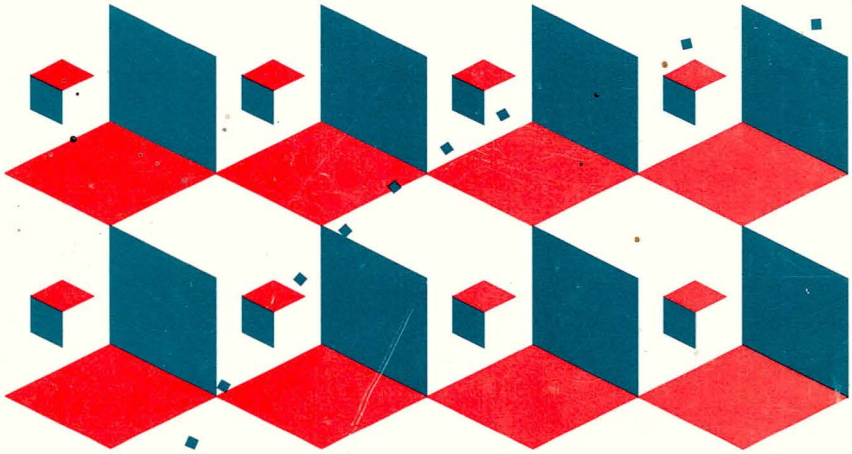


CONVEX

Exemplar Programming Guide

First Edition

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214) 497-4000

CONVEX Exemplar Programming Guide

Order No. DSW-067

First Edition

October 1994

CONVEX Press
Richardson, Texas
United States of America

CONVEX
Exemplar
Programming Guide

Order No. DSW-067

Copyright © 1994 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
C Series is a trademark of CONVEX Computer Corporation.
CXdb and CXpa are trademarks of CONVEX Computer Corporation.
Exemplar and CONVEX Exemplar are trademarks of CONVEX Computer Corporation
SPP 1000 Series and SPP Series are trademarks of CONVEX Computer Corporation
SPP-UX is a trademark of CONVEX Computer Corporation
UNIX is a registered trademark of UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc.
HP is a registered trademark of Hewlett Packard Corporation

 This entire book is recyclable.

Printed in the United States of America

Revision Information for

CONVEX Exemplar Programming Guide

Edition	Document No.	Description
First	710-030230-002	First production version of the Exemplar Programming Guide, released with CONVEX Fortran Version 9.1 and CONVEX C Version 6.1.

Contents

How to use this guide

Purpose and audience

This guide describes efficient methods for programming in CONVEX C and Fortran on Exemplar (also known as SPP Series) computers. The first 4 chapters cover basic concepts, including automatic optimizations and simple manual optimizations that require minimal programmer intervention. After this, more advanced topics are covered, including advanced manual optimizations, using ConvexPVM on Exemplar, and the Compiler Parallel Support Library.

The *Exemplar Programming Guide* is for experienced Fortran and C programmers. This early edition primarily uses Fortran in code examples; future editions will cover C more thoroughly. Readers need not be familiar with the CONVEX scalable parallel architecture, programming model, or optimization concepts; this book addresses these topics in the necessary detail.

Scope

This guide covers programming methods for CONVEX Fortran Version 9.1 and CONVEX C Version 6.1, which run under SPP-UX Version 2.1 or higher. These releases of CONVEX Fortran and C also require the CONVEX SPP Series Assembler, Loader and Libraries (ALL) Version 3.4 or higher.

This guide is concerned with producing programs that efficiently exploit the features of both the Exemplar architecture and the SPP Series compilers that run on it. 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 you have chosen the best possible algorithm for your problem and helps you obtain the best possible performance from that algorithm.

Organization

This document consists of the following chapters:

- Chapter 1 introduces the CONVEX Exemplar computer, discusses how it differs from other parallel computers, presents its programming model, and provides an overview of the optimizations available using Exemplar compilers.
- Chapter 2 presents a programmer's overview of the Exemplar architecture.
- Chapter 3 discusses optimization levels, the options used to specify them, the automatic optimizations performed at each level, and the directives used to control these optimizations.
- Chapter 4 discusses basic manual optimizations and programming constructs you can use to increase efficiency using shared memory.
- Chapter 5 discusses the memory classes available for partitioning data on SPP Series machines, and how to use them.
- Chapter 6 discusses advanced manual optimizations and programming constructs you can use to increase efficiency using shared memory.
- Chapter 7 discusses message passing programming using ConvexPVM on Exemplar machines.
- Chapter 8 discusses common optimization problems that you may encounter and provides suggestions for how to deal with them.
- Chapter 9 discusses the aggressive optimizations performed when the `-uo` option is specified.
- Appendix A lists and describes compiler directives and pragmas available for use with the SPP Series Fortran and C compilers.
- Appendix B lists and describes the optimization options available for use on SPP Series machines.
- Appendix C explains the optimization report.
- Appendix D discusses the Compiler Parallel Support library.
- An Index is included at the end of the document.

Suggested reading order

This book takes a bottom-up approach to presenting information. Chapters 1, 2 and 3 provide background

information which will help you understand how the Exemplar architecture and 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, more complicated programming techniques to even further improve performance; Chapter 7 discusses message passing on Exemplar machines, which requires even more manual intervention. Chapters 8 and 9 tell 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), which is the most programmer-dependent optimization tool available for SPP Series computers.

If you are interested in a general, comprehensive overview of programming for the Exemplar computer, 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, start with chapters 3 and 4, and 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 system 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, start with Chapter 7; you may also want to read chapters 2 through 6 to see how the compilers can help you with nonparallel performance.

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

Notational conventions

This section discusses notational conventions used in this book.

General conventions

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

- *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

  Note that except where noted, the directives and pragmas described in this book can be used with both the C and Fortran compilers. In general discussion, these directives and pragmas are presented in lower case type, but either compiler will recognize them regardless of their case.
- 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, you would type:

```
man 1 mnpgname
```

Note

A Note highlights important supplemental information.

Caution

A Caution highlights procedures or information necessary to avoid damage to equipment, software, or data.

Command syntax

Consider this example:

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

1. `COMMAND` must be typed as it appears.
2. *input_file* indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
4. Either `a` or `b` must be supplied.
5. [*output_file*] indicates an optional file name.

Associated documents

CONVEX Computer Corporation provides the following documents to help you use the Fortran and C compilers and associated tools:

- For more information about the CONVEX Fortran compiler, refer to the *Fortran User's Guide* (DSW-038), *Fortran Language Reference Manual* (DSW-037), and *Release Notice, Fortran Compiler V9.1*.
- For more information about the CONVEX C compiler, refer to the *C User's Guide* (DSW-086) and *Release Notice, CONVEX C V6.1*.
- For more information about CXpa, refer to the *CXpa Reference* (DSW-253).
- For more information on the CXdb debugger, refer to the *CXdb Reference* (DSW-472).
- For more information on the Exemplar architecture, refer to the *Exemplar Architecture* manual (DHW-014).
- For more information on administering SPP-UX, refer to the *SPP-UX System Administration Guide* (DSW-853).
- For more information on using SPP-UX, refer to the *HP-UX Reference* (Hewlett-Packard order number B2355-90004).

Ordering documentation

To order this document or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, Texas 75083-3851 U.S.A.

Order documents by title, requesting the most recent edition. In some situations, you may not want the current edition. To receive a specific edition of a manual, contact the local CONVEX office or call the Technical Assistance Center (TAC).

Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC):

- Within the continental U.S., use (800)952-0379.
- Outside continental U.S., contact your local CONVEX office.

Acknowledgments

This book is the product of unprecedented cooperation among CONVEX development, marketing, support and technical writing staffs. While it is impossible to list every person that contributed ideas to the *Exemplar Programming Guide*, the author would like to especially thank the following people:

- Joel Williamson, whose *SPP-1 Programming Model* document served as a catalyst, basic outline, and technical reference for this book. Without his guidance and patience, this book could never have been written.
- Greg Astfalk, who provided the basis for Appendix A and spent many hours explaining concepts and reviewing drafts.
- Aaron Potler, who provided understandable explanations of complex features and performed thorough, timely reviews.
- The compiler, operating system and library development teams, who provided countless (often repetitive) explanations and spent many hours reviewing the book: Gary Applegate, Rich Bleikamp, Terry Caracuzzo, Robert Cox, Tony Linthicum, Randy Meyer, Geoffrey Rogers, Perry Schmidt, Mark Schroeder, and Bill Torkelson.
- The technical reviewers, whose comments improved the accuracy and readability of the book: Marianne Chamberlain, Robert Ellis, James Hansen, Paco Romero, Adam Schwartz, and Sharon Shaw.

- The editors, who improved the overall quality of writing in the book: Jerry Hill and Kristi Knox.
- The managers who provided the vision, resources and guidance necessary to make this book and the compilers it documents possible: Matt Blanton, Steve Fieler, Marilyn Lutz, Tom McClendon, Steve Rowan, and Presley Smith
- Other CONVEX personnel and preferred partner customers that offered suggestions and opportunities to solicit them: Frank Baetke, Don Davis, Wayne Greenberg, Phil Merkey, Daniel Savarese, and Thomas Sterling.

—Vince Cavasin, October 1994

Introduction 1

The distinguishing characteristic of CONVEX supercomputers has always been their ease-of-use. The CONVEX Exemplar family of computers brings this ease-of-use legacy to a massively parallel architecture. With the Exemplar family, the full power of massive parallelism can finally be realized without the need to employ complicated and highly platform-dependent programming paradigms. Exemplar compilers generate efficient parallel code with very little intervention on your part; this efficiency can be even further increased by using the techniques discussed in this book.

This chapter provides a general overview of the Exemplar architecture as compared to other parallel architectures, the optimizations available on Exemplar, and the applicable programming models.

Scalable parallel processing

CONVEX Exemplar systems implement massively parallel processing (MPP) using scalable parallel processing (SPP) technology. The Exemplar SPP1000 Series is the first generation in the Exemplar line. It consists of the compact SPP1000/CD system, and the full-size SPP1000/XA system.

As the name implies, scalable parallel machines can be scaled to meet your specific needs. Many MPP systems can only be expanded by doubling the number of processors, or require a large number of processors in their base configuration; CONVEX Exemplar SPP1000 Series systems can contain as few as 2 processors and as many as 128, and can be scaled by as few as 2. These processors are arranged in 1 to 16 hypernodes, with each hypernode containing 2 to 8 processors. Processors communicate with each other, with memory and with I/O devices via the Coherent Toroidal Interconnect (CTI). The CTI consists of a nonblocking crossbar on each hypernode for intrahypernode communication, and four high-speed CTI rings

which link the hypernodes together for interhypernode communication. The CTI ring design is derived from the IEEE standard 1596-1992, SCI (Scalable Coherent Interface), but this Exemplar implementation sacrifices complete SCI compatibility to provide lower latencies.

SPP1000 Series memory is also easily scalable; each hypernode can support 128 Mbytes, 256Mbytes, 512 Mbytes, 1 Gbyte or 2 Gbytes of physical memory. This allows for a maximum of 32 Gbytes on a 16-hypernode system. Each process can access its full 32 bit (4 Gbyte) virtual address space, and, if necessary, programs can be written in a manner that allows them to surpass the 4 Gbyte limit and access all the memory on a system.

SPP vs. traditional 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 SPP architectures below, but the same differences apply in principle to all vector/parallel machines.

Architectural differences

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, cross bars 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 which speed loop computations involving arrays by performing array arithmetic on up to 128 elements per vector instruction. Machines containing multiple CPUs can further reduce time to solution by adding parallelism at the process, loop, and code-region level.

The SPP1000 Series architecture takes a different approach. The SPP1000/CD systems support 2, 4, 8, 12 or 16 Hewlett-Packard PA-RISC 7100 microprocessors in one or two hypernodes. The SPP1000/XA systems support 4 to 128 PA-RISC processors in 1 to 16 hypernodes. Rather than using vector units to exploit fine-grain parallelism, these processors speed scalar processing by using a reduced set of high-speed instructions coupled with pipelining, 1Mbyte instruction and data caches, and a large register set.

Two-dimensional parallelism, which can benefit nested loops or parallel regions of code containing loops, is also possible on multihypernode SPP1000 Series machines. Rather than

implementing the first dimension in the vector unit and the second across processors (as in C Series), the SPP1000 Series can implement the first level within a hypernode and the second across hypernodes. Single-dimensional parallelism that spans hypernodes is also possible.

Memory

Because of the large number of processors possible on a multihypernodal SPP1000 Series system, memory access via a system-wide crossbar is not practical. Instead, low latency, high-bandwidth memory access is provided by globally shared memory (GSM). 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; 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 commonly 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 will optimize well on the SPP1000 Series with little manual intervention. CONVEX SPP1000 Series compilers are designed to automatically exploit opportunities for parallelism and data localization in programs written for shared memory machines. Chapters 3 through 7 discuss manual optimizations that can yield even more performance from such programs.

SPP vs. clustered workstations

While the SPP1000 Series architecture uses the same processors found in HP workstations, its GSM, automatic optimizing compilers, high-speed interconnections, shared peripherals, and user-configurability sharply distinguish it from clustered workstations. The following subsections discuss each distinguishing feature in detail.

Memory

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

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

Optimizing compilers

Programs for clustered workstations are compiled using the workstations' compilers. If the cluster contains workstations that require different executables (i.e. 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.

CONVEX compilers have long been highly regarded for their ability to automatically optimize code. CONVEX Exemplar compilers continue this legacy and add to it fully automatic node- and thread-wise parallelism and several new data localization optimizations designed to improve memory usage and aid parallelization. Additionally, a rich set of directives allows you to even further enhance the automatic optimizations performed on your shared memory program.

CONVEX Exemplar compilers give the highest performance with little or no programmer intervention from generic programs which exploit GSM. Message passing programs, with their parallelism explicitly coded, will also benefit from all Exemplar compiler optimizations.

Interprocess communication

To communicate among themselves or access each other's data, the processors in a cluster of workstations must communicate over low-performance networks and access distributed memory. Communication can only be handled 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 do efficiently on clusters because of the need for frequent data accesses and synchronization.

Exemplar systems are suitable for both coarse- and fine-grained parallelism. Programs containing potential parallelism will, when compiled with CONVEX Exemplar compilers, automatically exploit the parallelism, spawning threads to run on as many processors (or even hypernodes) as are available, and rejoining these threads upon completion. This fine-grained parallelism will take full advantage of Exemplar's fully coherent caches and high-speed interconnects. While message passing is supported, and can be used to speed certain applications (see Chapter 7), with GSM, it is not necessary for most programs. When message passing is used on Exemplar, the high-speed interconnects can give a substantial performance increase over traditional networks. This makes message passing programs that exploit fine-grained parallelism practical.

SPP-UX automatically schedules threads to idle and underused processors as necessary, insuring a balanced machine load and exploiting both thread- and process-level parallelism.

Peripherals

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

Configurability

Exemplar systems can be configured by system administrators into one or more *subcomplexes*. A subcomplex is a virtual machine consisting of a specified number of processors on specified hypernodes. Subcomplexes allow Exemplar systems to be configured into several virtual machines which make the most effective use of available resources given the computational tasks at hand. Subcomplexes can be reconfigured

as needed without rebooting the machine; this allows time- or task-dependent reconfigurations, or immediate correction of inefficient configurations. No similar capability is available for clusters.

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 Exemplar, on the other hand, increases unhindered by a traditional network or private-memory limits. Adding peripherals and memory to an Exemplar system can also provide improved absolute performance, since all processors can access both, whereas memory and peripherals are processor-specific on clusters.

The Exemplar programming model

The Exemplar programming model provides two perspectives from which a programmer can write (or adapt) code to run on Exemplar. Those perspectives are the shared memory and message passing paradigms.

The shared memory paradigm

In the shared memory paradigm, the compilers handle optimizations, and, if requested, parallelization. Data distribution is taken care of by hardware via the CTIcache. Numerous compiler directives and pragmas (which are discussed in detail in chapters 4 and 5, and listed in Appendix A) are available to even further increase optimization opportunities. Since the shared memory approach is used on most non-MPP machines, Exemplar's support of it allows programs from such machines to be easily ported to the Exemplar architecture. Most programmers will also use the familiar shared memory paradigm for programs written from scratch for Exemplar, since it automates many of the tasks for which the message passing paradigm requires explicit coding.

Chapters 4 and 6 cover shared memory programming in detail.

The message passing paradigm

The Exemplar message passing paradigm uses a version of the ConvexPVM (Parallel Virtual Machine) library, which is also

available for use with clustered workstations. However, on Exemplar, with its tightly-coupled processors, interprocess communication is much faster, and fine-grain parallelism is more practical.

Under the message passing paradigm, ConvexPVM functions are used to explicitly spawn parallel processes, share data among them, and coordinate their activities. There is no shared memory; each process has its own private 4Gbyte address space (this is determined by the HP 7100's 32-bit address space) and any data that must be shared must be explicitly passed between processors. Since processes cannot access each other's memory space, cache coherency and its inherent duplication of data is not necessary, so a larger amount of physical memory can be made available to the program as a whole.

Support of message passing allows programs written under this paradigm for distributed memory machines to be easily ported to Exemplar. Programmers familiar with using the message passing style may choose to write new programs for Exemplar using this paradigm rather than shared memory, and they can realize a substantial performance boost over conventional message-passing machines, even when coding fine-grained parallelism. Lastly, the few programs that require more per-process memory than possible using shared memory will benefit from the manually-tuned message passing style.

Chapter 7 covers message passing programming in detail.

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 the process-private memory benefits of message passing are exploited.

Overview of Exemplar optimizations

CONVEX Exemplar compilers perform a broad 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 into which there is a single entry point and out of which there is 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 main memory accesses.

The compilers can perform a number of transformations on loops to facilitate more efficient data localization. The most fundamental of these transformations is *blocking*. Blocking breaks a loop whose data is too large to fit entirely into the processor data cache into a loop nest; the inner loop can then make efficient use of the cache while the outer loop runs the inner loop as many times as necessary to cover the entire original trip count.

Advanced scalar optimizations include several other 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 detail in Chapter 3, "Compiler optimizations."

Advanced scalar optimizations implicitly include all basic scalar optimizations.

Parallelization

It is through parallelization that you can realize the full power of a scalable parallel computer like Exemplar. Parallelization allows a program to be executed by as many processors as are available within its subcomplex, in most cases dramatically reducing time-to-solution. CONVEX Exemplar compilers can automatically locate and exploit loop-level parallelism in most programs, and, using the techniques described in Chapters 5, 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.

