly accessing a shared variable from parallel threads can create an unapparent dependence that can cause wrong answers.

Consider the following Fortran code:

```
      PROGRAM HOLDER
      REAL HOLD
C$DIR FAR_SHARED(HOLD)
C$DIR TASK_PRIVATE(X,Y)
C$DIR BEGIN_TASKS
      X = ...
      .
      .
      .
      CALL ADDHOLD(HOLD, X)
C$DIR NEXT_TASK
      Y = ...
      .
      .
      .
      CALL ADDHOLD(HOLD,Y)
C$DIR END_TASKS
      END

      SUBROUTINE ADDHOLD(HOLD,Z)
      REAL HOLD, Z
      HOLD = HOLD+Z
      END
```

Here, the `far_shared` variable `HOLD` is updated as a function of itself in the subroutine `ADDHOLD`, which is called from the potentially parallel tasks. If `HOLD` was updated within the tasks rather than in a subroutine, the dependence would be more obvious to the programmer, who may not have ready access to `ADDHOLD`'s source.

Isolating the assignment to `HOLD` inside a critical section would allow the tasks to safely parallelize, whether the assignment took place in a subroutine or inside the tasks themselves.

An analogous C example follows:

```
void addhold(float *hold, float z) {
  *hold = *hold + z;
}
main() {
  static far_shared float hold;
  static float x,y;
#pragma _CNX task_private(x,y)
#pragma _CNX begin_tasks
  x = ...;
  .
  .
  .
  addhold(&hold,x);
#pragma _CNX next_task
  y = ...;
  .
  .
  .
  addhold(&hold,y);
#pragma _CNX end_tasks
}
```

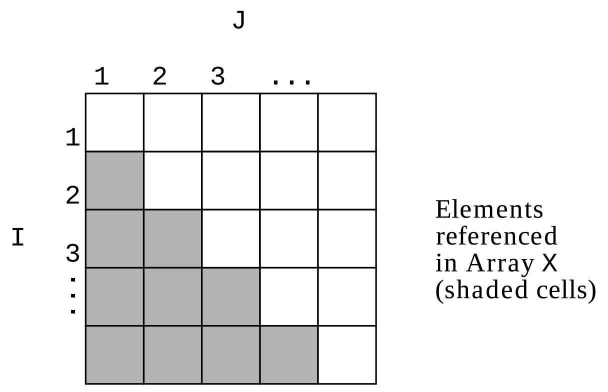
Always use caution when parallelizing a call to a procedure that passes the same shared variable from every thread.

Triangular loops

A *triangular loop* is a loop nest with an inner loop whose upper or lower bound (but not both) is a function of the outer loop's index. Examples of a lower triangular loop and an upper triangular loop are given below. To simplify explanations, only Fortran examples are given in this section.

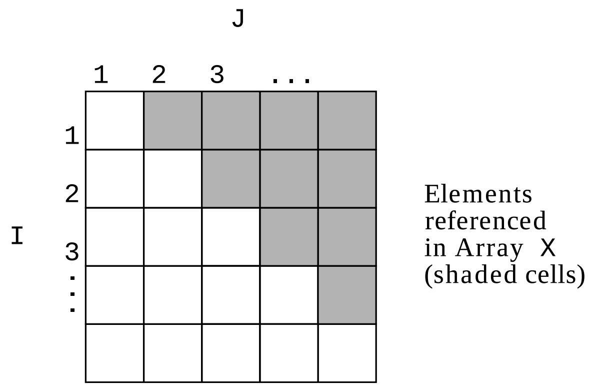
Lower triangular loop

```
DO J = 1, N
  DO I = J+1, N
    F(I) = F(I) + ... + X(I,J) + ...
```



Upper triangular loop

```
DO J = 1, N
  DO I = 1, J-1
    F(I) = F(I) + ... + X(I,J) + ...
```



While the compiler can usually auto-parallelize one of the outer or inner loops, there are typically performance problems in either case:

- If the outer loop is parallelized by assigning contiguous chunks of iterations to each of the threads, the load will be severely imbalanced. For example, in the lower triangular example above, the thread doing the last chunk of iterations does far less work than the thread doing the first chunk.
- If the inner loop is auto-parallelized, then on each outer iteration in the J loop, the threads are assigned to work on a different set of iterations in the I loop, thus losing access to some of their previously encached elements of F and thrashing each other's caches in the process.

By manually controlling the parallelization, you can greatly improve the performance of a triangular loop. Parallelizing the outer loop is generally more beneficial than parallelizing the inner loop. The next two sections ("Parallelizing the outer loop" and "Parallelizing the inner loop") explain how to achieve the enhanced performance.

Parallelizing the outer loop

Using directives you can control the parallelization of the outer loop in a triangular loop to optimize the performance of the loop nest.

For the outer loop, it is preferable to assign iterations to threads in a more balanced manner. The simplest method is to assign the threads one at a time using the attribute CHUNK_SIZE:

```
C$DIR PREFER_PARALLEL (CHUNK_SIZE = 1)
      DO J = 1, N
        DO I = J+1, N
          Y(I,J) = Y(I,J) + ...X(I,J)...
```

This causes each thread to execute in the following manner:

```
      DO J = MY_THREAD() + 1, N, NUM_THREADS()
        DO I = J+1, N
          Y(I,J) = Y(I,J) + ...X(I,J)...
```

where 0 <= MY_THREAD() < NUM_THREADS()

In this case, the first thread still does more work than the last, but the imbalance is greatly reduced. For example, assume N = 128 and there are 8 threads. Then the default parallel compilation would cause thread 0 to do J = 1 to 16, resulting in 1912 inner iterations, whereas thread 7 does J = 113 to 128, resulting in 120 inner iterations. With chunk_size = 1, thread 0 does 1072 inner iterations, and thread 7 does 1023.

Parallelizing the inner loop

If the outer loop cannot be parallelized, parallelize the inner loop if possible. There are two issues to be aware of when parallelizing the inner loop:

- Cache thrashing

  Consider the parallelization of the following inner loop:

  ```
  DO J = I+1, N
    F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
  ```

  where `I` varies in the outer loop iteration.

  The default iteration distribution has each thread processing a contiguous chunk of iterations of approximately the same number as every other thread. The amount of work per thread is about the same; however, from one outer iteration to the next, threads work on different elements in `F`, resulting in cache thrashing.

- The overhead of parallelization

  If the loop cannot be interchanged to be outermost (or at least outermore), then the overhead of parallelization is compounded by the number of outer loop iterations.

Below is a scheme that assigns “ownership” of elements to threads on a cache line basis so that threads always work on the same cache lines and retain data locality from one iteration to the next. In addition, the `parallel` directive (see the section “`parallel[(`*attribute_list*`)]`” on page 348) is used to spawn threads just once. The outer, nonparallel loop is replicated on all processors, and the inner loop iterations are manually distributed to the threads.

```
C F IS KNOWN TO BEGIN ON A CACHE LINE BOUNDARY
      NTHD = NUM_THREADS()
      CHUNK = 8               ! CHUNK * DATA SIZE (4 BYTES)
                              !    EQUALS PROCESSOR CACHE LINE SIZE;
                              !    A SINGLE THREAD WORKS ON CHUNK = 8
                              !    ITERATIONS AT A TIME
      NTCHUNK = NTHD * CHUNK  ! A CHUNK TO BE SPLIT AMONG THE THREADS
       ...
C$DIR PARALLEL,PARALLEL_PRIVATE(ID,JS,JJ,J,I)
      ID = MY_THREAD() + 1   ! UNIQUE THREAD ID
      DO I = 1, N
        JS = ((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
     >        + (ID-1) * CHUNK + 1
        DO JJ = JS, N, NTCHUNK
          DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)
            F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
          ENDDO
        ENDDO
      ENDDO
C$DIR END_PARALLEL
```

The idea is to assign a fixed ownership of cache lines of `F` and to assign a distribution of those cache lines to threads that keeps as many threads busy computing whole cache lines for as long as possible. Using `CHUNK = 8` for 4-byte data makes each thread work on 8 iterations covering a total of 32 bytes—the processor cache line size for V2200 and X2000 servers. In general, set `CHUNK` equal to the smallest value that multiplies by the data size to give a multiple of 32 (the processor cache line size on V2200 servers; also, the processor cache line size and CTIcache line size on X2000 servers). Smaller values of `CHUNK` keep most threads busy most of the time; however, setting `CHUNK` to obtain a multiple of 32 is better if the application is on an X2000 system and is executing on more than one hypernode, which implies that it is using the CTIcache.

When, because of the ever-decreasing work in the triangular loop, there are fewer cache lines left to compute than there are threads, threads drop out until there is only one thread left to compute those iterations associated with the last cache line. Compare this distribution to the default distribution that causes false cache line sharing (see the section “False cache line sharing” in this chapter) and consequent thrashing when all threads attempt to compute data into a few cache lines.

The scheme above maps a sequence of `NTCHUNK`-sized blocks over the `F` array. Within each block, each thread owns a specific cache line of data. The relationship between data, threads, and blocks of size `NTCHUNK` is shown in Figure 25.

Figure 25 **Data ownership by CHUNK and NTCHUNK blocks**

NTCHUNK 1

CHUNKs of F	Associated thread
F(1) ... F(8)	thread 0
F(9) ... F(16)	thread 1
F(17) ... F(24)	thread 2
F(33) ... F(40)	thread 3
F(41) ... F(48)	thread 4
F(25) ... F(32)	thread 5
F(49) ... F(56)	thread 6
F(57) ... F(64)	thread 7

NTCHUNK 2

CHUNKs of F	Associated thread
F(65) ... F(72)	thread 0
F(73) ... F(80)	thread 1
F(81) ...	...

CHUNK is the number of iterations a thread will work on at one time. The idea is to make a thread work on the same elements of F from one iteration of I to the next (except for those that are already complete). The scheme above causes thread 0 to do all work associated with the cache lines starting at F(1), F(1+NTCHUNK), F(1+2*NTCHUNK), and so on. Likewise, thread 1 does the work associated with the cache lines starting at F(9), F(9+NTCHUNK), F(9+2*NTCHUNK), and so on. Thus, if a thread assigns certain elements of F for I = 2, then it is certain that the same thread encached those elements of F in iteration I = 1, thus eliminating cache thrashing among the threads.

Examining the code

Having established the idea of assigning cache line ownership, consider the following Fortran example in more detail:

```
C$DIR PARALLEL,PARALLEL_PRIVATE(ID,JS,JJ,J,I)
        ID = MY_THREAD() + 1   ! UNIQUE THREAD ID
        DO I = 1, N
          JS = ((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
     >          + (ID-1) * CHUNK + 1
          DO JJ = JS, N, NTCHUNK
            DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)
              F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
            ENDDO
          ENDDO
        ENDDO
C$DIR END_PARALLEL
```

`C$DIR PARALLEL, PARALLEL_PRIVATE(ID,JS,JJ,J,I)`

The PARALLEL directive spawns threads, each of which begins executing the statements in the parallel region. Each thread has a private version of the variables ID, JS, JJ, J, and I.

`ID = MY_THREAD() + 1    ! UNIQUE THREAD ID`

This establishes a unique ID for each thread, in the range 1 to num_threads().

`DO I = 1, N`

All threads execute the I loop redundantly (instead of thread 0 executing it alone).

```
JS = ((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
      + (ID-1) * CHUNK + 1
```

For a given value of I+1, the above line determines in which NTCHUNK the value I+1 falls, then assigns a unique CHUNK of it to each thread ID. Suppose that there are *ntc* NTCHUNKs, where *ntc* is approximately N/NTCHUNK. Then the expression:

```
(I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK)
```

returns a value in the range 1 to *ntc* for a given value of I+1. Then the expression:

```
((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
```

identifies the start of an NTCHUNK that contains I+1 or is immediately above I+1 for a given value of ID.

For the NTCHUNK that contains I+1, if the cache lines owned by a thread either contain I+1 or are above I+1 in memory, this expression returns this NTCHUNK. If the cache lines owned by a thread are below I+1 in this NTCHUNK, this expression returns the next highest NTCHUNK. In other words, if there is no work for a particular thread to do in this NTCHUNK, then start working in the next one.

```
(ID-1) * CHUNK + 1
```

identifies the start of the particular cache line for the thread to compute within this NTCHUNK.

```
DO JJ = JS, N, NTCHUNK
```

Each thread does a unique set of cache lines starting at its specific JS and continuing into succeeding NTCHUNKs until all the work is done.

```
DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)
```

This does the work within a single cache line. If the starting index (I+1) is greater than the first element in the cache line (JS) then start with I+1. If the ending index (N) is less than the last element in the cache line, then finish with N.

More generally, notice that:

- Most of the “complicated” arithmetic is in outer loop iterations.
- Divides could be replaced, by the programmer, with shift instructions because they involve powers of two.
- If this application were to be run on an X2000 multihypernode system, choosing a chunk size of 8 for 4-byte data (32 bytes is the X2000 CTIcache line size) would be appropriate.

Compiler assumptions

Compiler assumptions can produce faulty optimized code when the source code contains:

- Iterations by zero
- Trip counts that may overflow at optimization levels +O2 and above

Descriptions of, and methods for, avoiding the items listed above are in the following sections.

Incrementing by zero

The compiler assumes that whenever a variable is being incremented on each iteration of a loop, the variable is being incremented by a loop-invariant amount other than zero. If the compiler parallelizes a loop that increments a variable by zero on each trip, the loop can produce incorrect answers or cause the program to abort. This error can occur when a variable used as an incrementation value is accidentally set to zero. If the compiler detects that the variable has been set to zero, the compiler does not parallelize the loop. If the compiler cannot detect the assignment, however, the symptoms described below occur.

The following Fortran example shows two loops that increment by zero:

```
CALL SUB1(0)
.
.
.
SUBROUTINE SUB1(IZR)
DIMENSION A(100), B(100), C(100)
J = 1
DO I = 1, 100, IZR ! INCREMENT VALUE OF 0 IS
                   ! NON-STANDARD
   A(I) = B(I)
ENDDO
PRINT *, A(11)
DO I = 1, 100
   J = J + IZR
   B(I) = A(J)
   A(J) = C(I)
ENDDO
PRINT *, A(1)
PRINT *, B(11)
END
```

Because `IZR` is an argument passed to `SUB1`, the compiler does not detect that `IZR` has been set to zero. Both loops parallelize at `+O3 +Oparallel +Onodynsel`.

The loops compile at `+O3`, but the first loop, which specifies the step as part of the `DO` statement (or as part of the `for` statement in C), attempts to parcel out loop iterations by a step of `IZR`. At runtime, this loop is infinite.

Due to dependences, the second loop would not behave predictably when parallelized—if it were ever reached at runtime. The compiler does not detect the dependences because it assumes J is an induction variable.

The analogous C code follows:

```
float a[100],b[100],c[100];

void sub1(int izr)
{
  int i,j = 1;

  for(i=0; i<100; i+=izr)
    a[i] = b[i];
  printf(“%f \n”, a[11]);
  for(i=0; i<100;i++) {
    j = j + izr;
    b[i] = a[j];
    a[j] = c[i];
  }
  printf(“%f \n”, a[1]);
  printf(“%f \n”, b[11]);
}

main()
{
  sub1(0);
}
```

Trip counts that may overflow

Some loop optimizations at +O2 and above may cause the variable on which the *trip count* is based to overflow. (A loop’s trip count is the number of times the loop executes.) The compiler assumes that each induction variable is increasing (or decreasing) without overflow during the loop. Any overflowing induction variable may be used by the compiler as a basis for the trip count. The following sections discuss when this overflow may occur and how to avoid it.

Linear test replacement

When optimizing loops, the compiler often disregards the original induction variable, using instead a variable or value that better indicates the actual *stride* of the loop. A loop's stride is the value by which the iteration variable increases on each iteration. By picking the largest possible stride, the compiler reduces the execution time of the loop by reducing the number of arithmetic operations within each iteration.

The Fortran code below contains an example of a loop in which the induction variable may be replaced by the compiler.

```
 ICONST = 64
 ITOT = 0
 DO IND = 1,N
   IPACK = (IND*1024)*ICONST**2
   IF(IPACK .LE. (N/2)*1024*ICONST**2)
>    ITOT = ITOT + IPACK
   .
   .
   .
 ENDDO
 END
```

Executing this loop using `IND` as the induction variable with a stride of 1 would be extremely inefficient, so the compiler picks `IPACK` as the induction variable and uses the amount by which it increases on each iteration, $1024*64^2$ or 2^{22}, as the stride.

The *trip count* (`N` in the example), or just *trip*, is the number of times the loop executes, and the *start* value is the initial value of the induction variable.

The following C function also contains an induction variable that may be replaced:

```
#include <math.h>
int ind, ipack, iconst, itot, n;
iconst = 64;
itot = 0;
for(ind=0; ind<n; ind++) {
  ipack = (ind*1024)*pow(iconst,2);
  if(ipack < (n/2)*1024*pow(iconst,2))
    itot += ipack;
  .
  .
  .
}
```

Here, as in the Fortran example, `ipack`, rather than `ind`, is used as the induction variable—again producing a stride of 2^{22}.

Linear test replacement, a standard optimization at levels `+O2` and above, normally does not cause problems. However, when the loop stride is very large, as in the examples above, a large trip count can cause the loop limit value (*start*+((*trip*-1)**stride*)) to overflow.

In the examples above, the induction variable is a 4-byte integer, which occupies 32 bits in memory. That means if *start*+((*trip*-1)**stride*) (1+((N-1)*2^{22})) is greater than 2^{31}-1, the value overflows into the sign bit and is treated as a negative number. If the stride value is negative, the absolute value of *start*+((*trip*-1)**stride*) must be not exceed 2^{31}. When a loop has a positive stride and the trip count overflows, the loop stops executing when the overflow occurs because the limit becomes negative—assuming a positive stride—and the termination test fails.

Because the largest allowable value for start+((*trip*-1)**stride*) is 2^{31}-1, the start value is 1, and the stride is 2^{22}, the maximum trip count for the loop can be found.

The stride, trip, and start values for a loop must satisfy the following inequality:

$$start + ((trip - 1) * stride) \leq 2^{31}$$

The start value is 1, so trip can be solved for as follows:

$$start + ((trip - 1) * stride) \leq 2^{31}$$
$$1 + (trip - 1) * 2^{22} \leq 2^{31}$$
$$(trip - 1) * 2^{22} \leq 2^{31} - 1$$
$$trip - 1 \leq 2^{9} - 2^{-22}$$
$$trip \leq 2^{9} - 2^{-22} + 1$$
$$trip \leq 512$$

The maximum value for `n` in the given loop, then, is 512.

If you find that certain loops give wrong answers at optimization levels `+O2` or higher, the problem may be test replacement. If you still want to optimize these loops at `+O2` or above, restructure them to force the compiler to choose a different induction variable.

Large trip counts at +O2 and above

When a loop is optimized at level `+O2` or above, its trip count must occupy no more than a signed 32-bit storage location. The largest positive value that can fit in this space is 2^{31} - 1 (2,147,483,647). Loops with trip counts that cannot be determined at compile time but that exceed 2^{31} - 1 at runtime will yield wrong answers.

This limitation only applies at optimization levels `+O2` and above.

A loop with a trip count that overflows 32 bits can be optimized by manually strip mining the loop.

A Standard HP compiler directives and pragmas

The standard Hewlett-Packard compilers provide the following directives and pragmas to control optimization levels and to inform the compiler about program behavior:

- `OPTIMIZE` directives
- `optimize` and `opt_level` pragmas
- `[no]inline` pragmas
- `allocs_new_memory` pragma
- `float_traps_on` pragma
- `[no]ptrs_strongly_typed` pragmas

This appendix discusses these directives and pragmas, which can be used in addition to the command-line optimization options described in Appendix D, “Optimization options.” However, unlike the command-line options, the directives and pragmas allow you to specify optimizations on a function-by-function basis.

See Appendix B, “Exemplar compiler directives and pragmas,” for an overview of the directives and pragmas that make up part of the Exemplar programming model.

The HP Exemplar compilers support most of the directives and pragmas available in the standard HP compilers. For information on those directives and pragmas, see the appropriate manual. (The section “Associated documents” on page xxv lists related manuals.)

Fortran OPTIMIZE directives

If you wish to control compilation in finer detail than what is allowed using command-line options, use the `OPTIMIZE` compiler directives. If you use one of the `OPTIMIZE` directives, you must also specify a command-line option that sets the optimization level (`+O1`, `+O2`, `+O3`, `+O4`, or `-O`). You cannot use an `OPTIMIZE` directive to raise the optimization level above the level specified by the command-line option; the compiler uses the lower of the two optimization levels.

The `OPTIMIZE` directives control which functions are optimized and, for Fortran 77, which set of optimizations is performed. Some directives must be placed before the function to be optimized, while others can appear anywhere within the function. The Fortran 77 `OPTIMIZE` directives allow you to control the level of optimization as well as the assumptions the optimizer makes when compiling a program.

If an optional `[ON|OFF]` is omitted, it defaults to an `ON` setting. Once turned on, `OPTIMIZE` directives remain in effect—for all program units that lexically follow them in the source file—until they are revoked by another `OPTIMIZE` directive.

Fortran 90 OPTIMIZE directives

The `$HP$ OPTIMIZE` directive enables or disables the level of optimization that was specified on the command line for the following program units.

The syntax and descriptions of the `OPTIMIZE` directives are given below.

```
!$HP$ OPTIMIZE [ON|OFF]
```

`!$HP$ OPTIMIZE OFF`

Specifies level 0 (`+O0`) optimizations.

`!$HP$ OPTIMIZE ON`

Specifies the level of optimization set by the command-line option `+O0`, `+O1`, `+O2`, `+O3`, `+O4`, or `-O`.

This directive is effective for all program units that follow it in your program. It should be placed outside and before the program units it is to affect.

Fortran 77 OPTIMIZE directives

In Fortran 77, `OPTIMIZE` directives using the `ASSUME_NO_SIDE_EFFECTS` or the `ASSUME_PARM_TYPES_MATCHED` options can appear anywhere within a program unit. All other directives must appear outside a program unit.

The options to the Fortran 77 `OPTIMIZE` directive are listed below:

- `[ON|OFF]`
- `LEVEL1 [ON|OFF]`
- `LEVEL2 [ON|OFF]`
- `LEVEL2_MIN [ON|OFF]`
- `LEVEL2_MAX [ON|OFF]`
- `LEVEL3 [ON|OFF]`
- `LEVEL4 [ON|OFF]`
- `[NO]INLINE=[`*namelist*`]`
- `ASSUME_NO_EXTERNAL_PARMS [ON|OFF]`
- `ASSUME_NO_FLOATING_INVARIANT [ON|OFF]`
- `ASSUME_NO_HIDDEN_POINTER_ALIASING [ON|OFF]`
- `ASSUME_NO_PARAMETER_OVERLAPS [ON|OFF]`
- `ASSUME_NO_SHARED_COMMON_PARMS [ON|OFF]`
- `ASSUME_NO_SIDE_EFFECTS [ON|OFF]`
- `ASSUME_PARM_TYPES_MATCHED [ON|OFF]`

The syntax and descriptions of the `OPTIMIZE` directives are given below.

`$OPTIMIZE OFF`

Specifies level 0 (`+O0`) optimizations. This directive is the default.

`$OPTIMIZE ON`

Specifies the level of optimization set by the command-line option `+O0`, `+O1`, `+O2`, `+O3`, `+O4`, or `-O`. If the level of optimization was not set by an option, this directive is ignored.

`$OPTIMIZE LEVEL1 [ON|OFF]`

Turns on or off level 1 (`+O1`) optimizations (optimizes only within each basic block).

`$OPTIMIZE LEVEL2 [ON|OFF]`

Turns on or off level 2 (`+O2`) optimizations, with the following `ASSUME` settings:

```
ASSUME_NO_PARAMETERS_OVERLAPS ON
ASSUME_PARM_TYPES_MATCHED ON
ASSUME_NO_EXTERNAL_PARMS ON
ASSUME_NO_SHARED_COMMON_PARMS ON
ASSUME_NO_SIDE_EFFECTS OFF
ASSUME_NO_FLOATING_INVARIANT ON
ASSUME_NO_HIDDEN_POINTER_ALIASING ON
```

`$OPTIMIZE LEVEL2_MIN [ON|OFF]`

Turns on or off level 2 (`+O2`) optimizations with all the `ASSUME` settings at `OFF`.

`$OPTIMIZE LEVEL2_MAX [ON|OFF]`

Turns on or off level 2 (`+O2`) optimizations with all the `ASSUME` settings at `ON`.

`$OPTIMIZE LEVEL3 [ON|OFF]`

Turns on or off level 3 (`+O3`) optimizations.

`$OPTIMIZE LEVEL4 [ON|OFF]`

Turns on or off level 4 (`+O4`) optimizations.

`$OPTIMIZE [NO]INLINE[=`*namelist*`]`

The compiler directive `$OPTIMIZE [NO]INLINE` is analogous to the `+O[no]inline` command-line option: it is used either to request inlining or to disable it. (Inlining can occur only at optimization levels `+O3` and above; it is enabled by default whenever you specify the `+O3` or `+O4` option.)

The syntax for using this directive is:

`$OPTIMIZE [NO]INLINE[=`*namelist*`]`

where

namelist	is a comma-separated list of routine names.

When you use the `$OPTIMIZE INLINE` directive to enable inlining, the optimizer treats the directive as a request to inline. If you specify a list of procedure names with the directive and inlining is already on by default (that is, you have also used the `+O3` or `+O4` option), the optimizer gives special consideration to the named procedures.

When you use the `$OPTIMIZE NOINLINE` directive to disable inlining and do not specify a list of names, the optimizer disables inlining for all procedures it encounters thereafter. If you specify a list of procedure names, the optimizer only disables inlining for the named procedures.

`$OPTIMIZE ASSUME_NO_EXTERNAL_PARMS [ON|OFF]`

Turns on or off the `ASSUME` setting that none of the parameters passed to the current procedure are from an external space. (External space refers to space that is different from the user's own data space.) Parameters can come from another space if they come from operating system space or if they are in a space shared by other users. If `ASSUME_NO_EXTERNAL_PARMS` is `OFF`, the compiler is unable to perform certain optimizations, such as array-accessing optimization.

`$OPTIMIZE ASSUME_NO_FLOATING_INVARIANT [ON|OFF]`

Turns on or off the `ASSUME` setting that no loop-invariant floating-point operations can cause an exception if executed outside the loop.

`$OPTIMIZE ASSUME_NO_HIDDEN_POINTER_ALIASING [ON|OFF]`

Turns on or off the `ASSUME` setting that Fortran pointers are not used to:

- Save Argument Addresses Across Procedure Calls
- Access variables whose addresses have not been explicitly taken with an address-returning intrinsic

This directive should be turned `OFF` when any of the following conditions are true:

- A subroutine or function saves the address of any of its arguments between invocations (either in `COMMON` or in `SAVE` variables).
- Any function returns the address of any of its arguments.
- A variable is modified through a pointer access even though its address was not explicitly taken with one of the address-returning intrinsics (`%LOC`, `LOC`, `BADDRESS`, `IADDR`). For example, assume that the address of a variable in `COMMON` is taken and the result incremented by some number of bytes. It is then assigned to a pointer in order to update another variable in that `COMMON` block. In such cases, the other variable is modified through a pointer without its address being explicitly taken.

Turning the `ASSUME_NO_HIDDEN_POINTER_ALIASING` setting `OFF` can increase the time it takes the compiler to optimize the program.

`$OPTIMIZE ASSUME_NO_PARAMETER_OVERLAPS [ON|OFF]`

Turns on or off the `ASSUME` setting that no actual parameters have overlapping storage in the calling program.

`$OPTIMIZE ASSUME_NO_SHARED_COMMON_PARMS [ON|OFF]`

Turns on or off the ASSUME setting that none of the parameters passed to the current procedure are from a shared COMMON block. If there is a shared COMMON block parameter, the compiler needs to be informed of it so that the compiler always returns to memory to access the values of these variables instead of keeping them as register variables. This directive should also be used when all of the following are true:

- The parameter passed to the current procedure is part of a COMMON block used by that procedure.
- The parameter is named differently than the variable name it has in the COMMON block.
- The parameter is reassigned with the same value within the procedure.

`$OPTIMIZE ASSUME_NO_SIDE_EFFECTS [ON|OFF]`

Turns on or off the ASSUME setting that the current procedure changes only local variables. It does not change any variables in COMMON, nor does it change parameters.

`$OPTIMIZE ASSUME_PARM_TYPES_MATCHED [ON|OFF]`

Turns on or off the ASSUME setting that formal and actual parameter pairs in the current procedure unit match in type.

C and C++ pragmas

The compiler pragmas discussed in this section, available in both the C and aC++ compilers, allow you to control compilation in finer detail than what is possible using command-line options. These pragmas also enable you to give information about your program to the compiler.

Pragmas cannot cross line boundaries and the word `pragma` must be in lowercase letters. The optimizer pragmas discussed in this section may not appear inside a function.

Optimizer control pragmas

The `OPTIMIZE` and `opt_level` pragmas control which functions are optimized and which set of optimizations are performed. These pragmas can be placed before any function definition and will override any previous pragma. These pragmas cannot raise the optimization level above the level specified on the command line. Once turned on, these directives remain in effect for the remainder of the file or until superseded by another pragma. For these pragmas to work, the source must be compiled with one of the optimization level options (`+O0`, `+O1`, `+O2`, `+O3`, `+O4`, or `-O`).

The `opt_level 1` and `opt_level 2` pragmas provide more control over optimization than the `+O1` and `+O2` compiler options because these pragmas can be used to raise or lower optimization on a function-by-function basis inside the source file using different levels for different functions. The `opt_level 3` and `opt_level 4` pragmas can only be used at the beginning of the source file.

Table 14 summarizes the values of `OPTIMIZE` and `opt_level`.

Table 14 **C and C++ optimizer control pragmas**

Pragma*	Description
`#pragma OPTIMIZE ON`	Turns optimization on (the compiler uses the optimization level specified on the command line)
`#pragma OPTIMIZE OFF`	Turns optimization off
`#pragma opt_level 1`	Optimize only within small blocks of code
`#pragma opt_level 2`	Optimize within each procedure
`#pragma opt_level 3`	Optimize across all procedures within a source file
`#pragma opt_level 4`	Optimize across all procedures within a program

*`opt_level 3` and `opt_level 4` must be specified at the top of the source file.

C pragmas

The compiler pragmas discussed in this section, available only in the C compiler, allow you to control compilation in finer detail than what is possible using command-line options. These pragmas also enable you to give information about your program to the compiler.

Pragmas cannot cross line boundaries and the word `pragma` must be in lowercase letters. The optimizer pragmas discussed in this section may not appear inside a function.