Architecture overview

2

This chapter provides an overview of the Exemplar system architecture for programmers. For more detailed information on all the topics discussed in this chapter, refer to the *Exemplar Architecture* manual, order number DHW-014.

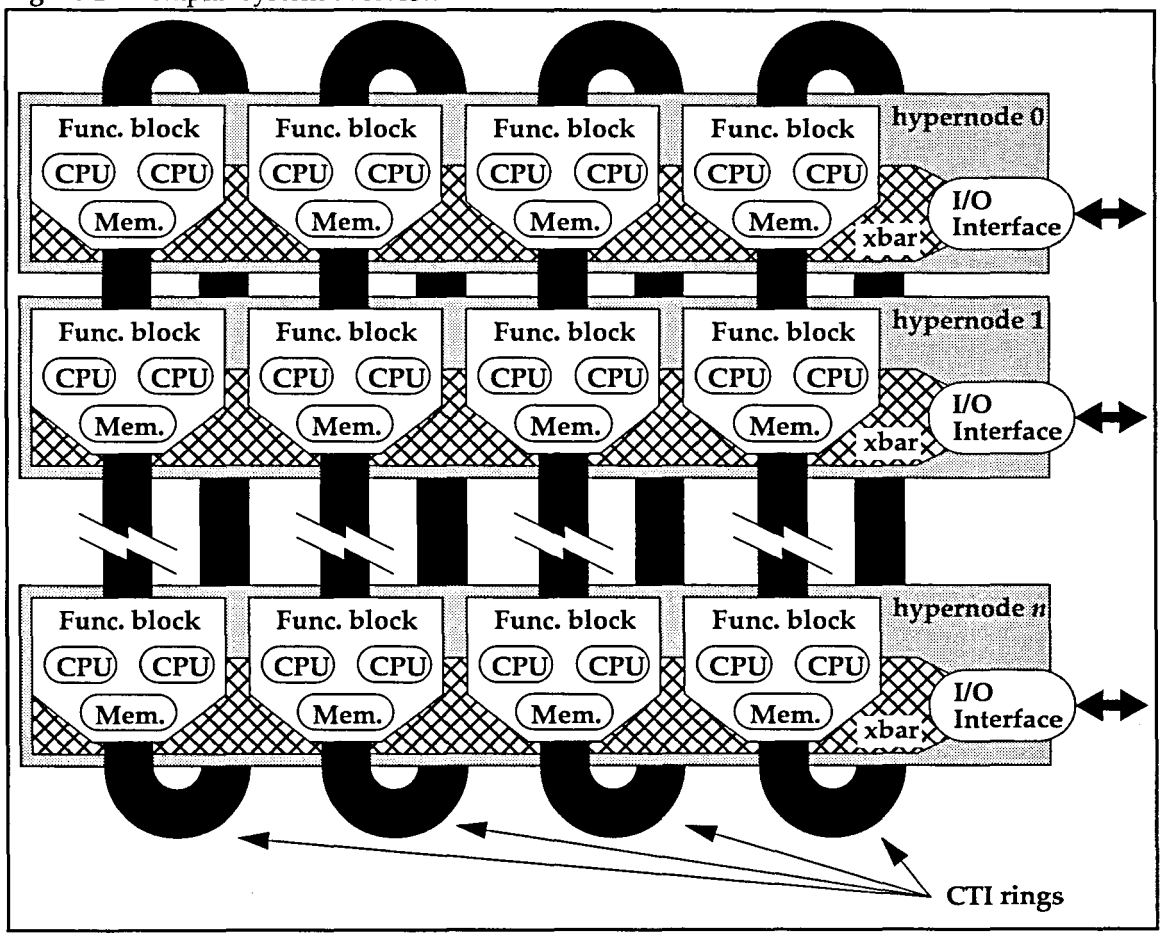
System organization

Think of an Exemplar SPP1000 system as a shared memory computer with two tiers of memory latency. The hypernode crossbar constitutes the first tier, and the CTI rings constitute the second. Exemplar SPP1000/XA systems consist of 1 to 16 hypernodes. Each hypernode contains 4 or 8 processors and 256 Mbytes, 512 Mbytes, 1 Gbyte or 2 Gbytes of physical memory. Processors are arranged in functional blocks, each containing two processors, 128 Mbytes to 512 Mbytes of memory, and some control devices. Functional blocks within a hypernode communicate with each other, with memory, and with peripherals via a 5 port nonblocking crossbar. Functional blocks communicate across hypernodes via four CTI rings.

Exemplar SPP1000/CD systems consist of 1 or 2 hypernodes, each containing 2, 4 or 8 processors, but are otherwise identical to SPP1000/XA systems.

Figure 1 shows an overview of a multihypernode Exemplar system containing 8 processors per hypernode.

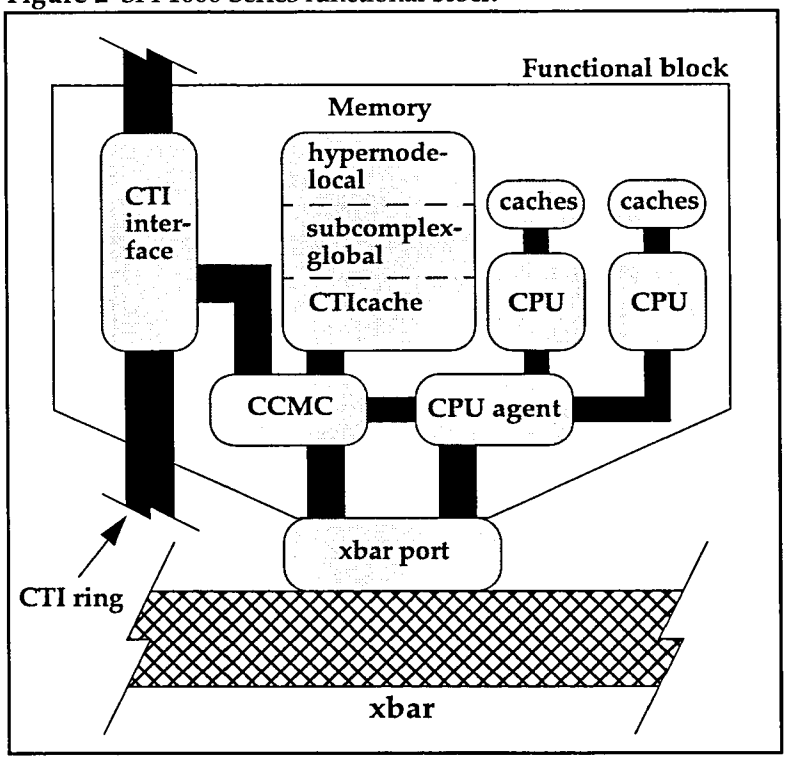
Figure 1 Exemplar system overview



Though it is difficult to illustrate in two dimensions, hypernodes are distributed evenly around the CTI rings. While the rings attach specific functional blocks within the hypernode as illustrated, any functional block can access memory on any other functional block by routing its request through its own crossbar to the functional block that is attached to the correct ring. Data is returned via the same ring and then routed via the crossbar back to the requesting functional block.

Figure 2 shows a single functional block in more detail.

Figure 2 SPP1000 Series functional block



CPUs communicate directly with their own instruction and data caches, which are each 1Mbyte in size and can be accessed by the CPU in 1 clock. The functional block's two CPUs communicate with the rest of the machine through the CPU agent. The CONVEX Coherent Memory Controller (CCMC) provides the interface between the functional block's 2 memory banks and the rest of the machine. All intrahypernode memory accesses take 56-60 clocks, even if they can be fulfilled from the functional block's own memory block. This is because they must traverse the crossbar, which gives equal access to all hypernode memory from all functional blocks.

Memory

Each shared memory process running on an SPP1000 Series system accesses its own 4 Gbyte virtual address space. Processes cannot access each other's virtual address space. This virtual memory maps to the physical memory of the subcomplex on which the process is running. Physical memory is partitioned on a subcomplex basis; subcomplexes cannot access each other's physical memory, just as processes cannot access each other's virtual memory.

Physical memory

All memory (excluding processor caches) on the SPP1000 Series system is implemented in the memory banks of the functional blocks. Each functional block contains 2 memory banks, for a total of 8 banks per 8-CPU hypernode. As shown in Figure 2, this memory is typically partitioned (by the system administrator) into hypernode-local, subcomplex-global, and CTIcache. 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 SPP-UX executables, as well as user process data that has been explicitly declared private, reside.

Subcomplex-global memory is accessible by all CPUs in a given subcomplex. This memory can be allocated by the system administrator during machine configuration as discussed in the "Subcomplexes" section, which follows.

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

Virtual memory

Virtual memory is divided into 5 classes. The compilers will choose default classes for your programs that provide good performance; programmers can also manually assign data to memory classes to improve data distribution and further increase performance. Note, however, that doing so also requires some other aspects of optimization, particularly loop parallelization, to be handled manually.

Briefly, the virtual memory classes and their physical memory mappings are:

- `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 private to the threads running on a given 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.

- 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 subcomplex. All data of a near_shared object maps to single physical addresses on a particular 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 subcomplex. Physically, far_shared data is distributed by pages round-robin to all the hypernodes in the subcomplex, 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 subcomplex. Physically, block_shared data is distributed in blocks equally among the hypernodes, one block per hypernode. block_shared memory must be dynamically allocated; the programmer can then easily insure 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 4, "Memory classes."

Cache lines

A *cache line* describes the size of a chunk of contiguous data that must be copied into 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. A *CTIcache line* is 64 bytes long; this is the amount of data moved from global memory to the CTIcache when a CTIcache miss occurs. A CTIcache line consists of two contiguous *processor cache lines*, which are 32 bytes long; when a processor cache miss occurs, this is the amount of data fetched from memory. 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 replicated in the CTIcache.

Cache thrashing

The SPP1000 Series uses 1 Mbyte 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. On the SPP1000 Series, cache addresses are computed within a process using the following formula:

$$cache\_address = \mathrm{MOD}(virtual\_address, 2^{20})$$

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 means that unique data objects whose virtual addresses share their least-significant 20 bits will have identical cache addresses. If such data objects are referenced from within the same loop, they will overwrite each other in the cache, generally resulting in constant cache misses, and causing unnecessary and costly main-memory accesses. This condition is known as *cache thrashing*.

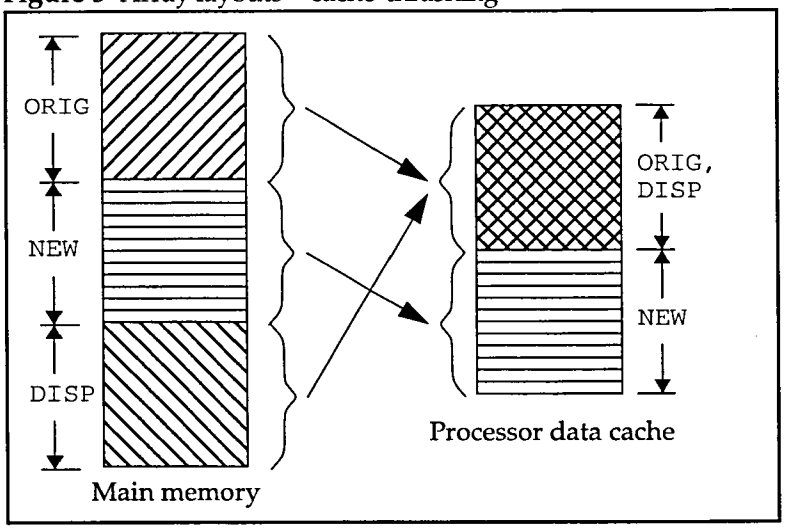
Cache thrashing can become a problem when two encachable data objects are exactly 1 Mbyte apart in memory; to eliminate the problem, you need only insure that your data is not spaced this way.

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 will overwrite each other. 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 .5 Mbyte (8 × 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 3.

Figure 3 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 main memory into the cache. Since 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.

Since main 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.

But the problem is easily fixed by increasing the distance between the arrays. This can be done by either increasing the array sizes or inserting a padding array between each original array. The following example illustrates the padding approach.

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

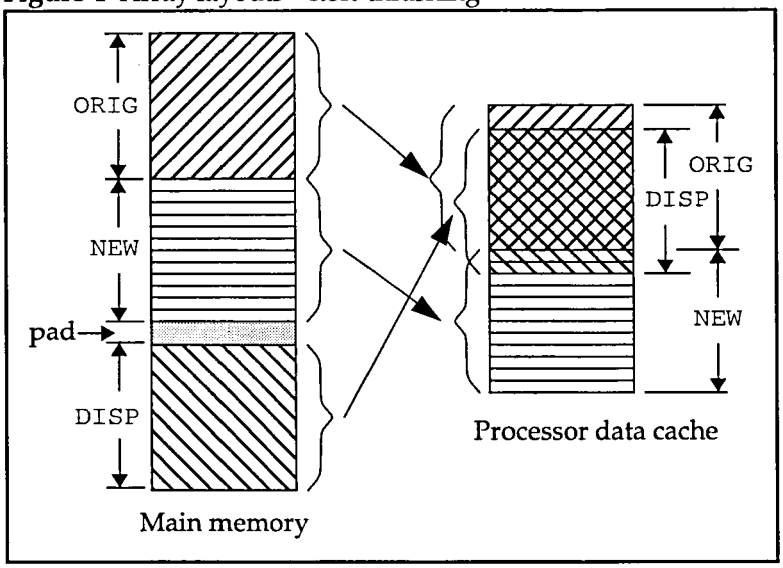
Here, the array `P(8)` moves `NEW` and `DISP` 64 bytes further from `ORIG` in memory. Now no two elements of the same index share a cache address, and for the given loop, this postpones cache overwriting until the entire current cache line is completely exploited. `P` is 8 elements, or 64 bytes, which prevents CTIcache as well as processor cache thrashing; a size of 32 bytes would work to prevent processor cache thrashing, but there is a slim chance CTIcache thrashing could still occur since a CTIcache line is 64 bytes.

The alternate approach involves increasing the size of `ORIG` or `NEW` by 8 elements (64 bytes), as shown in the following example.

```
REAL*8 ORIG(65536),NEW(65544),DISP(65536)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
```

Here, `NEW` has been increased by 8 elements, providing the padding necessary to prevent `ORIG` from sharing cache addresses with `DISP`. Figure 4 shows how both solutions prevent thrashing. Note that the pad never gets encached, whether it consists of the `P` array or an extended `NEW` array, because it is never referenced. If it was referenced, it would not solve the thrashing problem.

Figure 4 Array layouts—non-thrashing



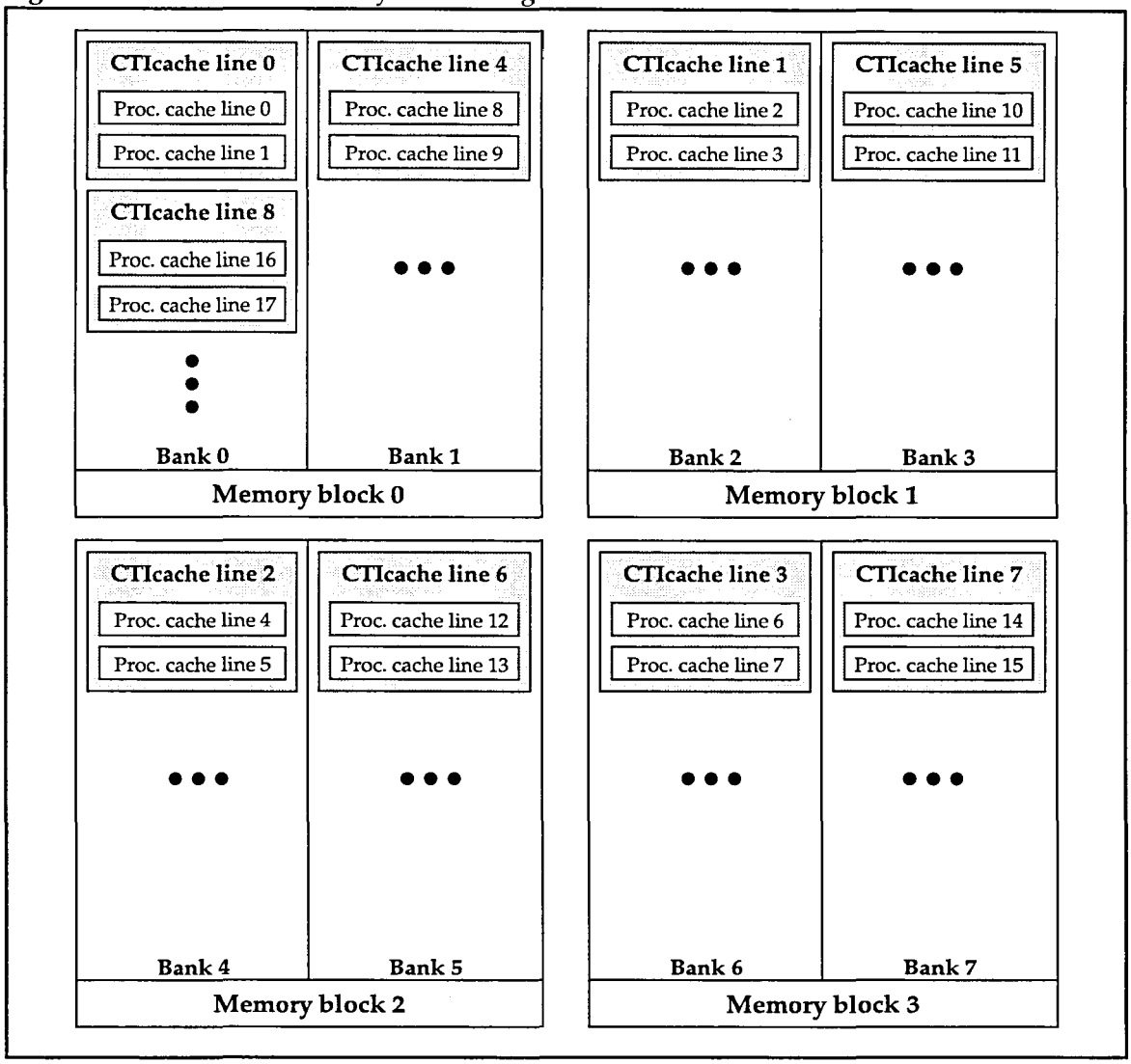
It is important to note that this is a highly simplified, worst-case example. On SPP1000 Series machines, thrashing can happen any time two data items that are referenced in the same loop are an integral multiple of 1 Mbyte apart in virtual memory. This can happen with data that is not stored in common, 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, 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 will not necessarily eliminate the problem completely. Using common blocks in Fortran can also help because 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 8 banks of a hypernode by CTIcache lines. Contiguous CTIcache lines are assigned in round-robin fashion, first to the even banks, then to the odd, as shown in Figure 5.