`[no]inline` pragmas

The syntax for the `[no]inline` pragma is:

`#pragma [no]inline [`*namelist*`]`

where

namelist	is a comma-separated list of function names.

When `inline` is specified without *namelist*, any function can be inlined. When specified with *namelist*, the functions given in *namelist* are candidates for inlining.

The `noinline` pragma disables inlining for all functions or those functions specified in *namelist*.

For example, to specify inlining of the two subprograms `checkstat` and `getinput`, use:

```
#pragma inline checkstat, getinput
```

To specify that an infrequently called routine should not be inlined when compiling at `+O3` or `+O4`, use:

```
#pragma noinline opendb
```

See the related `+O[no]inline` optimization option in the section "`+O[no]inline[=`*namelist*`]`" on page 373.

`allocs_new_memory` pragma

The compiler gathers information about each function (such as information about function calls, variables, parameters, and return values) and passes this information to the optimizer. The `allocs_new_memory` pragma tells the optimizer to make assumptions it cannot normally make, resulting in improved compile-time and runtime speed. The pragma changes the default information the compiler collects.

The `allocs_new_memory` *function_name* pragma states that the function *function_name* returns a pointer to new memory that either it allocates or a routine that it calls allocates. The new memory must be memory that was either newly allocated or was previously freed and is now reallocated. For example, the standard routines `malloc()` and `calloc()` satisfy this requirement.

If used, the `allocs_new_memory` pragma should appear before the first function defined in a file and is in effect for the entire file.

The `allocs_new_memory` pragma has the form:

`#pragma allocs_new_memory` *namelist*

where

namelist	is a comma-separated list of function names.

Large applications might have routines that are layered above `malloc()` and `calloc()`. These interface routines make the calls to `malloc()` and `calloc()`, initialize the memory, and return the pointer that `malloc()` or `calloc()` returns. For example, consider the program below:

```
struct_type *get_new_record(void) {
  struct_type *p;

  if ((p=malloc(sizeof(*p))) == NULL) {
    printf("get_new_record():out of memory\n");
    abort();
  }
  else {
  /* initialize the struct */
  .
  .
  .
  return p;
  }
}
```

The routine `get_new_record` falls under this category, and can be included in *namelist* in the `allocs_new_memory` *namelist* pragma.

`float_traps_on` pragma

This pragma informs the compiler that the function(s) may enable floating-point trap handling. When the compiler is so informed, it will not perform loop-invariant code motion on floating-point operations in the function(s) named in the pragma. This pragma is required for proper code generation when floating-point traps are enabled.

The pragma has the following form:

`#pragma float_traps_on [`*namelist*`]`

where

namelist	is a comma-separated list of function names.

For example:

```
#pragma float_traps_on xyz,abc
```

informs the compiler and optimizer that `xyz` and `abc` have floating-point traps turned on and therefore loop-invariant code motion should not be performed.

`[no]ptrs_strongly_typed` pragmas

The `ptrs_strongly_typed` pragma allows you to specify when a subset of types are type-safe. This provides a finer level of control than the `+O[no]ptrs_strongly_typed` command-line option that is discussed in Appendix D, “Optimization options.”

The `ptrs_strongly_typed` pragma has the following form:

```
#pragma ptrs_strongly_typed begin
      ...
#pragma ptrs_strongly_typed end
```

Similarly, the `noptrs_strongly_typed` pragma has the form:

```
#pragma noptrs_strongly_typed begin
      ...
#pragma noptrs_strongly_typed end
```

The type-safe assumptions apply to all types that are defined between a `#pragma ptrs_strongly_typed begin/end` pair. These pragmas are not allowed to nest. For each `begin`, an associated `end` must be defined in the compilation unit.

The pragma takes precedence over the `+O[no]ptrs_strongly_typed` command-line option. Although, sometimes both are required as shown in the example below.

In this example only two types, pointer-to-`int` and pointer-to-`float` will be assumed to not be type-safe. Assume the following program is named foo.c:

```
double *d;
.
.
.
#pragma noptrs_strongly_typed begin
int *i;
float *f;
#pragma noptrs_strongly_typed end
.
.
.
main(){
.
.
.
}
```

If foo.c is compiled as in the following command line:

```
% cc +Optrs_strongly_typed foo.c
```

then all types are assumed to be type-safe except the types bracketed by `#pragma noptrs_strongly_typed`.

B Exemplar compiler directives and pragmas

This appendix presents an alphabetical list of all the Fortran 90 directives, Fortran 77 directives, and C pragmas that are part of the Exemplar programming model. See Appendix A, “Standard HP compiler directives and pragmas,” for information on additional directives and pragmas available in the Exemplar compilers.

The HP aC++ compiler does not support the pragmas described in this appendix; however, it does support the typedefs (`barrier_t` and `gate_t`) and storage class specifiers (`far_shared`, `near_shared`, `node_private`, and `thread_private`) explained in this appendix.

Overview

This appendix provides a brief overview of the directives and pragmas available in the Exemplar programming model. More specific information and examples can be found elsewhere in this guide. The Fortran directives not supported as C pragmas are expressed in C as either storage class extensions (`thread_private`, etc.) or as typedefs (`gate_t`, `barrier_t`, etc.) in the spp_prog_model.h file and are described in Chapter 5, “Memory classes,” and Chapter 6, “Advanced shared-memory programming.”

NOTE The forms of the directives and pragmas discussed in this appendix differ from the forms for the directives and pragmas described in Appendix A, “Standard HP compiler directives and pragmas.”

The form of an Exemplar Fortran compiler directive is:

`C$DIR` *directive-list*

The form of an Exemplar C pragma is:

`#pragma _CNX` *directive-list*

where

directive-list
: is a comma-separated list of one or more of the directives/pragmas described in this chapter.

Directive names are presented here in lowercase; they may be specified in either case in both languages, but `#pragma` must always appear in lowercase in C. In the sections that follow, *namelist* represents a comma-separated list of names. These names can be variables, arrays, or `COMMON` blocks. In the case of a `COMMON` block, its name must be enclosed within slashes. The occurrence of a lowercase *n* or *m* is used to indicate an integer constant. Occurrences of *gate_var* are for variables that have been, or are being, defined as gates. Any parameters that appear within square brackets (`[ ]`) are optional.

Directives and pragmas

Brief descriptions of the available directives and pragmas follow in this section. Where appropriate, cross references to chapters containing more detailed information are included.

`align_cti(`***namelist***`)`

This directive or pragma aligns the variables and arrays listed in *namelist* on CTIcache boundaries. This allows for more efficient data reuse. The CTIcache is 32 bytes on X2000 systems. (V2200 servers and nonscalable SMPs do not use a CTIcache.)

`barrier(`***namelist***`)`

This Fortran directive denotes a list of variables, as given in *namelist*, that will be used as the synchronization variables for the barrier routines. This does not imply any synchronization in itself, it is simply defining the barrier variables. In C, `barrier` is a typedef (`barrier_t`), rather than a pragma. For more information, refer to Chapter 6, “Advanced shared-memory programming.”

`begin_tasks[(`***attribute_list***`)]`

This directive or pragma defines the beginning of a section (or sections; see `next_task`) of code that will be executed as an independent, parallel task. Each task is executed by a separate thread. `begin_tasks` must have an accompanying `end_tasks` in the same program unit.

The optional *attribute_list* can be any of the following legal combinations (*m* is an integer constant):

- `threads` (default)
- `nodes`
- `dist`
- `ordered`
- `max_threads=`*m*
- `threads, ordered`
- `nodes, ordered`
- `dist, ordered`
- `threads, max_threads=`*m*
- `nodes, max_threads=`*m*
- `dist, max_threads=`*m*
- `ordered, max_threads=`*m*
- `threads, ordered, max_threads=`*m*
- `nodes, ordered, max_threads=`*m*
- `dist, ordered, max_threads=`*m*

Attributes may be listed in any order. The compilers flag any attribute combinations other than those listed above with a warning and ignore the directive.

Refer to Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” for a complete discussion of parallel tasking.

`block_loop[(block_factor=`*n*`)]`

This directive or pragma indicates a specific loop to block, and optionally, the block factor *n* (*n* must be an integer constant greater than or equal to 2) that will be used in the compiler's internal computation of loop nest based data reuse. If no `block_factor` is specified, the compiler uses a heuristic to determine the `block_factor`. Refer to Chapter 3, "Compiler optimizations," for more information on blocking.

`block_shared(`*allocatable_array_namelist*`)`

This Fortran directive is used to declare arrays as being of type `block_shared`. Block-shared arrays are sized to be an integral multiple of the page size. The pages of the array are distributed in same-size blocks across the hypernodes on which the process is executing in the system. If the user-specified size is not an integral multiple of page size × `num_nodes()`, then the size is automatically rounded up to meet this criterion. Refer to Chapter 5, "Memory classes," for more information.

`critical_section[(`*gate_var*`)]`

This directive or pragma defines the beginning of a code block in which only one thread may be executing at a time. The end of the code block must be indicated by an `end_critical_section` directive or pragma, which must appear in the same flow of control within the same program unit. The optional *gate_var* can be used to differentiate between parallel tasks. Refer to Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," for more information.

`dynsel[(`*`trip_count`*`=`*`n`*`)]`

This directive or pragma enables workload-based dynamic selection for the immediately following loop. *trip_count* represents either the `thread_trip_count` or `node_trip_count` attribute, and *n* is an integer constant. When `thread_trip_count =` *n* is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the thread-parallel version is run. When `node_trip_count =` *n* is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the node-parallel version is run—assuming `+Onodepar` is specified. Refer to Chapter 3, “Compiler optimizations” for more information on dynamic selection.

`end_critical_section`

This directive or pragma defines the end of the critical section that was begun with the `critical_section` directive or pragma. `critical_section` and `end_critical_section` must appear as a pair. Refer to Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” for more information.

`end_ordered_section`

This directive or pragma defines the end of the ordered section that was begun with the `ordered_section` directive or pragma. `ordered_section` and `end_ordered_section` must appear as a pair. Refer to Chapter 6, “Advanced shared-memory programming,” for more information on ordered sections.

`end_parallel`

This directive or pragma signifies the end of a parallel region. The `parallel` directive signifies the beginning of a parallel region. Refer to Chapter 4, “Basic shared-memory programming,” for more information.

`end_tasks`

This directive or pragma terminates the specification of parallel tasks indicated by `begin_tasks` and `next_task`. It must appear at the end of the last section of parallel code defined by these directives or pragmas. All of these must appear in the same program unit. Refer to Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” for more information.

`far_shared(`***namelist***`)`

This Fortran directive causes the compiler to place the data objects in *namelist* (variables, arrays, or `COMMON` blocks) into `far_shared` memory. `far_shared` memory is the most general form that is distributed on a page basis across the memories of all hypernodes in a system. The `far_shared` data objects of a process are addressable by all threads of that process. In C and C++, `far_shared` is a storage class specifier. Refer to Chapter 5, “Memory classes,” for more information on memory classes.

`far_shared_pointer(`***namelist***`)`

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointers to the allocated objects (specified in *namelist*) in `far_shared` memory, regardless of the memory classes to which the respective objects are allocated.

This directive applies only to Fortran 90 allocatable data objects and to Fortran 90-style allocatable data objects used in HP Fortran 77 programs. Refer to Chapter 5, “Memory classes,” for more information on memory classes.

`gate(`***namelist***`)`

This Fortran directive specifies a list of gate variables that will be subsequently used in a critical section, ordered section, or passed as an argument to the synchronization intrinsics. In C and C++, `gate` is a typedef (`gate_t`), rather than a pragma. Refer to Chapter 6, “Advanced shared-memory programming,” for more information.

`loop_parallel[(`*attribute_list*`)]`

This directive or pragma is an explicit instruction to the compiler to parallelize the immediately following loop. The loop iterations will be run in an indeterminate order unless the optional `ordered` attribute appears. The user is responsible for any required data privatization and loop synchronization, as described in Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming.” The optional *attribute_list* can be any of the following combinations (*n* and *m* are integer constants):

- `threads` (default)
- `nodes`
- `dist`
- `ordered`
- `max_threads=`*m*
- `chunk_size=`*n*
- `threads, ordered`
- `nodes, ordered`
- `dist, ordered`
- `threads, max_threads=`*m*
- `nodes, max_threads=`*m*
- `dist, max_threads=`*m*
- `ordered, max_threads=`*m*
- `threads, chunk_size=`*n*
- `nodes, chunk_size=`*n*
- `dist, chunk_size=`*n*
- `threads, ordered, max_threads=`*m*
- `nodes, ordered, max_threads=`*m*
- `dist, ordered, max_threads=`*m*
- `chunk_size=`*n*`, max_threads=`*m*
- `threads, chunk_size=`*n*`, max_threads=`*m*

- `nodes, chunk_size=`*n*`, max_threads=`*m*
- `dist, chunk_size=`*n*`, max_threads=`*m*
- `ivar=`*indvar*

The `ivar=`*indvar* attribute is:

- Required for all loops in C and for `DO WHILE` and hand-rolled loops in Fortran
- Optional for Fortran `DO` loops
- Compatible with any other attribute

Attributes may be listed in any order. The compilers flag any attribute combinations other than those listed above with a warning and ignore the directive.

Refer to Chapter 6, “Advanced shared-memory programming,” for more information.

`loop_private(`***namelist***`)`

This directive or pragma declares a list of variables and/or arrays private to the immediately following loop. No values may be carried into the loop by `loop_private` variables. To be loop private, the variables and/or arrays must be assigned before they are used on each iteration of the immediately following loop. These private data items are distinct from the shared items of the same name that exist outside the loop. Values assigned to `loop_private` variables on the final iteration (that is, the *n*th iteration of a loop with *n* iterations) may be saved into the shared variables of the same name if the `save_last` directive or pragma also appears on this loop. If `save_last` is not used, then the value of any shared variable declared to be `loop_private` is undefined at loop termination. Refer to Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” for more information.

`near_shared(`***namelist***`)`

When applied to static variables at compile-time, this Fortran directive will cause all pages of the data objects in *namelist* to be mapped to physical pages on logical hypernode 0. If applied to allocatable arrays, then the pages of such arrays will be mapped to physical pages on the hypernode of the allocating thread. `near_shared` data can be addressed by any thread of a process on any hypernode in the system but it is “closer” (in terms of access latency) to the threads on the hypernode that allocates the data. In C and C++, `near_shared` is a storage class specifier. Refer to Chapter 5, “Memory classes,” for more information on memory classes.

`near_shared_pointer(`***namelist***`)`

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointers to the allocated objects (specified in *namelist*) in `near_shared` memory, regardless of the memory classes to which the objects are allocated.

This directive applies only to Fortran 90 allocatable data objects and to Fortran 90-style allocatable data objects used in HP Fortran 77 programs. Refer to Chapter 5, “Memory classes,” for more information on memory classes.

`next_task`

This directive or pragma starts a block of code following a `begin_tasks` block that will be executed as a parallel task. The end of the code block is marked by another `next_task` or by an `end_tasks` directive or pragma.

This directive must appear within a `begin_tasks` and `end_tasks` pair. There is no limit on the number of `next_task` directives that can appear. Refer to Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” for more information.

`no_block_loop`

This directive or pragma disables loop blocking on the immediately following loop. Refer to Chapter 3, “Compiler optimizations,” for more information on loop blocking.

`no_distribute`

This directive or pragma disables loop distribution for the immediately following loop. Refer to Chapter 3, “Compiler optimizations,” for more information on loop distribution.

`no_dynsel`

This directive or pragma disables workload-based dynamic selection for the immediately following loop. Refer to Chapter 3, “Compiler optimizations,” for more information on dynamic selection.

`no_loop_dependence(`*namelist*`)`

This directive or pragma informs the compiler that the arrays in *namelist* do not have any dependences for iterations of the immediately following loop. Use `no_loop_dependence` for arrays only; use `loop_private` to indicate dependence-free scalar variables.

This directive or pragma causes the compiler to ignore any dependences that it perceives to exist. This can enhance the compiler’s ability to optimize the loop, including the possibility of parallelization.

Refer to Chapter 3, “Compiler optimizations,” and Chapter 8, “Programming conventions for optimal code,” for more information.

`no_loop_transform`

This directive or pragma prevents the compiler from performing reordering transformations on the following loop. The compiler will not distribute, fuse, block, interchange, unroll, unroll and jam, or parallelize a loop on which this directive appears. Refer to Chapter 3, “Compiler optimizations,” for more information.

`no_parallel`

This directive or pragma prevents the compiler from generating parallel code for the immediately following loop. Refer to Chapter 3, “Compiler optimizations,” for more information.

`no_side_effects(`***funclist***`)`

This directive or pragma informs the compiler that the functions appearing in *funclist* have no side effects wherever they appear lexically following the directive. Side effects include modifying a function argument, modifying a Fortran `COMMON` variable, performing I/O, or calling another routine that does any of the above. The compiler can sometimes eliminate calls to procedures that have no side effects; also the compiler may be able to parallelize loops with calls when informed that the called routines do not have side effects.

`no_unroll_and_jam`

This directive or pragma disables loop unroll and jam for the immediately following loop. Refer to the section“Loop unroll and jam” on page 95 for more information.

`node_private(`***namelist***`)`

This Fortran directive causes the variables and arrays specified in *namelist* to be replicated in the physical memory of each hypernode on which the process is executing. Thus, while each data object has a single image in virtual memory, it maps to a different physical location on each hypernode. The threads of a process within a hypernode all share access to the copy on their hypernode and cannot access the copies on other hypernodes. In C and C++, `node_private` is a storage class specifier. Refer to Chapter 5, “Memory classes,” for more information.

`node_private_pointer(`***namelist***`)`

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointers to the allocated objects (specified in *namelist*) in `node_private` memory, regardless of the memory classes to which the objects are allocated.

This directive applies only to Fortran 90 allocatable data objects and to Fortran 90-style allocatable data objects used in HP Fortran 77 programs. Refer to Chapter 5, “Memory classes,” for more information.

`ordered_section(`***gate_var***`)`

This directive or pragma defines the beginning of an ordered section. An ordered section is the same as a critical section (a code block in which only one thread may be executing at a time) with the additional restriction that the threads must pass through the ordered section in iteration order. The end of the code block must be indicated by an `end_ordered_section` directive or pragma. Ordered sections must appear within the control flow of a `loop_parallel(ordered)` directive. Refer to Chapter 6, “Advanced shared-memory programming,” for more information.

`parallel[(`***attribute_list***`)]`

This directive or pragma signifies the beginning of a parallel region of code. All code up to the following `end_parallel` directive or pragma will be run on all available threads. No loop transformations, data privatization, or parallelization analysis will be performed by the compiler on the code in the region.

The optional attribute_list can be any of the following legal combinations (*m* is an integer constant):

- `threads` (default)
- `nodes`
- `max_threads=`*m*
- `threads, max_threads=`*m*
- `nodes, max_threads=`*m*

Attributes may be listed in any order. The compilers flag any attribute combinations other than those listed above with a warning and ignore the directive.

Refer to Chapter 4, “Basic shared-memory programming,” for more information.

`parallel_private(`***namelist***`)`

This directive or pragma declares a list of variables or arrays private to the immediately following parallel region. It serves the same purpose for parallel regions that `task_private` serves for tasks. The privatized variables and arrays will not carry their values beyond the `end_parallel` directive or pragma. Refer to Chapter 4, “Basic shared-memory programming,” for more information.

`prefer_parallel[(`***attribute_list***`)]`

This directive or pragma instructs the compiler to parallelize the following loop but only if it is safe to do so. A loop is safe to parallelize if it has an iteration count that can be determined at runtime before loop invocation, and contains no loop-carried dependences (LCDs), procedure calls, or I/O operations. Refer to Chapter 4, “Basic shared-memory programming,” for more information.

The optional *attribute_list* can be any of the following combinations (*n* and *m* are integer constants):

- `threads` (default)
- `nodes`
- `dist`
- `max_threads=`*m*
- `chunk_size=`*n*
- `threads, max_threads=`*m*
- `nodes, max_threads=`*m*
- `dist, max_threads=`*m*
- `threads, chunk_size=`*n*
- `nodes, chunk_size=`*n*
- `dist, chunk_size=`*n*
- `chunk_size=`*n*`, max_threads=`*m*
- `threads, chunk_size=`*n*`, max_threads=`*m*
- `nodes, chunk_size=`*n*`, max_threads=`*m*
- `dist, chunk_size=`*n*`, max_threads=`*m*

Attributes may be listed in any order. The compilers flag any attribute combinations other than those listed above with a warning and ignore the directive.

`reduction(`*namelist*`)`

This directive or pragma—which is only to be used with `loop_parallel`—specifies that the scalar variables in the comma-separated *namelist* are involved in reductions. The `reduction` directive and pragma are used to inform the compiler of reductions in `loop_parallel` loops. Once the compiler is informed of the reductions, the compiler generates code to perform the reduction while parallelizing the loop—assuming no other parallelization inhibitors are present in the loop. Refer to the section “Reductions” on page 114 for more information.

`save_last[(`*list*`)]`

This directive or pragma specifies that the variables in the comma-separated *list* that are also named in an associated `loop_private(`*namelist*`)` directive or pragma must have their last values saved into the “shared” variable of the same name at loop termination. (A variable’s last value in a loop of *n* iterations is the value it is assigned in the *n*th iteration.)

If the optional *list* is not used, `save_last` specifies that all variables named in an associated `loop_private(`*namelist*`)` directive or pragma must have their last values saved into the “shared” variable of the same name at loop termination.

If `save_last` is not specified then the values in any privatized variables or arrays are indeterminate at loop termination. Refer to Chapter 6, “Advanced shared-memory programming,” for more information.

`scalar`

This directive or pragma prevents the compiler from performing reordering transformations on the following loop. The compiler will not distribute, fuse, block, interchange, unroll, unroll and jam, or parallelize a loop on which this directive appears.

The `no_loop_transform` directive or pragma provides the same functionality as the `scalar` directive or pragma and is recommended in place of the `scalar` directive or pragma.

`sync_routine(`***`routinelist`***`)`

This directive or pragma indicates to the compiler that the routines listed in *routinelist* are user-defined synchronization routines, so that the compiler does not attempt to move code across these routine calls. Use `sync_routine` anytime you hide a call to a compiler synchronization function inside another routine call, or anytime you use CPSlib functions for synchronization.

`sync_routine` is only effective for the listed routines in the file in which it appears.

`task_private(`***`namelist`***`)`

This directive or pragma will privatize the variables and arrays specified in *namelist* for each task specified in the immediately following `begin_tasks`/`end_tasks` block. If a `task_private` data object is referenced within a task, it must have been assigned a value previously in that task. The privatized variables and arrays do not carry their values beyond the `end_tasks` directive or pragma. Refer to Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” for more information.

`thread_private(`***`namelist`***`)`

This Fortran directive will cause the variables and arrays specified in *namelist* to be treated as being `thread_private`. `thread_private` data objects map to unique `node_private` addresses for each thread of a process. In C and C++, `thread_private` is a storage class specifier. Refer to Chapter 5, “Memory classes,” for more information.

`thread_private_pointer(`***`namelist`***`)`

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointers to the allocated objects (specified in *namelist*) in `thread_private` memory, regardless of the memory classes to which the objects are allocated.

This directive applies only to Fortran 90 allocatable data objects and to Fortran 90-style allocatable data objects used in HP Fortran 77 programs. Refer to Chapter 5, “Memory classes,” for more information.

`unroll_and_jam[(unroll_factor=`*`n`*`)]`

This directive or pragma causes one or more noninnermost loops in the immediately following nest to be partially unrolled (to a depth of *n* if `unroll_factor` is specified), then fuses the resulting loops back together. It must be placed on a loop that ends up being noninnermost after any compiler-initiated interchanges. Refer to the section“Loop unroll and jam” on page 95 for more information.

C SGI directives

This appendix describes SGI directives that are supported in HP Fortran 90. It also provides a mapping between SGI directives and Exemplar directives. This appendix does not address any of the parallelization pragmas from SGI's Power C.

SGI Directives in HP Fortran 90

HP Fortran 90 supports the SGI compiler directives listed in this section to facilitate compiling and running SGI-native Fortran codes on HP platforms.

The following SGI directives are supported by HP Fortran 90.

Table 15 SGI Directives

`{!|C}*$*ASSERT DO`	`{!|C}*$*ASSERT DO PREFER`
`{!|C}$DOACROSS`	`{!|C}*$*[NO]VECTORIZE`
`{!|C}*$*NOINLINE`	

These SGI directives are supported with the restrictions noted in the following descriptions.

ASSERT DO(SERIAL)

Forces the immediately following loop to run serially (not in parallel).

Syntax

```
!*$* ASSERT DO(SERIAL)
```

The prefix `!*$*` may be specified as `C*$*` in fixed-format programs.

ASSERT DO(CONCURRENT)

Instructs the compiler to ignore any assumed data dependences in the immediately following loop. Unless known data dependences exist in the loop, the compiler parallelizes the loop when the directive `ASSERT DO(CONCURRENT)` is specified.

Syntax

```
!*$* ASSERT DO(CONCURRENT)
```

The prefix `!*$*` may be specified as `C*$*` in fixed-format programs.

ASSERT DO(VECTOR)

Causes the compiler to attempt to replace certain loops with calls to the math library whenever possible. This directive applies to the immediately following loop (or set of nested loops).

You must specify the `+Ovectorize` option for this directive to be effective.

Syntax

```
!*$* ASSERT DO(VECTOR)
```

The prefix `!*$*` may be specified as `C*$*` in fixed-format programs.

ASSERT DO PREFER(CONCURRENT)

Causes the immediately following `DO` loop to be run in parallel if no dependences or other conditions prevent the compiler from parallelizing it.

Syntax

```
!*$*ASSERT DOPREFER(CONCURRENT)
```

The prefix `!*$*` may be specified as `C*$*` in fixed-format programs.

ASSERT DO PREFER(SERIAL)

Forces the immediately following `DO` loop to run in serial (not parallel).

Syntax

```
!*$* ASSERT DO PREFER(SERIAL)
```

The prefix `!*$*` may be specified as `C*$*` in fixed-format programs.

DOACROSS

The `C$DOACROSS` directive causes the compiler to generate parallel code for the immediately following `DO` loop.

Syntax

`!$DOACROSS [`*clause* `[,` *clause*`] ...]`

The prefix `!$` may be specified as `C$` in fixed-format programs.

clause can have the following values:

- `IF(`*expression*`)`

  When *clause* is `IF`, the logical expression (*expression*) determines whether the immediately following `DO` loop is parallelized. If *expression* evaluates to true, the loop is parallelized. If *expression* is false, it is run serially.

- `SHARE(`*namelist*`)`

  For `SHARE`, variables listed in *namelist*, the same copy of the variable is used for all iterations of the loop.

- `LOCAL(`*namelist*`)`

  For `LOCAL`, variables listed in *namelist*, an uninitialized copy of the variable is used for each iteration of the loop.

- `LASTLOCAL(`*namelist*`)`

  `LASTLOCAL` specifies that variables in *namelist* have their final values from the last iteration of the loop saved, for possible use following the loop (however, an uninitialized copy of the variable is used for each iteration of the loop).

- `MP_SCHEDTYPE=`*schedtype*

NOTE The `MP_SCHEDTYPE` clause is not supported in HP Fortran 90.

- `CHUNK=`*n*

  The `CHUNK` clause causes the loop to be divided into sections of *n* iterations, where *n* is the value specified. Each group of *n* iterations is executed independently. Once a process completes a group of iterations, it begins on the next available set of iterations.

  (Note that because the `MP_SCHEDTYPE` clause is not supported, when the `CHUNK` clause is specified a scheduling type of `DYNAMIC` is assumed.)

- `REDUCTION(`*scalarlist*`)`

NOTE The `REDUCTION` clause is not supported in HP Fortran 90.

NOINLINE [(*routinelist*)]

Specifies that the listed routines are not to be inlined.

Syntax

`!*$*NOINLINE [(`*routinelist*`)] {HERE|ROUTINE|GLOBAL}`

The prefix `!*$*` may be specified as `C*$*` in fixed-format programs.

routinelist is a comma-separated list of routine names. If *routinelist* is not specified, the directive applies to all routines.

If `HERE` is specified, the directive applies only to the next line of code.

`ROUTINE` indicates that occurrences of the listed routines should not be inlined within the current routine.

`GLOBAL` causes the listed routines to not be inlined throughout the entire source file.

`[NO]VECTORIZE`

Causes the compiler to attempt to replace certain loops with calls to the math library whenever possible. This directive applies to the immediately following loop (or set of nested loops).

You must specify the `+Ovectorize` option for this directive to be effective.

Syntax

```
!*$* [NO]VECTORIZE
```

The prefix `!*$*` may be specified as `C*$*` in fixed-format programs.

SGI directives and their Exemplar equivalents

This section presents the SGI Fortran 77 directives that can be mapped to Exemplar directive(s) and/or command-line options. Each SGI directive is briefly described and its corresponding Exemplar Fortran directive, directive combination, and/or command-line option are given. HP Fortran 90 directly supports the SGI directives described in the section “SGI Directives in HP Fortran 90” on page 353. For more information on the Exemplar directives, follow the cross references to their respective descriptions. The SGI directives listed in this section are available only in SGI’s Power Fortran 77.

NOTE The mappings between SGI and Exemplar directives presented in this section are approximate; there may be slight differences in functionality.

In the sections below, *namelist* represents a comma-separated list of variable names. Any Exemplar directive mentioned below is available when compiling with `+Oparallel` at `+O3` and above; additionally, in Fortran 77 Version 1.2.3 and C Version 1.2.3, the `+Onoexemplar_model` option must not be specified.

The SGI directives discussed in this section are:

- `C$DOACROSS [`*clause* `[,` *clause*`] ...]`
- `C*$*ASSERT DO(SERIAL)`
- `C*$*ASSERT DO(CONCURRENT)`
- `C*$*ASSERT DO PREFER(SERIAL)`
- `C*$*ASSERT DO PREFER(CONCURRENT)`
- `C*$*ASSERT NO RECURRENCE(`*variable*`)`
- `C*$*CONCURRENTIZE`
- `C*$*NOCONCURRENTIZE`

C$DOACROSS [*clause* [, *clause*] ...]

This directive causes the SGI compiler to parallelize the immediately following `DO` loop.

Valid values for the optional *clause* of `C$DOACROSS` are given below. These clauses may be the default behavior for the Exemplar compilers or may require you to use additional Exemplar directives or directive attributes, as indicated in each description.

IF (*logical_expression*)

Determines if the following loop is executed in parallel or serially. If *logical_expression* is true, the loop is parallelized; otherwise, the loop runs serially.

Equivalent Exemplar directive: This clause has no equivalent Exemplar directive.