Each memory block shown in Figure 5 is located on a separate functional block within the hypernode.

Figure 5 SPP1000 Series memory interleaving



Interleaving speeds memory accesses by allowing several processors to access contiguous data simultaneously. This is extremely 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.

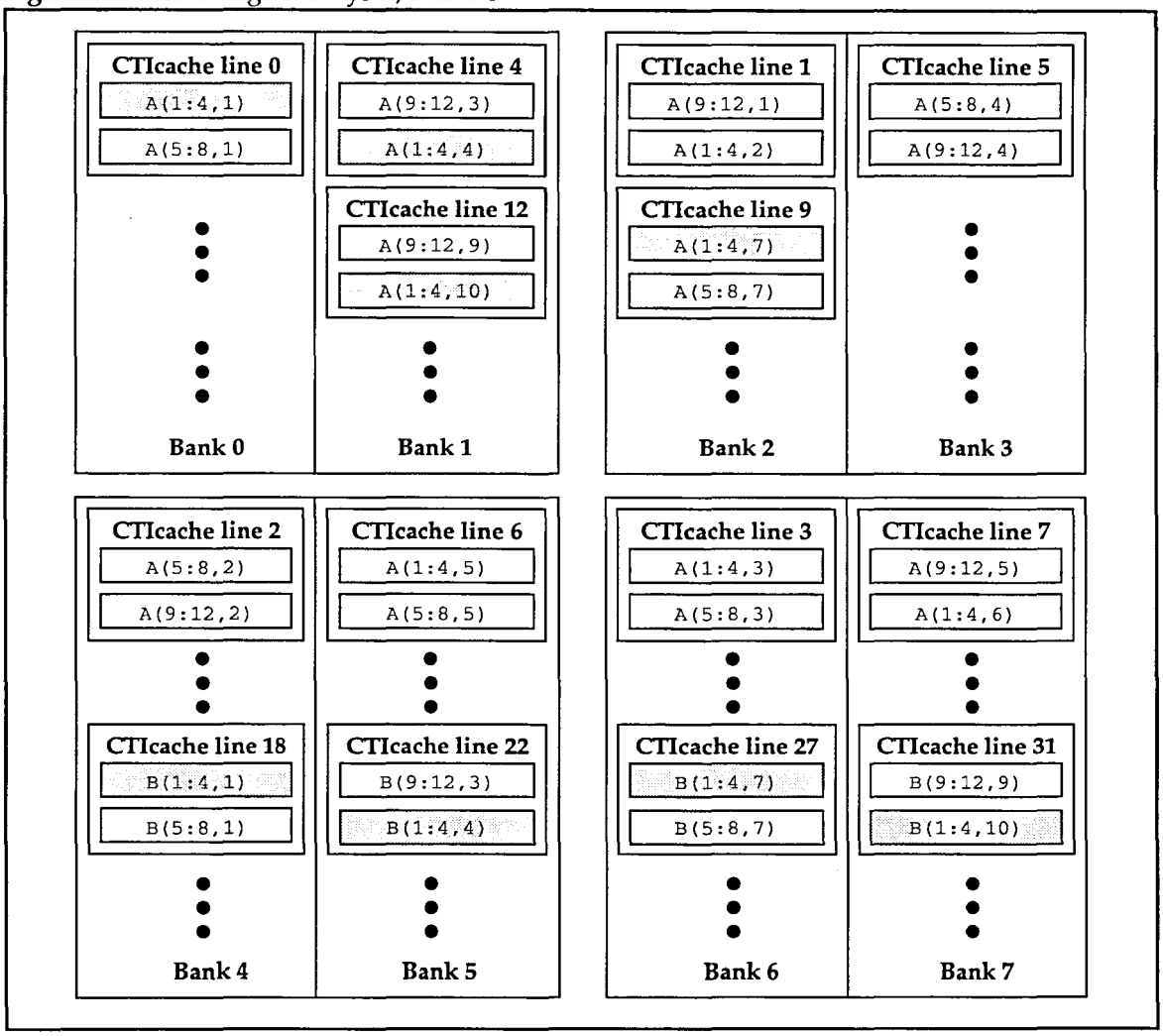
```
REAL*8 A(12,12), B(12,12)
...
DO J = 1, 12
  DO I = 1, 12
    A(I,J) = B(I,J)
  ENDDO
ENDDO
```

Assume that arrays A and B are stored contiguously in main memory, with A starting in bank 0, CTIcache line 0, processor cache line 0, as shown in Figure 6.

Assume the CONVEX SPP1000 Series Fortran compiler parallelizes the J loop to run on as many processors as are available in the subcomplex (up to 12). Assuming there are four processors available when the program is run, the J loop would be divided into four new loops, each with 3 iterations. Each new loop would run to completion on a separate processor. We'll refer to these four processors as CPU0 through CPU3.

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.

Figure 6 Interleaving of arrays A, B and C



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)`. 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 6 shows the first 8 CTIcache lines consecutively; after these, only the initial cache lines for each processor are shown. Each processor's initial cache line is shaded.

Without interleaving, all four CPUs would request their arrays simultaneously from the same bank. Arbitration logic would order the CPUs; whichever ended up going first would get its requested cache line while the second experienced a 28 clock

delay, the third a 56 clock delay, and the fourth a 84 clock delay. Before the second CPU had even received its cache line, the first would likely be back in the queue waiting for its next cache line.

Interleaving helps to eliminate such contention. `A(1,1)`, which is the first element of the chunk needed by CPU0, is on cache line 0 in bank 0; `A(1,4)`, the first element needed by CPU1, is on cache line 4 in bank 1; `A(1,7)`, the first element needed by CPU2, is on cache line 9 in bank 2, and `A(1,10)`, the first element needed by CPU3, is on cache line 12 in bank 1. Contention only exists between CPU1 and CPU3, and after one of these CPUs gets its cache line and moves on, it will proceed at full speed with the other CPU 28 clocks behind it. In other words, after the initial access, the contention will cease. Contention may resurface occasionally as the processors make their way through the data, but the resulting delays are minimal compared to what could be expected without interleaving.

The initial chunks of `B` are even more favorably distributed, in banks 4, 5, 6, and 7, respectively. No initial contention exists for `B`.

Subcomplexes

On Exemplar systems, processes run on virtual machines called subcomplexes, which are arbitrary collections of processors. Subcomplexes are highly configurable, and configuration is done using the CONVEX Subcomplex Manager. For more information on the Subcomplex Manager, refer to the *SPP-UX System Administrator's Guide.*

Subcomplexes allow the system administrator to tailor processors and memory to your specific application needs, making the most efficient use of your Exemplar system resources. For example, a nonparallel or non-time-critical application can be allocated a single processor; an application containing a lot of fine-grain parallelism can be allocated many processors, and an application requiring large amounts of memory can be allocated processors on several hypernodes. Because subcomplexes can be managed without rebooting, you can adjust configurations to automatically exploit periodic changes in system usage such as late-night or weekend idle time.

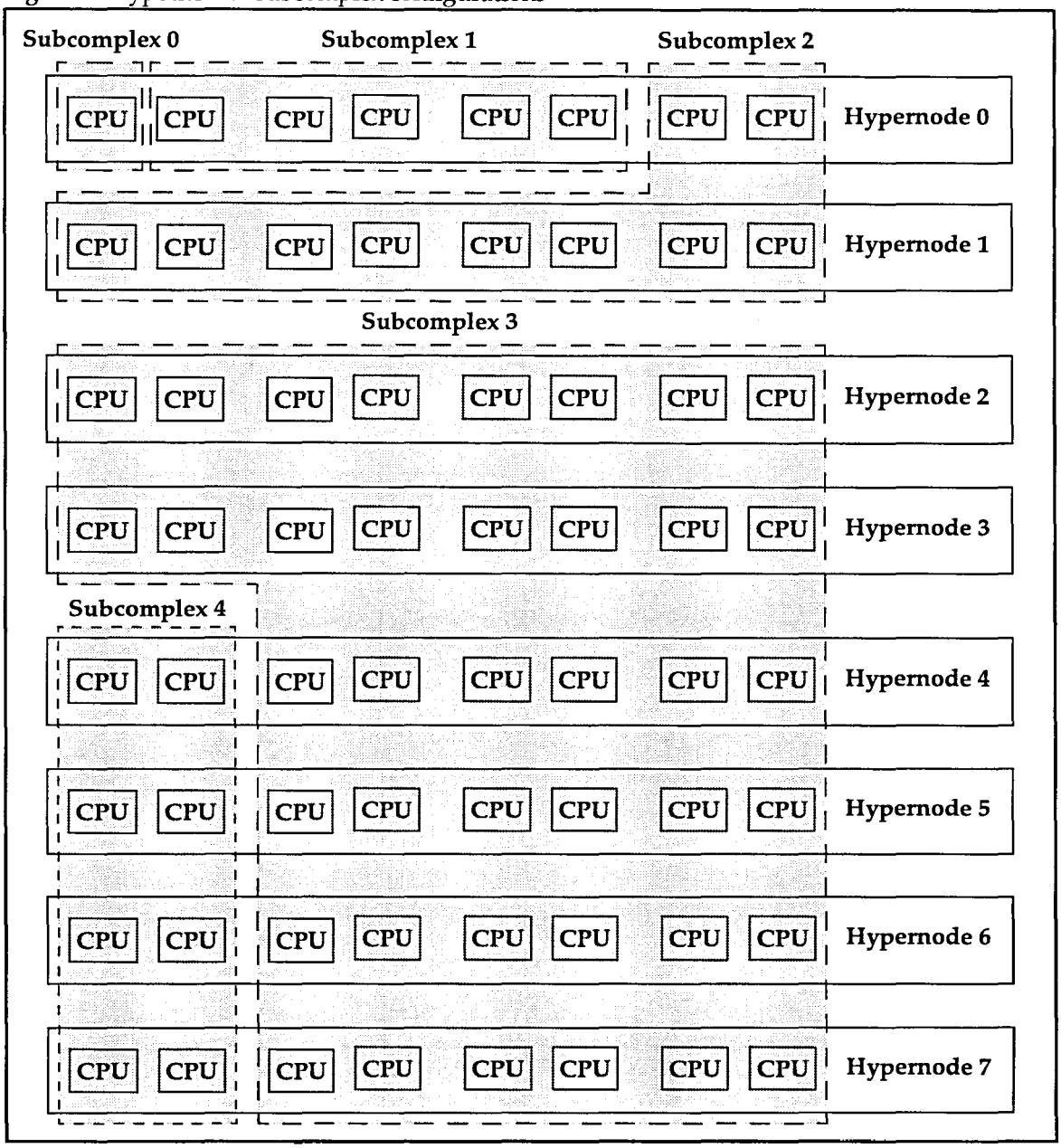
Physical configuration

Subcomplexes can consist of as few as one processor or as many as the total number installed on the machine. Hypernodes can be split among as many subcomplexes as there are processors in the hypernode, and subcomplexes can be subsets or supersets of

hypernodes. Processors can only belong to one subcomplex at a time, and every processor must belong to a subcomplex.

Figure 7 shows an 8-hypernode, 64-processor system split into 5 subcomplexes.

Figure 7 Hypothetical subcomplex configurations



Processors can be added to or removed from a subcomplex at any time; depending on the global memory requirements of the hypernodes making up the subcomplex, the subcomplex may or may not need to go idle. Idling a subcomplex requires killing all of its processes.

Subcomplex memory can also be configured using the Subcomplex Manager. Global memory can be allocated when the subcomplex is idle, in 16Mbyte increments, up to the total amount available, from any hypernode which has one or more processors in the subcomplex.

Also configurable are subcomplex permissions, which are similar to file permissions and can be set at the user, group, and world level. Read permissions on a subcomplex allow you to read the subcomplex configuration; write permissions allow you to change the subcomplex's job scheduling policies, such as process execution priority; execute permissions allow you to run a process on the subcomplex. Only root can configure or reconfigure a subcomplex.

Subcomplex memory

GSM is global to a subcomplex, and processes run entirely within their assigned subcomplex. For a program to run on more than one subcomplex, message passing via sockets (not the CTI rings) must be used to communicate between the subcomplexes; this is much slower than interhypernode communication within a subcomplex.

The virtual address spaces accessible from within a subcomplex are unique; this prevents processes running on one subcomplex from accessing memory on another subcomplex. Furthermore, the virtual address spaces accessible to each process running on a subcomplex are unique; each process is assigned a 4Gbyte virtual space, and processors cannot access each other's memory.

When two or more subcomplexes share a single hypernode, the physical memory pages assigned as global memory for each subcomplex are unique, preventing contention over global memory. However, the physical pages associated with hypernode-local and CTIcache memory are shared among multiple subcomplexes running on one hypernode; this can result in contention over these resources.

Compiler optimizations 3

This chapter discusses the various optimization options available for use with the SPP1000 Series Fortran and C compilers and provides detailed explanations of the optimizations performed under each option.

Optimization options

Five optimization options are available for use with the SPP1000 Series C and Fortran compilers. These options have identical names and perform identical optimizations regardless of which compiler you're using. They are specified on the compiler command line along with any other options you wish to use. SPP1000 Series compiler optimization levels are summarized in Table 1.

Table 1 SPP1000 Series C and Fortran optimization options

Option	Description
`-no`	Machine instruction level scalar optimizations. This option is the default.
`-O0`	Basic block level scalar optimizations.
`-O1`	Program unit level scalar optimizations and global register allocation.
`-O2`	Global instruction scheduling and data localization optimizations.
`-O3`	Parallel optimizations.

These options are cumulative; each option retains the optimizations of the previous option. For example, entering the following command line:

```
% fc -O2 foo.f
```

Would compile the Fortran program `foo.f` with all `-O2`, `-O1`, `-O0`, and `-no` level optimizations shown in Table 1.

`-no` level optimizations

At optimization level `-no`, the compiler performs scalar optimizations that span no more than a single source statement. These optimizations create object code that fully uses the scalar features of the PA-RISC architecture.

Instruction scheduling

Instruction scheduling rearranges machine instructions to use the computer's functional units most effectively. Each CPU on an SPP1000 Series system has multiple functional units on which operations execute simultaneously.

At optimization level `-no`, the compiler rearranges the machine instructions generated by no more than a single source statement to maximize use of the functional units.

Concurrent execution of machine instructions on multiple functional units, within a single processor, is distinct from parallel processing, which occurs on multiple processors.

Span-dependent instructions

There are several ways to specify a branch in the PA-RISC machine language; short branches can be more efficiently executed than long branches. The SPP1000 Series compilers automatically generate branches using the most efficient and appropriate branching techniques.

For more information on branch instructions, see the Hewlett Packard *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*.

Register allocation

The SPP1000 Series compilers use a technique for allocating registers that fully exploits the PA-RISC register set. This allows grouping of register loads and concurrent execution of instructions (*pipelining*), and reduces register conflicts.

Tree-height reduction

The compiler represents expressions internally as trees. When they involve floating point numbers, these trees are optimized

by *tree-height reduction* or *balancing*. For example, consider this Fortran expression involving REAL variables:

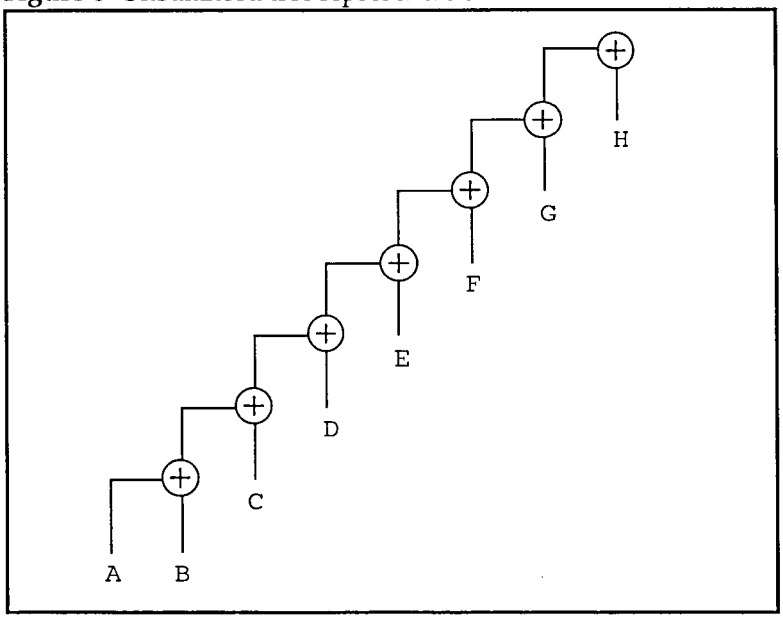
```
A + B + C + D + E + F + G + H
```

The expression can be evaluated as follows:

```
((((((A + B) + C) + D) + E) + F) + G) + H)
```

(A+B) is evaluated first, the result is added to C, and so on until the expression is fully evaluated. Figure 8 shows how the compiler represents this order internally.

Figure 8 Unbalanced tree representation



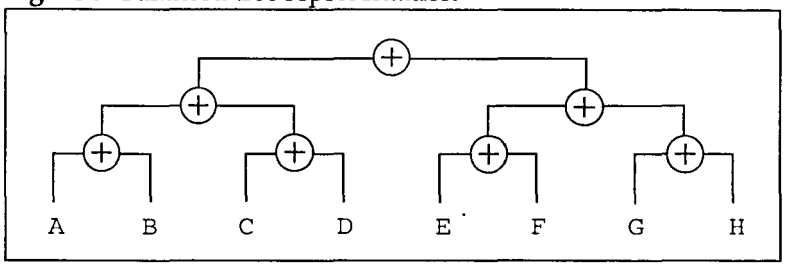
The PA-RISC processor's add functional unit takes two clocks to complete an add, but pipelining allows it to issue one add result per clock if the pipeline is kept full. The unit does not chain, so the result of an add in progress cannot be used in the next add. Since each addition in this example depends on the result of the addition to the right, pipelining in the add functional unit cannot be exploited; it can then only issue one result every two clocks.

Another way to evaluate the expression is

```
((A + B) + (C + D)) + ((E + F) + (G + H)))
```

This is represented internally as shown in Figure 9.