Although the Exemplar compilers do not have a directive that is equivalent to SGI's `IF` clause, they do provide the `DYNSEL(`*trip_count*=*n*`)` directive. This directive allows you to specify that a loop should (or should not) run in parallel based on the iteration count given by the integer *n*. For more information on the `DYNSEL` directive, see the section "`dynsel[(`*trip_count*=*n*`)]`" on page 341.

{LOCAL | PRIVATE} (*namelist*)

Specifies that each variable in *namelist* must be a thread-private variable for the loop.

Equivalent Exemplar directive: LOOP_PARALLEL (IVAR attribute may be needed); LOOP_PRIVATE may be needed

If the variable from the LOCAL (or PRIVATE) clause is not the primary loop induction variable, place it in a LOOP_PRIVATE argument list and use the LOOP_PARALLEL directive on the loop. For more information on the LOOP_PRIVATE directive, see the section "loop_private(*namelist*)" on page 344.

If the variable is the primary loop induction variable, specify it as the induction variable using the IVAR attribute to the LOOP_PARALLEL directive. For more information on the LOOP_PARALLEL directive, see the section "loop_parallel[(*attribute_list*)]" on page 343.

{SHARE | SHARED} (*namelist*)

Causes all iterations of the loop to use the same copy of the variables given in *namelist*.

Equivalent Exemplar directive: default when using LOOP_PARALLEL

The SHARE (or SHARED) clauses represent the default behavior of the Exemplar compilers when using the LOOP_PARALLEL directive. You only need to use LOOP_PARALLEL to achieve this behavior.

{LASTLOCAL | LAST LOCAL} (*namelist*)

Causes each iteration of the loop to have its own copy of the variables given in *namelist* and saves the variable's last value. (A variable's last value in a loop of *n* iterations is the value it is assigned in the *n*th iteration.)

Equivalent Exemplar directive combination:
LOOP_PARALLEL, LOOP_PRIVATE(*namelist*), SAVE_LAST(*namelist*)

To duplicate this behavior in an Exemplar compiler, place the variables in *namelist* in the argument lists of both the LOOP_PRIVATE and SAVE_LAST directives and apply the LOOP_PARALLEL directive to the loop.

Consider the following example:

```
C$DOACROSS LASTLOCAL(ATEMP)
      DO I = 1, 1000
        X(I) = Y(I) + Z(I)
        ATEMP = Z(I)
      ENDDO
```

Using Exemplar directives, this example is as follows:

```
C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(ATEMP), SAVE_LAST(ATEMP)
      DO I = 1, 1000
        X(I) = Y(I) + Z(I)
        ATEMP = Z(I)
      ENDDO
```

For more information on the `LOOP_PARALLEL` directive, see the section “`loop_parallel[(`*attribute_list*`)]`” on page 343. For more information on the `LOOP_PRIVATE` directive, see the section “`loop_private(`*namelist*`)`” on page 344. For more information on the `SAVE_LAST` directive, see the section “`save_last[(`*list*`)]`on page 350.

`REDUCTION (`***namelist***`)`

Specifies that the variables given in *namelist* are involved in a reduction operation.

Equivalent Exemplar directive:
`LOOP_PARALLEL`, `REDUCTION(`*namelist*`)`

To duplicate this behavior in an Exemplar compiler, place the variables in *namelist* in the argument list of the `REDUCTION` directive and apply the `LOOP_PARALLEL` directive to the loop. For more information on the `LOOP_PARALLEL` directive, see the section “`loop_parallel[(`*attribute_list*`)]`” on page 343. For more information on the `REDUCTION` directive, see the section “`reduction(`*namelist*`)`” on page 350.

MP_SCHEDTYPE=*mode*

Schedules work in a parallel loop according to *mode*, where *mode* has one of the following values:

{SIMPLE | STATIC}

Schedules approximately equal-sized contiguous chunks of iterations to the threads.

Equivalent Exemplar directive: default when using LOOP_PARALLEL

This mode represents the default behavior of the Exemplar compilers when using the LOOP_PARALLEL directive. You only need to use LOOP_PARALLEL to achieve this behavior.

DYNAMIC

Causes threads to compete for CHUNK-sized assignments, where CHUNK is the SGI directive clause.

Equivalent Exemplar directive: This clause has no equivalent Exemplar directive.

{INTERLEAVE | INTERLEAVED}

Distributes each chunk of iterations among the available threads by interleaving. The number of consecutive iterations given to each thread is determined by SGI's CHUNK clause, which is shown later in this list.

Equivalent Exemplar directive: LOOP_PARALLEL (CHUNK_SIZE=*n*)

Use the CHUNK_SIZE=*n* attribute to LOOP_PARALLEL (where *n* is an integer constant) to duplicate this behavior. For more information on the LOOP_PARALLEL directive, see the section "loop_parallel[(*attribute_list*)]" on page 343.

{GUIDED | GSS}

Dynamically allocates chunks of iterations to threads; the chunks initially distributed are larger than the chunks distributed at the end.

Equivalent Exemplar directive: This clause has no equivalent Exemplar directive.

`RUNTIME`

Instructs the compiler to use environment variables to manage scheduling.

Equivalent Exemplar directive: This clause has no equivalent Exemplar directive.

{CHUNK=*integer_expression* | BLOCKED *integer_expression*}

Specifies that *integer_expression* or fewer iterations are to be executed by each thread.

Equivalent Exemplar directive:
`LOOP_PARALLEL (CHUNK_SIZE=`*n*`)`, when applicable

If `CHUNK` or `BLOCKED` is used with `INTERLEAVE` (or `INTERLEAVED`), use the `LOOP_PARALLEL` directive with the attribute `CHUNK_SIZE=`*n*, where *n* is an integer constant. Otherwise, there is no Exemplar equivalent. (Although these SGI clauses accept integer expressions, the equivalent Exemplar directive requires integer constants.)

C*$*ASSERT DO(SERIAL)

This directive instructs the SGI compiler to run the following loop serially.

Equivalent Exemplar directive: `NO_PARALLEL`

Use the Exemplar `NO_PARALLEL` directive to duplicate the SGI behavior. `NO_PARALLEL` applies only to the following loop; it does not affect any loops enclosing the following loop, as the SGI directive does. For more information on the `NO_PARALLEL` directive, see the section “`no_parallel`” on page 346.

C*$*ASSERT DO(CONCURRENT)

This directive forces the SGI compiler to ignore any apparent dependences in the following loop that could prevent the loop from being parallelized.

Equivalent Exemplar directive: `LOOP_PARALLEL`

Use the `LOOP_PARALLEL` directive to duplicate this behavior in an Exemplar compiler. For more information on the `LOOP_PARALLEL` directive, see the section “`loop_parallel[(`*attribute_list*`)]`” on page 343.

C*$*ASSERT DO PREFER(SERIAL)

This directive instructs the SGI compiler to execute the following `DO` loop serially.

Equivalent Exemplar directive: `NO_PARALLEL`

Use the `NO_PARALLEL` directive to duplicate this behavior in an Exemplar compiler. For more information on the `NO_PARALLEL` directive, see the section “`no_parallel`” on page 346.

C*$*ASSERT DO PREFER(CONCURRENT)

Used in a loop nest, this SGI directive requests that the compiler run a particular loop in the nest in parallel.

Equivalent Exemplar directive: `PREFER_PARALLEL`

Use the `PREFER_PARALLEL` directive to duplicate this behavior in an Exemplar compiler. For more information on the `PREFER_PARALLEL` directive, see the section “`prefer_parallel[(`*attribute_list*`)]`” on page 349.

C*$*ASSERT NO RECURRENCE(*variable*)

This directive instructs the SGI compiler to ignore all data dependences (apparent and real) associated with *variable*.

Equivalent Exemplar directive: LOOP_PRIVATE(*namelist*)

Use the LOOP_PRIVATE directive, placing *variable* in LOOP_PRIVATE's *namelist*, to duplicate this behavior in an Exemplar compiler. For more information on the LOOP_PRIVATE directive, see the section "loop_private(*namelist*)" on page 344.

C*$*CONCURRENTIZE

This directive causes the SGI compiler to parallelize eligible loops.

Equivalent Exemplar directive or options: PREFER_PARALLEL or +O3 +Oparallel

If the SGI directive is used on a single loop, use PREFER_PARALLEL. If C*$*CONCURRENTIZE is used globally, compile using the command-line options +O3 +Oparallel—without specifying +Onoautopar. For more information on the PREFER_PARALLEL directive, see the section "prefer_parallel[(*attribute_list*)]" on page 349. For more information on the command-line options, see the sections "Optimization level options" on page 365 and "+O[no]parallel" on page 379.

C*$*NOCONCURRENTIZE

This directive prevents the SGI compiler from parallelizing the following loop.

Equivalent Exemplar directive or option: NO_PARALLEL or +Onoautopar

If the SGI directive is used on a single loop, use NO_PARALLEL. If it is used globally, use the command-line option +Onoautopar to duplicate the behavior in an Exemplar compiler. For more information on the NO_PARALLEL directive, see the section "no_parallel" on page 346. For more information on the +Onoautopar option, see the section "+O[no]autopar" on page 367.

D Optimization options

This appendix lists and briefly describes the optimization options available for use with Exemplar compilers. Refer to the section “Using the optimizer” on page 42 for information on coding guidelines that assist the optimizer. See the f90(1), f77(1), cc(1), and aCC(1) man pages for information on compiler options in general. The options described in this appendix are available in all Exemplar compilers unless otherwise stated.

Optimization level options

The options listed in this section specify the level of optimization desired.

`+O0`	(Machine instruction-level optimizations) Constant folding and simple register assignment
`+O1`	(Block-level optimizations) `+O0` optimizations, plus instruction scheduling and optimizations on basic blocks
`+O2`	(Routine-level optimizations) `+O1` optimizations, plus optimizations within a single subprogram; loop optimizations to reduce pipeline stalls; analysis of data flow, memory usage, loops, and expressions
`+O3`	(File-level optimizations) `+O2` optimizations, plus full optimizations across all subprograms (including inlining) within a single file; use of parallelism-related directives and pragmas from the Exemplar programming model when `+Oparallel` is also specified
`+O4`	(Cross-module optimizations) `+O3` optimizations, plus full optimizations across all files in the application that have been compiled at `+O4`; optimizations include

inlining across the entire application; optimizations are performed at link time (This option is not available in Fortran 90.)

Controlling specific optimizer features

At each optimization level, you can turn specific optimizations on or off using the `+O[no]`*optimization* option. The *optimization* parameter is the name of a specific optimization described below. The optional prefix `[no]` disables the specified optimization.

The following sections describe the optimizations that can be turned on or off, their defaults, and the optimization levels at which they may be used. In syntax descriptions, *namelist* represents a comma-separated list of names.

+O[no]aggressive

Optimization level(s): `+O2`, `+O3`, `+O4`

Default: `+Onoaggressive`

The `+O[no]aggressive` option enables optimizations that can result in significant performance improvement, but that can change a program's behavior. These optimizations include the optimizations invoked by the following advanced options (which are discussed separately in this appendix):

- `+OSIGNEDPOINTERS` (C and C++)
- `+oentrysched`
- `+Onofltacc`
- `+Olibcalls`
- `+Onoinitcheck`
- `+Ovectorize`

`+O[no]all`

Optimization level(s): can be used at any level

Default: `+Onoall`

The `+Oall` option performs maximum optimization, including aggressive optimizations and optimizations that can significantly increase compile time and memory usage. Specifying the `+Oall` option is equivalent to specifying the following list of options: `+O4 +Oaggressive +Onolimit`.

`+O[no]autopar`

Optimization level(s): `+O3`, `+O4` (`+Oparallel` must be specified to enable `+O[no]autopar`)

Default: `+Oautopar`

When used with `+Oparallel` option, `+Oautopar` (the default) causes the compiler to automatically parallelize loops that are safe to parallelize.

A loop is safe to parallelize if it has an iteration count that can be determined at runtime before loop invocation, and contains no loop-carried dependences, procedure calls, or I/O operations.
A loop-carried dependence exists when one iteration of a loop assigns a value to an address that is referenced or assigned on another iteration.

You can use Fortran directives and C pragmas to improve on the automatic optimizations and to assist the compiler in locating additional opportunities for parallelization.

When used with `+Oparallel`, the `+Onoautopar` option causes the compiler to parallelize only those loops marked by the `loop_parallel` or `prefer_parallel` directives or pragmas. Because the compiler does not automatically find parallel tasks or regions, user-specified task and region parallelization is not affected by this option.

Because parallelization takes places only at `+O3` and above, `+O[no]autopar` is useful only at `+O3` and above.

+O[no]conservative

Optimization level(s): +O2, +O3, +O4

Default: +Onoconservative

The +O[no]conservative option causes the optimizer to [not] make conservative assumptions about the code when optimizing it. Use +Oconservative when conservative assumptions are necessary due to the coding style, as with programs that are not standard-compliant. (Specifying +Oconservative disables any optimizations that assume standard-compliant code.)

+Oconservative is equivalent to +Onoaggressive.

+O[no]dataprefetch

Optimization level(s): +O2, +O3, +O4

Default: +Onodataprefetch

When +Odataprefetch is enabled, the optimizer will insert instructions within innermost loops to explicitly prefetch data from memory into the data cache. For cache lines containing data that will be written, +Odataprefetch prefetches the cache lines so that they are valid for both read and write access. Data prefetch instructions will be inserted only for data referenced within innermost loops using simple loop varying addresses (that is, in a simple arithmetic progression). It is only available for PA-RISC 2.0 targets.

The math library libm contains special prefetching versions of vector routines. If you have a PA-RISC 2.0 application that contains operations on arrays larger than 1 megabyte in size, using +Ovectorize in conjunction with +Odataprefetch may improve performance substantially.

Use the +Odataprefetch option for applications that have high data cache miss overhead.

+O[no]dynsel

Optimization level(s): +O3, +O4 (+Oparallel must be specified to enable +O[no]dynsel)

Default: +Odynsel

When specified with +Oparallel, +Odynsel enables workload-based dynamic selection. For parallelizable loops whose iteration counts are known at compile time, +Odynsel causes the compiler to generate either a parallel or a serial version of the loop—depending on which is more profitable.

This optimization also causes the compiler to generate both parallel and serial versions of parallelizable loops whose iteration counts are unknown at compile time. At runtime, the loop's workload is compared to parallelization overhead, and the parallel version is run only if it is profitable to do so.

The +Onodynsel option disables dynamic selection and tells the compiler that it is profitable to parallelize all parallelizable loops. The dynsel directive and pragma can be used to enable dynamic selection for specific loops when +Onodynsel is in effect. See the section "Dynamic selection" on page 109 for additional information.

+O[no]entrysched

Optimization level(s): +O1, +O2, +O3, +O4

Default: +Onoentrysched

The +Oentrysched option optimizes instruction scheduling on a procedure's entry and exit sequences. Enabling this option can speed up an application. The option affects unwinding in the entry and exit regions.

This option can change the behavior of programs that perform exception-handling or that handle asynchronous interrupts. The behavior of setjmp() and longjmp() is not affected.

+O[no]exemplar_model

Optimization level(s): +O0, +O1, +O2, +O3, +O4

Default: +Oexemplar_model

This option is available only in Exemplar Fortran 77 Version 1.2.3 and Exemplar C Version 1.2.3.

+Oexemplar_model (the default) causes the compiler to accept the Exemplar programming model. This option allows you to use the directives, pragmas, and associated command-line options that make up the programming model. At lower optimization levels (+O0, +O1, +O2), this option enables only the following components of the programming model:

- Synchronization directives (Fortran)
- Synchronization pragmas (C)
- Synchronization typedefs (C and C++)
- Memory class directives (Fortran)
- Memory storage class specifiers (C and C++)

At +O3 and +O4, using +Oexemplar_model enables (in addition to the features enabled at the lower levels) the parallelism-related directives and pragmas. See Appendix B, “Exemplar compiler directives and pragmas,” for additional information.

The +Onoexemplar_model option turns off support for the Exemplar programming model. If you use this option, directives and pragmas from the Exemplar programming model will be ignored.

+O[no]fail_safe

Optimization level(s): +O1, +O2, +O3, +O4

Default: +Ofail_safe

The +Ofail_safe option allows compilations with internal optimization errors to continue by issuing a warning message and restarting the compilation at +O0.

Use +Onofail_safe when you want the internal optimization errors to abort your compilation.

The `+Ofail_safe` option is disabled when compiling for parallelization (that is, when you specify `+Oparallel` with `+O3` or `+O4`).

+O[no]fastaccess

Optimization level(s): `+O0`, `+O1`, `+O2`, `+O3`, `+O4`

Default: `+Onofastaccess` at `+O0`, `+O1`, `+O2` and `+O3`;
`+Ofastaccess` at `+O4`

The `+Ofastaccess` option optimizes for fast access to global data items.

Use `+Ofastaccess` to improve execution speed at the expense of longer compile times.

+O[no]fltacc

Optimization level(s): `+O2`, `+O3`, `+O4`

Default: neither (See Table 16.)

The `+O[no]fltacc` option [enables] disables optimizations that cause imprecise floating-point results.

Use `+Onofltacc` to improve execution speed at the expense of floating-point precision. The `+Onofltacc` option allows the compiler to perform floating-point optimizations that are algebraically correct but that may result in numerical differences. In general, these differences will be insignificant. The `+Onofltacc` option also enables the optimizer to generate Fused Multiply-Add (FMA) instructions.

`+Ofltacc` disables optimizations that cause imprecise floating-point results. Specifying `+Ofltacc` disables the generation of FMA instructions as well as other floating-point optimizations. Use `+Ofltacc` if it is important that the compiler evaluates floating-point expressions according to the order specified by the language standard.

If you are optimizing code at `+O2` or higher and do not specify `+Onofltacc` or `+Ofltacc`, the optimizer will use FMA instructions, but will not perform floating-point optimizations that involve expression reordering. FMA is implemented by the PA-8*x*00 instructions `FMPYFADD` and `FMPYNFADD` and improves performance but occasionally produces results that may differ in accuracy from results produced by code without FMA. In general, the differences are slight.

Table 16 presents a summary of the preceding information.

Table 16 **+O[no]fltacc and floating-point optimizations**

Option specified*	FMA optimizations	Other floating-point optimizations
+Ofltacc	Disabled	Disabled
+Onofltacc	Enabled	Enabled
neither option is specified	Enabled	Disabled

*+O[no]fltacc is only available at +O2 and above.

+O[no]global_ptrs_unique[=namelist]

Optimization level(s): +O2, +O3, +O4

Default: +Onoglobal_ptrs_unique

This option is not available in C++.

Use this C compiler option to identify unique global pointers, so that the optimizer can generate more efficient code in the presence of unique pointers, for example by using copy propagation and common subexpression elimination. A global pointer is unique if it does not alias with any variable in the entire program.

This option supports a comma-separated list of unique global pointer variable names, represented by *namelist* in +O[no]global_ptrs_unique[=*namelist*]. If *namelist* is not specified, using +O[no]global_ptrs_unique informs the compiler that all [no] global pointers are unique.

The example below states that no global pointers are unique except a and b:

```
+Oglobal_ptrs_unique=a,b
```

The next example says that all global pointers are unique except a and b:

```
+Onoglobal_ptrs_unique=a,b
```

`+O[no]info`

Optimization level(s): `+O0`, `+O1`, `+O2`, `+O3`, `+O4`

Default: `+Onoinfo`

`+Oinfo` displays informational messages about the optimization process. This option can be used at all optimization levels, but is most useful at `+O3` and `+O4`.

`+O[no]initcheck`

Optimization level(s): `+O2`, `+O3`, `+O4`

Default: unspecified

The initialization checking feature of the optimizer has three possible states: on, off, or unspecified. When on (`+Oinitcheck`), the optimizer initializes to zero any local, scalar, nonstatic variables that are uninitialized with respect to at least one path leading to a use of the variable.

When off (`+Onoinitcheck`), the optimizer issues warning messages when it discovers definitely uninitialized variables, but does not initialize them.

When unspecified, the optimizer initializes to zero any local, scalar, nonstatic variables that are definitely uninitialized with respect to all paths leading to a use of the variable.

`+O[no]inline[=`*namelist*`]`

Optimization level(s): `+O3`, `+O4`

Default: `+Oinline`

The Fortran 90 and aC++ compilers accept only `+O[no]inline`; no *namelist* values are accepted.

When `+Oinline` is specified without a name list, any function can be inlined. For inlining to be successful, follow the prototype definitions for function calls in the appropriate header files.

When specified with a name list, the named functions are important candidates for inlining. For example, saying

```
+Oinline=foo,bar +Onoinline
```

indicates that inlining be strongly considered for `foo` and `bar`; all other routines will not be considered for inlining because `+Onoinline` is given.

When this option is disabled with a name list, the compiler will not consider the specified routines as candidates for inlining. For example, by stating:

```
+Onoinline=baz,x
```

indicates that inlining should not be considered for `baz` and `x`; all other routines will be considered for inlining because `+Oinline` is the default.

Use this option when you need to precisely control which subprograms are inlined. Use of this option can be guided by knowledge of the frequency with which certain routines are called and may be warranted by code size concerns.

`+Oinline_budget=`*n*

Optimization level(s): `+O3`, `+O4`

Default: `+Oinline_budget=100`

In `+Oinline_budget=`*n*, *n* is an integer in the range 1 to 1000000 that specifies the level of aggressiveness, as follows:

n = 100	Default level of inlining.
n > 100	More aggressive inlining. The optimizer is less restricted by compilation time and code size when searching for eligible routines to inline.
n = 1	Only inline if it reduces code size.

The `+Onolimit` and `+Osize` options also affect inlining. Specifying the `+Onolimit` option implies specifying `+Oinline_budget=200`. The `+Osize` option implies `+Oinline_budget=1`. Note, however, that the `+Oinline_budget` option takes precedence over both of these options. This means that you can override the effects on inlining of the `+Onolimit` and `+Osize` options by specifying the `+Oinline_budget` option on the same compile line.

+O[no]libcalls

Optimization level(s): +O0, +O1, +O2, +O3, +O4

Default: +Onolibcalls at +O0 and +O1;
+Olibcalls at +O2, +O3, and +O4

Use the +Olibcalls option to increase the runtime performance of code that calls standard library routines in simple contexts. The +Olibcalls option expands the following library calls inline:

- strcpy()
- sqrt()
- fabs()
- alloca()

Inlining will take place only if the function call follows the prototype definition in the appropriate header file. A single call to printf() may be replaced by a series of calls to putchar(). Calls to sprintf() and strlen() may be optimized more effectively, including elimination of some calls producing unused results. Calls to setjmp() and longjmp() may be replaced by their equivalents _setjmp() and _longjmp(), which do not manipulate the process's signal mask.

Using the +Olibcalls option invokes millicode versions of frequently called math functions. Currently, there are millicode versions for the following functions:

acos	asin	atan	atan2
cos	exp	log	log10
pow	sin	tan	

See the *HP-UX Floating-Point Guide* for the most up-to-date listing of the math library functions.

Use +Olibcalls to improve the performance of selected library routines only when you are not performing error checking for these routines. The calling code must not expect to access ERRNO after the function's return.

Using +Olibcalls with +Ofltacc will give different floating-point calculation results than those given using +Olibcalls without +Ofltacc.

+O[no]limit

Optimization level(s): +O2, +O3, +O4

Default: +Olimit

The +Olimit option suppresses optimizations that significantly increase compile-time or that can consume a considerable amount of memory.

The +Onolimit option allows optimizations to be performed regardless of their effects on compile-time and memory usage. Specifying the +Onolimit option implies specifying +Oinline_budget=200. See the section "+Oinline_budget=*n*" on page 374 for more information.

+O[no]loop_block

Optimization level(s): +O3, +O4

Default: +Onoloop_block

This option is not available in Fortran 77 Version 1.2.3 or
C Version 1.2.3.

The +O[no]loop_block option enables [disables] blocking of eligible loops for improved cache performance. The +Onoloop_block option disables both automatic and directive-specified loop blocking. For more information on loop blocking, see the section "Loop blocking" on page 83.

+O[no]loop_transform

Optimization level(s): +O3, +O4

Default: +Oloop_transform

The +O[no]loop_transform option enables [disables] transformation of eligible loops for improved cache performance. The most important transformation is the reordering (interchange) of nested loops to make the inner loop unit stride, resulting in fewer cache misses. The other transformations affected by +O[no]loop_transform are loop distribution, loop blocking, loop fusion, loop unroll, and loop unroll and jam. See Chapter 3, "Compiler optimizations," for information on loop transformations.

+Onoloop_transform may be a helpful option if you experience any problem while using +Oparallel.

+O[no]loop_unroll[=*unroll factor*]

Optimization level(s): +O2, +O3, +O4

Default: +Oloop_unroll=4

The +Oloop_unroll option turns on loop unrolling. When you use +Oloop_unroll, you can also suggest the *unroll factor* to control the code expansion. The default unroll factor is 4; in other words, the loop body is replicated four times. By experimenting with different factors, you may improve the performance of your program. In some cases, the compiler uses its own unroll factor. The +Onoloop_unroll option turns off partial and complete unrolling. Loop unrolling improves efficiency by eliminating loop overhead and can create opportunities for other optimizations, such as improved register use and more efficient scheduling. See the section “Loop unrolling” on page 61 for more information on unrolling.

+O[no]loop_unroll_jam

Optimization level(s): +O3, +O4

Default: +Oloop_unroll_jam

This option is not available in Fortran 77 Version 1.2.3 or
C Version 1.2.3.

The +O[no]loop_unroll_jam option enables [disables] loop unrolling and jamming. The +Onoloop_unroll_jam option disables both automatic and directive-specified unroll and jam. Loop unrolling and jamming increases register exploitation. For more information on the unroll and jam optimization, see the section “Loop unroll and jam” on page 95.

+O[no]moveflops

Optimization level(s): +O2, +O3, +O4

Default: +Omoveflops

Allows [disallows] moving conditional floating point instructions out of loops. The behavior of floating-point exception handling may be altered by this option.

Use +Onomoveflops if floating-point traps are enabled and you do not want the behavior of floating-point exceptions to be altered by the relocation of floating-point instructions.

+O[no]nodepar

Optimization level(s): +O3, +O4 (+Oparallel must be specified to enable +O[no]nodepar)

Default: +Ononodepar

The +Ononodepar option disables node-parallelism by causing the compiler to generate code for a single-node machine. When this option is used, serial code is generated for node-parallel constructs. Specifying the +Ononodepar option prevents the compiler from implementing node-parallelism, but allows the implementation of both automatic and directive-specified thread-parallelism.

The +Onodepar option causes the compiler to perform node-parallelism where it has been specified using the nodes attribute with the loop_parallel, prefer_parallel, parallel, or begin_tasks directives or pragmas. Also, the +Onodepar option causes the compiler to honor the node_trip_count attribute to the dynsel directive or pragma.

The +O[no]nodepar option is effective only when specified with the +Oparallel option at +O3 and above.

+O[no]parallel

Optimization level(s): +O3, +O4

Default: +Onoparallel

NOTE If you compile one or more files in an application using +Oparallel, then the application must be linked (using the compiler driver) with the +Oparallel option to link in the proper start-up files and runtime support.

The +Oparallel option causes the compiler to:

- Honor the directives and pragmas of the Exemplar programming model that involve parallelism, such as begin_tasks, loop_parallel, and prefer_parallel. (These directives and pragmas are not recognized in Fortran 77 Version 1.2.3 or C Version 1.2.3 if the +Onoexemplar_model option is also specified.)
- Look for opportunities for parallel execution in loops.

The following methods can be used to specify the number of processors used in executing your parallel programs:

- loop_parallel(max_threads=*m*) directive and pragma
- prefer_parallel(max_threads=*m*)directive and pragma

  For more information on the directives and pragmas see the chapters “Basic shared-memory programming” and “Advanced shared-memory programming.” (These pragmas are not available in the aC++ compiler.)
- MP_NUMBER_OF_THREADS environment variable, which is read at runtime by your program. If this variable is set to some positive integer *n*, your program executes on *n* processors; *n* must be less than or equal to the number of processors in the system where the program is executing.

The +Oparallel option is valid only at optimization level +O3 and above. For information on parallelization, see the section "Parallelization" on page 100.

Using the +Oparallel option disables +Ofail_safe, which is on by default. See the section "+O[no]fail_safe" on page 370 for more information.

The +Onoparallel option is the default for all optimization levels. This option disables automatic and directive-specified parallelization.

+O[no]parmsoverlap

Optimization level(s): +O2, +O3, +O4

Default for Fortran: +Onoparmsoverlap

Default for C: +Oparmsoverlap

The option +Oparmsoverlap causes the optimizer to assume that the actual arguments of function calls overlap in memory.

+O[no]pipeline

Optimization level(s): +O2, +O3, +O4

Default: +Opipeline

Enables [disables] software pipelining.

Use +Onopipeline if program size is more important than execution speed.

Software pipelining is particularly useful for loops that contain arithmetic operations on REAL or REAL*8 variables in Fortran or on float or double variables in C and C++.

+O[no]procelim

Optimization level(s): +O0, +O1, +O2, +O3, +O4

Default: +Onoprocelim at +O0, +O1, +O2, +O3;
+Oprocelim at +O4

When +Oprocelim is specified, procedures that are not referenced by the application are eliminated from the output executable file. The +Oprocelim option reduces the size of the executable file, especially when optimizing at +O3 and +O4, at which inlining may have removed all of the calls to some routines.

When +Onoprocelim is specified, procedures that are not referenced by the application are not eliminated from the output executable file.

If the +Oall option is enabled, the +Oprocelim option is enabled.

+O[no]ptrs_ansi

Optimization level(s): +O2, +O3, +O4

Default: +Onoptrs_ansi

This option is not available in C++.

Use the C compiler option +Optrs_ansi to make the following two assumptions, which the more aggressive +Optrs_strongly_typed does not make:

- int *p is assumed to point to an int field of a struct or union.
- char * is assumed to point to any type of object.

When both +Optrs_ansi and +Optrs_strongly_typed are specified, +Optrs_ansi takes precedence.

+O[no]ptrs_strongly_typed

Optimization level(s): +O2, +O3, +O4

Default: +Onoptrs_strongly_typed

This option is not available in C++.

Use the C compiler option +Optrs_strongly_typed when pointers are type-safe. The optimizer can use this information to generate more efficient code.

Type-safe (that is, strongly-typed) pointers are pointers to a specific type that only point to objects of that type. For example, a pointer declared as a pointer to an int is considered type-safe if that pointer points to an object only of type int.

Based on the type-safe concept, a set of groups are built based on object types. A given group includes all the objects of the same type.

The term *type-inferred aliasing* is a concept that means any pointer of a type in a given group (of objects of the same type) can only point to any object from the same group; it cannot point to a typed object from any other group.

Type casting to a different type violates type-inferring aliasing rules. See Example 2 below.

Dynamic casting is allowed. See Example 3 below.

For finer detail, see the use the [no]ptrs_strongly_typed pragma as discussed in Appendix A, “Standard HP compiler directives and pragmas.”

Example 1: How data types interact

The optimizer generally spills all global data from registers to memory before any modification to global variables or any loads through pointers. However, you can instruct the optimizer on how data types interact so that it can generate more efficient code.

Consider the following example (line numbers are provided for reference):

```
1  int *p;
2  float *q;
3  int a,b,c;
4  float d,e,f;
5  foo()
6  {
7    for (i=1;i<10;i++) {
8               d=e;
9              *p=...;
10              e=d+f;
11              f=*q;
12   }
13 }
```

With `+Onoptrs_strongly_typed` turned on, the pointers `p` and `q` will be assumed to be disjoint because the types they point to are different types. Without type-inferred aliasing, `*p` is assumed to invalidate all the definitions. So, the use of `d` and `f` on line 10 have to be loaded from memory. With type-inferred aliasing, the optimizer can propagate the copy of `d` and `f` and thus avoid two loads and two stores.

This option can be used for any application involving the use of pointers, where those pointers are type safe. To specify when a subset of types are type-safe, use the `ptrs_strongly_typed` pragma. The compiler issues warnings for any incompatible pointer assignments that may violate the type-inferred aliasing rules discussed in the section “C aliasing options” on page 394.

Example 2: Unsafe type cast

Any type cast to a different type violates type-inferred aliasing rules. Do not use +Optrs_strongly_typed with code that has these “unsafe” type casts. Use the [no]ptrs_strongly_typed pragma to prevent the application of type-inferred aliasing to the unsafe type casts.

```
struct foo{
        int a;
        int b;
      } *P;

     struct bar {
        float a;
        int b;
        float c;
      } *q;

     P = (struct foo *) q;
        /* Incompatible pointer assignment
      through type cast */
```

Example 3: Generally applying type aliasing

Dynamic casting is allowed with +Optrs_strongly_typed or +Optrs_ansi. A pointer dereference is called dynamic casting if a cast is applied on the pointer to a different type.

In the example below, type-inferred aliasing is applied on P generally, not just to the particular dereference. Type-aliasing will be applied to any other dereferences of P.

```
     struct s {
        short int a;
        short int b;
        int c;
      } *P
      * (int *)P = 0;
```

For more information about type aliasing see the section “C aliasing options” on page 394.

+O[no]ptrs_to_globals[=*namelist*]

Optimization level(s): +O2, +O3, +O4

Default: +Optrs_to_globals

This option is not available in C++.

By default, global variables are conservatively assumed to be modified anywhere in the program. Use the C compiler option +Onoptrs_to_globals to specify which global variables are not modified through pointers, so that the optimizer can make your program run more efficiently by incorporating copy propagation and common subexpression elimination.

This option can be used to specify all global variables as not modified via pointers, or to specify a comma-separated list of global variables as not modified via pointers.

Note that the on state for this option disables some optimizations, such as aggressive optimizations on the program's global symbols.

For example, use the command-line option +Onoptrs_to_globals=a,b,c to specify global variables a, b, and c as not being accessed through pointers. No pointer can access these global variables. The optimizer will perform copy propagation and constant folding because storing to *p will not modify a or b.

```
int a, b, c;
      float *p;
      foo()
      {
         a = 10;
         b = 20;
        *p = 1.0;
         c = a + b;
      }
```

If all global variables are unique, use the +Onoptrs_to_globals option without listing the global variables (in other words, without using *namelist*).

In the example below, the address of b is taken. This means b can be accessed indirectly through the pointer. You can still use +Onoptrs_to_globals as:
+Onoptrs_to_globals +Optrs_to_globals=b.

```
int b,c;
int *p;

p=&b;

foo()
```

For more information about type aliasing see the section “C aliasing options” on page 394.

+O[no]regreassoc

Optimization level(s): +O2, +O3, +O4

Default: +Oregreassoc

The +O[no]regreassoc option enables [disables] register reassociation. Register reassociation is a technique for folding and eliminating integer arithmetic operations within loops, especially those used for array address computations. This optimization is a code-improving transformation that supplements loop-invariant code motion and strength reduction. When performed in conjunction with software pipelining, register reassociation can yield significant performance improvement.

`+O[no]report[=`*report_type*`]`

Optimization level(s): `+O3`, `+O4`

Default: `+Onoreport`

`+Oreport[=`*report_type*`]` specifies the contents of the Optimization Report. Values of *report_type* and the Optimization Reports they produce are shown in Table 17.

Table 17 Optimization Report contents

***report_type* value**	**Report contents**
`all`	Loop Report and Privatization Table
`loop`	Loop Report
`private`	Loop Report and Privatization Table
report_type not given (default)	Loop Report

The Loop Report gives information on optimizations performed on loops and calls. Using `+Oreport` (without =*report_type*) also produces the Loop Report.

The Privatization Table provides information on loop variables that are privatized by the compiler.

The `+Oreport[=`*report_type*`]` option is active only at `+O3` and above. The `+Onoreport` option does not accept any of the *report_type* values. For more information about the Optimization Report, refer to Appendix E, “Optimization Report.”

The option `+Oinfo` also displays information on the various optimizations being performed by the compilers. `+Oinfo` can be used at any optimization level but is most useful at `+O3` and above. The default, at all optimization levels, is `+Onoinfo`.

+O[no]sharedgra

Optimization level(s): +O2, +O3, +O4

Default: +Osharedgra

The +Onosharedgra option disables global register allocation for shared-memory variables that are visible to multiple threads. This option may help if a variable shared among parallel threads is causing wrong answers. See the section “Global register allocation” on page 52 for more information.

Global register allocation (+Osharedgra) is enabled by default at optimization level +O2 and higher.

+O[no]signedpointers

Optimization level(s): +O2, +O3, +O4

Default: +Onosignedpointers

The C and C++ option +O[no]signedpointers requests that the compiler perform [does not perform] optimizations related to treating pointers as signed quantities. Applications that allocate shared memory and that compare a pointer to shared memory with a pointer to private memory may run incorrectly if this optimization is enabled.

Use +Osignedpointers to improve application runtime speed.

+O[no]size

Optimization level(s): +O2, +O3, +O4

Default: +Onosize

The +Osize option suppresses optimizations that significantly increase code size. Specifying +Osize implies specifying +Oinline_budget=1. See the section “+Oinline_budget=*n*” on page 374 for additional information.

The +Onosize option does not prevent optimizations that can increase code size.

`+O[no]static_prediction`

Optimization level(s): `+O0`, `+O1`, `+O2`, `+O3`, `+O4`

Default: `+Onostatic_prediction`

`+Ostatic_prediction` turns on static branch prediction for PA-RISC 2.0 targets.

PA-RISC 2.0 has two means of predicting which way conditional branches will go:

- Dynamic branch prediction

  Dynamic branch prediction uses a hardware history mechanism to predict future executions of a branch from its last three executions. It is transparent and quite effective, unless the hardware buffers involved are overwhelmed by a large program with poor locality.

- Static branch prediction

  With static branch prediction on, each branch is predicted based on implicit hints encoded in the branch instruction itself. Static branch prediction's role is to handle large codes with poor locality for which the small dynamic hardware facility will prove inadequate. Use `+Ostatic_prediction` to better optimize large programs with poor instruction locality, such as operating system and database code.

+O[no]vectorize

Optimization level(s): +O3, +O4

Default: +Onovectorize

This option is not available in C++.

+Ovectorize allows the compiler to replace certain loops with calls to vector routines.

Use +Ovectorize to increase the execution speed of loops.

When +Onovectorize is specified, loops are not replaced with calls to vector routines.

Because the +Ovectorize option may change the order of floating-point operations in an application, it may also change the results of those operations slightly. See the *HP-UX Floating-Point Guide* for details.

The math library contains special prefetching versions of vector routines. If you have a PA2.0 application that contains operations on very large arrays (larger than 1 megabyte in size), using +Ovectorize in conjunction with +Odataprefetch may improve performance substantially.

+Ovectorize is also included as part of the +Oaggressive and +Oall options.

+O[no]volatile

Optimization level(s): +O1, +O2, +O3, +O4

Default: +Onovolatile

The C and C++ option +Ovolatile implies that memory references to global variables cannot be removed during optimization.

The +Onovolatile option implies that all globals are not of volatile class. This means that references to global variables can be removed during optimization.

Use this option to control the volatile semantics for all global variables.

`+O[no]whole_program_mode`

Optimization level(s): `+O4`

Default: `+Onowhole_program_mode`

This option is not available in Fortran 90 or C++.

The `+Owhole_program_mode` option enables the assertion that only the files that are compiled with this option directly reference any global variables and procedures that are defined in these files. In other words, this option asserts that there are no unseen accesses to the globals.

When this assertion is in effect, the optimizer can hold global variables in registers longer and delete inlined or cloned global procedures.

All files compiled with `+Owhole_program_mode` must also be compiled with `+O4`. If any of the files were compiled with `+O4` but were not compiled with `+Owhole_program_mode`, the linker disables the assertion for all files in the program.

The default, `+Onowhole_program_mode`, disables the assertion.

Use this option to increase performance speed, but only when you are certain that only the files compiled with `+Owhole_program_mode` directly access any globals that are defined in these files.

`+tm` ***target***

Optimization level(s): `+O0`, `+O1`, `+O2`, `+O3`, `+O4`

Default *target* value: corresponds to the machine on which you invoke the compiler

This option specifies the target machine architecture for which compilation is to be performed. Using this option causes the compiler to perform architecture-specific optimizations. *target* takes one of the following values:

- `K7200` to specify K-Class servers using PA-7200 processors
- `K8000` to specify K-Class servers using PA-8000 processors
- `V2200` to specify V2200 servers
- `S2000` to specify S2000 servers
- `X2000` to specify X2000 servers

Exemplar Fortran 77 and Exemplar C Version 1.2.3 accept only the `S2000` and `X2000` *target* values.

This option is valid at all optimization levels. The default *target* value corresponds to the machine on which you invoke the compiler.

Using the `+tm` *target* option implies `+DA` and `+DS` settings as described in Table 18. `+DA`*architecture* causes the compiler to generate code for the architecture specified by *architecture*. `+DS`*model* causes the compiler to use the instruction scheduler tuned to *model*. See the f77(1) man page or the cc(1) man page for more information on the `+DA` and `+DS` options.

Table 18 **`+tm` *target* and `+DA`/`+DS`**

***target* value specified**	**`+DA`*architecture* implied**	**`+DS`*model* implied**
`K7200`	`1.1`	`1.1`
`K8000`	`2.0`	`2.0`
`V2200`	`2.0`	`2.0`
`S2000`	`2.0`	`2.0`
`X2000`	`2.0`	`2.0`

If you specify `+DA` or `+DS` on the compiler command line, your setting takes precedence over the setting implied by `+tm` *target*.

C aliasing options

To be conservative, the optimizer assumes that a pointer can point to any object in the entire application. Command-line options to the C compiler are available to inform the optimizer of an application's pointer usage. Using this information, the optimizer can generate more efficient code, due to the elimination of some false assumptions. Pointer behavior can be communicated to the optimizer by using the following options (which are discussed earlier in this appendix):

- `+O[no]ptrs_strongly_typed`
- `+O[no]ptrs_to_globals[=`*namelist*`]`
- `+O[no]global_ptrs_unique[=`*namelist*`]`
- `+O[no]ptrs_ansi`

where

namelist	is a comma-separated list of global variable names.

Here are the type-inferred aliasing rules:

- Type-aliasing optimizations are based on the assumption that pointer dereferences obey their declared types.
- A C variable is considered address-exposed if and only if the address of that variable is assigned to another variable or passed to a function as an actual parameter. In general, address-exposed objects are collected into a separate group based on their declared types. Global and static variables are considered address-exposed by default. Local variables and actual parameters are considered address-exposed only if their addresses have been computed using the address operator &.

- Dereferences of pointers to a certain type will be assumed to only alias with the corresponding equivalent group. An equivalent group includes all the address-exposed objects of the same type. The dereferences of pointers are also assumed to alias with other pointer dereferences associated with the same equivalent group.

  For example, in the following line:

  ```
  int *p, *q;
  ```

  `*p` and `*q` are assumed to alias with any objects of type `int`. Also, `*p` and `*q` are assumed to alias with each other.

- Signed/unsigned type distinctions are ignored in grouping objects into an equivalent group. Likewise, `long` and `int` types are considered to map to the same equivalent group. However, the `volatile` type qualifier is considered significant in grouping objects into equivalent groups (for example, a pointer to `int` will not be considered to alias with a `volatile int` object).

- If two type names reduce to the same type, they are considered synonymous.

  In the following example, both types `type_old` and `type_new` will reduce to the same type, `struct foo`.

  ```
  typedef struct foo_st type_old;
  typedef type_old type_new;
  ```

- Each field of a structure type is placed in a separate equivalent group that is distinct from the equivalent group of the field's base type. (The assumption here is that a pointer to `int` will not be assigned the address of a structure field whose type is `int`). The actual type name of a structure type is not considered significant in constructing equivalent groups (for example, dereferences of a `struct foo` pointer and a `struct bar` pointer will be assumed to alias with each other even if `struct foo` and `struct bar` have identical field declarations).

- All fields of a `union` type are placed in the same equivalent group, which is distinct from the equivalent group of any of the field's base types. (Thus, all dereferences of pointers to a particular `union` type will be assumed to alias with each other, regardless of which union field is being accessed.)

- Address-exposed array variables are grouped into the equivalent group of the array element type.

- Applying an explicit pointer typecast to an expression value causes any later use of the typecast expression value to be associated with the equivalent group of the typecast expression value.

  For example, an `int` pointer that is typecast into a `float` pointer and then dereferenced will be assumed to potentially access objects in the `float` equivalent group—and not the `int` equivalent group.

  However, type-incompatible assignments to pointer variables will not alter the aliasing assumptions on subsequent references of such pointer variables.

  In general, type-incompatible assignments can potentially invalidate some of the type-safe assumptions, and such constructs may elicit compiler warning messages.

E Optimization Report

This appendix provides a complete description of the Optimization Report produced by the HP Exemplar Fortran 90, HP Exemplar aC++, HP Exemplar Fortran 77, and HP Exemplar C compilers. When you compile a program with the `+Oreport[=`*report_type*`]` option at `+O3` (or `+O4`), the compiler generates an Optimization Report for each program unit. The `+Oreport[=`*report_type*`]` option determines the report's contents based on the value of *report_type*, as shown in Table 19.

Table 19 **Optimization Report contents**

***report_type* value**	**Report contents**
`all`	Loop Report and Privatization Table
`loop`	Loop Report
`private`	Loop Report and Privatization Table
report_type not given (default)	Loop Report

The +Onoreport option does not accept any of the *report_type* values. Examples of Optimization Reports are given in the section "Examples" on page 404.

Loop Report

The Loop Report lists the optimizations that were performed on loops and calls; and, if appropriate, the report gives reasons why a possible optimization was not performed. Loop nests are reported in the order in which they are encountered and separated by a blank line. A description of each column of the Loop Report follows:

`Line Num.`

Specifies the source line of the beginning of the loop, or of the loop from which it was derived. For cloned calls and inlined calls, the `Line Num.` column specifies the source line at which the call statement appears.

`Id Num.`

Specifies a unique ID number for every optimized loop and for every optimized call. This ID number can then be referenced by other parts of the report. Both loops appearing in the original program source and loops created by the compiler are given loop ID numbers; loops created by the compiler are also enumerated in the `New Id Nums` column as described later. No distinction between compiler-generated loops and loops that existed in the original source is made in the `Id Num` column; loops are assigned unique, sequential numbers as they are encountered.

`Var Name`

Specifies the name of the iteration variable controlling the loop or the called procedure if the line represents a call. If the variable is compiler-generated, its name is listed as `*VAR*`. If it consists of a truncated variable name followed by a colon and a number, the number is a reference to the variable name footnote table which appears after the Loop Report and Analysis Table in the Optimization Report.

`Reordering Transformation`

Indicates which reordering transformations were performed. Reordering transformations are performed on loops, calls, and loop nests, and typically involve reordering and/or duplicating sections of code to facilitate more efficient execution. This column has one of the values shown in Table 20.

Table 20 Reordering transformations reported in opt. report

Value	Explanation
`Block`	Loop blocking was performed. The new loop order will be indicated under the `Optimizing/Special Transformation` column, as shown in Table 21.
`Cloned call`	A call to a subroutine was cloned.
`Dist`	Loop distribution was performed.
`DynSel`	Dynamic selection was performed. The numbers in the `New Id Nums` column correspond to the loops created; for parallel loops, these generally include a `PARALLEL` and a `Serial` version.
`Fused`	The loops were fused into another loop and no longer exist. The original loops and the new loop will be indicated under the `Optimizing/Special Transformation` column, as shown in Table 21.
`Inlined call`	A call to a subroutine was inlined.
`Interchange`	Loop interchange was performed. The new loop order will be indicated under the `Optimizing/Special Transformation` column, as shown in Table 21.
`None`	No reordering transformation was performed on the call.
`PARALLEL`	The loop runs in thread-parallel mode.
`PAR-NODE`	The loop runs in node-parallel mode.
`Peel`	The first or last iteration of the loop was peeled in order to fuse the loop with an adjacent loop.
`Promote`	Test promotion was performed.

Value	Explanation
`Serial`	No reordering transformation was performed on the loop.
`Unroll and Jam`	The loop was unrolled and the nested loops were jammed (fused).
`VECTOR`	The loop was fully or partially replaced with more efficient calls to one or more vector routines.
`*`	Appears at left of loop-producing transformation optimizations (distribution, dynamic selection, blocking, fusion, interchange, call cloning, call inlining, peeling, promotion, unroll and jam).

`New Id Nums`

Specifies the ID number(s) for loops or calls created by the compiler. These ID numbers are listed in the `Id Num`. column and can be referenced in other parts of the report; however, the loops and calls they represent were not present in the original source code. In the case of loop fusion, the number in this column indicates the new loop created by merging all the fused loops. New ID numbers are also created for cloned calls, inlined calls, loop blocking, loop distribution, loop interchange, loop unroll and jam, dynamic selection, and test promotion.

`Optimizing / Special Transformation`

Indicates which, if any, optimizing transformations were performed. An optimizing transformation reduces the number of operations executed, or replaces operations with simpler operations. A special transformation allows the compiler to optimize code under special circumstances. When appropriate, this column has one of the values shown in Table 21.

Table 21 Optimizing/special transformations in opt. report

Value	Explanation
`Fused`	The loop was fused into another loop and no longer exists.
`Reduction`	The compiler recognized a reduction in the loop.
`Removed`	The compiler removed the loop.
`Unrolled`	The loop was completely unrolled.
(*OrigOrder*) -> (*InterchangedOrder*)	This information appears when `Interchange` is reported under `Reordering Transformation`. *OrigOrder* indicates the order of loops in the original nest; *InterchangedOrder* indicates the new order that occurs due to interchange. *OrigOrder* and *InterchangedOrder* consist of user iteration variables presented in outermost to innermost order.
(*OrigLoops*)->(*NewLoop*)	This information appears when `Fused` is reported under `Reordering Transformation`. *OrigLoops* indicates the original loops that were fused by the compiler to form the loop indicated by *NewLoop*. *OrigLoops* and *NewLoop* refer to loops based on the values from the `Id Num.` and `New Id Nums` columns in the Loop Report.
(*OrigLoopNest*)->(*BlockedLoopNest*)	This information appears when `Block` is reported under `Reordering Transformation`. *OrigLoopNest* indicates the order of the original loop nest containing a loop that was blocked. *BlockedLoopNest* indicates the order of loops after blocking. *OrigLoopNest* and *BlockedLoopNest* refer to user iteration variables presented in outermost to innermost order.

Supplemental tables

The tables described in this section are included in the Optimization Report—if necessary—to provide information supplemental to the Loop Report.

Analysis Table

If necessary, an Analysis Table is included in the Optimization Report to further elaborate on optimizations reported in the Loop Report. A description of each column of the Analysis Table follows:

`Line Num.`	Specifies the source line of the beginning of the loop or call.
`Id Num.`	References the ID number assigned to the loop or call in the Loop Report.
`Var Name`	Specifies the name of the iteration variable controlling the loop, `*VAR*` (as discussed in the `Var Name` description in the section “Loop Report” on page 398), or the name of the called procedure (as described in the “Loop Report” section of this appendix).
`Analysis`	Indicates why a transformation or optimization was not performed, or additional information on what was done.

Privatization Table

This table reports any user variables contained in a parallelized loop that are privatized by the compiler. Because the Privatization Table refers to loops, the Loop Report is automatically provided with it. A description of each column in the Privatization Table follows:

`Line Num.`
: Specifies the source line of the beginning of the loop.

`Id Num.`
: References the ID number assigned to the loop in the loop table.

`Var Name`
: Specifies the name of the iteration variable controlling the loop. `*VAR*` may also appear in this column, as discussed in the `Var Name` description in the section “Loop Report” on page 398.

`Priv Var`
: Specifies the name of the privatized user variable. Compiler-generated variables that are privatized are not reported here.

`Privatization Information for Parallel Loops`
: Provides more detail on the variable privatizations performed.

Variable Name Footnote Table

Variable names that are too long to fit in the `Var Name` columns of the other tables are truncated and followed by a colon and a footnote number. These footnotes are explained in the Variable Name Footnote Table. The headings in the Variable Name Footnote Table are explained below.

`Footnoted Var Name`	Specifies the truncated variable name and its footnote number.
`User Var Name`	Specifies the full name of the variable as given by the user in the source code.

Examples

The following Fortran examples enumerate the contents of the Optimization Report. In discussing the examples, loops are referred to by their ID numbers.

While only Fortran examples are given, analogous C and C++ code would produce similar Optimization Reports.

Example 1

Consider the following program. (Line numbers are provided for reference.)

```
1      PROGRAM EXAMPLE1
2      REAL A(100), B(100), C(100)
3      CALL SUB1(A,B,C)
4      END
5
6      SUBROUTINE SUB1(A,B,C)
7      REAL A(100), B(100), C(100)
8      DO ILOOPINDEX=1,100
9        A(ILOOPINDEX) = ILOOPINDEX
10     ENDDO
11     DO JLOOPINDEX=1,100
12       B(JLOOPINDEX) = A(JLOOPINDEX)**2
13     ENDDO
14     DO KLOOPINDEX=1, 100
15       C(KLOOPINDEX) = A(KLOOPINDEX) + B(KLOOPINDEX)
16     ENDDO
17     PRINT *, A(1), B(50), C(100)
18     END
```

The following Optimization Report is generated by compiling the program EXAMPLE1 with the command-line options `+O3 +Oparallel +Oreport=all +Oinline=sub1`:

```
% f77 +O3 +Oparallel +Oreport=all +Oinline=sub1 example1.f
            Optimization for example1

Line      Id    Var       Reordering          New       Optimizing / Special
Num.      Num.  Name      Transformation      Id Nums   Transformation
-----------------------------------------------------------------------------
   3         1  sub1     *Inlined call        (2-4)
   8         2  iloopi:1  Serial                        Fused
  11         3  jloopi:2  Serial                        Fused
  14         4  kloopi:3  Serial                        Fused
                         *Fused               (5)       (2 3 4) -> (5)
   8         5  iloopi:1  PARALLEL

Footnoted   User
Var Name    Var Name
-----------------------------------------------------------------------------
iloopi:1    iloopindex
jloopi:2    jloopindex
kloopi:3    kloopindex


            Optimization for sub1

Line      Id    Var       Reordering          New       Optimizing / Special
Num.      Num.  Name      Transformation      Id Nums   Transformation
-----------------------------------------------------------------------------
   8         1  iloopi:1  Serial                        Fused
  11         2  jloopi:2  Serial                        Fused
  14         3  kloopi:3  Serial                        Fused
                         *Fused               (4)       (1 2 3) -> (4)
   8         4  iloopi:1  PARALLEL

Footnoted   User
Var Name    Var Name
-----------------------------------------------------------------------------
iloopi:1    iloopindex
jloopi:2    jloopindex
kloopi:3    kloopindex
```

Optimization Report interpretation

This section uses fragments from the program EXAMPLE1 to explain the information in the Optimization Report for EXAMPLE1. Again, line numbers are listed for reference.

EXAMPLE1's Loop Report provides the following information:

The first line of the Loop Report:

```
3        1  sub1     *Inlined call      (2-4)
```

tells us that the call to sub1 was inlined. The inlining produced three new loops in EXAMPLE1: Loop #2, Loop #3, and Loop #4. Internally, the EXAMPLE1 module that originally looked like:

```
1      PROGRAM EXAMPLE1
2      REAL A(100), B(100), C(100)
3      CALL SUB1(A,B,C)
4      END
```

now looks like the following:

```
      PROGRAM EXAMPLE1
      REAL A(100), B(100), C(100)
      DO ILOOPINDEX=1,100                   !Loop #2
        A(ILOOPINDEX) = ILOOPINDEX
      ENDDO
      DO JLOOPINDEX=1,100                   !Loop #3
        B(JLOOPINDEX) = A(JLOOPINDEX)**2
      ENDDO
      DO KLOOPINDEX=1, 100                  !Loop #4
        C(KLOOPINDEX) = A(KLOOPINDEX) + B(KLOOPINDEX)
      ENDDO
      PRINT *, A(1), B(50), C(100)
      END
```

The next four lines indicate that the new loops have been fused; the fourth line informs us that the three loops were fused into one new loop, `Loop #5`.

```
  8         2  iloopi:1  Serial                          Fused
 11         3  jloopi:2  Serial                          Fused
 14         4  kloopi:3  Serial                          Fused
                        *Fused               (5)         (2 3 4) -> (5)
```

After fusing, the code internally appears as the following:

```
      PROGRAM EXAMPLE1
      REAL A(100), B(100), C(100)
      DO ILOOPINDEX=1,100                  !Loop #5
        A(ILOOPINDEX) = ILOOPINDEX
        B(ILOOPINDEX) = A(ILOOPINDEX)**2
        C(ILOOPINDEX) = A(ILOOPINDEX) + B(ILOOPINDEX)
      ENDDO
      PRINT *, A(1), B(50), C(100)
      END
```

As indicated by the following Loop Report line:

```
8         5  iloopi:1  PARALLEL
```

`Loop #5` uses `iloopi:1` as the iteration variable (referencing the Variable Name Footnote Table, we see that `iloopi:1` corresponds to `iloopindex`). From that same line in the report, we learn that the newly-created `Loop #5` was parallelized.

According to the Variable Name Footnote Table (duplicated below), the original variable `iloopindex` is abbreviated—to fit in the `Var Name` columns of other reports—by the compiler as `iloopi:1`; similarly, `jloopindex` and `kloopindex` are abbreviated as `jloopi:2` and `kloopi:3`, respectively. These names are used throughout the report to refer to these iteration variables.

```
Footnoted   User
Var Name    Var Name
-----------------------------------------------------------------------------
iloopi:1    iloopindex
jloopi:2    jloopindex
kloopi:3    kloopindex
```

Example 2

The following Fortran code provides an example of other transformations the compiler performs. (Line numbers are listed for reference.)

```
1      PROGRAM EXAMPLE2
2
3      INTEGER IA1(100), IA2(100), IA3(100)
4      INTEGER I1, I2
5
6      DO I = 1, 100
7        IA1(I) = I
8        IA2(I) = I * 2
9        IA3(I) = I * 3
10     ENDDO
11
12     I1 = 0
13     I2 = 100
14     CALL SUB1 (IA1, IA2, IA3, I1, I2)
15     END
16
17     SUBROUTINE SUB1(A, B, C, S, N)
18     INTEGER A(N), B(N), C(N), S, I, J
19       DO J = 1, N
20         DO I = 1, N
21           IF (I .EQ. 1) THEN
22             S = S + A(I)
23           ELSE IF (I .EQ. N) THEN
24             S = S + B(I)
25           ELSE
26             S = S + C(I)
27           ENDIF
28         ENDDO
29       ENDDO
30     END
```

The following Optimization Report is generated by compiling the program EXAMPLE2 for parallelization:

```
% f90 +O3 +Oparallel +Oreport=all example2.f
```

```
                 Optimization for SUB1

Line      Id     Var         Reordering         New       Optimizing / Special
Num.      Num.   Name        Transformation     Id Nums   Transformation
-----------------------------------------------------------------------------
   19        1   j          *Interchange        (2)       (j i) -> (i j)
   20        2   i          *DynSel             (3-4)
   20        3   i           PARALLEL                     Reduction
   19        5   j          *Promote            (6-7)
   19        6   j           Serial
   19        7   j           Serial

   20        4   i           Serial
   19        8   j          *Promote            (9-10)
   19        9   j           Serial
   19       10   j          *Promote            (11-12)
   19       11   j           Serial
   19       12   j           Serial

Line      Id     Var        Analysis
Num.      Num.   Name
-----------------------------------------------------------------------------
   19        5   j          Test on line 21 promoted out of loop
   19        8   j          Test on line 21 promoted out of loop
   19       10   j          Test on line 23 promoted out of loop
```

The report is continued on the next page.

```
            Optimization for clone 1 of SUB1 (6_e70_cl_sub1)

Line      Id    Var         Reordering        New       Optimizing / Special
Num.      Num.  Name        Transformation    Id Nums   Transformation
-----------------------------------------------------------------------------
   19       1   j          *Interchange       (2)       (j i) -> (i j)
   20       2   i           PARALLEL                    Reduction
   19       3   j          *Promote           (4-5)
   19       4   j           Serial
   19       5   j          *Promote           (6-7)
   19       6   j           Serial
   19       7   j           Serial

Line      Id    Var        Analysis
Num.      Num.  Name
-----------------------------------------------------------------------------
   19       3   j          Test on line 21 promoted out of loop
   19       5   j          Test on line 23 promoted out of loop

            Optimization for example2

Line      Id    Var         Reordering        New       Optimizing / Special
Num.      Num.  Name        Transformation    Id Nums   Transformation
-----------------------------------------------------------------------------
    6       1   i           Serial

   14       2   sub1       *Cloned call       (3)
   14       3   sub1        None

Line      Id    Var        Analysis
Num.      Num.  Name
-----------------------------------------------------------------------------
   14       2   sub1       Call target changed to clone 1 of SUB1 (6_e70_cl_sub1)
```

Optimization Report interpretation

This section uses fragments from the program `EXAMPLE2` to explain the information in the Optimization Report for `EXAMPLE2`. Again, line numbers are listed for reference.