Figure 9 Balanced tree representation



In Figure 8, the depth of the tree is seven; in Figure 9, the depth of the tree is three. The code generated for the tree in Figure 8 executes slower than that generated for the tree in Figure 9.

This is because none of the four additions in the innermost parentheses (represented as leaf nodes on the tree in Figure 9) requires the result of another addition. The additions can therefore be fed to the add functional unit so that they fill its pipeline, allowing it to issue one result per clock after the first add and up until the last. `(A+B)` is evaluated on the first clock, followed by `(C+D)` on the second; on the third, (A+B) is done and the next add is starting. This pattern continues until the expression is fully evaluated.

The deeper the tree representing the expression, the more time is required to evaluate the expression. If a particular evaluation order is not specified with parentheses, the compilers choose one that minimizes the depth of the expression and maximizes instruction pipelining, while maintaining the arithmetically correct evaluation order. Because the compilers choose evaluation order to ensure the most efficient execution, you can write expressions in any order; if your expression requires a certain order, be sure to indicate it with parentheses.

Short-circuit evaluation of conditionals in Fortran

Short-circuiting the evaluation of conditionals is ANSI standard behavior in C, but CONVEX Fortran also performs this optimization.

Short circuiting increases the efficiency of `IF` statements by skipping irrelevant tests when logical operators are involved in the conditional. CONVEX Fortran short-circuits evaluation of `IF` statements that contain `.AND.` and `.OR.` operators that have logical operands and are used in a logical context. Take, for example, the following Fortran `IF` statement:

```
IF ((A .EQ. B) .OR. F(G)) THEN
```

If (A .EQ. B) evaluates to true, the evaluation of F(G) is skipped, and the THEN portion of the statement is evaluated.

Similarly, given the Fortran code

```
IF ((A .EQ. B) .AND. F(G)) THEN
```

if (A .EQ. B) evaluates to false, the evaluation of F(G) and the THEN portion of the statement is skipped.

Short-circuit evaluation works with all types of IF statements (arithmetic, logical, and block). Performing arithmetic (+, -, *, /) on a logical expression disables short circuiting within that expression. Logical-valued expressions used as arguments to function calls within an IF statement's conditional expression may not be short circuited. Note that the binary operators .EQ., .NE., .LT., .LE., .GT., and .GE. always produce a logical result.

The compiler short-circuits the evaluation of conditionals by default. You can disable short-circuiting in Fortran by specifying the -nosc flag on the compiler command line.

-O0 Level optimizations

At optimization level -O0, the compiler performs scalar optimizations within a basic block; a basic block is a sequence of statements with only one entry point and one exit. Basic blocks may end with a branch, but they cannot contain branches. The compiler also continues to perform the optimizations performed at -no.

Instruction scheduling

At optimization level -O0 and above, instructions from *multiple* statements, as well as those from single statements, are scheduled as a group.

Redundant-assignment elimination

Redundant-assignment elimination removes unnecessary assignments to a variable. When a variable is not used between two assignments, the first assignment is eliminated. The code in

the following Fortran example contains a redundant assignment, X=Y+C, which the compiler removes.

Original code	Optimized code
`X = Y + C`	`!(statement eliminated)`
`!(X not used)`	.
.	.
.	.
`X = 3.1416`	`X = 3.1416`
.	.
.	.
.	.
`Y = (X + 7) * 2.15`	`Y = (X + 7) * 2.15`

Assignment substitution

Assignment substitution eliminates redundant loads. The compiler stores the value assigned to a variable in a register and references the register in subsequent references to the variable. A Fortran example appears below.

Original code	Optimized code
`X = Y + C`	`REG = Y + C`
`X = X * 4.4`	`REG = REG * 4.4`
`T = X * B + 12.4`	`T = REG * B + 12.4`
`X = 4.179`	`X = 4.179`

After the machine instructions for the first statement execute, the value of Y+C remains in a register. The compiler replaces subsequent references to X with references to this register until the value of X changes or until the end of the basic block is reached. This optimization eliminates repeated loading and storing of X into a register, which increases performance and provides opportunities for further optimization. In this example, assignment substitution makes the first assignment to X redundant, so the compiler eliminates the assignment.

Because the compiler substitutes assignments, you rarely need to optimize a program by replacing a variable reference with a constant in the source code.

Common-subexpression elimination

The compiler recognizes subexpressions that repeat within a basic block. The compiler retains the value of the subexpression in a register, which eliminates redundant computations and register loads. For example, the compiler recognizes `B+C` as a common subexpression of `A+B+C+D` and `B+E+C`, and calculates the subexpression only once.

The compiler also eliminates redundant array address calculations. As with assignment substitution, you do not need to manually create a temporary variable in which to store the value of a common subexpression. The compiler performs that function automatically.

Redundant-use elimination

This optimization is a special case of common subexpression elimination where the subexpression is a variable. The compiler detects multiple references to a variable between assignments and retains the value of the variable in a register. This action helps eliminate redundant register loads.

Constant propagation and folding

After assigning a constant to a variable, the compiler replaces subsequent references to the variable with the constant. For example, if you write `X=5`, the compiler replaces `X` with 5 within that basic block or until a new value is assigned to the variable. This is known as *constant propagation*, which is a form of assignment substitution.

A Fortran example of constant propagation and folding follows.

Original code	Optimized code
`I = 5`	`I = 5`
`J = 0`	`!assignment eliminated`
.	.
.	.
.	.
`J = J + 2`	`J = 2`
.	.
.	.
.	.
`K = K + I * J`	`K = K + 10`

The compiler also replaces operations on constants with the result of the operation. This is known as *constant folding*. For example, it replaces `Y=5+7` with `Y=12`. It then propagates the constant value to replace future references to `Y` within the basic block. The Fortran compiler also propagates and folds values assigned to names in `PARAMETER` statements.

The compiler folds the most frequently used Fortran intrinsics and C library routines when they are applied to constant arguments. For example, `SIN(0.0)` becomes `0.0`. The compiler also folds exponentiation involving constants. For example, `3**3` becomes `27`.

The compiler type-converts constants, if necessary, before propagating and folding them. If a Fortran program contains the expression `X=1`, where `X` is `REAL`, the compiler converts `1` to `1.0` before propagating it.

If an integer overflow occurs as a result of constant folding in Fortran, the compiler reports "Integer constant truncation." To see such a message in C, you must supply the `-d integer_overflow=e` option on the cc command line. If a floating-point overflow occurs, both compilers reports "Real constant either too large or too small." Floating-point underflow always results in zero. If any of these messages or conditions occur, eliminate the offending operation or bring the value of the constant within acceptable bounds.

Algebraic and trigonometric simplification

The compiler simplifies algebraic and trigonometric expressions, as shown in the following Fortran examples.

Original expression	Optimized expression
`X + 0`	`X`
`X * 1`	`X`
`X * 0`	`0`
`K .AND. -1`	`K`
`K .AND. 0`	`0`
`K .OR. -1`	`-1`
`K .OR. 0`	`K`
`-1 * X`	`-X`
`X - X`	`0`
`X / -1`	`-X`
`(-1) ** K`	`1-((K .AND. 1) * 2)`
`X ** 0.5`	`SQRT(X)`
`X ** 0`	`1`
`1 ** X`	`1`
`X / X`	`1`
`0 - X`	`-X`
`0 / X`	`0`
`SIN(X) * COS(X)`	`0.5 * SIN(2X)`
`SIN(X) / COS(X)`	`TAN(X)`

The compiler performs obvious variations of these operations for the commutative operators. For example, in Fortran the compiler converts `X+(0+Y)` to `X+Y`.

-O1 Level optimizations

At optimization level `-O1`, global optimization is done across a group of basic blocks but within a single procedure. The `-O1`

option performs global, basic-block, and machine instruction level optimizations.

Constant propagation and folding

Propagating and folding constants at the procedure level is analogous to performing the same operations at the basic-block level. The scope of the optimization is now an entire procedure.

A Fortran example of constant propagation and folding follows.

Original code

```
      INTEGER A,B,C
      A = 5
      B = 15
      READ *, I
      IF (I) 10,10,15
10    A = 6
      C = A
      GOTO 20
15    C = A + B
      GOTO 25
20    B = A + C
      GOTO 30
25    B = A + B + C

30    PRINT *,A,B,C
      END
```

Optimized code

```
      INTEGER A,B,C
      A = 5
      B = 15
      READ *, I
      IF (I) 10,10,15
10    A = 6
      C = 6   !A=6
      GOTO 20
15    C = 20 !A=5,B=15
      GOTO 25
20    B = 12 !A=6,C=6
      GOTO 30
25    B = 40
      !A=5,B=15,C=20
30    PRINT *,A,B,C
      END
```

The compiler propagates and folds constants globally at optimization level `-O1` and higher, which eliminates the need to propagate constants by hand in programs compiled at these levels.

Redundant-assignment elimination

At optimization level `-O1`, the compiler eliminates assignments to variables that do not have subsequent references within the program unit. The following Fortran example shows how the compiler eliminates redundant assignments to the variable `A`.

Original code

```
      SUBROUTINE FOO
      INTEGER A
      .
      .
      .
      X = Y * Z
      A = Y**3
      ASSIGN 10 TO B
      .
      .
      .
      IF (A .GT. 0) THEN
        .
        .
        .
        A = X * Y + 3.1416
        GOTO B
      ELSE
        .
        .
        .
        X = (X + 7) * Z + 3.1416 ! X USED LATER
      ENDIF
      .
      .
      .
C     A is not used later in this routine
      END
```

As shown in the optimized code that follows, the Fortran compiler does not eliminate ASSIGN statements and assignments to dummy arguments, function names, and common variables.

Optimized code

```
SUBROUTINE FOO
INTEGER A
.
.
.
X = Y * Z
A = Y**3
ASSIGN 10 TO B
.
.
.
IF (A .GT. 0)THEN
   .
   .   ! ASSIGNMENT TO A REMOVED
   .
  GOTO B
ELSE
   .
   .
   .
  X = (X + 7) * Z + 3.1416 ! X USED LATER
ENDIF
.
.
.
END
```

If the right side of a redundant assignment statement contains a procedure call, the compiler eliminates the assignment and retains the call, as in the following Fortran example.

Original code	Optimized code
`SUBROUTINE FOO`	`SUBROUTINE FOO`
.	.
.	.
.	.
`I = INTFUN(X)`	`<NULL> = INTFUN(X)`
.	.
.	.
.	.
`Comment: I not used`	

If a C or Fortran function appearing in an assignment has no side effects, the compiler can eliminate the function call, as well as the assignment, improving efficiency. A procedure free of side effects if it

- doesn't modify the value of an argument
- doesn't modify the value of a global or Fortran common variable
- doesn't modify the value of local static variables used on subsequent calls
- doesn't perform input or output
- doesn't call another procedure that has side effects

Existing procedure compilers cannot automatically determine whether a side effect exists. The CONVEX Fortran and C compilers eliminate procedure calls only if you explicitly request this behavior with the `NO_SIDE_EFFECTS` directive or pragma.

The form of this Fortran directive is

`C$DIR NO_SIDE_EFFECTS (`*func_list*`)`

and the form of the C pragma is

`#pragma _CNX no_side_effects (`*func_list*`)`

where *func_list* is a list of procedure names separated by commas. The directive must precede the procedure call that does not contain side effects.

Caution

Do not use the NO_SIDE_EFFECTS directive or pragma on a call to a procedure that:

- **Changes the value of an argument**
- **Changes the value of a COMMON or global variable**
- **Performs input or output**
- **Changes the value of a static local variable in C**
- **Changes the value of a SAVED local variable in Fortran**
- **Calls another procedure that performs one of these operations**

For more information about the NO_SIDE_EFFECTS directive, see Appendix A, "Compiler directives."

The CONVEX Application Compiler is capable of recognizing side effects in procedure calls. Refer to the *Application Compiler User's Guide* for more information.

Dead-code elimination

If, as a result of constant propagation and folding, the compiler can fold an arithmetic or logical expression in a Fortran IF statement to .TRUE. or .FALSE., or if the expression can be folded to a constant in C, the compiler eliminates any unreachable code that results.

Copy propagation

The compiler can replace a variable with another variable to which it has been equated. This is called *copy propagation*. For example, after evaluating the statement X=Y, the compiler replaces later occurrences of X with Y.

In the following Fortran example, if the compiler determines that X and Y are unchanged between the assignment and the reference, it replaces X with Y.

```
X = Y
.
.
.
W = Z - X
```

becomes

```
.
.
.
W = Z - Y
```

Common subexpression elimination

The compiler eliminates common subexpressions at the global level. The compiler retains the value of the common subexpression in a register if one is available; otherwise, the compiler assigns the value to a temporary variable. The compiler then replaces subsequent occurrences of the common subexpression with references to the register or temporary variable.

In the following Fortran example, the compiler determines that the subexpression must be calculated whether the condition associated with the `IF` statement evaluates to `.TRUE.` or `.FALSE.`

```
SUBROUTINE GCSE2
.
.
.
IF (K .LT. L) THEN
  A = (C * 4) / -(J * B + SQRT(C))
ELSE
  E = (E * 4) / -(J * B + SQRT(C))
ENDIF
F = (B * 4) / -(J * B + SQRT(C))
.
.
.
END
```

The compiler saves the value of the common subexpression in the temporary variable `T1` and uses the variable to compute the value for assignment to `A`, `E`, and `F`, as follows.

```
SUBROUTINE GCSE2
.
.
.
T1 = -(J * B + SQRT(C))
IF (K .LT. L) THEN
  A = (C * 4) / T1
ELSE
  E = (E * 4) / T1
ENDIF
F = (B * 4) / T1
.
.
.
END
```

The analogous C code follows.

```
void gcse2()
{
    .
    .
    .
    if (k < l)
      a = (c * 4) / -(j * b + sqrt(c));
    else
      e = (e * 4) / -(j * b + sqrt(c));
    f = (b * 4) / -(j * b + sqrt(c));
    .
    .
    .
}
```

As with the Fortran example, the compiler recognizes that the common subexpression is used before and after the `if` statement. It saves the value of the subexpression in the temporary variable `t1` before the `if` statement and uses this variable to compute the values of `a`, `e`, and `f`, as shown in the following example.

```
void gcse2()
{
   ...
   t1 = -(j * b + sqrt(c));
   if (k < 1)
     a = (c * 4) / t1;
   else
     e = (e * 4) / t1;
   f = (b * 4) / t1;
   ...
}
```

Code motion

Code motion is the movement of invariant expressions out of loops. An invariant expression yields the same result on every iteration of a loop.

In the following Fortran example, all variables used in the assignment to A remain invariant within the loop.

```
SUBROUTINE GCM
REAL AR(10)
.
.
.
DO I = 1, 10
  A = C / (-(E * B) + SQRT(C))
  AR(I) = A + B * C
ENDDO
.
.
.
END
```

The compiler recognizes this and moves the calculations and assignments out of the loop, performing these costly calculations only once.

The optimized code follows.

```
SUBROUTINE GCM
REAL AR(10)
.
.
.
A = C / (-(E * B) + SQRT(C))
DO I = 1, 10
  AR(I) = A + B * C
ENDDO
.
.
.
END
```

In C:

```
void gcm() {
  float ar[10]
  .
  .
  .
  for(i=0;i<10;i++) {
    a = c/(-(e*b) + sqrt(c));
    ar[i] = a+b+c;
  }
  .
  .
  .
}
```

After optimization:

```
void gcm() {
  float ar[10]
  .
  .
  .
  a = c/(-(e*b) + sqrt(c));
  for(i=0;i<10;i++)
    ar[i] = a+b+c;
  .
  .
  .
}
```

If an invariant expression does not lie on a path to all loop exits, the compiler does not move the invariant expression unless you use the `-uo` (potentially unsafe optimizations) compiler option.

For more information about using the `-uo` option, refer to Chapter 9, "Potentially unsafe optimizations."

Strength reduction

In some cases, the compiler can replace an arithmetic operation with an equivalent operation (possibly nonarithmetic) that executes more quickly. Such replacements are called strength reductions.

Explicit arithmetic reductions

The compiler can reduce the strength of various arithmetic operations. For example, the compiler transforms integer multiplication on positive numbers by 2, 4, 8, and 16 into integer shifts, as shown in the following Fortran code:

`J * 2` becomes `IISHFT(J, 1)`
`J * 4` becomes `IISHFT(J, 2)`

Multiplication involving integer constants is reduced to addition:

`X * 2` becomes `X + X`

When the `-uo` (potentially unsafe optimizations) command line option is specified, division by a constant is reduced to multiplication, which is substantially faster:

`X/C` becomes `D*X` where `D=1/C`

Because `C` is a constant, `D` also is a constant, which can be computed at compile time.

Induction variables and constants

The compiler can reduce the strength of operations to optimize loop induction variables and loop constants. Multiplications within a loop that calculate the address of a subscripted variable are often candidates for strength reduction.

The compiler does not reduce operations that only involve `REAL`, `float` or `double` variables. Because floating-point arithmetic is imprecise, reduced operations do not always yield equivalent results. If an expression does not lie on a path to all loop exits, the

compiler does not reduce the expression unless you use the `-uo` option.

In the following Fortran example, the compiler recognizes that `I` is incremented by 2 on each iteration and that `X` is incremented by `2*C`, a loop constant.

```
      SUBROUTINE GSR
      I = 1       !induction var
10    X = I * C   !loop induction value
      .
      .
      .
      I = I + 2
      IF(I .LE. 100) GOTO 10
      .
      .
      .
      END
```

The analogous C code follows:

```
void gsr()
{
   int i = 1;       /* induction var */
   ...
   do {
      x = i * c;   /* loop induction value */
      ...
      i += 2;
   } while( i<=100 );
}
```

As shown in the following optimized version, the compiler produces code that calculates `2*C` only once and increments `X` by the value saved in `T2` instead of calculating `I*C` on every iteration.

```
      SUBROUTINE GSR
      I = 1
      T1 = C
      T2 = 2 * C
10    X = T1
      .
      .
      .
      T1 = T1 + T2
      I = I + 2
      IF(I .LE. 100) GOTO 10
      .
      .
      .
      END
```

In C:

```
void gsr()
{
   int i = 1;
   ...
   t1 = c;
   t2 = c + c;
   do {
      x = t1;
      ...
      t1 += t2;
      i += 2;
   } while( i <= 100 );
}
```

Global register allocation

Scalar variables can often be stored in registers, eliminating the need for costly main 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. Consider the following Fortran example:

```
DO I = 1, N
  A(I) = X
  .
  .
  .
ENDDO
```

Here, X is referenced on every iteration of the loop. Eliminating loads and stores of X to main memory can substantially improve performance. Using GRA, the compiler generates code equivalent to that shown below:

```
REG = X
DO I = 1, N
  A(I) = REG
   .
   .
   .
ENDDO
X = REG
```

Where REG represents a register.