The Optimization Report first shows Optimization Reports for the subroutine and its clone. We shall now examine the optimizations to the subroutine. Originally, the subroutine appeared as below:

```
17     SUBROUTINE SUB1(A, B, C, S, N)
18     INTEGER A(N), B(N), C(N), S, I, J
19       DO J = 1, N
20         DO I = 1, N
21           IF (I .EQ. 1) THEN
22             S = S + A(I)
23           ELSE IF (I .EQ. N) THEN
24             S = S + B(I)
25           ELSE
26             S = S + C(I)
27           ENDIF
28         ENDDO
29       ENDDO
30     END
```

First, the compiler performs loop interchange (listed as `Interchange` in the report) to maximize cache performance:

```
19          1  j        *Interchange      (2)       (j i) -> (i j)
```

Internally, the subroutine then becomes the following:

```
17     SUBROUTINE SUB1(A, B, C, S, N)
18     INTEGER A(N), B(N), C(N), S, I, J
19       DO I = 1, N                          ! Loop #2
20         DO J = 1, N                        ! Loop #1
21           IF (I .EQ. 1) THEN
22             S = S + A(I)
23           ELSE IF (I .EQ. N) THEN
24             S = S + B(I)
25           ELSE
26             S = S + C(I)
27           ENDIF
28         ENDDO
29       ENDDO
30     END
```

The program is then optimized for parallelization. The compiler would like to parallelize the outermost loop in the nest (which is now the `I` loop). However because the value of `N` is not known, the compiler does not know how many times the `I` loop needs to be executed. To ensure that the loop is executed as efficiently as possible at runtime, the compiler replaces the `I` loop nest with two new copies of the `I` loop nest, one to be run in parallel, the other to be run serially. An `IF` is then inserted to select the more efficient version of the loop to execute at runtime. This method of making one copy for parallel execution and one copy for serial execution is known as dynamic selection, which is enabled by default when `+O3 +Oparallel` is specified. (See “Dynamic selection” on page 109 for more information.) This optimization is reported in the Loop Report in the line:

```
20         2  i        *DynSel            (3-4)
```

According to the report, `Loop #2` was used to create the new loops, `Loop #3` and `Loop #4`. Internally, the code now is represented as follows:

```
SUBROUTINE SUB1(A, B, C, S, N)
INTEGER A(N), B(N), C(N), S, I, J
IF (N .GT. some_threshold) THEN
  DO (parallel) I = 1, N               ! Loop #3
    DO J = 1, N                        ! Loop #5
      IF (I .EQ. 1) THEN
        S = S + A(I)
      ELSE IF (I .EQ. N) THEN
        S = S + B(I)
      ELSE
        S = S + C(I)
      ENDIF
    ENDDO
  ENDDO
ELSE
  DO I = 1, N                          ! Loop #4
    DO J = 1, N                        ! Loop #8
      IF (I .EQ. 1) THEN
        S = S + A(I)
      ELSE IF (I .EQ. N) THEN
        S = S + B(I)
      ELSE
        S = S + C(I)
      ENDIF
    ENDDO
  ENDDO
ENDIF
END
```

As indicated by the `Optimizing / Special Transformation` column of the report, `Loop #3` (which was parallelized) also contained one or more reductions. The `Reordering Transformation` column indicates that the `IF` statements were promoted out of `Loop #5`, `Loop #8`, and `Loop #10`.

The Analysis Table, which immediately follows the Loop Report, shows the line numbers of the IF statements that were promoted. The first test in Loop #5 was promoted, creating two new loops, Loop #6 and Loop #7. Similarly, Loop #8 has a test promoted, creating Loop #9 and Loop #10. The test remaining in Loop #10 is then promoted, thereby creating two additional loops. (A “promoted test” is an IF statement that is hoisted out of a loop. See the section “Test promotion” on page 66 for more information.) The Analysis Table contents are shown below:

```
19         5  j         Test on line 21 promoted out of loop
19         8  j         Test on line 21 promoted out of loop
19        10  j         Test on line 23 promoted out of loop
```

Moving to the Optimization Report for EXAMPLE2, we find that the following DO loop:

```
6     DO I = 1, 100
7       IA1(I) = I
8       IA2(I) = I * 2
9       IA3(I) = I * 3
10    ENDDO
```

does not undergo any reordering transformation. This fact is reported by the line

```
6        1  i         Serial
```

The call to the subroutine sub1 is cloned. As indicated by the asterisk (*), the compiler produced a new call. The new call is given the ID (3) listed in the New Id Nums column. The new call is then listed; None indicates that no reordering transformation was performed on the call to the new subroutine.

```
14         2  sub1     *Cloned call       (3)
14         3  sub1      None
```

The Analysis Table is then appended to the Loop Report to elaborate on the Cloned call transformation. This line informs us that the clone was called in place of the original subroutine.

```
14      2  sub1    Call target changed to clone 1 of SUB1 (6_e70_cl_sub1)
```

Example 3

This last example shows loop blocking, loop peeling, loop distribution, and loop unroll and jam, among other transformations. (Line numbers are listed for reference.)

```
1      PROGRAM EXAMPLE3
2
3      REAL*8 A(1000,1000), B(1000,1000), C(1000)
4      REAL*8 D(1000), E(1000)
5      INTEGER M, N
6
7      N = 1000
8      M = 1000
9
10     DO I = 1, N
11       C(I) = 0
12       DO J = 1, M
13         A(I,J) = A(I,J) + B(I,J) * C(I)
14       ENDDO
15     ENDDO
16
17     DO I = 1, N-1
18       D(I) = I
19     ENDDO
20
21     DO J = 1, N
22       E(J) = D(J) + 1
23     ENDDO
24
25     PRINT *, A(103,103), B(517, 517), D(11), E(29)
26
27     END
```

The following Optimization Report is generated by compiling program EXAMPLE3 as follows:

```
% f90 +O3 +Oreport +Oloop_block example3.f
            Optimization for example3

Line     Id    Var         Reordering         New       Optimizing / Special
Num.     Num.  Name        Transformation     Id Nums   Transformation
-----------------------------------------------------------------------------
   10       1  i:1        *Dist               (2-3)
   10       2  i:1         Serial

   10       3  i:1        *Interchange        (4)       (i:1 j:1) -> (j:1 i:1)
   12       4  j:1        *Block              (5)       (j:1 i:1) -> (i:1 j:1 i:1)
   10       5  i:1        *Promote            (6-7)
   10       6  i:1         Serial                       Removed
   10       7  i:1         Serial
   12       8  j:1        *Unroll And Jam     (9)
   12       9  j:1        *Promote            (10-11)
   12      10  j:1         Serial                       Removed
   12      11  j:1         Serial
   10      12  i:1         Serial

   17      13  i:2         Serial                       Fused
   21      14  j:2        *Peel               (15)
   21      15  j:2         Serial                       Fused
                          *Fused              (16)      (13 15) -> (16)
   17      16  i:2         Serial

Line     Id    Var        Analysis
Num.     Num.  Name
-----------------------------------------------------------------------------
   10       5  i:1        Loop blocked by 56 iterations
   10       5  i:1        Test on line 12 promoted out of loop
   10       6  i:1        Loop blocked by 56 iterations
   10       7  i:1        Loop blocked by 56 iterations
   12       8  j:1        Loop unrolled by 8 iterations and jammed into the
innermost loop
   12       9  j:1        Test on line 10 promoted out of loop
   21      14  j:2        Peeled last iteration of loop
```

Optimization Report interpretation

This section uses fragments from program EXAMPLE3 to explain the information in the Optimization Report for EXAMPLE3. Again, line numbers are listed for reference.

Notice that in this report, the `Var Name` column has entries such as `i:1`, `j:1`, `i:2`, and `j:2`. This type of entry appears when a variable is used more than once. For example, in EXAMPLE3, `I` is used as an iteration variable twice; consequently, `i:1` refers to the first occurrence, and `i:2` refers to the second occurrence.

The first line of the report tells us that the loop on line 10 of EXAMPLE3 (that is, `Loop #1`) is distributed to create `Loop #2` and `Loop #3`:

```
10        1  i:1       *Dist              (2-3)
```

Initially, `Loop #1` appears as shown below.

```
DO I = 1, N                            ! Loop #1
  C(I) = 0
  DO J = 1, M
    A(I,J) = A(I,J) + B(I,J) * C(I)
  ENDDO
ENDDO
```

It is then distributed as follows:

```
DO I = 1, N                            ! Loop #2
  C(I) = 0
ENDDO

DO I = 1, N                            ! Loop #3
  DO J = 1, M
    A(I,J) = A(I,J) + B(I,J) * C(I)
  ENDDO
ENDDO
```

The third line informs us that `Loop #3` is then interchanged to create `Loop #4`:

```
10        3  i:1       *Interchange      (4)       (i:1 j:1) -> (j:1 i:1)
```

Now, the loop looks like the following code:

```
DO J = 1, M                           ! Loop #4
  DO I = 1, N
    A(I,J) = A(I,J) + B(I,J) * C(I)
  ENDDO
ENDDO
```

The next line of the Optimization Report indicates that the nest rooted at `Loop #4` is blocked:

```
12      4  j:1      *Block          (5)      (j:1 i:1) -> (i:1 j:1 i:1)
```

The blocked nest internally appears as follows:

```
DO IOUT = 1, N, 56                    ! Loop #5
  DO J = 1, M
    DO I = IOUT, IOUT + 55
      A(I,J) = A(I,J) + B(I,J) * C(I)
    ENDDO
  ENDDO
ENDDO
```

`Loop #5` (the loop with iteration variable `i:1`) is the loop that was actually blocked. The report shows `*Block` on `Loop #4` (the `j:1` loop) because the entire nest rooted at `Loop #4` is replaced by the blocked nest.

The `IOUT` variable is introduced to facilitate the loop blocking. The compiler uses a step value of 56 for the `IOUT` loop as reported in the Analysis Table:

```
10       5  i:1       Loop blocked by 56 iterations
```

The next three lines of the report tell us that a test was promoted out of `Loop #5`, creating `Loop #6` (which is removed) and `Loop #7` (which is run serially). This test—which does not appear in the source code—is an implicit test that the compiler inserts in the code to ensure that the loop iterates at least once.

```
10       5  i:1       *Promote         (6-7)
10       6  i:1        Serial                    Removed
10       7  i:1        Serial
```

This test is referenced again in the following line from the Analysis Table:

```
10      5  i:1      Test on line 12 promoted out of loop
```

Looking back to the report, we see that the J is unrolled and jammed, creating Loop #9:

```
12        8  j:1       *Unroll And Jam    (9)
```

The Analysis Table also tells us that the J loop is unrolled and jammed, stating that the loop was unrolled by 8 iterations:

```
12     8  j:1     Loop unrolled by 8 iterations and jammed into the innermost loop
```

The unrolled and jammed loop looks like the following code:

```
DO IOUT = 1, N, 56                      ! Loop #5
  DO J = 1, M, 8                        ! Loop #8
    DO I = IOUT, IOUT + 55              ! Loop #9
      A(I,J) = A(I,J) + B(I,J) * C(I)
      A(I,J+1) = A(I,J+1) + B(I,J+1) * C(I)
      A(I,J+2) = A(I,J+2) + B(I,J+2) * C(I)
      A(I,J+3) = A(I,J+3) + B(I,J+3) * C(I)
      A(I,J+4) = A(I,J+4) + B(I,J+4) * C(I)
      A(I,J+5) = A(I,J+5) + B(I,J+5) * C(I)
      A(I,J+6) = A(I,J+6) + B(I,J+6) * C(I)
      A(I,J+7) = A(I,J+7) + B(I,J+7) * C(I)
    ENDDO
  ENDDO
ENDDO
```

The Optimization Report informs us that the compiler-inserted test in Loop #9 is promoted out the loop, creating Loop #10 and Loop #11.

```
12        9  j:1       *Promote         (10-11)
12       10  j:1        Serial                           Removed
12       11  j:1        Serial
```

According to the report, the last two loops in the program are fused (once an iteration is peeled off the second loop), then the new loop is run serially.

```
17       13  i:2        Serial                       Fused
21       14  j:2       *Peel          (15)
21       15  j:2        Serial                       Fused
                       *Fused         (16)       (13 15) -> (16)
17       16  i:2        Serial
```

Combining that information with the following line from the Analysis Table:

```
21       14  j:2        Peeled last iteration of loop
```

we know that initially, `Loop #14` has an iteration peeled to create `Loop #15`, as shown below. (The loop peeling is performed to enable loop fusion.)

```
DO I = 1, N-1                               ! Loop #13
  D(I) = I
ENDDO

DO J = 1, N-1                               ! Loop #15
  E(J) = D(J) + 1
ENDDO
```

`Loop #13` and `Loop #15` are then fused to produce `Loop #16`:

```
DO I = 1, N-1                               ! Loop #16
  D(I) = I
  E(I) = D(I) + 1
ENDDO
```

+Oinfo option

The `+Oinfo` option, which displays informational messages about the optimization process, can also be used to provide information. This option is only useful at `+O3` and above. The default is `+Onoinfo`.

F Compiler Parallel Support Library

Introduction

The Compiler Parallel Support Library (CPSlib) is a library of thread management and synchronization routines for use in controlling parallelism. Most programs can fully exploit their parallelism via higher-level devices such as automatic parallelization, compiler directives, and message-passing; CPSlib is provided for those few cases requiring a lower-level interface. Using CPSlib requires you to manually control all aspects of parallelism, synchronization, and data partitioning.

This appendix includes a discussion of the forms of parallelism available via CPSlib, instructions for accessing CPSlib, a brief description of each routine included in CPSlib, and examples of common programming constructs as implemented using CPSlib routines. For further information, refer to the section 3 man pages for the routine in question, or to the cps(3) man page for an overview.

CPSlib supports two forms of parallelism: symmetric and asymmetric.

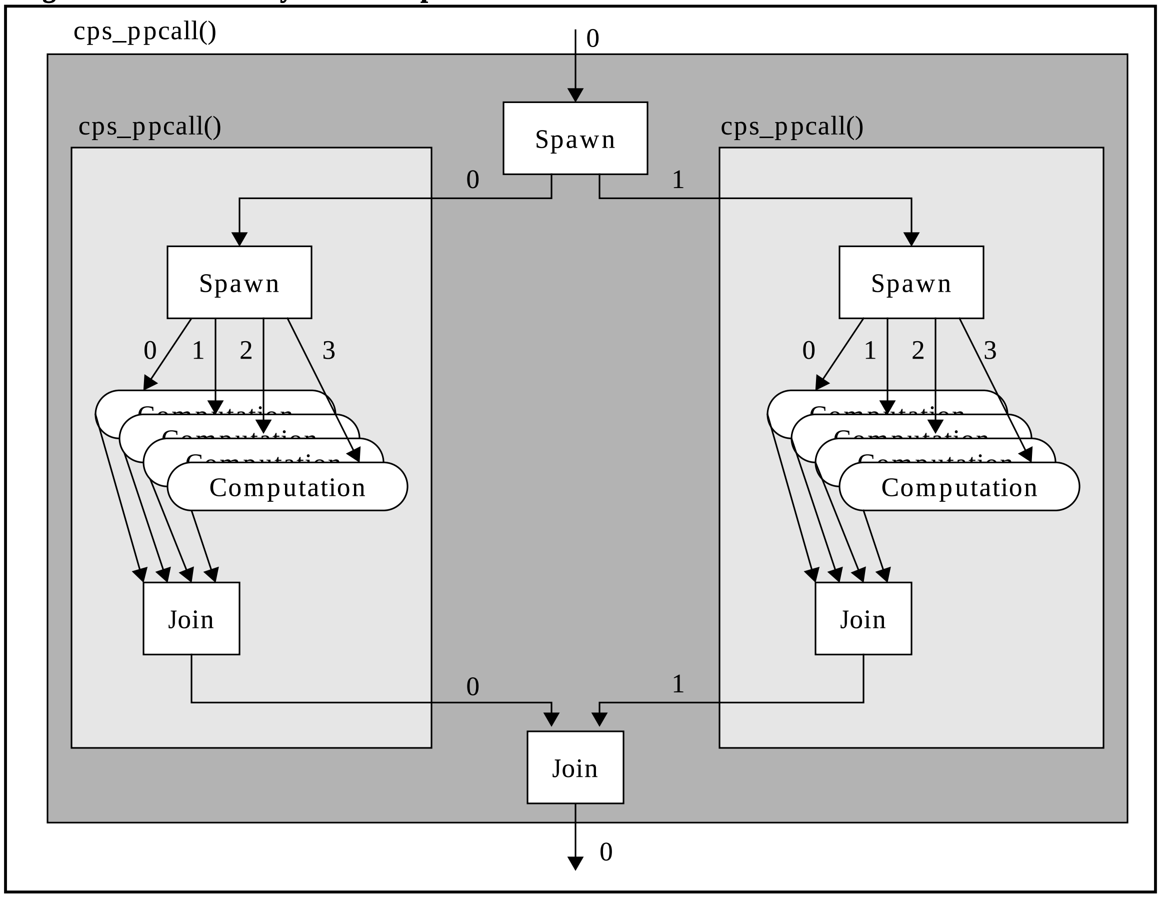
Symmetric parallelism

In symmetric parallelism, several threads execute the same instruction stream. Symmetric parallelism is typically used by the compilers to parallelize a loop; the description of parallelism given in Chapter 3, “Compiler optimizations,” is a description of symmetric parallelism.

Symmetric parallel threads are spawned using `cps_ppcall()` or `cps_ppcalln()`, which, along with all CPSlib routines, are described in detail further on. These functions automatically spawn a given number of threads, which call a specified routine in parallel. All parallel work must occur in the called routine. When the routine returns, `cps_ppcall()` or `cps_ppcalln()` automatically executes a join and the program proceeds in serial.

Figure 26 shows a `cps_ppcall()` that spawns two threads, each of which in turn spawns four threads. The arrows represent a thread’s instruction flow; the numbers labeling the arrows indicate the spawn thread IDs. Kernel and spawn thread IDs are also discussed in the section “Thread ID assignments” on page 223.

Figure 26 **Symmetric parallelism**



Shaded boxes represent operations hidden from the user by `cps_ppcall()`. As shown in the left branch of the first spawn, when a `cps_ppcall()` or `cps_ppcalln()` is processed, the parent thread is allocated to the computation as spawn thread ID 0; its kernel thread ID, which is uninteresting to the user, is unchanged. Additional peer threads in both spawned threads are spawned and assigned spawn thread IDs from one to the number of threads spawned minus one.

When the threads join, the original parent thread (spawn thread ID 0) leaves the join after all other threads that were spawned last have also joined. The join operation contains an implicit barrier, so that all threads must reach the join before the original thread continues.

Spawns and joins may thus be arbitrarily nested; however, each thread that was allocated as a result of a spawn must eventually join for correct program operation. The spawn thread ID has a scope between the immediately enclosing spawn/join pair; a single thread may change its spawn thread ID as the result of executing a spawn or join operation.

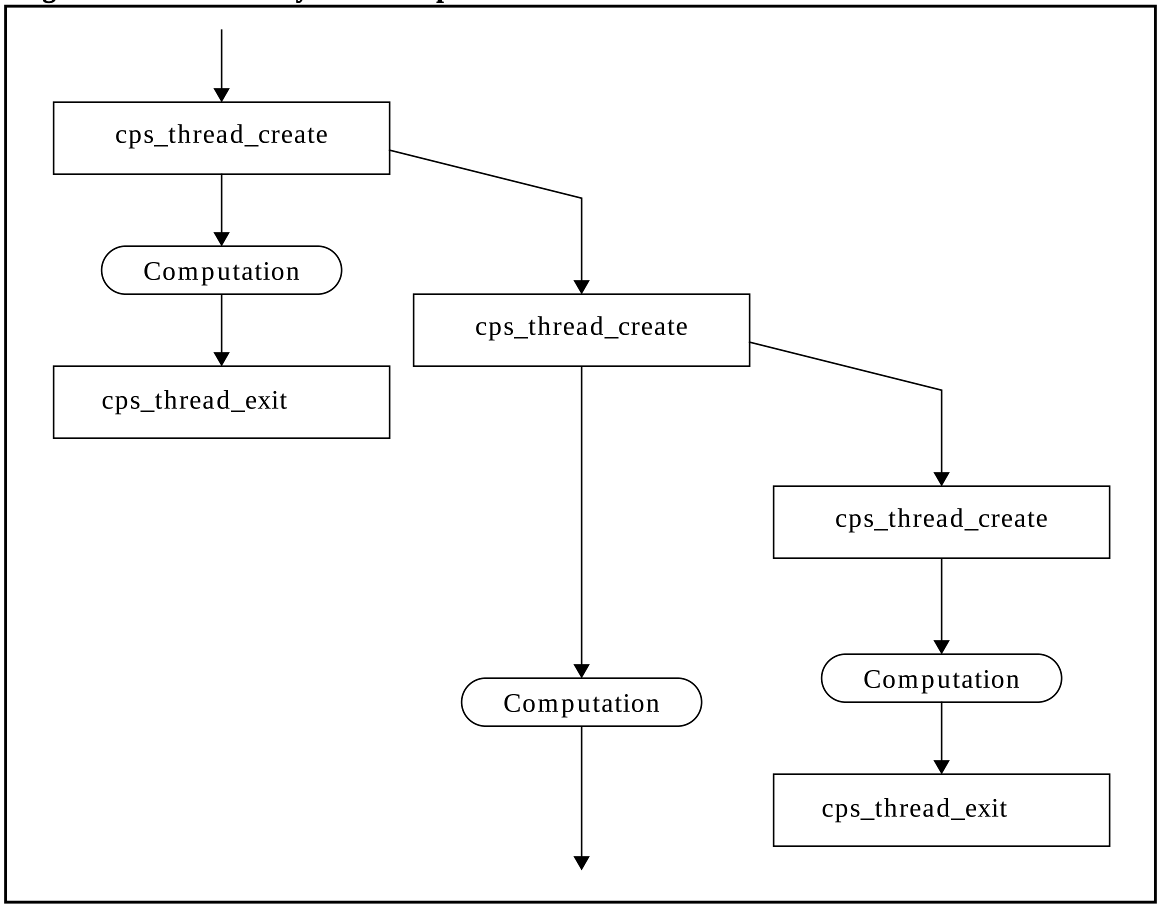
Asymmetric parallelism

Asymmetric parallelism is used when each thread executes a different, independent instruction stream. Asymmetric threads are analogous to the Unix `fork` system call construct; the threads are disjoint. Either the parent or the child may terminate first in any order. Either or both the parent and the child may spawn additional symmetric or asymmetric threads.

Asymmetric threads are spawned using the `cps_thread_create()` function and terminated using the `cps_thread_exit()` function. Asymmetric threads cannot join with their parent thread; they terminate separately. If you do not specifically call `cps_thread_exit()` to terminate an asymmetric thread, the thread will automatically terminate when the procedure called by `cps_thread_create()` successfully terminates.

Figure 27 shows an asymmetric thread tree. Asymmetrically spawned child threads do not have spawn thread IDs; they do, however, have unique kernel thread IDs, which are assigned in no particular order. The parent which executed the `cps_thread_create()` retains the same kernel thread ID it had before it spawned the child thread.

Figure 27 **Asymmetric parallelism**



You can spawn symmetric threads from asymmetric threads using `cps_ppcall` and `cps_ppcalln`. In this case, the parent thread retains its kernel thread ID and, along with the symmetric threads, receives a spawn thread ID. These symmetric threads must join before the asymmetric parent can exit; if an asymmetric parent attempts to exit while its symmetric children are still active, it will join instead.

Accessing CPSlib

C and C++ programs using CPSlib functions must include the following header file:

```
#include <cps.h>
```

To access `errno` symbolic constant values in C or C++, you must also include the following header file:

```
#include <errno.h>
```

The Fortran 90, Fortran 77, aC++, and C compilers share a common interface to CPSlib; the same library, libcps.sl, allows access to the CPSlib functions from any of the languages.

In the Exemplar compilers, CPSlib is automatically linked in only at `+O3` (and above) when `+Oparallel` is specified. If your program explicitly calls CPSlib routines or calls other libraries that use CPS routines and you are not linking `at +O3` (or `+O4`) with `+Oparallel`, you must explicitly link in CPSlib as shown in the example below.

Assume prog.c contains calls to CPSlib routines:

```
% cc prog.c -lpthread -lcps -lpthread
```

Linking in CPSlib requires specifying—in the order given—all of the string `-lpthread -lcps -lpthread`. The `pthread` library is required because CPSlib uses Pthreads routines. Pthreads routines can be used independently of CPSlib. For more information on using Pthreads, see the pthread(3t) man page or the book *Programming with Threads on HP-UX* (B2355-90060).

CPSlib and MP_NUMBER_OF_THREADS

CPSlib checks the value of the MP_NUMBER_OF_THREADS environment variable to determine how many processors to generate code for. This variable is read at runtime. If this variable is set to some positive integer *n*, your program executes on *n* processors; *n* must be less than or equal to the number of processors in the system where the program is executing. The default value for MP_NUMBER_OF_THREADS is 1.

The following command line shows the C shell syntax to use when setting the variable to 8 processors:

```
% setenv MP_NUMBER_OF_THREADS 8
```

CPS library functions

CPSlib provides thread-management functions, high-level synchronization functions, and low-level synchronization functions. This section briefly describes each function and its arguments.

Default versions of the functions presented here return 4-byte values and take 4-byte arguments. 8-byte versions are also available; the names of these functions are suffixed with _8 (for example, the 8-byte version of `cps_ppcall` is `cps_ppcall_8`), and the functions take eight-byte arguments. Refer to the appropriate man pages for more information.

All C/C++ examples presented here assume that `cps.h` is included in the program. Note that the NULL value in C and C++ is equivalent to 0 (zero) in Fortran.

NOTE

CPSlib routines are incompatible with system functions of the form `cnx_*`; mixing the two will cause wrong answers, deadlock or runtime errors. Mixing CPSlib routines and certain compiler directives may cause wrong answers at `+O1` or higher; if this happens, lower the optimization level or refer to the section “Global register allocation” on page 52. Do not mix CPSlib routines and Pthreads routines.

Thread-management functions

These functions allow you to spawn and join or terminate threads.

Symmetric thread functions

Symmetric threads are spawned, execute in parallel, and join via a single CPSlib function call. By definition, all parallel symmetric threads must execute to completion before the join operation can take place; therefore no exit or wait functions are provided.

cps_ppcall and cps_ppcalln

The `cps_ppcall` and `cps_ppcalln` functions allow you to spawn symmetrically parallel threads. In Fortran, these functions have the following forms:

```
INTEGER FUNCTION CPS_PPCALL(PARAMS, FUNC, ARG)
INTEGER PARAMS(4) !ELEMENTS ARE ANALOGOUS TO ELEMENTS OF params
                  !STRUCTURE IN C CODE
EXTERNAL FUNC
INTEGER FUNCTION CPS_PPCALLN(PARAMS,FUNC,n,ARG1,...,ARGn)
INTEGER PARAMS(4) !ELEMENTS ARE ANALOGOUS TO ELEMENTS OF params
                  !STRUCTURE IN C CODE
EXTERNAL FUNC
```

Because the subprogram `FUNC` is used as an actual argument in the function reference, it must be declared `EXTERNAL`.

In C and C++:

```
typedef struct {
  int node;        /* node to place allocated threads on */
  int min;         /* minimum number of threads to allocate */
  int max;         /* maximum number of threads to allocate */
  int threadscope; /* thread scope attributes */
} spawn_sym_t;
int cps_ppcall(spawn_sym_t *params,void (*func)(void *),void *arg);
int cps_ppcalln(spawn_sym_t *params,void (*func)(void *),
                const int *n, void *arg1, ..., void *argn);
```

The elements `params->node` and `PARAMS(1)` control what hypernodes threads are allocated on. They can contain the logical hypernode ID of the hypernode on which to allocate the threads, or they can take one of the values shown in Table 22.

Table 22 **params->node/PARAMS(1) values**

C/C++ symbolic constant name	Value	Meaning
CPS_SAME_NODE	-1	Allocate threads on same hypernode as calling thread
CPS_ANY_NODE	-2	Allocate threads on any hypernode
CPS_DIFFERENT_NODE	-3	Allocate threads on different hypernode than that of the calling thread

params->min and PARAMS(2) specify the minimum number of threads to allocate; params->max and PARAMS(3) specify the maximum number of threads to allocate.

params->threadscope and PARAMS(4) control the creation strategy for new threads. Table 23 shows acceptable values for these parameters. Except where noted, the logical or of two or more of these values can be used.

Table 23 **params->threadscope/PARAMS(4) values**

C/C++ symbolic constant name	Value	Meaning
CPS_THREAD_PARALLEL	1	Allocate multiple threads per hypernode; mutually exclusive with CPS_NODE_PARALLEL
CPS_NODE_PARALLEL	2	Allocate one thread per hypernode; mutually exclusive with CPS_THREAD_PARALLEL
CPS_OVER_SUBSCRIBE	4	Allows multiple threads per CPU; if not set, no more than one thread per CPU can be started
CPS_IGNORE_STACKSCOPE	8	Allows the spawning of new threads which may not be able to validly address their parent's stack or thread_private data

cps_ppcall spawns the number of threads designated in the argument params and arranges for each thread, including the calling thread, to call the argument function func with the argument arg. After returning from `func`, each thread automatically joins. When all threads have joined, the parent continues executing the code following the `cps_ppcall` or `cps_ppcalln`.

`cps_ppcalln` is identical to `cps_ppcall`, except that it will pass *n* arguments to the function `func`, where *n* ranges from 0 to 256.

The thread that calls `cps_ppcall` or `cps_ppcalln` is considered the parent thread; if the call is successful, it will become spawn thread 0 prior to calling the function. Other threads spawned will be assigned increasing spawn thread IDs ranging from 1 to *m*-1, where *m* is the number of threads spawned.

After all of the threads return from `func`, the parent thread will restore its previous thread state and continue execution after `cps_ppcall` or `cps_ppcalln`.

If successful, these functions return the number of threads spawned including the parent. If an error occurs, they return -1 and, in C and C++, `errno` is set as shown in Table 24.

Table 24 **`errno` values for `cps_ppcall` and `cps_ppcalln`**

`errno` value	Meaning
`EAGAIN`	The minimum number of threads could not be allocated
`EINVAL`	Either the hypernode specified by `params->node` (or `PARAMS(1)`) is not a valid logical hypernode ID, or the value of `params->min` (`PARAMS(2)`) or `params->max` (`PARAMS(3)`) is less than 0

Nested `cps_ppcall` and/or `cps_ppcalln` calls are permitted.

NOTE Each thread spawned to run `func` receives a default local stack of 80 Mbytes. If `func` declares local variables that occupy more than 80 Mbytes, you must change this default using the `CPS_STACK_SIZE` environment variable. `CPS_STACK_SIZE` specifies the default stack size for spawned parallel functions in kbytes. It is read once at program startup; the spawn thread stack size cannot be changed during execution. Thread 0’s default stack size is specified by HP-UX.

For more information, refer to the cps_ppcall(3) man page.

Asymmetric thread functions

By definition, asymmetric threads execute independently of one another. Rather than join, asymmetric threads *terminate*; that is, because their execution is independent, one asymmetric thread has no need to know when another has completed, so no join is necessary or possible.

cps_thread_create and cps_thread_createn

The cps_thread_create and cps_thread_createn functions allow you to spawn asymmetrically parallel threads. In Fortran, these functions have the following form:

```
INTEGER FUNCTION CPS_THREAD_CREATE(NODE, FUNC, ARG)
INTEGER NODE
EXTERNAL FUNC
INTEGER FUNCTION CPS_THREAD_CREATEN(NODE, FUNC, n, ARG1, ..., ARGn)
INTEGER NODEEXTERNAL FUNC
```

Because the subprogram FUNC is used as an actual argument in the function reference, it must be declared EXTERNAL.

In C and C++:

```
int cps_thread_create(const int *node, void (*func) (void *),
                      void *arg);
int cps_thread_createn(const int *node, void (*func) (void*),
                       const int *n, void *arg1, ..., void *argn);
```

cps_thread_create spawns a single asymmetric thread on the hypernode specified by node and arranges for the asymmetric thread to call the argument function func with the parameter arg. On return from func, the asymmetric thread will automatically call cps_thread_exit(); this function can also be manually called from within func to terminate the asymmetric thread.

node takes the same CPS_SAME_NODE and CPS_ANY_NODE values as the node argument to cps_ppcall, which is described in the section “Symmetric thread functions” on page 427.

cps_thread_createn is identical to cps_thread_create, except that it will pass *n* arguments to the function func, where *n* ranges from 0 to 256.

If successful, these functions return the kernel thread ID of the newly spawned thread. No assumptions can be made about kernel thread IDs except that they are unique. Asymmetric threads do not have spawn thread IDs.

NOTE cps_thread_create and cps_thread_createn return immediately after initiating func, and both func and the parent thread execute in parallel until one terminates. If the parent thread manipulates arg after returning from cps_thread_create[n] but before func exits, you must ensure that this manipulation is synchronized between the parent and func, or the value of arg will be indeterminate.

If unsuccessful, these functions return -1 and, in C and C++, set errno as shown in Table 25.

Table 25 **errno values for cps_thread_create[n]**

errno value	Meaning
EAGAIN	The asymmetric thread could not be allocated
EINVAL	node is not a valid logical hypernode ID in the program-assigned system

Nested calls to cps_thread_create and/or cps_thread_createn are permitted.

For more information, refer to the cps_thread_create(3) man page.

cps_thread_exit

This function terminates the asymmetric thread call made by cps_thread_create. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_THREAD_EXIT()
```

In C and C++:

```
int cps_thread_exit(void);
```

Upon successful completion of func (specified in cps_thread_create), asymmetric threads spawned with cps_thread_create will automatically call cps_thread_exit. The function is provided for cases in which you wish to terminate an asymmetric thread before func normally returns.

If successful, cps_thread_exit does not return. If unsuccessful it returns -1 and, in C and C++, sets errno to EINVAL if it was called from a symmetric thread.

For more information, refer to the cps_thread_create(3) man page.

cps_thread_wait

This function can be used to wait until all asymmetric threads have terminated, or to find out the number of active asymmetric threads. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_THREAD_WAIT(FLAG)
```

In C and C++:

```
int cps_thread_wait(const int *flag);
```

If `flag` is set, this routine waits until all asymmetric threads in the program have terminated. If `flag` is not set, it returns the number of active asymmetric threads in the program.

`cps_thread_wait` cannot be called with `flag` set from an asymmetric thread because the active state of the calling thread will prevent the function from returning, resulting in deadlock.

To use `cps_thread_wait` in an asymmetric thread to wait for all child asymmetric threads to terminate, you must know how many asymmetric threads were spawned before the calling thread. With this information, you can construct a loop that calls `cps_thread_wait` with `flag` equal to 0 until the number of active threads is equal to the number of previously spawned threads plus 1 (the calling thread), as shown in the following Fortran example:

```
10     IWAIT = CPS_THREAD_WAIT(0) ! FIND NUMBER OF ASYM THREADS
       IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
       IF(IWAIT .GT. PREVTHRDS+1) GOTO 10  ! SPIN UNTIL ALL
                                           ! CHILDREN TERMINATE
```

Here, `CPS_THREAD_WAIT` returns the total number of asymmetric threads at line 10; the next line is a routine error trap, and the third line checks the returned number against the known number of threads that are not children of the calling thread. `PREVTHRDS` is a user-defined and user-incremented variable. The loop cannot terminate until the returned number indicates that all of the calling thread's children have terminated.

If unsuccessful, `cps_thread_wait` returns -1 and, in C and C++, sets `errno` to `EDEADLK` if it was called from an asymmetric thread or the child of an asymmetric thread with `flag` set.

For more information, refer to the cps_thread_create(3) man page.

Thread information and attribute functions

These functions provide information on active parallel threads, the system configuration, and CPS attributes. For information beyond that which follows, refer to the cps_info(3) man page.

`cps_stid`

This function returns the spawn thread ID of the calling thread. In Fortran `cps_stid` has the following form:

```
INTEGER FUNCTION CPS_STID()
```

In C and C++:

```
stid_t cps_stid(void);
```

Spawn thread IDs range from 0..*n*-1, where *n* is the number of currently active symmetric threads in the current spawn context (refer to `cps_nsthreads`).

If unsuccessful, this function returns -1. Because asymmetric threads have no spawn thread IDs, `cps_stid()` returns -1 when called from an asymmetric thread.

`cps_ktid`

This function returns the kernel thread ID of the calling thread. In Fortran, `cps_ktid` has the following form:

```
INTEGER FUNCTION CPS_KTID()
```

In C and C++:

```
tid_t cps_ktid(void);
```

Note that kernel threads IDs are generated with no regularity; they are simply unique IDs.

cps_nsthreads

This function returns the number of threads in the current CPS *spawn context*. Each spawn establishes a spawn context and the number returned by `cps_nsthreads` can vary from spawn to spawn. For example, if you have a `cps_ppcall` that spawns 2 threads nested within a `cps_ppcall` that spawns 4 threads, `cps_nsthreads` returns 2 when called from the inner spawn context, and 4 when called from the outer spawn context.

In Fortran, `cps_nsthreads` has the following form:

```
INTEGER FUNCTION CPS_NSTHREADS()
```

In C and C++:

```
int cps_nsthreads(void);
```

cps_plevel

This function can be used to determine the current level of parallelism. In Fortran, it has the following form:

```
INTEGER FUNCTION CPS_PLEVEL()
```

In C and C++:

```
int cps_plevel(void);
```

The return value is a bit mask. The possible return values are a sum of those shown in Table 26.

Table 26 **cps_plevel return values**

C/C++ symbolic constant name	Value	Meaning
CPS_PL_NONE	0	No parallel threads
CPS_PL_PARALLEL	1	Asymmetric parallel threads are active
CPS_PL_NODE	2	Hypernode-parallel threads are active
CPS_PL_NTHREAD	4	Parallel threads are active on the current hypernode
CPS_PL_THREAD	8	Parallel threads are active across multiple hypernodes
CPS_PL_ASYMMETRIC	16	Current thread is an asymmetric thread or a child of one

cps_node_id

This function returns the logical ID of the hypernode on which the calling thread is executing. In Fortran, `cps_node_id` has the following form:

```
INTEGER CPS_NODE_ID()
```

In C and C++:

```
int cps_node_id(void);
```

Logical hypernode IDs range from 0..*n*-1, where *n* is the number of available hypernodes in the system. Logical IDs are assigned in the order in which your program occupies the system. The hypernode that your program's thread 0 runs on is considered logical hypernode 0; any hypernodes it expands to later are assigned increasing logical ID numbers. Because the operating system starts a program on the least-loaded hypernode, logical hypernode IDs can differ between programs due to load balancing; thus two programs running on the same system are unlikely to address identical hypernodes with identical logical IDs.

Logical hypernode IDs have no correlation to physical hypernode IDs, which are unique for each hypernode at the machine level.

cps_node_cpus

This function returns the number of threads available to the caller on the hypernode on which it is running. In Fortran it has the following form:

```
INTEGER CPS_NODE_CPUS()
```

In C and C++:

```
int cps_node_cpus(void);
```

Note that the return value represents the number of threads visible to the calling application, and does not necessarily indicate the total number of threads on the hypernode.

cps_node_nthreads

This function returns the number of active threads belonging to the calling application that are running on the hypernode on which the call is executing. In Fortran it has the following form:

```
INTEGER CPS_NODE_NTHREADS()
```

In C and C++:

```
int cps_node_nthreads(void);
```

Active threads include threads spawned by `cps_ppcall`, `cps_ppcalln`, or `cps_thread_create`, as well as threads spawned automatically due to compiler-generated parallelism.

`cps_is_parallel`

This function returns 1 if the program has parallel code and can go parallel; otherwise it returns 0. In Fortran it has the following form:

```
INTEGER CPS_IS_PARALLEL()
```

In C and C++:

```
int cps_is_parallel(void);
```

`cps_complex_cpus`

This function returns the total number of threads available to the application. In Fortran, it has the following form:

```
INTEGER CPS_COMPLEX_CPUS()
```

In C and C++:

```
int cps_complex_cpus(void);
```

`cps_complex_nthreads`

This function returns the number of active threads belonging to the calling application that are running on the system on which the call is executing. In Fortran, it has the following form:

```
INTEGER CPS_COMPLEX_NTHREADS()
```

In C and C++:

```
int cps_complex_nthreads(void);
```

Active threads include threads spawned by `cps_ppcall`, `cps_ppcalln`, or `cps_thread_create`, as well as threads spawned automatically due to compiler-generated parallelism.

`cps_complex_nodes`

This function returns the total number of logical hypernodes available to the application from which it is called in the application's system. In Fortran it has the following form:

```
INTEGER CPS_COMPLEX_NODES()
```

In C and C++:

```
int cps_complex_nodes(void);
```

cps_set_threads

This function sets the number of threads CPS has in its pool of free threads to `count`, where `count` is in the range from 1 to the value returned by `cps_complex_cps`. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_SET_THREADS(COUNT)
INTEGER COUNT
```

In C and C++:

```
int cps_set_threads(const int *count)
```

If successful, `cps_set_threads` returns 0. Otherwise, -1 is returned, and, in C and C++, `errno` is set to:

- `EINVAL` if `count` is greater than the value returned by `cps_complex_cpus` or if `count` is less than or equal to 0
- `EBUSY` if there are any active CPS threads (other than the main thread)

cps_topology

This function fills an array with the application's view of the topology of the system on which it is running. In Fortran it has the following form:

```
INTEGER CPS_TOPOLOGY(NODES,N)INTEGER NODES(N)
```

In C and C++:

```
int cps_topology(int nodes[], int n)
```

Each element of the `NODES` or `nodes` array corresponds to a logical hypernode, with the first element corresponding to logical hypernode 0, the second to logical hypernode 1, etc. Therefore, given default subscripting in Fortran, `NODES(1)` represents logical hypernode 0. Given default subscripting in C and C++, `nodes[0]` corresponds to logical hypernode 0. On return from `cps_topology`, each element contains the number of threads running on the corresponding hypernode that belong to the calling application.

`N` or `n` represents the number of elements in the array. If there are more elements than actual hypernodes, the remaining elements are set to 0.

cps_wait_attr

As described in Chapter 3, “Compiler optimizations,” idle threads can either be suspended or spin-waiting. This function allows you to get or change the current CPSlib thread wait attributes. In Fortran it has the following form:

```
INTEGER CPS_WAIT_ATTR(CMD, WAIT)
INTEGER CMD, WAIT(2)
```

In C and C++:

```
typedef struct {
  int wait_attr; /* idle wait attribute */
  int wait_time; /*wait time in milliseconds*/
} cps_spin_state_t;
int cps_wait_attr(int *cmd,cps_wait_attr_t *wait);
```

Where `CMD` or `cmd`, depending on its contents, indicates whether to get or change the current wait attributes. Accepted values are shown in Table 27.

Table 27 Accepted CMD/cmd values

C/C++ symbolic constant name	Value	Meaning
`CPS_GETWAIT`	0	Get the current wait attribute values and store them in the `WAIT` array or in the structure pointed to by `wait`
`CPS_SETWAIT`	1	Set the wait attributes to the values in the `WAIT` array or pointed to by `wait`; these values become effective after the next join or `cps_thread_exit` is executed
`CPS_SETWAITI`	2	Set the wait attributes as `CPS_SETWAIT` does, but force all active threads to join immediately

The `WAIT` array or the structure pointed to by `wait` contain the new or current wait attributes, depending on the value of `cmd`. `WAIT(2)` and `wait->wait_attr` take one of the values shown in Table 28.

Table 28 **WAIT(2)/wait->wait_attr values**

C/C++ symbolic constant name	Value	Meaning
CPS_SUSPEND	1	Suspend the thread after waiting the time specified in WAIT(2) or wait->wait_time
CPS_SPINWAIT	2	Spin-wait until the thread is reactivated

WAIT(2) and wait->wait_time take an integer representing the number of milliseconds the thread is to spin-wait before suspending itself (assuming the wait attribute is set to CPS_SUSPEND). These values default to 50 ms for non-oversubscribed threads and 0 ms for oversubscribed threads.

If unsuccessful, cps_wait_attr returns -1 and, in C and C++, sets errno to EINVAL if:

- wait_time is greater than CPS_WAIT_MAX, and wait_attr is set to CPS_SUSPEND
- wait_attr is not set to CPS_SUSPEND or CPS_SPINWAIT or cmd is not CPS_GETWAIT, CPS_SETWAIT, or CPS_SETWAITI

High-level synchronization functions

These routines manipulate the barriers and mutexes that are used for synchronization. CPSlib barriers can be used to construct barriers such as those that can be constructed with compiler directives as described in Chapter 6, “Advanced shared-memory programming.” Mutexes are areas of mutual exclusion, and are analogous to directive-controlled critical sections described in Chapter 6.

These constructs offer a lower degree of automation and a higher degree of control than those directive-specified constructs described in Chapter 6. However, they offer a higher degree of automation and a lower degree of control than the constructs that can be manually built using low-level synchronization functions described in the following section.

All of the functions described in this section return 0 on success and -1 on failure.

For information beyond that which follows, refer to the cps_barrier(3) and cps_mutex(3) man pages.

`cps_barrier_alloc`

This function allocates the barrier `barr` and sets its associated shared counter to zero. When each thread reaches the barrier, it increments the counter; when the counter equals the number of parallel threads, all threads may proceed.

In Fortran this function has the form:

```
INTEGER FUNCTION CPS_BARRIER_ALLOC(BARR)
INTEGER BARRIER
```

In C and C++:

```
int cps_barrier_alloc(barrier_t *barr);
```

If this function fails when called from C or C++, `errno` is set to `ENOMEM` if the memory required for `barr` cannot be allocated.

`cps_barrier_free`

This function releases the barrier `barr`. In Fortran it has the form:

```
INTEGER FUNCTION CPS_BARRIER_FREE(BARR)
INTEGER BARR
```

In C and C++:

```
int cps_barrier_free(barrier_t *barr);
```

If this function fails when called from C or C++, `errno` is set to `EINVAL` if `barr` was not allocated with a CPSlib barrier allocation function, or to `EBUSY` if the counter associated with `barr` is nonzero.

`cps_barrier`

This function increments the shared counter associated with the barrier `barr`. When the value of the shared counter is equal to the argument `nthreads`, the function returns, and the counter is set to zero. In Fortran, `cps_barrier` has the following form:

```
INTEGER FUNCTION CPS_BARRIER(BARR,NTHREADS)
INTEGER BARR, NTHREADS
```

In C and C++:

```
int cps_barrier(barrier_t *barr, const int *nthreads);
```

If this function fails when called from C or C++ because `barr` was not allocated with a CPSlib barrier allocation function, `errno` is set to `EINVAL`.

`cps_mutex_alloc`

This function allocates the mutex `mutx` and unlocks it. In Fortran it has the form:

```
INTEGER FUNCTION CPS_MUTEX_ALLOC(MUTX)
INTEGER MUTX
```

In C and C++:

```
int cps_mutex_alloc(cps_mutex_t *mutx);
```

This function does not check whether `mutx` is already allocated; therefore, when successful, it always allocates a new mutex. When it fails, `mutx` is set to NULL (in Fortran, `MUTX` is set to 0).

If this function fails when called from C or C++, it sets `errno` to `ENOMEM` if it cannot allocate the required memory.

`cps_mutex_free`

This function releases the mutex `mutx`. In Fortran it has the form:

```
INTEGER FUNCTION CPS_MUTEX_FREE(MUTX)
INTEGER MUTX
```

In C and C++:

```
int cps_mutex_free(cps_mutex_t *mutx);
```

If this function is successful when called from C or C++, `mutx` is set to NULL. In Fortran, `MUTX` is undefined.

If unsuccessful when called from C or C++, it sets `errno` to `EINVAL` if mutx was not allocated by a CPSlib allocation function, or to `EBUSY` if `mutx` has already been acquired.

`cps_mutex_lock`

If the mutex `mutx` is unlocked, this function acquires it and returns; if it is locked by another thread, `cps_mutex_lock` will wait until it is acquired before returning.

In Fortran, `cps_mutex_lock` has the following form:

```
INTEGER FUNCTION CPS_MUTEX_LOCK(MUTX)
INTEGER MUTX
```

In C and C++:

```
int cps_mutex_lock(cps_mutex_t *mutx);
```

If the calling thread has already acquired `mutx` this function returns -1 and, in C and C++, sets `errno` to `EDEADLK`.

`cps_mutex_unlock`

This function releases the mutex `mutx` so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_MUTEX_UNLOCK(MUTX)
INTEGER MUTX
```

In C and C++:

```
int cps_mutex_unlock(cps_mutex_t *mutx);
```

cps_mutex_trylock

This function attempts to acquire `mutx`; if `mutx` is already locked by another thread, `cps_mutex_trylock` will return -1.

In Fortran, `cps_mutex_trylock` has the following form:

```
INTEGER FUNCTION CPS_MUTEX_TRYLOCK(MUTX)
INTEGER MUTX
```

In C and C++:

```
int cps_mutex_trylock(cps_mutex_t *mutx);
```

If the calling thread has already acquired `mutx` this function returns -1 and, in C and C++, sets `errno` to `EDEADLK`.

Low-level synchronization functions

These functions manipulate the counters and semaphores used for low-level synchronization. These functions require you to create and manually control your own synchronization semaphores.

Semaphores can be cache- or memory-based. Cache-based semaphores can be stored in the processor data cache, making them best for use in situations that generate minimal semaphore contention. Memory-based semaphores are never brought into the processor data cache, making them preferable in situations that generate semaphore contention.

Because each semaphore has an associated counter, it can be used both as a lock (to implement, for example, a critical section) and a counter (to implement, for example, an ordered section).

For information beyond that which follows, refer to the cps_sema(3) man page.

c_init32

This function allocates the cache-based semaphore `cs` and initializes its associated counter to `value`. In Fortran it has the following form:

```
INTEGER FUNCTION C_INIT32(CS, VALUE)
INTEGER CS, VALUE
```

In C and C++:

```
int c_init32(cache_sema_t *cs,const int *value);
```

If successful, `c_init32` returns the counter value; otherwise it returns -1.

`c_free32`

This function frees the cache-based semaphore `cs` and sets it to NULL on success. In Fortran it has the following form:

```
INTEGER FUNCTION C_FREE32(CS)
INTEGER CS
```

In C and C++:

```
int c_free32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

`c_lock`

This function acquires a cache-based semaphore `cs`. In Fortran it has the following form:

```
INTEGER FUNCTION C_LOCK(CS)
INTEGER CS
```

In C and C++:

```
int c_lock(cache_sema_t *cs);
```

If the semaphore is already acquired, `c_lock` will wait until the semaphore is released before returning. It may or may not give up the processor in the interim.

`c_unlock`

This function releases the cache-based semaphore `cs` so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION C_UNLOCK(CS)
INTEGER CS
```

In C and C++:

```
int c_unlock(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

c_cond_lock

If the cache-based semaphore `cs` is available, this function acquires it; otherwise, it returns -1 without waiting and allows execution to continue in the calling thread. In Fortran it has the following form:

```
INTEGER FUNCTION C_COND_LOCK(CS)
INTEGER CS
```

In C and C++:

```
int c_cond_lock(cache_sema_t *cs);
```

c_fetch32

This function returns the value of the counter associated with the cache-based semaphore `cs`. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH32(CS)
INTEGER CS
```

In C and C++:

```
int c_fetch32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

c_fetch_and_inc32

This function increments the value of the counter associated with the semaphore `cs` and returns the old value.

In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_INC32(CS)
INTEGER CS
```

In C and C++:

```
int c_fetch_and_inc32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

`c_fetch_and_dec32`

This function decrements the value of the counter associated with the semaphore `cs` and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_DEC32(CS)
INTEGER CS
```

In C and C++:

```
int c_fetch_and_dec32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

`c_fetch_and_clear32`

This function returns the current value of the counter associated with the semaphore `cs` and clears the counter. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_CLEAR32(CS)
INTEGER CS
```

In C and C++:

```
int c_fetch_and_clear32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

`c_fetch_and_add32`

This function adds `value` to the counter associated with the semaphore `cs` and returns the old `value`. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_ADD32(CS, VALUE)
INTEGER CS, VALUE
```

In C and C++:

```
int c_fetch_and_add32(cache_sema_t *cs,
                      const int *value);
```

If unsuccessful, this function returns -1.

c_fetch_and_set32

This function returns the current value of the counter associated with the semaphore `cs`, and sets the semaphore to the new value contained in `newval`. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_SET32(CS, NEWVAL)
INTEGER CS, NEWVAL
```

In C and C++:

```
int c_fetch_and_set32(cache_sema_t *cs,
                      const int *newval);
```

If unsuccessful, this function returns -1.

m_init32

This function allocates the memory-based semaphore `ms` and initializes the counter associated with it to value. In Fortran it has the following form:

```
INTEGER FUNCTION M_INIT32(MS,VALUE)
INTEGER MS, VALUE
```

In C and C++:

```
int m_init32(mem_sema_t *ms,const int *value);
```

If successful, this function returns the counter value; otherwise it returns -1.

m_free32

This function releases the memory-based semaphore `ms` and sets it to NULL on success. In Fortran it has the following form:

```
INTEGER FUNCTION M_FREE32(MS)
INTEGER MS
```

In C and C++:

```
int m_free32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

m_lock

This function acquires the memory-based semaphore `ms`. If the semaphore is already acquired, `m_lock` will wait until the semaphore is released before returning. It may or may not give up the processor in the interim. In Fortran it has the following form:

```
INTEGER FUNCTION M_LOCK(MS)
INTEGER MS
```

In C and C++:

```
int m_lock(mem_sema_t *ms);
```

m_unlock

This function releases the memory-based semaphore `ms` so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION M_UNLOCK(MS)
INTEGER MS
```

In C and C++:

```
int m_unlock(mem_sema_t *ms)
```

If unsuccessful, this function returns -1.

m_cond_lock

If the memory-based semaphore ms is available, this function acquires it; otherwise it returns -1 without waiting and allows execution to continue in the calling thread. In Fortran it has the following form:

```
INTEGER FUNCTION M_COND_LOCK(MS)
INTEGER MS
```

In C and C++:

```
int m_cond_lock(mem_sema_t *ms);
```

m_fetch32

This function returns the value of the counter associated with the memory-based semaphore ms. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH32(MS)
INTEGER MS
```

In C and C++:

```
int m_fetch32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

m_fetch_and_inc32

This function increments the value of the counter associated with the semaphore ms and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_INC32(MS)
INTEGER MS
```

In C and C++:

```
int m_fetch_and_inc32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

m_fetch_and_dec32

This function decrements the value of the counter associated with the semaphore ms and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_DEC32(MS)
INTEGER MS
```

In C and C++:

```
int m_fetch_and_dec32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

m_fetch_and_clear32

This function returns the current value of the counter associated with the semaphore ms and clears the counter. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_CLEAR32(MS)
INTEGER MS
```

In C and C++:

```
int m_fetch_and_clear32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

sync_routine directive and pragma

Among the most basic optimizations performed by an Exemplar compiler is code motion, which is described in Chapter 3, “Compiler optimizations.” This optimization can move some code across routine calls. If the routine call is to a synchronization or parallelization function and the code moved must execute on a certain side of it, this movement can cause wrong answers. Anytime you use CPSlib functions in Fortran or C rather than the directives or functions described in Chapter 4, “Basic shared-memory programming,” and Chapter 6, “Advanced shared-memory programming,” to synchronize or parallelize code, you must identify the functions with a sync_routine directive or pragma. sync_routine should be used to identify all CPSlib functions, as well as any user-written routines that accomplish synchronization or parallelization or hide calls to any synchronization or parallelization routines.

In Fortran, `sync_routine` has the following form:

`C$DIR SYNC_ROUTINE (`*routinelist*`)`

In C, it has the following form:

`#pragma _CNX sync_routine (`*routinelist*`)`

where

routinelist	is a comma-separated list of synchronization procedures.

`sync_routine` is only effective for the listed routines that lexically follow it in the file in which it appears.

Consider the following Fortran example:

```
      SUBROUTINE WORK(ARG1, ARG2, MUTX)
      INTEGER ARG1, ARG2, MUTX, CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK
C$DIR SYNC_ROUTINE(CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK)
      .
      .
      .
      DO I = 1, N
        .
        .
        .
        LCK = CPS_MUTEX_LOCK(MUTX)
        .
        .
        .
        LCK = CPS_MUTEX_UNLOCK(MUTX)
      ENDDO
      .
      .
      .
      END
```

Here, the subroutine WORK is called in parallel and contains a loop that contains a critical section protected by calls to CPSlib functions. Listing these CPSlib functions in a SYNC_PARALLEL directive at the beginning of the subroutine prevents the compiler from moving code out of the critical section.

An analogous C example follows:

```
#include <spp_prog_model.h>
work(int arg1, int arg2, int mutx) {
  int i, lck;
#pragma _CNX sync_routine(cps_mutex_lock, cps_mutex_unlock)
  .
  .
  .
#pragma _CNX loop_parallel(ivar=i)
  for(i=0; i<n; i++) {
    .
    .
    .
    lck = cps_mutex_lock(&mutx);
    .
    .
    .
    lck = cps_mutex_unlock(&mutx);
  }
}
```

Examples

The examples presented here demonstrate various constructs that can be programmed using the CPSlib functions described in the previous sections.

Symmetric parallelism

There are two forms of symmetric parallelism: block parallelism, and cyclic parallelism. The CPSlib functions used in the examples that follow are described in detail in the section “CPS library functions” on page 426.

Block parallelism

Block parallelism is the most commonly used form of parallelism; it is the form generated by default by Exemplar compilers. It involves splitting up the iterations of a loop into iteration blocks of similar size, and running each block on a separate processor.

A simple Fortran example that uses CPSlib to implement block parallelism follows. The CPSlib functions used here are described in detail in the “CPS library functions” section.

```
      PROGRAM CPSBLOCK
      REAL X(1000), Y(1000), Z(1000)
      INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS)
      EXTERNAL PARBLK ! REQUIRED BECAUSE PARBLK IS AN ARGUMENT
C INITIALIZE PARGS ARRAY:
      PARGS(1) = -2  ! ALLOCATE THREADS ON CALLING THREAD’S NODE
      PARGS(2) = 2   ! MINIMUM OF 2 THREADS
      PARGS(3) = CPS_NODE_CPUS() ! MAXIMUM # OF THREADS
      PARGS(4) = 1       ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
C SPAWN THREADS:
      ITHREAD = CPS_PPCALLN(PARGS, PARBLK, 4, X, Y, Z, NTHR)
C IF SPAWN FAILS, REPORT:
      IF (ITHREAD .LT. 0) PRINT *,’PPCALLN FAILED’
      .
      .  ! SERIAL CODE
      .
      END

      SUBROUTINE PARBLK (X, Y, Z, NTHR)
      REAL X(1000), Y(1000), Z(1000)
      INTEGER CPS_NSTHREADS, CPS_STID, STID, NTHR
C$DIR SYNC_ROUTINE(CPS_NSTHREADS, CPS_STID)
      STID = CPS_STID()      ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
      ITPERPROC = 1000/NTHR  ! COMPUTE ITERATIONS PER THREAD
      IEXCESS = 1000-ITPERPROC*NTHR  ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DO NOT HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = STID*ITPERROC + IEXCESS + 1
        MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
      ELSE
        MYSTART = STID * (ITPERPROC+1) + 1
        MYEND = MYSTART + (ITPERPROC+1) - 1
      ENDIF
C ACTUAL COMPUTATION:
      DO J = MYSTART, MYEND
        Z(J) = X(J) + Y(J)
      ENDDO
      RETURN
      END
```

This example calls CPS_NODE_CPUS to find the number of available threads, then calls CPS_PPCALLN to spawn parallel threads to run the subroutine PARBLK.

PARBLK then determines the number of iterations necessary per processor; if the number of processors does not integrally divide the number of iterations, it automatically adjusts some blocks to handle the extra iterations. Finally, the loop in PARBLK performs its body in parallel, with each thread operating on the appropriate iteration range.

Note the error trap immediately after the call to CPS_PPCALLN; this is important, as it provides the only means of knowing if the spawn failed.

Cyclic parallelism

Cyclic parallelism distributes consecutive iterations of a loop to separate processors. It is similar to the parallelism achieved through use of the `loop_parallel(ordered)` directive and pragma, but it does not order the iterations automatically; you must handle any necessary ordering manually. `loop_parallel(ordered)` is discussed in the section “`loop_parallel(ordered)`” on page 233 in Chapter 6, “Advanced shared-memory programming.”

A simple Fortran example that uses CPSlib to implement cyclic parallelism follows. The CPSlib functions used here are described in detail in the “CPS library functions” section.