The analogous C example follows:

```
for(i=0;i<n;i++) {
  a[i] = x;
   .
   .
   .
}
```

After GRA:

```
REG = x;
for(i=0;i<n;i++){
  a[i] = REG;
   .
   .
   .
}
x = REG;
```

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

GRA can sometimes cause wrong answers in Fortran code that violates ANSI standard argument-passing conventions. The problem arises when a constant is passed into a subroutine which potentially attempts to assign to it. If such an assignment executes, a bus error will occur; however, some codes may conditionally execute the assignment based on whether the argument is a variable or constant in the calling program.

In such cases, GRA may be unaware that the subroutine's dummy arguments may represent constant actual arguments. If the dummy arguments are allocated registers, a bus error will result even if the executable never attempts to assign to the constant, because GRA will always generate code to store the register back to the dummy argument variable at some point.

Consider the following Fortran example:

```
.
.
.
CALL ASSIGNER(1,IMAX,A)
CALL ASSIGNER(0,10,A)
.
.
.
SUBROUTINE ASSIGNER(IVAR,I,A)
INTEGER A(100,100)
DO ICNT = 1,10
  IF (TEST(IVAR) .EQ. 1) THEN
    .
    .
    I = ...
    .
    .
  ELSE
    .
    .   ! NO ASSIGNMENT TO I
    .
  ENDIF
ENDDO
RETURN
END

INTEGER FUNCTION TEST(ITEST)
.
.
.
```

Here, the `IVAR` argument indicates whether the `I` dummy argument is a variable or constant. If it is a variable, it is assigned in the `IF` statement.

GRA, unaware that `I` may be a constant, will allocate a register for it (assuming nothing in the missing code prevents it from doing so). When the register is stored back to `I` at runtime, a runtime error results.

The following Fortran example shows code that is equivalent to code produced by the GRA optimization for the subroutine ASSIGNER.

```
SUBROUTINE ASSIGNER(IVAR,I,A)
INTEGER A(100,100)
REG = I          ! I PLACED IN REGISTER
DO ICNT = 1,10
  IF (TEST(IVAR) .EQ. 1) THEN
    .
    .
    I = ...
    .
    .
  ELSE
    .
    .   ! NO ASSIGNMENT TO I
    .
  ENDIF
ENDDO
I = REG ! REG STORED BACK TO I--IMPOSSIBLE
RETURN  ! IF I IS A CONSTANT
END
```

This situation can be avoided by specifying the `-nga` compiler option. `-nga` disables global register allocation for arguments passed by reference. If the above example is compiled with `-nga`, `I` will never be allocated a register and the code will work as expected.

GRA can also 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 main memory will be updated correctly as each thread executes.

Parallel assignments to the same shared variables from multiple threads only make sense 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 delimited by compiler directives, but for those created using CPSlib (refer to Appendix D), or for conditional assignments based on thread ID, GRA may allocate registers which may cause wrong answers when stored.

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

GRA is enabled by default at optimization levels `-O1` and higher.

-O2 Level optimizations

The primary goal of `-O2` optimizations is *data localization*.Data localization keeps heavily used data in the processor data cache, thus eliminating the need for more costly CTIcache or main 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 level `-O2` to achieve localization. These transformations are explained in detail in this section.

In addition to localization, the compiler performs global instruction scheduling and all basic block and machine-instruction level optimizations.

Note

Most of the subsections that follow present code examples that 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. The compiler actually generates an intermediate language after performing these optimizations.

Why localize?

The benefits of localization are best illustrated through an example.

Consider the following Fortran code:

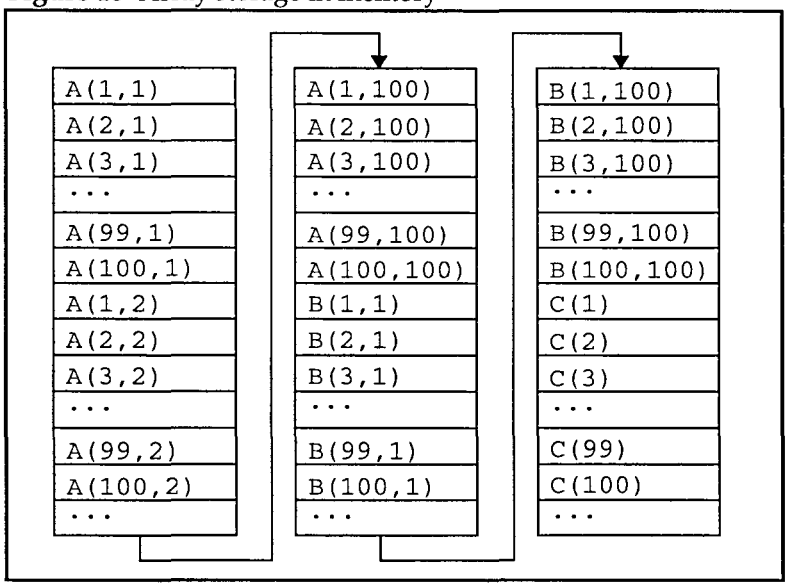
```
REAL*4 A(100, 100), B(100, 100), C(100)
COMMON /BLK1/ A, B, C
...
DO J = 1, 100
  DO I = 1, 100
    A(I, J) = B(I, J) * C(I)
  ENDDO
ENDDO
```

The compiler will recognize that this loop nest is too small to benefit from blocking. As the loop is written, all three arrays can be reused.

First, notice that this loop operates on three REAL*4 arrays, which total less than 80 kbyte in size. All the elements of these arrays will fit easily into the processor data cache, so elements being overwritten will not be a problem.The arrays are stored in COMMON to eliminate the possibility of cache thrashing, as described in Chapter 2.

Now let's examine how these arrays are stored in memory. Fortran arrays are stored in column-major order, meaning that the elements of the first column are stored contiguously, followed by the elements of the second column and so on, as shown in Figure 10. Note that contiguous storage of the arrays in this illustration is due to the COMMON block storage of the arrays.

Figure 10 Array storage in memory



The boxes in Figure 10 represent contiguous virtual addresses; these could map to physical addresses in the CTIcache or in main memory. In any case, assuming the arrays have not been accessed before we reach the J loop and therefore are not in the processor data cache, they must be encached there as the loop executes.

On the first iteration of the loop, A(1,1), B(1,1) and C(1) will be fetched from memory as part of separate 32-byte cache lines. These elements will be positioned at random relative to the cache line boundaries; in other words, there is no way of knowing whether A(1,1) is the first element in its cache line, or if it is positioned elsewhere. While it is likely that each cache line will contain some reusable elements of the array it is fetching from, this is not guaranteed on the first fetch. However, when any present reusable elements are used and the second fetch

from memory takes place, the cache line in question will begin with the element being fetched and also contain the 7 following elements. Thus, after the first element is fetched for each array, fetching `A(I,1)` will encache `A(I:I+7,1)`; fetching `B(I,1)` will encache `B(I:I+7,1)` and fetching `C(I)` will encache `C(I:I+7)`. The `I` loop will then iterate 7 more times without needing to look beyond the processor data cache for data, boosting performance significantly.

Another significant performance boost happens when the `I` loop finishes executing for `J = 1`. At this point, all of `C` has been encached. While each successive column of `A` and `B` will have to be fetched a cache line at a time from memory, `C` will always be available immediately from the processor data cache, saving, in this simple example, 13 out-of-cache memory accesses.

More complicated loops, such as the matrix multiply algorithm, and loops that manipulate much larger arrays can often be transformed by the compiler such that far more data reuse is possible, resulting in greater performance.

The following sections explain the loop transformations that aid data localization.

Strip mining

Strip mining is a fundamental `-O2` transformation that is used by loop blocking 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, i.e. to achieve 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, e.g. 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.

An analogous C example follows.

```
for(i=0;i<10000;i++)
  a[i] = a[i] * b[i];
```

After strip mining with a strip length of 1000

```
for(iout=0;iout<10000;iout+=1000)
  for(istrip=0;istrip<iout +1000;istrip++)
    a[istrip] = a[istrip] * b[istrip];
```

In and of itself, strip mining is not profitable. However, strip mining is essential to the highly profitable loop blocking optimization, which is described later in a following section.

Loop distribution

Loop distribution is another fundamental `-O2` 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
  B(I, 1) = 0
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
  D(I) = E(I) + A(I)
ENDDO
```

Loop distribution creates three copies of the `I` loop, separating the nested `J` loop from the assignments to arrays `B` and `D`. In this way, all three assignments are moved to innermost loops, as shown in the following transformed code.

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

An analogous C example follows.

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

This loop is distributed as shown below.

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

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

Loop interchange

The compiler interchanges 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 (at optimization level `-O3` only).
- 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 below, 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 accesses arrays in row-major order. An analogous example in 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

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 sized to optimally 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 and *temporal* reuse.

Spatial reuse is using data which was encached as a result of fetching another piece of data from memory. Remember that on an SPP1000 Series system, data is fetched by cache lines; 32 bytes of data is encached on every fetch. 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. Starting with the second memory fetch (which will not happen until any usable elements obtained on the first fetch are used), 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 7 following elements are encached on each fetch after the first. If any of these 7 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 may never synch so that all the elements they contain are usable.

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 containing either temporal or spatial reuse are candidates for blocking. Blocking exploits spatial reuse by insuring that once fetched, cache lines are not overwritten until their spatial reuse is exhausted. Temporal reuse is similarly maximized.

Reuse example

The following Fortran loop contains arrays that are candidates for 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 gives spatial reuse on the A and B arrays, and both spatial and temporal reuse on the C array. Spatial reuse is achieved on the A array because every 8th iteration of the I loop (after whatever elements received with first fetch of A are used) 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, must be fetched from memory. Some reuse may be possible if the element occurs before the end of the containing cache line. On subsequent iterations of J, whenever a cache line is fetched from memory, 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. Since 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 fetches do not synch up as nicely as they do for A, spatial reuse is still fully exploited.

Figure 12 on page 60 illustrates spatial reuse for both A and B in the context of the blocking example discussed in the following section.

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.

Note that this loop does not require blocking to achieve this reuse because the arrays only occupy a total of less than 80 kbytes, so they fit easily into the cache.

Blocking example—simple loop

In order to achieve reuse in more realistic examples which manipulate arrays that won't all fit in the cache, blocking strip mines the inner loop. The new innermost loop has an iteration space selected so that all the array elements it references will fit into the cache without overwriting themselves. The new loop nest is then interchanged such that the innermost loop is "blocked" from overwriting its strips.

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, and 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
```

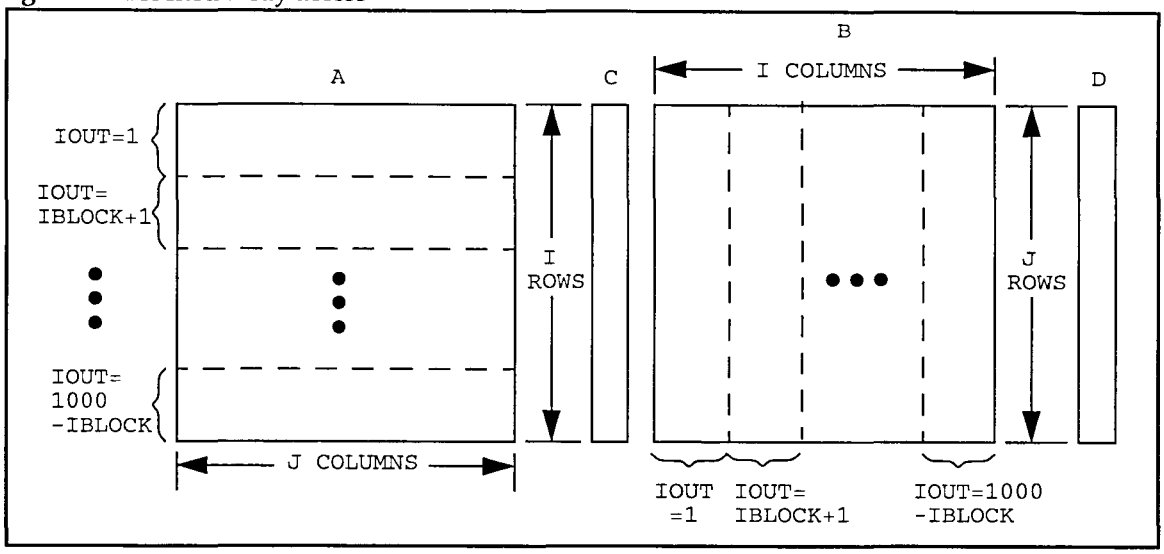
`IBLOCK` is the block factor (it's also the strip mine length) the compiler chooses based on the size of the arrays and size of the cache. Note that this example assumes the chosen `IBLOCK` divides 1000 evenly.

Now the compiler interchanges the `J` and `IOUT` loops to block the `I` loop.

```
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
```

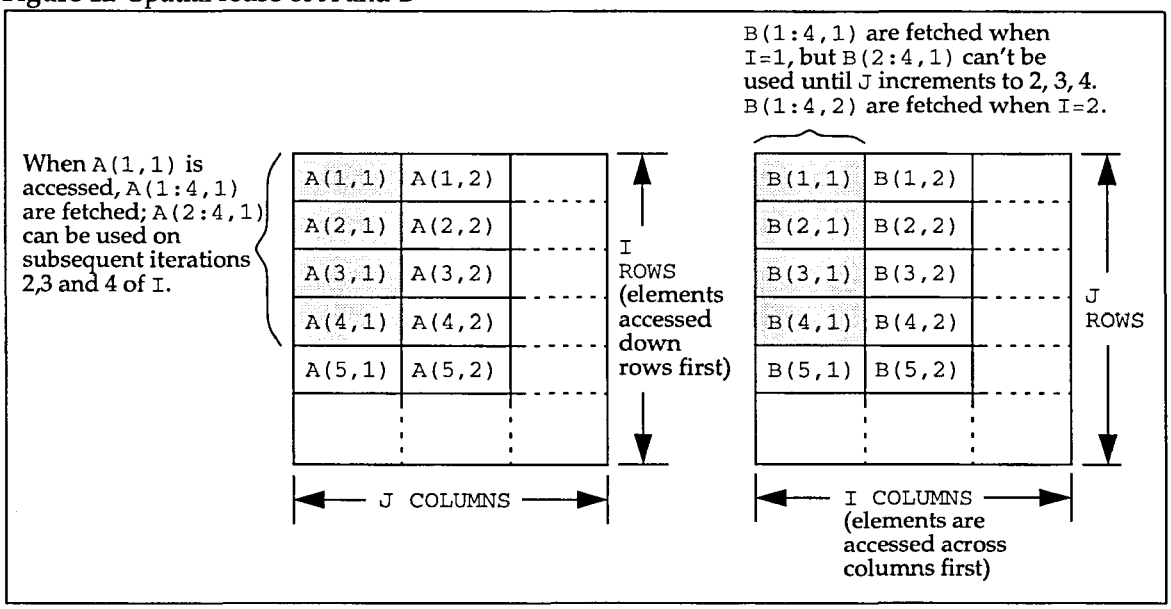
This new nest now accesses `IBLOCK` rows of `A` and `IBLOCK` columns of `B` for every iteration of `J`. At every iteration of `IOUT`, the nest accesses 1000 `IBLOCK`-length columns of `A`, or an `IBLOCK` × 1000 chunk of `A`, and 1000 `IBLOCK`-width rows of `B` are accessed. This is illustrated in Figure 11.

Figure 11 Blocked array access



Fetches of A occurring after the first fetch encache the needed element and the 3 elements that are used in the 3 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 to maximize spatial reuse of both A and B. Figure 12 illustrates how cache lines of each array are fetched, assuming that A and B both start on cache line boundaries. The shaded area represents the initial cache line fetched.

Figure 12 Spatial reuse of A and B



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; and temporal reuse of C with respect to the K loop.

An analogous C example follows.

```
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];
```

CONVEX C interchanges and blocks this example to provide optimal access efficiency for the row-major 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's different array storage. This code yields spatial reuse of `b` with respect to the `j` 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; and 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, the loop may still be blocked, but those loops will not be interchanged with each other. They may, however, be interchanged with other loops in the nest that are candidates for interchange, in which case blocking can proceed to some extent.

Blocking directives, pragmas and options

Loop blocking can be disabled for specific loops using the `NO_BLOCK_LOOP` compiler directive and pragma. You can advise the compiler to use a specific block factor via the `BLOCK_LOOP` directive and pragma. In Fortran, these directives have the following form:

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

In C, these directives have the following 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 an integer constant or constant expression. 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 absence of the block_factor argument, this directive is useful for indicating which loop in a nest to block. In this case, the compiler will pick an appropriate block factor.

These directives 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),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 × 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.

To disable blocking for all loops, use the -noblock compiler option. To specify a blocking factor for all loops, use the -blockloop *n* compiler option, where *n* is the blocking factor to be used for all loops that the compiler blocks.

Loop unrolling

Loop unrolling involves increasing a loop's step value and replicating the loop body, with each replication 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, replicating the loop body a number of times equal to the iteration count, and replacing the iteration variable with constants. This only makes sense for loops with small iteration counts. Consider the following Fortran example.

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

This loop is completely unrolled as shown in the following example.

```
A(1) = B(1) + C(1)
A(2) = B(2) + C(2)
A(3) = B(3) + C(3)
A(4) = B(4) + C(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 *unrolling depth* (also known as 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 4 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 4 values of `A` instead of 1 value.

An analogous C example follows.

```
for(i=0;i<100;i++)
  a[i] = b[i] + c[i];
```

This can be unrolled to a depth of 4 as shown in the following code.