```
      PROGRAM CPSCYCLE
      REAL X(1000), Y(1000), Z(1000), SUM
      INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS)
      EXTERNAL PARCYC ! PARCYC IS AN ARGUMENT
      READ *, NPROCSC
C INITIALIZE PARGS ARRAY:
      PARGS(1) = -2  ! ALLOCATE THREADS ON CALLING THREAD’S NODE
      PARGS(2) = 2   ! MINIMUM OF 2 THREADS
      PARGS(3) = CPS_NODE_CPUS() ! MAXIMUM # OF THREADS
      PARGS(4) = 1      ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
C SPAWN THREADS:
      ITHREAD = CPS_PPCALLN (PARGS, PARCYC, 4, X, Y, Z, NTHR)
C IF SPAWN FAILS, REPORT:
      IF (ITHREAD .LT. 0) PRINT *,’PPCALLN FAILED’
      .
      .  ! SERIAL CODE
      .
      END

      SUBROUTINE PARCYC (X, Y, Z, NTHR)
      REAL X(1000), Y(1000), Z(1000)
      INTEGER CPS_NSTHREADS, CPS_STID, STID, NTHR
C$DIR SYNC_ROUTINE(CPS_NSTHREADS, CPS_STID)
      STID = CPS_STID()      ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
C ACTUAL COMPUTATION:
      DO J = 1+STID, 1000, NTHR ! STEP BY NUMBER OF THREADS
         Z(J) = X(J) + Y(J)
      ENDDO
      RETURN
      END
```

This example works exactly like the block parallelism example, except the loop in PARCYC, its parallel subroutine, is cyclically parallel. Cyclic parallelism is accomplished here by offsetting the loop start value by spawn thread ID and stepping the loop by the number of parallel threads. This ensures that each thread computes a unique array element on every step of the loop; NTHR elements are computed per step. Contiguous STIDs compute contiguous elements.

Asymmetric parallelism

A simple Fortran program that implements asymmetric parallelism follows. The CPSlib functions used here are described in detail in the "CPS library functions" section.

```
      PROGRAM ASYM
      REAL X1(1000), X2(1000), Y1(1000), Y2(1000), Z(1000)
      INTEGER CPS_THREAD_CREATE, CPS_THREAD_WAIT
C$DIR SYNC_ROUTINE(CPS_THREAD_CREATE, CPS_THREAD_WAIT)
      COMMON /POINTS/ X1, X2, Y1, Y2
      EXTERNAL DISTANCE ! DISTANCE IS AN ARGUMENT
      .
      . ! SERIAL CODE
      . ! EXAMPLE CONTINUED
C SPAWN ASYMMETRIC THREAD TO EXECUTE SUBROUTINE DISTANCE:
      ITHREAD = CPS_THREAD_CREATE(-2, DISTANCE, Z)
      IF (ITHREAD .LT. 0) PRINT*, "THREAD_CREATE FAILED IN MAIN"
      .
      . ! THIS CODE RUNS IN PARALLEL WITH DISTANCE
      .
      IWAIT = CPS_THREAD_WAIT(1) ! WAIT FOR ALL ASYMMETRIC
                                 ! THREADS TO TERMINATE
      IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
      .
      . ! THIS CODE RUNS SERIALLY AFTER PARALLEL THREADS
      . ! TERMINATE
      END
      SUBROUTINE DISTANCE(Z)
      REAL X1(1000), X2(1000), Y1(1000), Y2(1000), Z(1000)
      REAL X3(1000), Y3(1000)
      INTEGER CPS_THREAD_CREATE, CPS_THREAD_WAIT
C$DIR SYNC_ROUTINE(CPS_THREAD_CREATE, CPS_THREAD_WAIT)
      COMMON /POINTS/ X1, X2, Y1, Y2
      EXTERNAL FINDX ! FINDX IS AN ARGUMENT
C SPAWN ASYMMETRIC THREAD TO EXECUTE SUBROUTINE FINDX:
      JTHREAD = CPS_THREAD_CREATE(-2, FINDX, X3)
      IF (JTHREAD .LT. 0) PRINT*, "THREAD_CREATE FAILED IN DISTANCE"

      DO I = 1, 1000 ! COMPUTE Y3 IN PARALLEL WITH FINDX
        Y3(I) = (Y2(I) - Y1(I))**2
      ENDDO
10    IWAIT = CPS_THREAD_WAIT(0) ! FIND NUMBER OF ASYM THREADS
      IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
      IF(IWAIT .GT. 1) GOTO 10  ! SPIN UNTIL ONLY THIS THREAD
                                ! IS ACTIVE
      DO I = 1, 1000 ! COMPUTE Z SERIALLY AFTER X3 AND Y3
        Z(I) = SQRT(X3(I) + Y3(I))
      ENDDO
      RETURN
      END

      SUBROUTINE FINDX(X3) ! RUNS IN PARALLEL WITH COMPUTATION
                           ! OF Y3
      REAL X1(1000), X2(1000), Y1(1000), Y2(1000), X3(1000)
      COMMON /POINTS/ X1, X2, Y1, Y2
      DO I = 1, 1000
        X3(I) = (X2(I) - X1(I))**2
      ENDDO
      RETURN
      END
```

In this example, the arrays X3 and Y3 must be computed before the array Z can be computed. The main program spawns an asymmetric parallel thread to run DISTANCE, which spawns an asymmetric thread to run FINDX. DISTANCE then computes Y3 while FINDX computes X3; all the while, the main program can be doing other work in parallel with both subroutines. When DISTANCE is done with Y3, it waits until FINDX is done with X3, then computes Z. The main program waits until DISTANCE is done, then proceeds with more work.

Note the way in which `CPS_THREAD_WAIT` is used when called from `DISTANCE`; this is explained further in the “CPS library functions” section.

Synchronization using high-level functions

This section demonstrates how to use barriers and mutexes to synchronize symmetrically parallel code.

Barriers

Remember that, when you use `cps_ppcall()` to spawn symmetric parallelism, before the function returns, a join operation takes place after all spawned threads terminate. This join is an implicit barrier, since thread 0 cannot proceed until all parallel threads terminate. In many cases, this is the only barrier synchronization you will require.

However, the `cps_barrier()` high-level synchronization functions allow you to explicitly create barriers if necessary.

The following Fortran example is similar to the symmetric parallelism example in the section “Block parallelism” on page 454 except that instead of relying on the implicit barrier contained in the call to CPS_PPCALL( ), it contains an explicit CPS_BARRIER( ) in the subroutine SUMMER.

```
      PROGRAM BAR
      REAL A(1000)
      REAL SUM(:), TOTSUM
      INTEGER PARGS(4), SUMBAR, CPS_NODE_CPUS, CPS_PPCALLN
      INTEGER CPS_BARRIER_ALLOC, CPS_BARRIER_FREE
C$DIR SYNC_ROUTINE(CPS_BARRIER_ALLOC,CPS_BARRIER_FREE)
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS,CPS_PPCALLN)
      EXTERNAL SUMMER
      ALLOCATABLE SUM
      NCPUS = CPS_NODE_CPUS()
      ALLOCATE(SUM(0:NCPUS-1)) ! ONE ELEMENT FOR EACH CPU
      PARGS(1) = -2     ! ALLOCATE THREADS ON CALLING THREAD’S NODE
      PARGS(2) = 2      ! MINIMUM OF 2 THREADS PER NODE
      PARGS(3) = NCPUS  ! MAXIMUM # OF THREADS PER NODE
      PARGS(4) = 1      ! ALLOCATE MULTIPLE THREADS PER NODE
      DO I = 0, NCPUS-1 ! INITIALIZE SUM
        SUM(I) = 0.0
      ENDDO
      .
      .  ! SERIAL CODE
      .
      IERR = CPS_BARRIER_ALLOC(SUMBAR) ! ALLOCATE BARRIER
      IF (IERR .LT. 0) PRINT*, "BARRIER ALLOCATION FAILED"


C SPAWN PARALLEL THREADS:
      IERR = CPS_PPCALLN(PARGS,SUMMER,5,A,SUM,TOTSUM,SUMBAR,NCPUS)
      IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
      IERR = CPS_BARRIER_FREE(SUMBAR) ! FREE BARRIER
      IF (IERR .LT. 0) PRINT*, "BARRIER FREE FAILED"
      .
      .  ! SERIAL CODE
      .
      END
      SUBROUTINE SUMMER(A,SUM,TOTSUM,SUMBAR,NCPUS)
      INTEGER  STID, NTHR, SUMBAR
      INTEGER CPS_STID, CPS_NSTHREADS, CPS_BARRIER
C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHREADS, CPS_BARRIER)
      REAL A(1000), SUM(0:NCPUS-1), TOTSUM
      STID = CPS_STID()      ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
      ITPERPROC = 1000/NTHR  ! COMPUTE ITERATIONS PER THREAD
      IEXCESS = 1000-ITPERPROC*NTHR  ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DO NOT HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = STID*ITPERROC + IEXCESS + 1
        MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
      ELSE
        MYSTART = STID * (ITPERPROC+1) + 1
        MYEND = MYSTART + (ITPERPROC+1) - 1
      ENDIF
C ACTUAL COMPUTATION:
      DO J = MYSTART, MYEND ! EACH THREAD COMPUTES LOCAL SUM
        SUM(STID) = SUM(STID) + A(J)
      ENDDO
C WAIT UNTIL ALL THREADS ARE DONE COMPUTING THEIR PORTION OF SUM:
      IERR = CPS_BARRIER(SUMBAR, NTHR)
      IF (IERR .LT. 0) PRINT*, "BARRIER FAILED"
      IF(STID .EQ. 0) THEN ! THREAD 0 COMPUTES TOTAL SUM
        DO I = 0, NTHR-1
          TOTSUM = TOTSUM + SUM(I)
        ENDDO
      ENDIF
      RETURN
      END
```

Here, the subroutine SUMMER is called in parallel to compute the sum of the elements of array A. Each parallel thread computes its own sum in an element of the array SUM. The CPS_BARRIER function is used to prevent execution of any further code until all threads have finished computing their portion of SUM. When CPS_BARRIER returns, thread 0 computes TOTSUM, and SUMMER returns.

Mutexes

CPSlib mutexes allow you to limit access to the sections of code they delimit to one thread at a time, allowing you to construct critical sections similar to those discussed in Chapter 4, “Basic shared-memory programming.”

In the following Fortran example, the routine SUMMER performs the same task it did in the preceding barrier example. However, here access to the TOTSUM computation takes place in fully parallel code; it is limited to one thread at a time by the mutex SUMMUTEX. This eliminates the need for each thread to compute independent SUM arrays as in the preceding barrier example.

```
      PROGRAM MUT
      REAL A(1000)
      REAL TOTSUM
      INTEGER PARGS(4), SUMMUTEX
      INTEGER CPS_NODE_CPUS,CPS_PPCALLN
      INTEGER CPS_MUTEX_ALLOC,CPS_MUTEX_FREE
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS,CPS_PPCALLN)
C$DIR SYNC_ROUTINE(CPS_MUTEX_ALLOC,CPS_MUTEX_FREE)
      EXTERNAL SUMMER
      NCPUS = CPS_NODE_CPUS()
      PARGS(1) = -2     ! ALLOCATE THREADS ON CALLING THREAD’S NODE
      PARGS(2) = 2      ! MINIMUM OF 2 THREADS PER NODE
      PARGS(3) = NCPUS  ! MAXIMUM # OF THREADS PER NODE
      PARGS(4) = 1      ! ALLOCATE MULTIPLE THREADS PER NODE
      TOTSUM = 0.0      ! INITIALIZE TOTSUM
      .
      .   ! SERIAL CODE
      .
      IERR = CPS_MUTEX_ALLOC(SUMMUTEX)  ! ALLOCATE MUTEX
      IF (IERR .LT. 0) PRINT*, "MUTEX ALLOCATION FAILED"
```

```
C SPAWN PARALLEL THREADS:
      IERR = CPS_PPCALLN(PARGS,SUMMER,3,A,TOTSUM,SUMMUTEX)
      IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
      IERR = CPS_MUTEX_FREE(SUMMUTEX)   ! FREE MUTEX
      IF (IERR .LT. 0) PRINT*, "MUTEX FREE FAILED"
      .
      .   ! SERIAL CODE
      .
      END

      SUBROUTINE SUMMER(A,TOTSUM,SUMMUTEX)
      INTEGER  STID, NTHR, SUMMUTEX
      INTEGER CPS_STID, CPS_NSTHREADS
      INTEGER CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK
C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHREADS)
C$DIR SYNC_ROUTINE(CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK)
      REAL A(1000),TOTSUM
      STID = CPS_STID()      ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
      ITPERPROC = 1000/NTHR  ! COMPUTE ITERATIONS PER THREAD
      IEXCESS = 1000-ITPERPROC*NTHR  ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DO NOT HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = STID*ITPERROC + IEXCESS + 1
        MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
      ELSE
        MYSTART = STID * (ITPERPROC+1) + 1
        MYEND = MYSTART + (ITPERPROC+1) - 1
      ENDIF
C ACTUAL COMPUTATION:
      DO J = MYSTART, MYEND
C MUTEX LIMITS ACCESS TO TOTSUM TO ONE THREAD AT A TIME:
        IERR = CPS_MUTEX_LOCK(SUMMUTEX)
        IF (IERR .LT. 0) PRINT*, "MUTEX LOCK FAILED"
        TOTSUM = TOTSUM + A(J)
        IERR = CPS_MUTEX_UNLOCK(SUMMUTEX)
        IF(IERR .LT. 0) PRINT*, "MUTEX UNLOCK FAILED"
      ENDDO
      RETURN
      END
```

Here, as in the barrier example, SUMMER is called in parallel. Each parallel thread then waits until it can lock SUMMUTEX before updating TOTSUM.

Synchronization using low-level functions

This section demonstrates how to use semaphores to synchronize symmetrically parallel code.

Critical sections

Critical sections like the one in the preceding mutex example can be implemented in a similar fashion using cache-based or memory-based semaphores.

The following Fortran example is identical to the mutex example, but implements the critical section using a memory-based semaphore instead of a mutex:

```
      PROGRAM SEM
      REAL A(1000)
      REAL TOTSUM
      INTEGER PARGS(4), SUMSEM, SEMCNT
      INTEGER CPS_NODE_CPUS, CPS_PPCALLN, M_INIT32, M_FREE32
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS, CPS_PPCALLN, M_INIT32, M_FREE32)
      EXTERNAL SUMMER
      NCPUS = CPS_NODE_CPUS()
      PARGS(1) = -2     ! ALLOCATE THREADS ON CALLING THREAD'S NODE
      PARGS(2) = 2      ! MINIMUM OF 2 THREADS PER NODE
      PARGS(3) = NCPUS  ! MAXIMUM # OF THREADS PER NODE
      PARGS(4) = 1      ! ALLOCATE MULTIPLE THREADS PER NODE
      TOTSUM = 0.0      ! INITIALIZE TOTSUM
      SEMCNT = 0 ! COUNTER FOR SEMAPHORE; VALUE IS IRRELEVANT
      .
      .  ! SERIAL CODE
      .
      IERR = M_INIT32(SUMSEM,SEMCNT) ! ALLOCATE SUMSEM
      IF (IERR .LT. 0) PRINT*, "SEMAPHORE ALLOCATION FAILED"
      IERR = CPS_PPCALLN(PARGS,SUMMER,3,A,TOTSUM,SUMSEM)
      IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
      IERR = M_FREE32(SUMSEM)
      IF (IERR .LT. 0) PRINT*, "SEMAPHORE FREE FAILED"
      .
      .  ! SERIAL CODE
      .
      END

      SUBROUTINE SUMMER(A,TOTSUM,SUMSEM)
      INTEGER  STID, NTHR, SUMSEM
      INTEGER CPS_STID, CPS_NSTHREADS, M_LOCK, M_UNLOCK
      REAL A(1000),TOTSUM


C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHREADS, M_LOCK, M_UNLOCK)
      STID = CPS_STID()      ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
      ITPERPROC = 1000/NTHR  ! COMPUTE ITERATIONS PER THREAD
      IEXCESS = 1000-ITPERPROC*NTHR  ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DO NOT HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = STID*ITPERROC + IEXCESS + 1
        MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
      ELSE
        MYSTART = STID * (ITPERPROC+1) + 1
        MYEND = MYSTART + (ITPERPROC+1) - 1
      ENDIF
C ACTUAL COMPUTATION:
      DO J = MYSTART, MYEND
C SEMAPHORE LIMITS ACCESS TO TOTSUM TO ONE THREAD AT A TIME:
        IERR = M_LOCK(SUMSEM)
        IF (IERR .LT. 0) PRINT*, "SEMAPHORE LOCK FAILED"
        TOTSUM = TOTSUM + A(J)
        IERR = M_UNLOCK(SUMSEM)
        IF(IERR .LT. 0) PRINT*, "SEMAPHORE UNLOCK FAILED"
      ENDDO
      RETURN
      END
```

Here as in the mutex example, SUMMER is called in parallel. Each parallel thread then waits until it can lock the memory-based semaphore SUMSEM before updating TOTSUM.

Ordered sections

Semaphores can also be used to construct ordered sections such as those constructed using the `loop_parallel(ordered)`, `begin_ordered_section` and `end_ordered_section` directives and pragmas, which are described in Chapter 6, “Advanced shared-memory programming.”

The parallel loop in the following Fortran example contains a backward LCD, which is isolated using low-level synchronization functions so that the threads must execute the LCD in iteration order.

```
      PROGRAM ORDERED ! DEMONSTRATES ORDERED SECTIONS USING CPS
                      ! LOW LEVEL SYNCHRONIZATION
      REAL X(1000), Y(1000)
      INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
      INTEGER M_INIT32, M_FREE32
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS, M_INIT32, M_FREE32)
      INTEGER ORDSEM, SEMCNT
      EXTERNAL ORDWORK
      PARGS(1) = -2  ! ALLOCATE THREADS CALLING THREAD'S NODE
      PARGS(2) = 2   ! MINIMUM OF 2 THREADS
      PARGS(3) = CPS_NODE_CPUS()  ! MAXIMUM OF NPROCS THREADS
      PARGS(4) = 1   ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
      SEMCNT = 0
      .
      .  ! SERIAL CODE
      .
      IERR = M_INIT32(ORDSEM,SEMCNT) ! ALLOCATE ORDSEM
      IF (IERR .LT. 0) PRINT*, "SEMAPHORE ALLOCATION FAILED"
C SPAWN THREADS:
      ITHREAD = CPS_PPCALLN(PARGS,ORDWORK,5,X,Y,NTHR,ORDSEM,SEMCNT)
      IF (ITHREAD .LT. 0) PRINT *,"PPCALLN FAILED"
      IERR = M_FREE32(ORDSEM)
      IF (IERR .LT. 0) PRINT*, "SEMAPHORE FREE FAILED"
      .
      .  ! SERIAL CODE
      .
      END
      SUBROUTINE ORDWORK (X, Y, NTHR,ORDSEM,SEMCNT)
      REAL X(1000), Y(1000)
      INTEGER CPS_NSTHREADS, CPS_STID, M_FETCH32
      INTEGER ORDSEM,SEMCNT,STID, NTHR, CNTVAL
      INTEGER M_FETCH_AND_INC32, M_FETCH_AND_CLEAR32
C$DIR SYNC_ROUTINE(CPS_NSTHREADS, CPS_STID, M_FETCH32)
C$DIR SYNC_ROUTINE(M_FETCH_AND_INC32, M_FETCH_AND_CLEAR32)
      STID = CPS_STID()      ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED

C ACTUAL COMPUTATION:
      DO J = 2+STID, 1000, NTHR ! CYCLIC DECOMPOSITION
        .
        .  ! DEPENDENCE-FREE PARALLEL CODE
        .
10      CNTVAL = M_FETCH32(ORDSEM) ! GET SEMAPHORE COUNTER VALUE
        IF(CNTVAL .EQ. STID) THEN  ! IF IT'S MY STID'S TURN
C PERFORM LCD COMPUTATION:
          X(J) = X(J-1) + Y(J)
          IF(CNTVAL .GE. NTHR-1) THEN          ! HIGHEST STID
            IERR = M_FETCH_AND_CLEAR32(ORDSEM) ! RESETS COUNTER
            IF(IERR .LT. 0) PRINT*, "FETCH-CLEAR FAILED"
          ELSE ! ALL OTHER STIDS INCREMENT COUNTER:
            IERR = M_FETCH_AND_INC32(ORDSEM)
            IF(IERR .LT. 0) PRINT*, "FETCH-INC FAILED"
          ENDIF
        ELSE
          GOTO 10 ! LOOP AND TRY AGAIN IF CNTVAL .NE. STID
        ENDIF
      ENDDO
      RETURN
      END
```

This example uses a cyclic decomposition in the parallel J loop because, by definition, ordered sections must be executed in iteration order, and this is impossible using a block decomposition.

As in the example in the “Cyclic parallelism” section, here the starting index is offset according to spawn thread ID and the loop steps by the number of parallel threads. This ensures that each thread computes a unique array element on every step of the loop; NTHR elements are computed per step. Contiguous STIDs compute contiguous elements.

The loop is ordered by the first `IF` statement in the loop, which only allows the body of the loop (including the LCD) to execute if the counter associated with the semaphore `ORDSEM` is equal to the current `STID`. This counter is incremented (or reset when the highest `STID` is reached) in the body of the loop, forcing the threads to execute in iteration order. The counter associated with `ORDSEM` controls access to the LCD; no explicit semaphore lock is needed.

Substantial nonordered work must be present in this loop to make the overhead of the ordered section worthwhile. Assuming this condition is met, once all the threads pass through the ordered section once, their execution of the nonordered code will be staggered such that they will stay busy while they are outside the ordered code.

The ordered parallelism described here is similar to that achieved through use of compiler directives in the “Ordered sections” section of Chapter 6, “Advanced shared-memory programming.”

Glossary

absolute address An address that does not undergo virtual-to-physical address translation when used to reference memory or the I/O register area.

accumulator A variable used to accumulate value. Accumulators are typically assigned a function of themselves, which can create dependences when done in loops.

actual argument In Fortran, a value that is passed by a call to a procedure (function or subroutine). The actual argument appears in the source of the calling procedure; the argument that appears in the source of the called procedure is a *dummy argument*. C and C++ conventions refer to actual arguments as *actual parameters*.

actual parameter In C and C++, a value that is passed by a call to a procedure (function). The actual parameter appears in the source of the calling procedure; the parameter that appears in the source of the called procedure is a *formal parameter*. Fortran conventions refer to actual parameters as *actual arguments*.

address A number used by the operating system to identify a storage location.

address space Memory space, either physical or virtual, available to a process.

agent The gate array on V2200 and X2000 servers that provides a high-speed interface between pairs of PA-RISC processors and the *crossbar*. Also called the *CPU Agent* and the *CPA*.

alias An alternative name for some object, especially an alternative variable name that refers to a memory location. Aliases can cause data dependences, which prevent the compiler from parallelizing parts of a program.

alignment A condition in which the address, in memory, of a given data item is integrally divisible by a particular integer value, often the size of the data item itself. Alignment simplifies the addressing of such data items.

allocatable array In Fortran 90, a named array whose rank is specified at compile time, but whose bounds are determined at run time.

allocate An action performed by a program at runtime in which memory is reserved to hold data of a given type. In Fortran 90, this is done through the creation of *allocatable arrays*. In C, it is done through the dynamic creation of memory blocks using `malloc`. In C++, it is done through the dynamic creation of memory blocks using `malloc` or `new`.

ALU Arithmetic logic unit. A basic element of the central processing unit (CPU) where arithmetic and logical operations are performed.

Amdahl's law A statement that the ultimate performance of a computer system is limited by the slowest component. In the context of Exemplar servers this is interpreted to mean that the serial component of the application code will restrict the maximum speed-up that is achievable.

American National Standards Institute (ANSI) A repository and coordinating agency for standards implemented in the U.S. Its activities include the production of Federal Information Processing (FIPS) standards for the Department of Defense (DoD).

ANSI See *American National Standards Institute*.

apparent recurrence A condition or construct that fails to provide the compiler with sufficient information to determine whether or not a recurrence exists. Also called a *potential recurrence*.

argument In Fortran, either a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called (*dummy argument*) or the variable or constant that is passed by a call to a procedure (*actual argument*). C and C++ conventions refer to arguments as *parameters*.

arithmetic logic unit (ALU) A basic element of the central processing unit (CPU) where arithmetic and logical operations are performed.

array An ordered structure of operands of the same data type. The structure of an array is defined by its rank, shape, and data type.

array section A Fortran 90 construct that defines a subset of an array by providing starting and ending elements and strides for each dimension. For an array `A(4,4)`, `A(2:4:2,2:4:2)` is an array section containing only the evenly indexed elements `A(2,2)`, `A(4,2)`, `A(2,4)`, and `A(4,4)`.

array-valued argument In Fortran 90, an *array section* that is an actual argument to a subprogram.

ASCII American Standard Code for Information Interchange. This encodes printable and non-printable characters into a range of integers.

assembler A program that converts assembly language programs into executable machine code.

assembly language A programming language whose executable statements can each be translated directly into a corresponding machine instruction of a particular computer system.

automatic array In Fortran, an array of explicit rank that is not a dummy argument and is declared in a subprogram.

bandwidth A measure of the rate at which data can be moved through a device or circuit. Bandwidth is usually measured in millions of bytes per second (Mbytes/sec) or millions of bits per second (Mbits/sec).

bank conflict An attempt to access a particular memory bank before a previous access to the bank is complete.

barrier A structure used by the compiler in barrier synchronization. Also sometimes used to refer to the construct used to implement barrier synchronization. See also *barrier synchronization*.

barrier synchronization A control mechanism used in parallel programming that ensures all threads have completed an operation before continuing with the next operation. On Exemplar servers, barrier synchronization can be automated by certain CPSlib routines and compiler directives. See also *barrier*.

basic block A linear sequence of machine instructions with a single entry and a single exit.

bit A binary digit.

block-shared memory Memory that is addressed by the same virtual address from any hypernode in the system on which the memory was allocated. This memory class is used to store arrays that are dynamically allocated at runtime, when the number of hypernodes on which the process is running is known. The virtual pages of the arrays are then divided into a number of chunks equal to the number of available hypernodes, and these chunks (which likely contain multiple contiguous pages each) are distributed to the system-global physical pages of the available hypernodes, 1 chunk per hypernode. If the number of pages of a `block_shared` array is not integrally divisible by the number of hypernodes, the array size is increased to allow integral division. Compare with *node-private memory*, *thread-private memory*, *near-shared memory*, and *far-shared memory*.

blocking factor Integer representing the stride of the outer strip of a pair of loops created by blocking.

branch A class of instructions which change the value of the program counter to a value other than that of the next sequential instruction.

byte A group of contiguous bits starting on an addressable boundary. Generally, a byte is 8 bits in length.

cache A small, high-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of the memory that are, or are believed to be, currently in use. Cache memory is physically separate from main memory and can be accessed with substantially less latency. The Exemplar series of computers employs separate data and instruction cache memories.

cache, direct mapped A form of cache memory that addresses encached data by a function of the data’s virtual address. On V2200 servers, the processor cache address is identical to the least-significant 21 bits of the data’s virtual address. This means cache thrashing can occur when the virtual addresses of two data items are an exact multiple of 2 Mbyte (21 bits) apart. On X2000 servers, the processor cache address is identical to the least-significant 20 bits of the data’s virtual address, meaning cache thrashing can occur when the virtual addresses of two data items are an exact multiple of 1 Mbyte (20 bits) apart.

cache hit A *cache hit* occurs if data to be loaded is residing in the cache.

cache line A chunk of contiguous data that is copied into a cache in one operation. On V2200 servers, processor cache lines are 32 bytes (V2200 servers and other single-node SMP servers do not employ CTIcache lines). On X2000 servers, both processor cache lines and CTIcache lines consist of 32 bytes of data. When a processor cache miss occurs and data must be fetched from outside the processor cache, the requested data is brought in as part of a 32-byte cache line.

cache memory A small, high-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of the memory that are, or are believed to be, currently in use. Cache memory is physically separate from main memory and can be accessed with substantially less latency. V2200 servers and X2000 servers employ separate data and instruction caches.

cache miss A *cache miss* occurs if data to be loaded is not residing in the cache.

cache purge The act of invalidating or removing entries in a cache memory.

cache thrashing *Cache thrashing* occurs when two or more data items that are frequently needed by the program map to the same cache address. In this case, each time one of the items is encached it overwrites another needed item, causing constant cache misses and impairing data reuse. Cache thrashing also occurs when two or more threads are simultaneously writing to the same cache line.

CMC The Coherent Memory Controller gate array, which, among other tasks, provides an interface to the CTI interface and to memory and maintains cache coherency information. Each pair of processors has a CMC.

central processing unit (CPU) The central processing unit (CPU) is that portion of a computer that recognizes and executes the instruction set.

clock cycle The duration of the square wave pulse sent throughout a computer system to synchronize operations.

clone A compiler-generated copy of a loop or procedure. When the Exemplar compilers generate code for a parallelizable loop, they generate two versions: a serial clone and a parallel clone. See also *dynamic selection*.

code A computer program, either in source form or in the form of an executable image on a machine.

coherency A term frequently applied to caches. If a data item is referenced by a particular processor on a multiprocessor system, the data is copied into that processor's cache and is updated there if the processor modifies the data. If another processor references the data while a copy is still in the first processor's cache, a mechanism is needed to ensure that the second processor does not use an outdated copy of the data from memory. The state that is achieved when both processors' caches always have the latest value for the data is called cache coherency. On V2200 servers and X2000 servers, an item of data may reside concurrently in several processors' caches.

column-major order Memory representation of an array such that the columns are stored contiguously. For example, given a two-dimensional array `A(3,4)`, the array element `A(3,1)` immediately precedes element `A(1,2)` in memory. This is the default storage method for arrays in Fortran.

compiler A computer program that translates computer code written in a high-level programming language, such as Fortran, into equivalent machine language.

Compiler Parallel Support library (CPSlib) A library of low-level parallelization and synchronization routines. Refer to Appendix F, "Compiler Parallel Support Library," for more information.

concurrent In parallel processing, threads that can execute at the same time are called concurrent threads.

conditional induction variable A loop induction variable that is not necessarily incremented on every iteration.

constant folding Replacement of an operation on constant operands with the result of the operation.

constant propagation The automatic compile-time replacement of variable references with a constant value previously assigned to that variable. Constant propagation is performed within a single procedure by conventional compilers.

conventional compiler A compiler that cannot perform interprocedural optimization.

counter A variable that is used to count the number of times an operation occurs. CPSlib semaphores are associated with counters so that they can facilitate barrier synchronization or the creation of ordered sections.

CPA CPU Agent. The gate array on V2200 servers and X2000 servers that provides a high-speed interface between pairs of PA-RISC processors and the *crossbar*. Also called the *CPU Agent* and the *agent*.

CPSlib (Compiler Parallel Support library) A library of low-level parallelization and synchronization routines. Refer to Appendix F, “Compiler Parallel Support Library,” for more information.

CPU Central processing unit. The central processing unit (CPU) is that portion of a computer that recognizes and executes the instruction set.

CPU Agent The gate array on V2200 servers and X2000 servers that provides a high-speed interface between pairs of PA-RISC processors and the *crossbar*. Also called the *agent* and the *CPA*.

CPU-private memory Data that is accessible by a single thread only (not shared among the threads constituting a process). A thread-private data object has a unique virtual address which maps to a unique physical address within each hypernode. Threads access the physical copies of thread-private data residing on their own hypernode when they access thread-private virtual addresses. Compare with *node-private memory*, *near-shared memory*, *far-shared memory*, and *block-shared memory*.

CPU time The amount of time the CPU requires to execute a program. Because programs share access to a CPU, the wall clock time of a program may not be the same as its CPU time. If a program can use multiple processors, the CPU time may be greater than the wall clock time. (See *wall clock time*.)

critical section A portion of a parallel program that can be executed by only one thread at a time.

crossbar A switching device that connects the CPUs, banks of memory, and I/O controller on a single hypernode of a V2200 server or of an X2000 server. Because the crossbar is nonblocking, all ports can run at full bandwidth simultaneously, provided there is not contention for a particular port.

CSR Control/Status Register. A CSR is a software-addressable hardware register used to hold control information or state.

CTIcache A partition of physical memory that exists on each hypernode and is used to store copies of global data fetched from other hypernodes.

CTI interface The hardware interface between the CTI rings and the CMC.

CTI (Coherent Toroidal Interface) ring The ring interconnect that connects all the hypernodes of a multihypernode Exemplar server together in a ring topology. While the CTI ring is derived from the IEEE SCI standard, complete compatibility is sacrificed to provide lower latencies.

data cache (Dcache) A small cache memory with a fast access time. This cache holds prefetched and current data. On X2000 servers, processors have 1-Mbyte off-chip caches. On V2200 servers, processors have 2-Mbyte off-chip caches. See also *cache, direct mapped*.

data dependence A relationship between two statements in a program, such that one statement must precede the other to produce the intended result. (See also *loop-carried dependence (LCD)* and loop-independent dependence (LID).)

data localization Optimizations designed to keep frequently used data in the processor data cache, thus eliminating the need for more costly CTIcache or memory accesses.

data type A property of a data item that determines how its bits are grouped and interpreted. For processor instructions, the data type identifies the size of the operand and the significance of the bits in the operand. Some example data types include `INTEGER`, `int`, `REAL`, and `float`.

Dcache Data cache. A small cache memory with a one clock cycle access time. This cache holds prefetched and current data. On X2000 servers, this cache is 1 Mbyte in size. On V2200 servers, this cache is 2 Mbytes.

deadlock A condition in which a thread waits indefinitely for some condition or action that cannot, or will not, occur.

direct memory access (DMA) A method for gaining direct access to memory and achieving data transfers without involving the CPU.

distributed memory A memory architecture used in multi-CPU systems, in which the system's memory is physically divided among the processors. In most distributed-memory architectures, distributed memory is accessible from only a single processor; sharing of data requires explicit message passing.

distributed part A loop generated by the compiler in the process of loop distribution.

DMA Direct memory access. A method for gaining direct access to memory and achieving data transfers without involving the CPU.

double A double-precision floating-point number that is stored in 64 bits in C and C++.

doubleword A primitive data operand which is 8 bytes (64 bits) in length. Also called a *longword*. See also *word*.

dummy argument In Fortran, a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called. The dummy argument appears in the source of the called procedure; the parameter that appears in the source of the calling procedure is an *actual argument*. C and C++ conventions refer to dummy arguments as *formal parameters*.

dynamic selection The process by which the compiler chooses the appropriate runtime clone of a loop. See also *clone*.

encache To copy data or instructions into a cache.

exception A hardware-detected event that interrupts the running of a program, process, or system. See also *fault*.

execution stream A series of instructions executed by a CPU.

far-shared memory Memory that is addressed by the same virtual address from any hypernode in the system on which the memory was allocated. Far-shared memory is physically distributed by pages, in a manner that is approximately round-robin, to all the hypernodes in the system, so the virtual address maps to a single physical address located on one of the hypernodes. Access latencies therefore vary as a function of hypernode and data element. Compare with *node-private memory*, *thread-private memory*, *near-shared memory*, and *block-shared memory*.

fault A type of *interruption* caused by an instruction requesting a legitimate action that cannot be carried out immediately due to a system problem.

floating point A numerical representation of a real number. On V2200 servers and X2000 servers, a floating point operand has a sign (positive or negative) part, an exponent part, and a fraction part. The fraction is a fractional representation. The exponent is the value used to produce a power of two scale factor (or portion) that is subsequently used to multiply the fraction to produce an unsigned value.

FLOPS Floating-point operations per second. A standard measure of computer processing power in the scientific community.

formal parameter In C and C++, a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called. The formal parameter appears in the source of the called procedure; the parameter that appears in the source of the calling procedure is an actual parameter. Fortran conventions refer to formal parameters as dummy arguments.

Fortran A high-level software language used mainly for scientific applications.

Fortran 90 The international standard for Fortran adopted in 1991.

function A procedure whose call can be imbedded within another statement, such as an assignment or test. Any procedure in C or C++ or a procedure defined as a FUNCTION in Fortran.

functional unit (FU) A part of a CPU that performs a set of operations on quantities stored in *registers*.

gate A construct that restricts execution of a block of code to a single thread. A thread locks a gate on entering the gated block of code and unlocks the gate on exiting the block. When the gate is locked, no other threads can enter. Compiler directives can be used to automate gate constructs; gates can also be implemented using *semaphores*.

Gbyte See *gigabyte*.

gigabyte 1073741824 (2^{30}) bytes.

global optimization A restructuring of program statements that is not confined to a single basic block. Global optimization, unlike interprocedural optimization, is confined to a single procedure. Global optimization is done by Exemplar compilers at optimization level `+O2` and above.

global register allocation (GRA) A method by which the compiler attempts to store commonly-referenced scalar variables in registers throughout the code in which they are most frequently accessed.

global variable A variable whose scope is greater than a single procedure. In C and C++ programs, a global variable is a variable that is defined outside of any one procedure. Fortran has no global variables per se, but `COMMON` blocks can be used to make certain memory locations globally accessible.

granularity In the context of parallelism, a measure of the relative size of the computation done by a thread or parallel construct. Performance is generally an increasing function of the granularity. In higher-level language programs, possible sizes are routine, loop, block, statement, and expression. Fine granularity is exhibited by parallel loops, tasks and expressions; coarse granularity is exhibited by parallel processes.

hand-rolled loop A loop, more common in Fortran than C or C++, that is constructed using `IF` tests and `GOTO` statements rather than a language-provided loop structure such as `DO`.

hidden alias An alias that, because of the structure of a program or the standards of the language, goes undetected by the compiler. Hidden aliases can result in undetected *data dependences*, which may result in wrong answers.

High Performance Fortran (HPF) An ad-hoc language extension of Fortran 90 that provides user-directed data distribution and alignment. HPF is not a standard, but rather a set of features desirable for parallel programming.

hoist An optimization process that moves a memory load operation from within a loop to the basic block preceding the loop.

HP Hewlett-Packard, the manufacturer of the PA-RISC chips used as processors in V2200 servers and X2000 servers.

HP-UX Hewlett-Packard's Unix-based operating system for its PA-RISC workstations and servers.

hypercube A topology used in some massively parallel processing systems. Each processor is connected to its binary neighbors. The number of processors in the system is always a power of two; that power is referred to as the dimension of the hypercube. For example, a 10-dimensional hypercube has 2^{10}, or 1,024 processors.

hypernode A set of processors and physical memory organized as a symmetric multiprocessor (SMP) running a single image of the operating system microkernel. Nonscalable servers and V2200 servers consist of one hypernode. X2000 servers consist of one or more hypernodes, with a high speed *CTI ring* connecting the hypernodes. When discussing multidimensional parallelism or memory classes, hypernodes are generally called nodes.

Icache Instruction cache. This cache holds prefetched instructions and permits the simultaneous decoding of one instruction with the execution of a previous instruction. On X2000 servers, this cache is 1 Mbyte in size. On V2200 servers, this cache is 2 Mbytes.

IEEE Institute for Electrical and Electronic Engineers. An international professional organization and a member of ANSI and ISO.

induction variable A variable that changes linearly within the loop, that is, whose value is incremented by a constant amount on every iteration. For example, in the following Fortran loop, `I`, `J` and `K` are induction variables, but `L` is not.