```
for(i=0;i<100;i+=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];
}
```

Loop unrolling improves efficiency by eliminating loop overhead; in totally unrolled loops, the overhead is completely eliminated, and in partially unrolled loops the number of tests and induction variable increments are reduced. Unrolling can also create opportunities for other optimizations, such as improved register use and more efficient scheduling.

CONVEX compilers perform loop unrolling by default at optimization levels `-O2` and higher. The compiler completely unrolls loops with iteration counts determinable at compile time to be less than 5. Loops with undeterminable iteration counts or determinable counts of 5 or more are partially unrolled.

Unrolling is enabled by default and through use of the `-ur` compiler option. It can be disabled for all loops by specifying the `-nur` compiler option. You can specify an unroll factor by using the `-urn` *n* option, where *n* is the unroll factor for all unrolled loops.

Unrolling can also be specified for individual loops using the `unroll` directive and pragma. In Fortran, this directive has the following form:

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

The C pragma has the following form:

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

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

Note

Unrolling is only performed on innermost loops. If you use the directive or pragma on a loop nest, you must specify it on the loop which ends up, after any interchanges performed by the compiler, to be innermost.

To insure that the directive or pragma is on the innermost loop, you should compile the nest in question and determine from the optimization report (which is discussed in Appendix C) which

loop is innermost after compilation, then place the directive or pragma on that loop in your source. Placing the directive or pragma on a loop that is not innermost after compilation will have no effect.

Loop unroll and jam

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.

Blocked loops are often prime candidates for unroll and jam because their deep nesting provides many candidates for temporal reuse with respect to the non-innermost loops.

Consider our blocked matrix multiply loop:

```
DO IOUT = 1, N, ISTRIP
  DO KOUT = 1, N, KSTRIP
    DO J = 1, N
      DO I = IOUT, IOUT+ISTRIP-1
        DO K = KOUT, KOUT+KSTRIP-1,
          C(I,J) = C(I,J) + A(I,K)*B(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

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

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

```
DO IOUT = 1, N, ISTRIP
 DO KOUT = 1, N, KSTRIP
  DO J = 1, N, 2
    DO I = IOUT, IOUT+ISTRIP-1, 2
      DO K = KOUT, KOUT+KSTRIP-1
        C(I,J) = C(I,J)+A(I,K)*B(K,J)
      ENDDO
      DO K = KOUT, KOUT+KSTRIP-1
        C(I+1,J) = C(I+1,J)+A(I+1,K)*B(K,J)
      ENDDO
    ENDDO
    DO I = IOUT, IOUT+ISTRIP-1, 2
      DO K = KOUT, KOUT+KSTRIP-1
        C(I,J+1) = C(I,J+1)+A(I,K)*B(K,J+1)
      ENDDO
      DO K = KOUT, KOUT+KSTRIP-1
        C(I+1,J+1)=C(I+1,J+1)+A(I+1,K)*B(K,J+1)
      ENDDO
    ENDDO
  ENDDO
 ENDDO
ENDDO
```

The "jam" part of unroll and jam occurs when the loops are fused back together, to create the following.

```
DO IOUT = 1, N, ISTRIP
 DO KOUT = 1, N, KSTRIP
  DO J = 1, N, 2
    DO I = IOUT, IOUT+ISTRIP-1, 2
      DO K = KOUT, KOUT+KSTRIP-1
       C(I,J) = C(I,J)+A(I,K)*B(K,J)
       C(I+1,J) = C(I+1,J)+A(I+1,K)*B(K,J)
       C(I,J+1) = C(I,J+1)+A(I,K)*B(K,J+1)
       C(I+1,J+1)= C(I+1,J+1)+A(I+1,K)*B(K,J+1)
      ENDDO
    ENDDO
  ENDDO
 ENDDO
ENDDO
```

This new loop exploits more registers and requires fewer loads and stores than the original. Recall that the original blocked loop could use no more than 3 registers. This unrolled-and-jammed loop can use 8, one for each of the following references: `C(I,J)`, `A(I,K)`, `B(K,J)`, `C(I+1,J)`, `A(I+1,K)`, `C(I,J+1)`, `B(K,J+1)`, `C(I+1,J+1)`. 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 `FMPYADD` instruction, which doubles the speed of the operations in the body of the loop by simultaneously performing unrelated adds and multiplies.

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.

Unroll and jam is enabled by default and through use of the `-uj` option at optimization levels `-O2` and higher. You can disable unroll and jam by specifying the `-nuj` option. You can specify an unrolling factor by using the `-ujn` *n* option, where *n* is the desired unrolling factor for all loops which the compiler unrolls and jams.

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
```

All of these directives and pragmas apply to the immediately following loop nest. Since unroll and jam is only performed on nested loops, *you must insure that the directive or pragma is specified on a loop that, after any compiler-initiated interchanges, ends up being non-innermost.* You can determine which loop in a nest will be innermost by compiling the nest without any directives and examining the optimization report.

IF-DO and if-for optimizations

These optimizations modify loops containing tests to improve performance. Tests can be promoted out of the loops or eliminated completely. By minimizing the number of tests within a loop, the compiler reduces the number of branches that must be executed, thereby improving performance.

In Fortran, these optimizations are performed on `DO` loops containing `IF` statements; in C, they are performed on `for` loops containing `if` statements. These optimizations fall into one of three categories: redundant test elimination, loop peeling and test promotion. Each of these is described in detail below.

Redundant-test elimination

Redundant-test elimination is the simplest of these optimizations. The compiler recognizes when a test against some index variable is evaluated more than once and eliminates that test as well as any accompanying redundant code.

This optimization is especially relevant when you are optimizing FORTRAN 66 programs that contain `DO` loops surrounded by `IF` tests, as shown in the following example.

```
DO I = 1, N
  IF (I .GT. 0) THEN
    DO J = 1, I
      A(I,J) = 0
    ENDDO
  ENDIF
ENDDO
```

With the test removed, the loop looks like this:

```
DO I = 1, N
  DO J = 1, I
    A(I,J) = 0
  ENDDO
ENDDO
```

Here the explicit test `IF (I .GT. 0)` is redundant, since the test is implicit in the `DO` loop. It is therefore removed during redundant-test elimination.

Similarly, the test in the following C code always succeeds:

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

The `if` test is redundant because of the loop control, which initializes `i` to 0 and increments it by one each iteration.The compiler recognizes that the condition always occurs and removes the test:

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

Redundant-test elimination is always performed at optimization levels `-O2` and above.

Loop boundary-value peeling

Loop boundary-value peeling involves removing the first iteration, last iteration, or first and last iterations of a loop to remove conditional tests from the loop. This is done when the loop contains a test or tests involving an explicit reference to the loop index variable that, when evaluated, either always causes or always prevents execution of the first iteration, last iteration, or both. Last-iteration peeling is also sometimes necessary when the compiler automatically privatizes a loop variable which is referenced after the loop. In this case, peeling is used to assign the last-iteration value to the private variable so that it can be later referenced.

Given the Fortran code shown below, the compiler automatically peels off the first and last tests and rewrites the loop to cover the remaining indexes.

```
DO I = 1, 100
  IF (I .EQ. 1) THEN
    A(I) = B(I)
  ELSE IF (I .EQ. 100) THEN
    A(I) = C(I)
  ELSE
    A(I) = -A(I)
  ENDIF
ENDDO
```

After peeling, the code looks like this:

```
A(1) = B(1)
DO I = 2, 99
  A(I) = -A(I)
ENDDO
A(100) = C(100)
```

The analogous C code follows.

```
for (i=0; i<100; i++)
  if (i==0)
    a[i]=b[i];
  else
  if (i==99)
    a[i]=c[i];
  else
    a[i]= -a[i];
```

After peeling:

```
a[0]= b[0];
for (i=1; i<99; i++)
  a[i]=-a[i];
a[99]=c[99];
```

In some cases, boundary-value peeling requires replicating large amounts of code, and this can greatly increase the size of the executable file. By default, the compiler peels boundary values and expands the code up to a predetermined conservative limit; you can increase this limit by using the `-peel` compiler option or, if you wish to do so on a loop-by-loop basis, the `PEEL` Fortran compiler directive or `peel` C pragma.

You can allow the compiler to expand code without bound by using the `-peelall` compiler option, the `PEEL_ALL` Fortran directive, or the `peel_all` C pragma. In codes containing large numbers of boundary-value operations, allowing code expansion without bound can greatly lengthen compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

Boundary-value peeling can be disabled completely with the `-nopeel` compiler option. Similarly, you can disable peeling on a loop-by-loop basis with the `NO_PEEL` Fortran compiler directive or the `no_peel` C pragma. See Appendix A, "Compiler directives and pragmas," for more information.

Note

Loop boundary-value peeling is not performed on loops that have no tests on boundary values. In other words, the compiler does not try to peel unpeelable loops.

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, N
  IF (FOO .EQ. BAR) THEN
    A(I) = B(I)
  ELSE
    A(I) = 0
  ENDIF
ENDDO
```

Test promotion produces the following code.

```
IF (FOO .EQ. BAR) THEN
  DO I = 1, N
    A(I) = B(I)
  ENDDO
ELSE
  DO I = 1, N
    A(I) = 0
  ENDDO
ENDIF
```

In C:

```
for(i=0; i<n; i++)
  if (foo==bar)
    a[i]=b[i];
  else
    a[i]=0;
```

After test promotion:

```
if (foo==bar)
  for(i=0; i<n; i++)
    a[i]=b[i];
else
  for (i=0; i<n; i++)
    a[i]=0;
```

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

You can control the amount of code replication and test promotion with compiler options and directives. By default, the compiler promotes tests and replicates code up to a predetermined conservative limit.

The `-ptst` compiler option increases this limit and can cause a noticeable increase in compile time.

The `-ptstall` option promotes all tests regardless of code replication. This can cause a large increase in compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

The `-noptst` option disables test promotion.

The `PROMOTE_TEST`, `PROMOTE_TEST_ALL` and `NO_PROMOTE_TEST` Fortran compiler directives and `promote_test`, `promote_test_all` and `no_promote_test` C pragmas provide similar functionality on a loop-by-loop basis. See Appendix A, "Compiler directives and pragmas," for more information about these directives.

At optimization levels `-O2` and above, the CONVEX compilers automatically perform these optimizations on Fortran `DO` and hand-rolled loops, and on `for`, `while`, `do-while` and hand-rolled loops in C. Simple and nested loops, and loops with exits, are handled.

Scalar replacement

Scalar replacement involves storing loop-invariant array elements in scalars, which can then be stored in registers by global register allocation. This substantially increases the performance of the loop by eliminating the necessity of accessing the array element in main memory or the cache.

Consider the following Fortran loop:

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

Here, `A(I)` is invariant with respect to the `J` loop. The compiler can therefore replace `A(I)` with a scalar before entering the `J` loop, and store the scalar back to `A(I)` between iterations of `I`. Global register allocation can then assign this scalar to a register for the duration of the iteration, further improving efficiency.

Employing scalar replacement with global register allocation, the compiler generates code equivalent to the following Fortran:

```
DO I = 1, N
  REG = A(I)    ! PLACE A(I) IN REGISTER
  DO J = 1, M
    REG = REG + B(J)
  ENDDO
  A(I) = REG    ! STORE REGISTER BACK TO A(I)
ENDDO
```

Here, the variable "REG" represents a register.

An analogous C example follows:

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

After scalar replacement:

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

Scalar replacement increases performance for this kind of loop by about 50%.

The compiler will not attempt scalar replacement if the loop-invariant array element is aliased or if the array element is contained in conditional code.

At optimization level -O2 and above, scalar replacement is enabled by default and through use of the -sr compiler option; it can be disabled by the -nsr compiler option.

Inhibitors of localization

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

- Loop-carried dependencies
- Aliased scalar or array variables
- Computed or assigned GOTO statements in Fortran
- Multiple loop entries or exits
- RETURN or STOP statements in Fortran
- Procedure calls
- I/O statements

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

Loop-carried dependencies

A loop-carried dependency (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 which are offset in opposite directions. The Fortran loop below contains an interchange-inhibiting LCD.

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

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) to compute A(I,J). Table 2 shows the sequence in which values of A are computed for this loop.

Table 2 Computation sequence of A(I,J)—original loop

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

As enumerated in Table 2, 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 dependency must be honored for the loop to yield its intended results. If a row element of A is computed before the previous row element is computed, the result will be incorrect.

Interchanging the I and J loops yields the following code.

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

After interchange, the loop computes values of A in the sequence shown in Table 3.

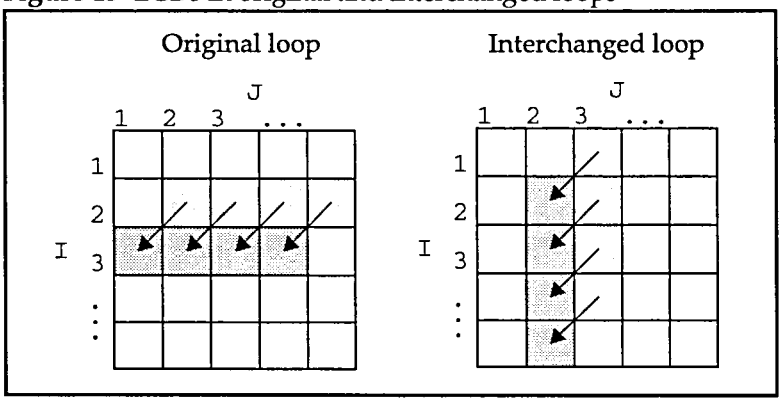
Table 3 Computation sequence of `A(I,J)`—interchanged loop

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

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

The problem here is that columns of A are being computed using elements that have not been written yet, which violates the dependency illustrated in Table 2. The element-to-element dependencies in both the original and interchanged loop are illustrated in Figure 13.

Figure 13 LCDs in original and interchanged loops



The arrows in Figure 13 represent dependencies from one element to another; the arrows point at elements which depend on the elements at the arrows' base. Shaded elements indicate a typical row or column computed in the inner loop; lightly shaded elements have not been computed yet, and darkly shaded elements have already 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 dependencies which change when the loop is interchanged. Most LCDs do not fall into this category and thus do not inhibit data localization.

Occasionally the compiler will encounter an apparent LCD. In this case, if it cannot determine whether the LCD actually inhibits interchange, it will conservatively avoid 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 dependency. 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 dependencies using the `NO_LOOP_DEPENDENCE` compiler directive or pragma. In Fortran, this directive has the following 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-delimited list of variables and/or arrays that have no dependencies for the immediately following loop.

The previous Fortran loop can be interchanged when the NO_LOOP_DEPENDENCE directive is specified for A and B 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 either IADD or JADD (but not both) acquire negative values at runtime, these loops may result in incorrect answers.

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 through use of the EQUIVALENCE statement, and in C through 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 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; since 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 equivalent of this loop follows. Keep in mind that C stores 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];
  }
}
```

C has nothing analogous to Fortran's EQUIVALENCE statement, but arrays can be effectively equivalenced through the use of pointers, as shown.

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

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

Caution

The following code 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 code shows the same argument-passing problem in C (this code is legal ANSI 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, and are often created in C using the `break` statement, and in Fortran using the `GOTO` statement.

Consider the following 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. 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.

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. Since an out-of-loop destination would be a loop exit, these statements often prevent loop interchange and therefore data localization.

Procedure calls

The standard CONVEX compilers are unaware of the side effects of most procedures, and therefore cannot determine whether they might interfere with loop interchange. These side effects may include data dependencies involving loop arrays, aliasing (as described in the "Aliasing" section), 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 14.

Figure 14 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 loops exhibit the same limitations. A C example that produces the data patterns shown in Figure 14 follows.

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

Preventing data localization

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

```
C$DIR SCALAR
```

In C it has the following form:

```
#pragma _CNX scalar
```

-O3 Level optimizations

The primary goal of -O3 optimizations is *parallelization*. Parallelization divides a program into threads. A *thread* is a sequence of instructions that must execute on a single CPU.

The CONVEX SPP1000 Series 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 subcomplex on which your program is running. You can specify a smaller number of processors via the mpa utility. Refer to the mpa(1) man page for more information. You can also specify a smaller number of processors using the LOOP_PARALLEL(MAX_THREADS=*m*) or PREFER_PARALLEL(MAX_THREADS=*m*) compiler directives and pragmas, as described in chapters 4 and 6.

Automatic parallelization is only useful when your program is written using the shared memory paradigm or a combination of message passing and shared memory. If you are writing your program entirely under the message passing paradigm, you should explicitly handle parallelism yourself as discussed in Chapter 7. Such pure message passing programs should be compiled at optimization level -O2 or lower.

Basic operation

Parallelism can exist at both the loop level and the task level. CONVEX SPP1000 Series compilers automatically exploit

loop-level parallelism. You can easily identify task-level parallelism using the BEGIN_TASKS, NEXT_TASK and END_TASKS directives, as discussed in the "Parallelizing code outside of loops" section later in this chapter. These directives allow the compiler to run identified sections of code in parallel.

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

Note

Only loops with iteration counts determinable 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.

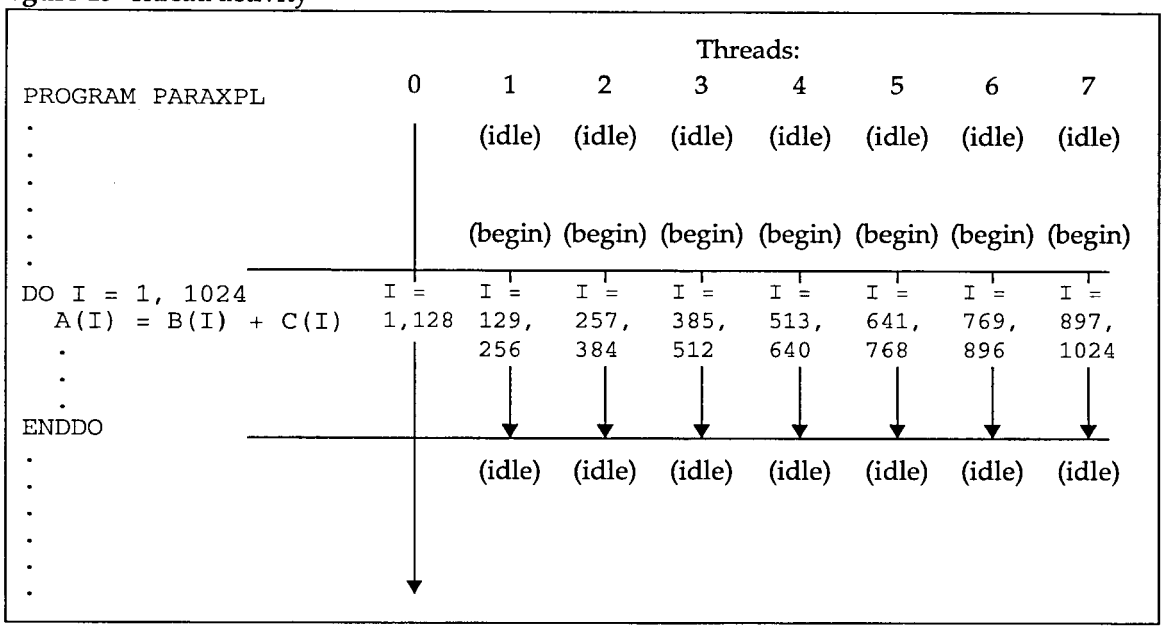
Consider the following Fortran code.

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

This code 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. The compiler generates code that will run on as many processors as are available. If the number of available processors does not evenly divide the number of iterations, some processors will perform fewer iterations than others.

On Exemplar, shared memory programs run as a collection of threads on multiple processors. When a program starts, a separate execution thread is started on each of the processors of the subcomplex on which the program is running. 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. When thread 0 encounters a parallel loop or task, it signals the other threads to begin execution. The threads then become active, run until their portion of the parallel code is finished, and go idle once again, as shown in Figure 15. If other processes are executing on the other processors when the threads become active, SPP-UX insures that all threads execute to completion before thread 0 continues.

Figure 15 Thread activity



To actually implement this, the compiler transforms the loop in a manner similar to strip mining. However, unlike the strip mining described in the `-O2` optimizations section, the outer loop is conceptual; since the strips will be executing on different CPUs, 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 processors are available and which processor 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
```

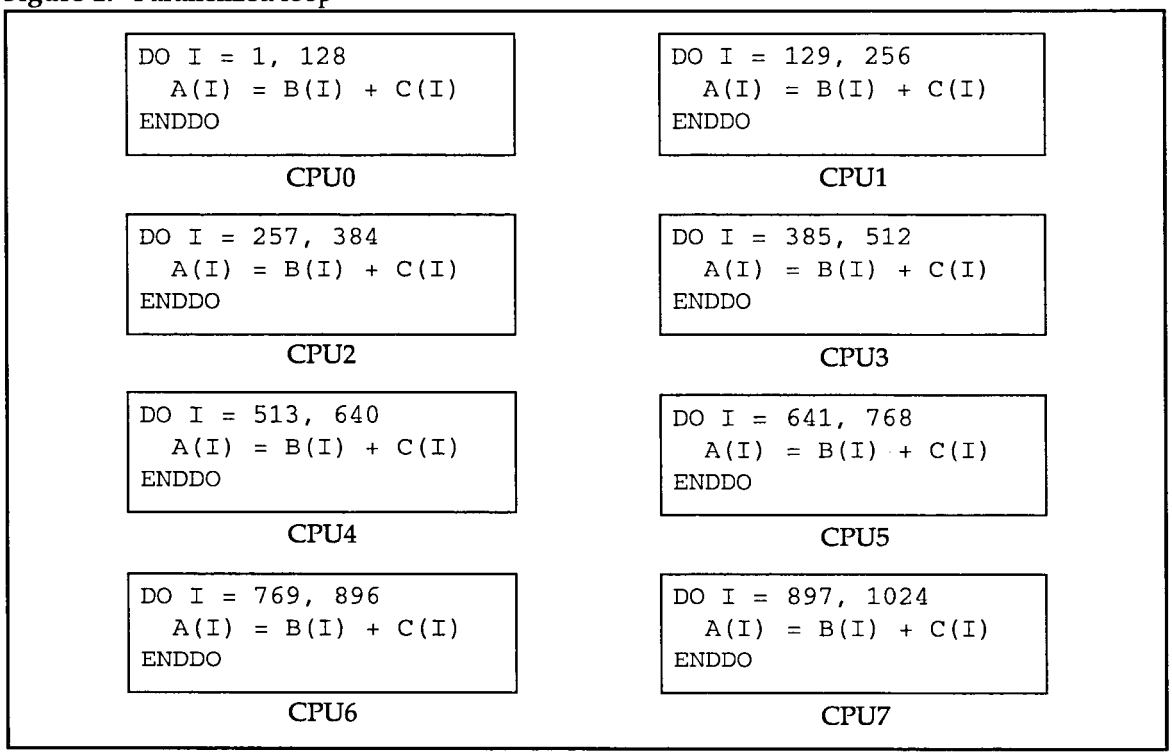
The code shown in Figure 16 is a conceptual representation of the transformation the compiler performs on this example when it is compiled for parallelization, assuming that `N` ≥ `NumProcs`. For `N` < `NumProcs`, `N` is used. If `NumProcs` is not an integral divisor of `N`, some processors will do fewer iterations than others.

Figure 16 Conceptual strip mine for parallelization

```
For each available processor do:
  DO I = ThreadID * (N/NumProcs) + 1, N/NumProcs + ThreadID * (N/NumProcs)
    A(I) = B(I) + C(I)
  ENDDO
ENDDO
```

NumProcs is the number of available processors. ThreadID is the ID number of the thread this particular loop will run in, which is between 0 and NumProcs-1. A unique ThreadID is assigned to each thread, and the ThreadIDs are consecutive. So, for NumProcs = 8, as in Figure 15, 8 loops would be spawned, with ThreadIDs = 0 through 7. These 8 loops are illustrated in Figure 17.

Figure 17 Parallelized loop



Note

The strip-based parallelism described here is the default. Stride-based parallelism is possible through use of the LOOP_PARALLEL compiler directive and pragma, which is described in Chapter 4.

Fortran 90 constructs

Fortran 90 array expressions are translated into loops by the compiler, and parallelism in these loops will be automatically exploited. For example, the following Fortran 90 statement:

```
X(1:M:2, 1:N) = Y(2:M+1:2,2:N+1)
```

is translated by the compiler into a loop similar to the following:

```
DO J = 1, M, 2
  DO I = 1, N
    X(J,I) = Y(J+1,I+1)
  ENDDO
ENDDO
```

which can then be automatically parallelized.

Masked array assignments are similarly parallelized. Consider the following WHERE statement.

```
REAL DATA(1000), LIMIT
LOGICAL NORMAL(1000)
.
.
.
WHERE(DATA .LE. LIMIT) NORMAL = .TRUE.
```

The compiler translates the WHERE statement into a loop similar to the following:

```
DO I = 1, 1000
  IF(DATA(I) .LE. LIMIT) NORMAL(I) = .TRUE.
ENDDO
```

which can then be automatically parallelized.

Parallel optimizations

At optimization level -O3, parallelism is, in effect, *the* optimization. All -O2 level optimizations are also performed, but parallelization is the only optimization added at -O3.

Simple loops can be parallelized without the need for extensive -O2 transformations, as shown in the "Basic operation" section. 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 such 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 insuring that each processor is working on non-overlapping array data.

Inhibitors of parallelization

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

- Loop-carried dependencies
- 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 dependencies can inhibit parallelization than data localization as described in the following sections.

Loop-carried dependencies

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

LCDs fall into three categories: *forward* LCDs, *backward* LCDs and *output* LCDs. The LCD that inhibits localization is a combination of these. All 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). Since the calculation depends on A(I+1) being uncomputed, this would produce wrong answers.

An analogous 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 loop follows:

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

Output LCDs

An 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 might 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 loop follows:

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

Apparent LCDs
As at optimization level `-O3`, 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 "`-O2` Level optimizations" section.

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];
```

Reduction

In many cases, the compiler can recognize and parallelize a special class of dependency 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., .EQV., or .NEQV.

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.

Preventing parallelization

Parallelization can be disabled on a loop basis by using the NO_PARALLEL directive or pragma. The Fortran directive has the following form:

```
C$DIR NO_PARALLEL
```

The C pragma has the following 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];
```

Other parallelization directives

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 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 will 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 dependencies.
- BEGIN_TASKS, NEXT_TASK and END_TASKS—allow you to parallelize regions of code outside of loops. Accepts attributes for node- and thread-parallelism, ordered execution, and maximum number of threads.
- CRITICAL_SECTION, 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, END_ORDERED_SECTION—allow you to isolate dependencies within a loop so that code contained within the ordered section executes in iteration order. Only useful when used with the `loop_parallel(ordered)` or `prefer_parallel(ordered)` directives or pragmas.

These directives and pragmas are discussed in detail in chapters 4 and 6.

Basic shared memory programming

4

This chapter discusses programming techniques that allow you to increase code efficiency with minimal effort.

Loop- and task-specific data privatization

Once assigned, the memory classes mentioned in Chapter 1 and discussed in detail in Chapter 5 are in effect throughout your entire program. Any loops that manipulate variables which 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 CONVEX SPP1000 Series C and Fortran compilers support the `loop_private` and `task_private` directives and pragmas. The `save_last` directive and pragma is provided to save the last value of data objects privatized with `loop_private` if necessary. These directives and pragmas allow you to easily privatize loop or parallel task data temporarily, 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 these 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`) as these classes inhibit automatic parallelization.

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 dependencies in the loops in which they are used. If dependencies exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6.

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 following form:

```
#pragma _CNX loop_private(namelist)
```

where *namelist* is a comma-delimited list of variables and/or arrays that are to be private to the immediately following loop. *namelist* cannot contain structures or dynamic, allocatable, or automatic arrays.

Consider the following Fortran example.

```
C$DIR LOOP_PRIVATE(S)
      DO I = 1, N
C       S IS ONLY PRIVATE IF AT LEAST
C       ONE IF TEST PASSES:
        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 backward-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 to the next 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.

Note that if on any iteration none of the IF tests pass, an actual backward-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];
}
```

The primary induction variable of a parallel loop should not be privatized using loop_private; secondary induction variables, however, should be. Both cases are discussed further in the "Denoting induction variables in parallel loops" section on page 110.

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 dependencies between the tasks in which they are used. If dependencies exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6.

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 following form:

```
#pragma _CNX task_private(namelist)
```

where *namelist* is a comma-delimited list of variables and/or arrays that are to be private to the immediately following tasks. *namelist* cannot contain dynamic, allocatable, or automatic arrays. Do not specify loop induction variables such as Fortran `DO` loop or C `for` loop indices in *namelist*.

Consider the following Fortran example.

```
      REAL*8 A(1000), B(1000), WRK(1000)
      .
      .
      .
C$DIR BEGIN_TASKS(NODES), TASK_PRIVATE(WRK)
      DO I = 1, N
        WRK(I) = A(I)
      ENDDO
      DO I = 1, N
        A(I) = WRK(N-I)
        .
        .
        .
      ENDDO
C$DIR NEXT_TASK
      DO J = 1, M
        WRK(J) = B(J)
      ENDDO
      DO J = 1, M
        B(J) = WRK(M-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. Note that these tasks run node-parallel, and that the compiler will automatically thread-parallelize the loops they contain.

An analogous C example follows:

```
float a[1000], b[1000], wrk[1000];
.
.
.
#pragma _CNX task_private(wrk)
#pragma _CNX begin_tasks(nodes)
for(i=0;i<n;i++)
  wrk[i] = a[i];
for(i=0;i<n;i++) {
  a[i] = wrk[n-1];
  .
  .
  .
}
#pragma _CNX next_task
for(j=0;j<m;j++)
  wrk[j] = b[j];
for(j=0;j<m;j++) {
  b[j] = wrk[m-j];
  .
  .
  .
}
#pragma _CNX end_tasks
```

save_last

The save_last directive and pragma allow you to save the final value of all loop_private data objects assigned in the last iteration of the immediately following loop. The values must be assigned in the last iteration; if the assignment is executed conditionally, it is the programmer's responsibility to insure that the condition is met and the assignment executes. Incorrect answers can result if the assignment does not execute. 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 following form.

```
C$DIR SAVE_LAST
```

In C, the pragma has the following form:

```
#pragma _CNX save_last
```

save_last must appear immediately before or after the associated loop_private directive or pragma, or on the same line.

Consider the following Fortran example.

```
C$DIR LOOP_PRIVATE(ATEMP), SAVE_LAST
      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
      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, the programmer must be sure that one of the conditions is met so that ATEMP is assigned on every iteration. When the loop terminates, the SAVE_LAST directive insures that ATEMP contains the value it is assigned on the last iteration, which is used later in the program.

An analogous C example follows:

```
#pragma _CNX loop_private(atemp), save_last
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;
}
.
.
.
if(atemp > amax) {
.
.
.
```

A `save_last` directive or pragma causes the compiler to peel the last iteration of the loop to facilitate saving. In this way, all but the last iteration of the loop can be parallelized; the final iteration runs serially, and carries out the final assignment to the saved variable.

The optimization report describes such peeled variables in the loop table, the test table, and the privatization table. Refer to Appendix C for more information on the optimization report.

Simple manual loop and task parallelization

The CONVEX SPP1000 Series compilers will automatically exploit strip-based loop parallelism in loops that are clearly dependency-free, as described in Chapter 3, "Compiler optimizations". The `prefer_parallel` and `loop_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 mentioned in Chapter 3 and discussed here allow you to specify regions of code that can be parallelized.

The subsections that follow discuss specifying simple, unordered loop and task parallelism using the `prefer_parallel`, `loop_parallel` and tasking directives and pragmas. Critical sections that do not rely on ordered execution are also covered here.

For a detailed discussion of ordered parallelism, parallel synchronization, and the effective use of memory classes, refer to chapters 5 and 6.

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 implemented in this chapter's examples using the `loop_private` directive discussed on page 96. Manual data privatization using memory classes is discussed in chapters 5 and 6.

Note

These directives should only be used on Fortran `DO` and C `for` loops that have iteration counts determinable prior to loop invocation at runtime.

`loop_parallel` and `prefer_parallel` generally take the same attributes. 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 following form:

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

and

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

where the optional *attribute-list* can contain one of the following case-insensitive attributes:

- `threads`—causes thread-parallelism.
- `nodes`—causes node-parallelism.
- `chunk_size = n`—divides the loop into chunks of *n* or fewer iterations, and distributes the chunks round-robin to the processors.
- `max_threads = m`—allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer which has a value at compile time.
- `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.
- `ordered, nodes`—causes ordered invocation of each iteration across hypernodes.
- `ordered, threads`—causes ordered invocation of each iteration across threads.
- `nodes, chunk_size = n`—node-parallelism by chunks.
- `threads, chunk_size = m`—thread-parallelism by chunks.
- `chunk_size = n, max_threads = m`—chunk parallelism on no more than *m* threads.

- `ordered, max_threads = m`—ordered parallelism on no more than *m* threads.
- `nodes, max_threads = m`—node-parallelize on no more than *m* hypernodes.
- `threads, max_threads = m`—thread-parallelism on no more than *m* threads.
- `ordered, nodes, max_threads = m`—ordered node-parallelism on no more than *m* hypernodes.
- `ordered, threads, max_threads = m`—ordered thread-parallelism on no more than *m* threads.
- `nodes, chunk_size = n, max_threads = m`—node-parallelize by chunks of size *n* on no more than *m* hypernodes.
- `threads, chunk_size = n, max_threads = m`—thread-parallelize by chunks of size *n* on no more than *m* threads.
- `ivar = indvar`—specifies primary loop induction variable; optional in Fortran but required in C.

Note

The values of *n* and *m* must be determinable at compile time for all of the above attributes in which they appear.

Only the listed combinations of attributes are allowed; however, 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 the pragma will be ignored. `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.

The optional `threads` attribute causes parallelization across threads; this is the default. If this 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 processors of each parallel hypernode.

The optional `nodes` attribute causes parallelization across hypernodes. 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.

The optional `chunk_size` 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 to each available thread. If combined with the `nodes` attribute, the chunks are distributed to one thread per available hypernode. 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 an integer constant or constant expression determinable at compile time.

This differs from the default strip-based parallelism described in Chapter 3, which 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` attribute allows each thread to do several noncontiguous chunks. Specifying `chunk_size` = (*number of iterations/number of threads*) is equivalent to default strip mining for parallelization.

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 18 uses Fortran 90 array syntax to illustrate the iterations performed by each thread, assuming 8 available threads.

Figure 18 Stride-parallelized loop

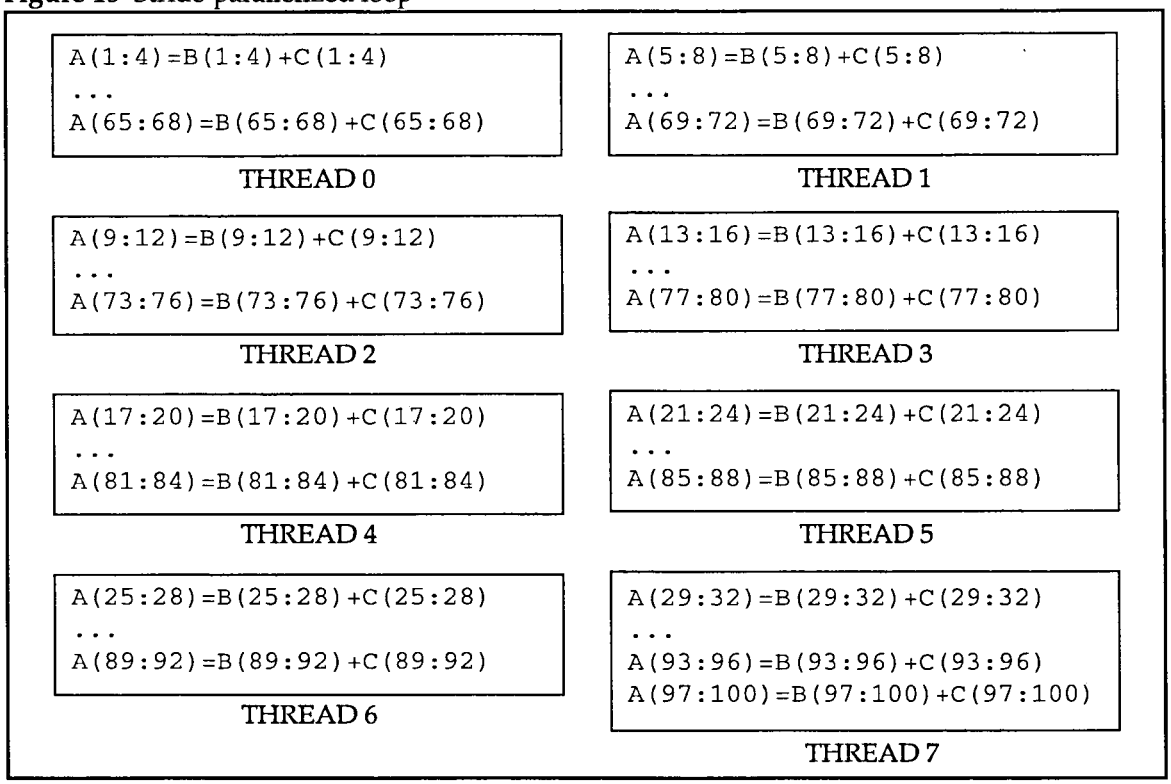


Figure 18 shows that the 100 iterations of `I` are parcelled out in chunks of 4 iterations to each of the 8 available threads; after the chunks are distributed evenly to all threads, there is one chunk left over (iterations 97:100), which executes on thread 7.