```
DO I = 1, N
   J = J + 2
   K = K + N
   L = L + I
ENDDO
```

inlining The replacement of a procedure (function or subroutine) call, within the source of a calling procedure, by a copy of the called procedure’s code.

Institute for Electrical and Electronic Engineers (IEEE) An international professional organization and a member of ANSI and ISO.

instruction One of the basic operations performed by a CPU.

instruction cache (Icache) This cache holds prefetched instructions and permits the simultaneous decoding of one instruction with the execution of a previous instruction. On X2000 servers, this cache is 1 Mbyte in size. On V2200 servers, this cache is 2 Mbytes.

instruction mnemonic A symbolic name for a machine instruction.

integral division Division that results in a whole number solution with no remainder. For example, 10 is integrally divisible by 2, but not by 3.

interface A logical path between any two modules or systems.

interleaved memory Memory that is divided into multiple banks to permit concurrent memory accesses. The number of separate memory banks is referred to as the memory stride.

interprocedural optimization Automatic analysis of relationships and interfaces between all subroutines and data structures within a program. Traditional compilers analyze only the relationships within the procedure being compiled.

interprocessor communication The process of moving or sharing data, and synchronizing operations between processors on a multiprocessor system.

intrinsic A function or subroutine that is an inherent part of a computer language. For example, `SIN` is a Fortran intrinsic.

job scheduler That portion of the operating system that schedules and manages the execution of all processes.

join The synchronized termination of parallel execution by spawned tasks or threads.

jump Departure from normal one-step incrementing of the program counter.

kbyte See *kilobyte*.

kernel The core of the operating system where basic system facilities, such as file access and memory management functions, are performed.

kernel thread identifier (ktid) A unique integer identifier (not necessarily sequential) assigned when a thread is created.

kilobyte 1024 (2^{10}) bytes.

latency The time delay between the issuing of an instruction and the completion of the operation. A common benchmark used for comparing systems is the latency of coherent memory access instructions. This particular latency measurement is believed to be a good indication of the *scalability* of a system; low latency equates to low system overhead as system size increases.

linker A software tool that combines separate object code modules into a single object code module or executable program.

load An instruction used to move the contents of a memory location into a register.

locality of reference An attribute of a memory reference pattern that refers to the likelihood of an address of a memory reference being physically close to the CPU making the reference.

local optimization Restructuring of program statements within the scope of a basic block. Local optimization is done by Exemplar compilers at optimization level `+O1` and above.

localization Data localization. Optimizations designed to keep frequently used data in the processor data cache, thus eliminating the need for more costly CTIcache or memory accesses.

logical address Logical address space is that address as seen by the application program.

logical hypernode ID Logical hypernode IDs range from 0..*n*-1, where *n* is the number of available hypernodes in the system. Logical IDs are assigned in the order in which your program occupies the system. The hypernode that your program’s thread 0 runs on is considered logical hypernode 0; any hypernodes it expands to later are assigned increasing logical ID numbers.

logical memory Virtual memory. The memory space as seen by the program, which may be larger than the available physical memory. The virtual memory of a V2200 server can be up to 16 Tbytes. The virtual memory of an X2000 server can be up to 4 Gbytes (however, through use of node-private memory, this 4 Gbytes can be mapped to a *larger* set of physical memory). HP-UX can map this virtual memory to a smaller set of physical memory, using disk space to make up the difference if necessary. Also called *virtual memory*.

longword (l) Doubleword. A primitive data operand which is 8 bytes (64 bits) in length. See also *word*.

loop blocking A loop transformation that strip mines and interchanges a loop to provide optimal reuse of the encachable loop data.

loop-carried dependence (LCD) A dependence between two operations executed on different iterations of a given loop and on the same iteration of all enclosing loops. A loop carries a dependence from an indexed assignment to an indexed use if, for some iteration of the loop, the assignment stores into an address that is referred to on a different iteration of the loop. To parallelize a loop containing an LCD, you generally must manually synchronize the LCD assignment and manually parallelize the loop.

loop constant A constant or expression whose value does not change within a loop.

loop distribution The restructuring of a loop nest to create simple loop nests. Loop distribution creates two or more loops, called distributed parts, which can serve to make parallelization more efficient by increasing the opportunities for loop interchange and isolating code that must run serially from parallelizable code. It can also improve data localization and other optimizations.

loop-independent dependence (LID) A dependence between two operations executed on the same iteration of all enclosing loops such that one operation must precede the other to produce correct results.

loop induction variable See *induction variable.*

loop interchange The reordering of nested loops. Loop interchange is generally done to increase the granularity of the parallelizable loop(s) present or to allow more efficient access to loop data.

loop invariant Loop constant. A constant or expression whose value does not change within a loop.

loop invariant computation An operation that yields the same result on every iteration of a loop.

loop replication The process of transforming one loop into more than one loop to facilitate an optimization. The optimizations that replicate loops are `IF-DO` and `if-for` optimizations, dynamic selection, loop unrolling, and loop blocking.

machine exception A fatal error in the system that cannot be handled by the operating system. See also *exception*.

main memory Physical memory other than the processor caches. On multinode servers, main memory is the physical memory other than the processor caches that is not allocated as part of the CTIcache.

main procedure A procedure invoked by the operating system when an application program starts up. The main procedure is the main program in Fortran; in C and C++, it is the function main().

main program In a Fortran program, the program section invoked by the operating system when the program starts up.

Mbyte See megabyte (Mbyte).

megabyte (Mbyte) 1048576 (2^{20}) bytes.

megaflops (MFLOPS) One million floating-point operations per second.

memory bank conflict An attempt to access a particular memory bank before a previous access to the bank is complete.

memory management The hardware and software that control memory page mapping and memory protection.

message Data copied from one process to another (or the same) process. The copy is initiated by the sending process, which specifies the receiving process. The sending and receiving processes need not share a common address space. (Note: depending on the context, a process may be a *thread*.)

Message-Passing Interface (MPI) A message-passing and process control library. For information on the Hewlett-Packard implementation of MPI, refer to the *HP MPI User's Guide* (B6011-90001).

message passing A type of programming in which program modules (often running on different processors or different hosts) communicate with each other by means of system library calls that package, transmit, and receive data. All message-passing library calls must be explicitly coded by the programmer.

MIMD (multiple instruction stream multiple data stream) A computer architecture that uses multiple processors, each processing its own set of instructions simultaneously and independently of others. MIMD also describes when processes are performing different operations on different data. Compare with SIMD.

multiprocessing The creation and scheduling of processes on any subset of CPUs in a system configuration.

mutex A variable used to construct an area (region of code) of *mutual exclusion*. When a mutex is locked, entry to the area is prohibited; when the mutex is free, entry is allowed.

mutual exclusion A protocol that prevents access to a given resource by more than one thread at a time.

near-shared memory Memory that is addressable by the same virtual address from any hypernode in the system on which the memory was allocated. Near-shared memory resides physically on the hypernode from which it was allocated, and is accessed with lowest latency from that hypernode. Access latencies are higher from other hypernodes. Compare with *thread-private memory*, *node-private memory*, *block-shared memory*, and *far-shared memory*.

negate An instruction that changes the sign of a number.

network A system of interconnected computers that enables machines and their users to exchange information and share resources. X2000 servers provide support for FDDI networks.

node On HP scalable and nonscalable servers, a node is equivalent to a *hypernode*. The term “node” is generally used in place of hypernode when discussing parallelism or memory classes.

node-private memory Memory residing on a hypernode that is accessible only by CPUs on the same hypernode. A node-private data object has a unique virtual address by which all threads on all hypernodes access it. This address maps to one physical address per hypernode; when a thread accesses the data, it receives the value contained in the physical memory of its own hypernode. Compare with *thread-private memory*, *near-shared memory*, *block-shared memory*, and *far-shared memory*.

non-uniform memory access (NUMA) This term describes memory access times in systems in which accessing different types of memory (for example, memory local to the current hypernode or memory remote to the current hypernode) results in non-uniform access times.

nonblocking crossbar A switching device that connects the CPUs, banks of memory, and I/O controller on a single hypernode. Because the crossbar is nonblocking, all ports can run at full bandwidth simultaneously provided there is not contention for a particular port.

NUMA Non-uniform memory access. This term describes memory access times in systems in which accessing different types of memory (for example, memory local to the current hypernode or memory remote to the current hypernode) results in non-uniform access times.

offset In the context of a process address space, an integer value that is added to a base address to calculate a memory address. Offsets in V2200 servers are 64-bit values, and must keep address values within a single 16-Tbyte memory space. Offsets in X2000 servers are 32-bit values, and must keep address values within a single 4-Gbyte memory space.

opcode A predefined sequence of bits in an instruction that specifies the operation to be performed.

operating system The program that manages the resources of a computer system. V2200 servers use the HP-UX operating system. X2000 servers use the SPP-UX operating system, which is compatible with the HP-UX operating system.

optimization The refining of application software programs to minimize processing time. Optimization takes maximum advantage of a computer's hardware features and minimizes input/output traffic and idle processor time.

optimization level The degree to which source code is optimized by the compiler. The Exemplar compilers have five levels of optimization: level `+O0`, `+O1`, `+O2`, `+O3`, and `+O4`. (The +O4 option is not available in Fortran 90.)

oversubscript An array reference that falls outside declared bounds.

oversubscription In the context of parallel threads, a process attribute that permits the creation of more threads within a process than the number of processors available to the process.

PA-RISC The Hewlett-Packard Precision Architecture reduced instruction set processor chip. This is the processor chip used in V2200 servers and X2000 servers.

packet A group of related items. A packet may refer to the arguments of a subroutine or to a group of bytes that is transmitted over a network.

page A page is the unit of virtual or physical memory controlled by the memory management hardware and software. On HP-UX servers, the default page size is 4 K (4,096) contiguous bytes. Valid page sizes are: 4 K, 16 K, 64 K, 256 K, 1 Mbyte, 4 Mbytes, 16 Mbytes, 64 Mbytes, and 256 Mbytes. See also *virtual memory*.

page fault A page fault occurs when a process requests data that is not currently in memory. This requires the operating system to retrieve the page containing the requested data from disk.

page frame A page frame is the unit of physical memory in which pages are placed. Referenced and modified bits associated with each page frame aid in memory management.

parallel optimization The transformation of source code into parallel code (parallelization) and restructuring of code to enhance parallel performance.

parallelization The process of transforming serial code to a form of code that can run simultaneously on multiple CPUs while preserving semantics. When `+O3 +Oparallel` is specified, the Exemplar compilers automatically parallelize loops in your program and recognize compiler directives and pragmas with which you can manually specify parallelization of loops, tasks, and regions.

parallelization, loop The process of splitting a loop into several smaller loops, each of which operates on a subset of the data of the original loop, and generating code to run these loops on separate processors in parallel.

parallelization, ordered The process of splitting a loop into several smaller loops, each of which iterates over a subset of the original data with a stride equal to the number of loops created, and generating code to run these loops on separate processors. Each iteration in an ordered parallel loop begins execution in the original iteration order, allowing dependences within the loop to be synchronized to yield correct results via gate constructs.

parallelization, stride-based The process of splitting up a loop into several smaller loops, each of which iterates over several discontiguous chunks of data, and generating code to run these loops on separate processors in parallel. Stride-based parallelism can only be achieved manually by using compiler directives or CPSlib functions.

parallelization, strip-based The process of splitting up a loop into several smaller loops, each of which iterates over a single contiguous subset of the data of the original loop, and generating code to run these loops on separate processors in parallel. Strip-based parallelism is the default for automatic parallelism and for directive-initiated loop parallelism in absence of the `chunk_size =` *n* or `ordered` attributes.

parallelization, task The process of splitting up source code into independent sections which can safely be run in parallel on available processors. Exemplar programming languages provide compiler directives and pragmas that allow you to identify parallel tasks in source code.

parameter In C and C++, either a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called (*formal parameter*) or the variable or constant that is passed by a call to a procedure (*actual parameter*). In Fortran, a symbolic name for a constant.

path An environment variable that you set in your shell configuration file that allows you to access commands in various directories without having to specify a complete path name.

physical address A unique identifier that selects a particular location in the computer's memory. Because HP-UX supports virtual memory, programs address data by its virtual address; HP-UX then maps this address to the appropriate physical address. See also *virtual address*.

physical address space The set of possible addresses for a particular physical memory.

physical memory Computer hardware that stores data. V2200 servers can contain up to 16 Gbytes of physical memory on a 16-processor hypernode. X2000 servers can contain up to 16 Gbytes of physical memory per hypernode, for a total of 64 Gbytes of physical memory on a full 4-hypernode system.

pipeline An overlapping operating cycle function that is used to increase the speed of computers. Pipelining provides a means by which multiple operations occur concurrently by beginning one instruction sequence before another has completed. Maximum efficiency is achieved when the pipeline is “full,” that is, when all stages are operating on separate instructions.

pipelining Issuing instructions in an order that best utilizes the pipeline.

procedure A unit of program code. In Fortran, a function, subroutine or main program; in C and C++, a function.

process A collection of one or more execution streams within a single logical address space; an executable program. A process is made up of one or more threads.

process memory The portion of system memory that is used by an executing process.

programming model A description of the features available to efficiently program a certain computer architecture.

program unit A procedure or main section of a program.

queue A data structure in which entries are made at one end and deletions at the other. Often referred to as first-in, first-out (FIFO).

rank The number of dimensions of an array.

read A memory operation in which the contents of a memory location are copied and passed to another part of the system.

recurrence A cycle of dependences among the operations within a loop in which an operation in one iteration depends on the result of a following operation that executes in a previous iteration.

recursion An operation that is defined, at least in part, by a repeated application of itself.

recursive call A condition in which the sequence of instructions in a procedure causes the procedure itself to be invoked again. Such a procedure must be compiled for reentrancy.

reduced instruction set computer (RISC) An architectural concept that applies to the definition of the instruction set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a compiler can generate highly optimized code. The PA-RISC processor used in V2200 servers and X2000 servers employs a RISC architecture.

reduction An arithmetic operation that performs a transformation on an array to produce a scalar result.

reentrancy The ability of a program unit to be executed by multiple threads at the same time. Each invocation maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables. Procedures must be compiled for reentrancy in order to be invoked in parallel or to be used for recursive calls. Exemplar compilers compile for reentrancy by default.

reference Any operation that requires a cache line to be encached; this includes load as well as store operations, because writing to any element in a cache line requires the entire cache line to be encached.

register A hardware entity that contains an address, operand, or instruction status information.

reuse, data In the context of a loop, the ability to use data fetched for one loop operation in another operation. In the context of a cache, reusing data that was encached for a previous operation; because data is fetched as part of a cache line, if any of the other items in the cache line are used before the line is flushed to memory, reuse has occurred.

reuse, spatial Reusing data that resides in the cache as a result of the fetching of another piece of data from memory. Typically, this involves using array elements that are contiguous to (and therefore part of the cache line of) an element that has already been used, and therefore is already encached.

reuse, temporal Reusing a data item that has been used previously.

RISC Reduced instruction set computer. An architectural concept that applies to the definition of the instruction set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a compiler can generate highly optimized code. The PA-RISC processor used in V2200 servers and X2000 servers employs a RISC architecture.

rounding A method of obtaining a representation of a number that has less precision than the original in which the closest number representable under the lower precision system is used.

row-major order Memory representation of an array such that the rows of an array are stored contiguously. For example, given a two-dimensional array `A[3][4]`, array element `A[0][3]` immediately precedes `A[1][0]` in memory. This is the default storage method for arrays in C.

SCI Scalable Coherent Interface. This is defined by IEEE standard 1596-1992. The interface is physically defined as a pair of 18-bit, differential ECL, unidirectional links. Each link provides 16 bits of data with two control signals. Data is sampled on both the rising and falling edges of the clock. This interface provides the basis for the CTI rings used in X2000 servers; however, total compatibility with the standard has been sacrificed to provide increased performance.

scope The domain in which a variable is visible in source code. The rules that determine scope are different for Fortran and C/C++.

semaphore An integer variable assigned one of two values: one value to indicate that it is "locked," and another to indicate that it is "free." Semaphores can be used to synchronize parallel threads. CPSlib provides a set of manipulation functions to facilitate this.

shape The number of elements in each dimension of an array.

shared virtual memory A memory architecture in which memory can be accessed by all processors in the system. This architecture can also support virtual memory.

shell An interactive command interpreter that is the interface between the user and the operating system.

SIMD (single instruction stream multiple data stream) A computer architecture that performs one operation on multiple sets of data. A processor (separate from the SMP array) is used for the control logic, and the processors in the SMP array perform the instruction on the data. Compare with *MIMD (multiple instruction stream multiple data stream).*

single A single-precision floating-point number stored in 32 bits. See also *double.*

SMP Symmetric multiprocessor. A multiprocessor computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no manager or worker processors; the operating system runs on any or all of the processors.

socket An endpoint used for interprocess communication.

socket pair Bidirectional pipes that enable application programs to set up two-way communication between processes that share a common ancestor.

source code The uncompiled version of a program, written in a high-level language such as Fortran or C.

source file A file that contains program source code.

space A contiguous range of virtual addresses within the system-wide virtual address space. Spaces are 4 Gbytes in size in X2000 servers and 16 Tbytes in V2200 servers.

spatial reference An attribute of a memory reference pattern that pertains to the likelihood of a subsequent memory reference address being numerically close to a previously referenced address.

spawn To activate existing threads.

spawn context A parallel loop, task list, or region that initiates the spawning of threads and defines the structure within which the threads' spawn thread IDs are valid.

spawn thread identifier (stid) A sequential integer identifier associated with a particular thread that has been spawned. stids are only assigned to spawned threads, and they are assigned within a spawn context; therefore, duplicate stids may be present amongst the threads of a program, but stids are always unique within the scope of their spawn context. stids are assigned sequentially and run from 0 to one less than the number of threads spawned in a particular spawn context.

SPMD Single program multiple data. A single program executing simultaneously on several processors. This is usually taken to mean that there is redundant execution of sequential scalar code on all processors.

stack A data structure in which the last item entered is the first to be removed. Also referred to as last-in, first-out (LIFO). HP-UX provides every thread with a stack which is used to pass arguments to functions and subroutines and for local variable storage.

store An instruction used to move the contents of a register to memory.

strip length, parallel In strip-based parallelism, the amount by which the induction variable of a parallel inner loop is advanced on each iteration of the (conceptual) controlling outer loop.

strip mining The transformation of a single loop into two nested loops. Conceptually, this is how parallel loops are created by default. A conceptual outer loop advances the initial value of the inner loop's induction variable by the parallel strip length. The parallel strip length is based on the trip count of the loop and the amount of code in the loop body. Strip mining is also used by the data localization optimization.

subroutine A software module that can be invoked from anywhere in a program.

superscalar A class of *RISC* processors that allow multiple instructions to be issued on each clock period.

Symmetric Multiprocessor (SMP) A multiprocessor computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no manager or worker processors; the operating system runs on any or all of the processors.

synchronization A method of coordinating the actions of multiple threads so that operations occur in the right sequence. When manually optimizing code, you can synchronize programs using compiler directives, calls to library routines, or assembly-language instructions. You do so, however, at the cost of additional overhead; synchronization may cause at least one CPU to wait for another.

system administrator (sysadmin) The system manager.

system manager The person responsible for the management and operation of a computer system. Also called the system administrator and the sysadmin.

Tbyte See terabyte (Tbyte).

terabyte (Tbyte) 1099511627776 (2^{40}) bytes.

term A constant or symbolic name that is part of an *expression*.

thread An independent execution stream that is executed by a CPU. One or more threads, each of which can execute on a different CPU, make up each process. Memory, files, signals, and other process attributes are generally shared among threads in a given process, enabling the threads to cooperate in solving the common problem. Threads are created and terminated by instructions that can be automatically generated by Exemplar compilers, inserted by adding compiler directives to source code, or coded explicitly using library calls or assembly-language.

thread create To activate existing threads.

thread identifier An integer identifier associated with a particular thread. See *thread identifier, kernel (ktid)* and *thread identifier, spawn (stid)*.

thread identifier, kernel (ktid) A unique integer identifier (not necessarily sequential) assigned when a thread is created.

thread identifier, spawn (stid) A sequential integer identifier associated with a particular thread that has been spawned. stids are only assigned to spawned threads, and they are assigned within a spawn context; therefore, duplicate stids may be present amongst the threads of a program, but stids are always unique within the scope of their spawn context. stids are assigned sequentially and run from 0 to one less than the number of threads spawned in a particular spawn context.

thread-private memory Data that is accessible by a single thread only (not shared among the threads constituting a process). A thread-private data object has a unique virtual address that maps to a unique physical address within each hypernode. Threads access the physical copies of thread-private data residing on their own hypernode when they access thread-private virtual addresses. Compare with *node-private memory*, *near-shared memory*, *far-shared memory*, and *block-shared memory*.

translation lookaside buffer A hardware entity that contains information necessary to translate a virtual memory reference to the corresponding physical page and to validate memory accesses.

TLB See *translation lookaside buffer*.

trip count The number of iterations a loop executes.

unsigned A value that is always positive.

user interface The portion of a computer program that processes input entered by a human and provides output for human users.

utility A software tool designed to perform a frequently used support function.

vector An ordered list of items in a computer's memory, contained within an array. A simple vector is defined as having a starting address, a length, and a stride. An indirect address vector is defined as having a relative base address and a vector of values to be applied as offsets to the base.

vector processor A processor whose instruction set includes instructions that perform operations on a *vector* of data (such as a row or column of an array) in an optimized fashion. Convex C Series systems employ vector processors; V2200 servers and X2000 servers do not.

virtual address The address by which programs access their data. HP-UX maps this address to the appropriate physical memory address. See also *space*.

virtual aliases Two different virtual addresses that map to the same physical memory address.

virtual machine A collection of computing resources configured so that a user or process can access any of the resources, regardless of their physical location or operating system, from a single interface.

virtual memory The memory space as seen by the program, which is typically larger than the available physical memory. The virtual memory of a V2200 server can be up to 16 Tbytes. The virtual memory of an X2000 server can be up to 4 Gbytes (however, through use of node-private memory, this 4 Gbytes can be mapped to a *larger* set of physical memory). The operating system maps this virtual memory to a smaller set of physical memory, using disk space to make up the difference if necessary. Also called *logical memory*.

wall clock time The chronological time an application requires to complete its processing. If an application starts running at 1:00 p.m. and finishes at 5:00 a.m. the following morning, its wall clock time is sixteen hours. Compare with *CPU time*.

word A contiguous group of bytes that make up a primitive data operand and start on an addressable boundary. In V2200 servers and X2000 servers, a word is four bytes (32 bits) in length. See also *doubleword*.

workstation A stand-alone computer that has its own processor, memory, and possibly a disk drive and can typically sit on a user's desk.

write A memory operation in which a memory location is updated with new data.

zero In floating point number representations, zero is represented by the sign bit with a value of zero and the exponent with a value of zero.

Index