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 attribute is most useful on loops in which the amount of work increases or decreases as a function of the iteration count. 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 J, 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] = ...
    .
    .
    .
  }
```

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 is useful when you know the maximum number of threads your loop will run on efficiently.

The ordered directive causes the iterations of the loop to be initiated in loop order across the processors. It is only useful in loops with manually-synchronized dependencies, so it is only useful with the loop_parallel directive. To achieve ordered parallelism, dependencies must be sychronized 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.

prefer_parallel

The `prefer_parallel` directive and pragma cause the compiler to parallelize the immediately following loop if it is free of dependencies 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.

`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 dependencies. `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++) {
    .
    .
    .
  }
```

While the `ordered` attribute can be used in a `prefer_parallel` directive, it is only useful if the loop contains synchronized dependencies, and `prefer_parallel` will not parallelize a loop containing any dependencies. This attribute is useful in the `loop_parallel` directive, as described in Chapter 6.

loop_parallel

The `loop_parallel` directive forces parallelization of the immediately following loop. The compiler does not check for data dependencies or perform variable privatization; you must synchronize any dependencies manually, and manually privatize loop data as necessary.

The `threads`, `nodes`, `chunk_size` and `max_threads` 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 dependencies as described in Chapter 6.

`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 (i.e. it does not modify its argument, it does not modify any `COMMON` variables, it performs no I/O, and it doesn't call any procedures that have side effects) and is compiled for reentrancy using the `-re` option (the default on SPP1000 Series compilers), this loop can be safely parallelized as shown. 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
  for(i=0;i<n;i++)
    x[i] = func(i);
```

The compiler will not parallelize any loop that does not have a number of iterations determinable 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)!WON'T PARALLELIZE
        .
        .
        A(I) = ...
        .
        .
      ENDDO
```

Since `A(I)` is assigned within the loop here, the compiler cannot determine the loop's iteration count prior to loop invocation, and cannot parallelize it.

An analogous C example follows:

```
#pragma _CNX loop_parallel
while(a(i) > 0) { /* won't parallelize */
  .
  .
  a[i] = ...
  .
  .
}
```

The "Critical sections" section contains an example of using `loop_parallel` to parallelize a loop with a dependency; the dependency is manually handled in a critical section.

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 for more information.

`CPS_STACK_SIZE`

Thread 0's stack size is always controlled by SPP-UX, but any threads it spawns (as the result of `loop_parallel` or tasking directives or pragmas, for example) receive a default stack size of 8 Mbytes. If the parallel construct contains a procedure, and if that procedure contains more than 8 Mbytes of local data, you must modify the stack size of the spawned threads via the `CPS_STACK_SIZE` environment variable. `CPS_STACK_SIZE` must be specified in kbytes. It is read at program startup; the value cannot be changed during execution.

The default stack size for thread 0 can be modified via the `mpa` utility. Refer to the mpa(1) man page for more information.

Denoting induction variables in parallel loops

To safely parallelize a loop, the compiler must be able to correctly determine the loop's primary induction variable.

The compiler can find Fortran `DO` loop induction variables; it may, however, have trouble with `DO WHILE` or hand-rolled Fortran loops, and with all 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 a loop's primary induction variable using the `IVAR=`*indvar* attribute to `loop_parallel`. Consider the following Fortran example:

```
      I = 0
C$DIR LOOP_PARALLEL(IVAR = I)
10    A(I) = ...
      .
      .           ! ASSUME NO DEPENDENCIES
      .
      I = I + 1
      IF(I .LE. N) GOTO 10
```

This is a hand-rolled loop that uses `I` as its primary induction variable. To insure parallelization, the `LOOP_PARALLEL` directive has been placed immediately before the start of the loop, and the induction variable, `I`, has been specified.

Induction variables in C loops can be difficult for the compiler to find, so `ivar` is required in all 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 dependencies */
    .
  }
}
```

Secondary induction variables are variables that are 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, 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 is *not* a legal secondary induction variable because it is not a function of I. It is not private because the value assigned to it in the current iteration is a function of its value in the last iteration. This example can be fixed 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 is a legal secondary induction variable 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) and increment it within a critical section inside 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.

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 `loop_parallel` loops.

A `critical_section` directive or pragma and its associated `end_critical_section` must appear in the same procedure, but they do not have to appear in the same procedure as the parallel construct in which they are used; i.e. 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 following form:

```
#pragma _CNX critical_section
#pragma _CNX end_critical_section
```

The `critical_section` directive and pragma can take an optional gate attribute which allows the declaration of multiple critical sections as described in Chapter 6; however, we will only discuss simple critical sections here.

Consider the following Fortran 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
```

Here, 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` insures this behavior.

An analogous C example follows:

```
#pragma _CNX loop_parallel
for(i=0;i<n;i++) {
   .
   .
   .
   #pragma _CNX critical_section
   sum = sum + x(i);
   #pragma _CNX end_critical_section
   .
   .
   .
}
```

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 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 following 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 attributes:

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

The `ordered` attribute causes the tasks to be initiated in their lexical order; i.e. 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. Note that while this argument insures 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 if necessary.

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

The `ordered, nodes` and `ordered, threads` attributes cause the tasks to run ordered node-parallel and ordered thread-parallel, respectively.

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.

Caution

The compiler performs no dependency checking or synchronization on the code delimited by the tasking directives and pragmas; you must insure the dependencies do not exist or insert your own synchronization code if necessary. Synchronization is discussed in Chapter 6.

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
```

This example specifies thread-parallelism by default. The compiler transforms the above code into a parallel loop and creates machine code equivalent to that shown below.

```
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 (or hypernodes), each thread (or hypernode) will execute multiple tasks; if there are more threads (or hypernodes) than tasks, some threads (or hypernodes) will not execute tasks.

The end_tasks directive and pragma acts as a barrier; all parallel tasks must complete before the code following the 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 one of the NODES attributes 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
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=1;i<=500;i++)
  c[i*2-1] = d[i];
#pragma _CNX end_tasks
```

Nested task parallelism is also possible. In order to nest any parallelism on SPP1000 Series machines, thread-parallelism must be nested within node-parallelism, so 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; it must, and will by default, go thread-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$DIR BEGIN_TASKS(NODES)
      ITOT = 0
      DO I = 1,N
        IF(B(I) .NE. 0) THEN
          A(I,J) = B(I)*C(I)
          ITOT = ITOT + 1
        ENDIF
      ENDDO
C$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 thread-parallelizable loop that goes parallel on the threads of the hypernode on which this task is running.The second node-parallel task contains three subroutine calls, each of which run thread-parallel within the hypernode. The third node-parallel task contains a Fortran 90 array section assignment which runs thread-parallel within the hypernode.

An analogous C example follows:

```
#pragma _CNX begin_tasks(nodes)
itot = 0;
for(i=0;i<n;i++)
  if(b[i] != 0) {
    a[i][j] = b[i]*c[i];
    itot = itot +1;
  }
#pragma _CNX next_task
  #pragma _CNX begin_tasks(threads)
  t1sub();
  #pragma _CNX next_task
  t2sub();
  #pragma _CNX next_task
  t3sub();
  #pragma _CNX end_tasks
#pragma _CNX next_task
for(i=0;i<1000;i++)
  x[i] = y[i];
#pragma _CNX end_tasks
```

Task parallelism can become even more involved, as described in Chapter 6, "Advanced shared memory programming."

Memory classes

5

Chapter 2, "Architecture overview," discusses the three classes of physical memory available on SPP1000 Series systems. These classes are:

- hypernode-local
- subcomplex-global
- CTIcache.

Hypernode-local memory is accessed via the `thread_private` and `node_private` virtual memory classes. Subcomplex-global memory is accessed via the `near_shared`, `far_shared` and `block_shared` virtual memory classes. CTIcache physical memory holds copies of shared memory data which is not resident in the hypernode's physical memory, but is accessed by threads running on the hypernode.

Caution

The memory classes discussed here are only 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. *Using these memory classes requires you to manually handle parallelization.*

Private vs. 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.

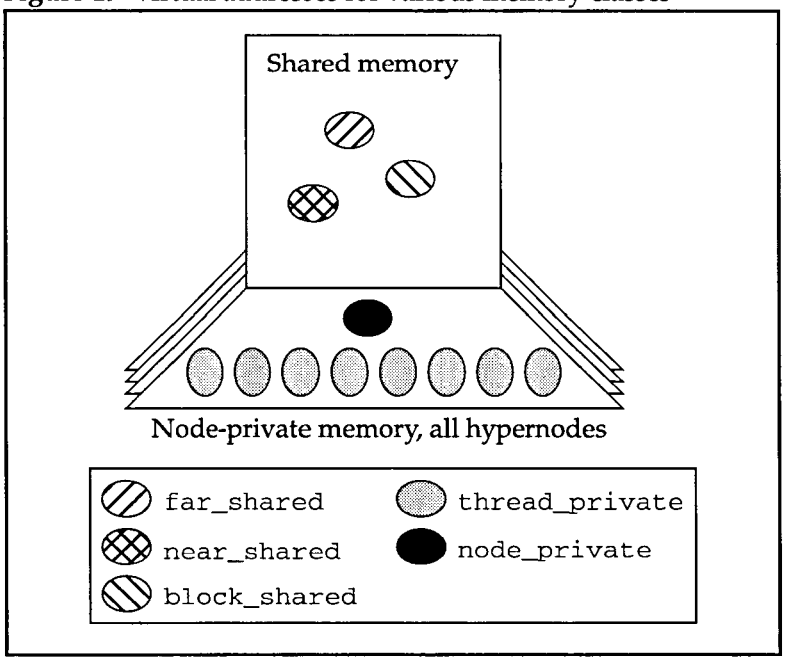
Both `node_private` and `thread_private` data are stored in hypernode-local memory, and are therefore inaccessible to any hypernode other than the one on which they reside (in the case of `thread_private`, access is further restricted to the declaring thread). Latency is identical for private data items that must be fetched from main memory. `near_shared`, `far_shared` and `block_shared` data, on the other hand, are stored in

subcomplex-global physical memory and are therefore accessible from any hypernode in the subcomplex on which the process is running. Main 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 19 shows the virtual addresses associated with a data item stored in each memory class by a single process running on a conceptual 4-hypernode, 8- processor-per-hypernode system. 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 19 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 8 unique virtual addresses, one for each thread within a hypernode.

Figure 20 shows the physical addresses associated with the data items shown in Figure 19, for an identical SPP1000 Series system. For illustrative purposes, all shared data in Figure 20 is assumed to be array data, and is represented by circles and portions of circles rather than ovals.

Figure 20 Physical addresses for various memory classes

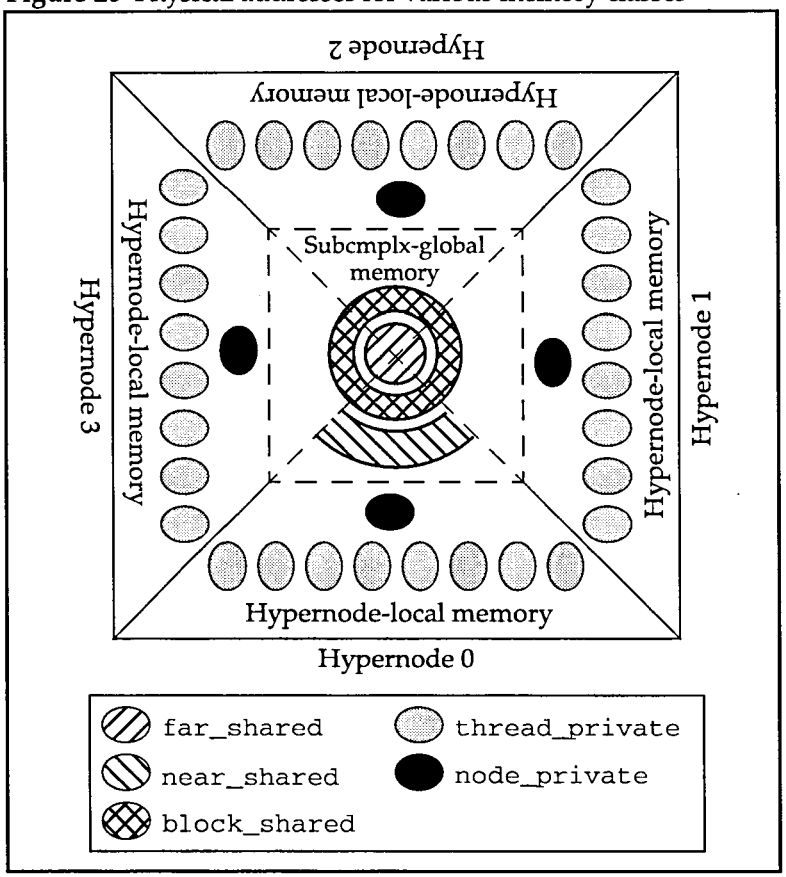


Figure 20 shows the physical memory replication of the private memory classes, as well as the distributed nature of the `far_shared` and `block_shared` classes. You can control hypernode placement of the `near_shared` class, shown here residing physically on hypernode 0. Note that all hypernodes can access this data, since it resides in subcomplex-global physical memory.

The subsections that follow 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 multihypernodal subcomplexes, 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, 8-processor-per-hypernode system, a single, statically allocated `thread_private` data item is accessed by 8 virtual addresses that map to 32 physical addresses. On a 7-hypernode, 8-processor-per-hypernode system, the same data item is accessed by 8 virtual addresses which map to 56 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 4 Gbyte 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 multihypernodal subcomplex, 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 subcomplex.

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

near_shared data is accessible by any thread running on any hypernode of a subcomplex, but is physically stored entirely within the subcomplex-global memory of a particular hypernode, allowing 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 within 60 clocks; threads running on other hypernodes must access the data via their CTIcache if it is encached there; if not, they must fetch it over the SCI rings. These accesses can take over 200 clocks, depending on whether any coherency or bank conflicts are encountered.

As explained in the "Static memory class assignments" section later in this chapter, the near_shared class is typically used for data which 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, and, since it is distributed evenly across the subcomplex-global memory of all hypernodes, it provides the best average access time when it is used equally by threads running on all 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 round-robin to physical memory pages on all the hypernodes in the subcomplex. The page size is 4KB.

As explained in the "Static memory class assignments" section later in this chapter, the far_shared class is typically used for data which is accessed equally by all the hypernodes in a subcomplex.

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 subcomplex. 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 subcomplex-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, memory classes are assigned through the use of syntax extensions to CONVEX C, which are defined in header file `/usr/include/spp_prog_model.h`. This file must be included in any C program which uses memory classes.

The general form for Fortran memory class directives is:

```
C$DIR class_name (namelist)
```

where *namelist* is a comma-delimited list of variables, arrays, and/or common block names to be assigned the class *class_name*. Common block names must be enclosed in slashes, and only entire common blocks can be assigned a class. This means that arrays and variables in *namelist* must not also appear in a common block, and must not be equivalenced to data objects in common blocks.

class_name can be `THREAD_PRIVATE`, `NODE_PRIVATE`, `NEAR_SHARED`, `FAR_SHARED`, or `BLOCK_SHARED`. For `BLOCK_SHARED`, *namelist* must include only allocatable arrays.

These Fortran memory class declarations must appear with other specification statements; they cannot appear within executable statements.

In C, CONVEX type-qualifier extensions are used, so memory classes are assigned in variable declarations. The general form for assigning memory classes in C is:

```
#include <spp_prog_model.h>
.
.
.
[storage_class_specifier] class_name  type_specifier  namelist
```

where *storage_class_specifier* specifies a nonautomatic storage class, *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`" section on page 145), *type_specifier* is a standard C data type (`int`, `float`, etc.), and *namelist* is a comma-delimited list of variables and/or arrays of type *type_specifier.*

In C, data objects which 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 C Programming Language* (DSW-046).

Note

All C code examples presented in this chapter assume that the line

```
#include <spp_prog_model.h>
```

appears above the C code presented. This header file maps user symbols to the implementation reserved space.

If you assign a memory class to a C structure, all structure members must be of the same class.

In both C and Fortran, once a data item is assigned a memory class, the class cannot be changed.

The CONVEX SPP1000 Series 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`

Since `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 data 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 example demonstrates several ways to declare thread_private storage in C. 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];
  /* tbc, a and b are declared elsewhere: */
  extern thread_private double tpc[1000],a,b;
  .
  .
  .
```

C's 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.

Assume a Fortran or C program containing the appropriate example is running on a 4-hypernode subcomplex with 8 processors per hypernode. Each data item will require 8 virtual addresses, for a total of 192,128 bytes of virtual space. These virtual addresses will map to 8 physical addresses per hypernode, or 32 total physical addresses per data item, requiring a total of 768,512 (32×24016) bytes of physical memory.

node_private

Since node_private variables are 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 occupies 24,016 bytes. The contents of BLK1 are placed in node_private memory along with XNP and YNP.

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.

```
/* 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 example.

For either language example, assuming an 8 processor 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 multihypernodal subcomplexes 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 16 Gbytes of physical memory on an 8 hypernode subcomplex (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 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.

near_shared
Static assignments of the near_shared memory class are of limited usefulness, because they place the declared near_shared memory on hypernode 0. The purpose of declaring the near_shared memory is defeated unless the code that most frequently accesses this data happens to be running on hypernode 0, which is unlikely on a multihypernodal subcomplex.

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 follows.

```
func() {
  static near_shared double xns[1000][1000];
  .
  .
  .
```

Here, xns is local to func(). Global declarations, and local declarations of the extern storage class, are also legal.

Both language examples allocate 8,000,000 bytes of virtual address space, which maps to 8,000,000 bytes of physical memory on 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 always be executed on hypernode 0 only.

The true power of near_shared memory is realized only when it is resident on the hypernode which 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 memory class assignments" section later in this chapter.

far_shared
Since 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 example follows.

```
static far_shared double xfs[1000][1000];
```

Here, xfs is local to func(). Global declarations, and local declarations of the extern storage class, are also legal.

These declarations allocate 8,000,000 bytes of virtual address space, which is mapped to physical memory by 4KB pages round-robin to each hypernode on a multihypernodal subcomplex, beginning at hypernode 0. When all hypernodes are accessing the far_shared data with relatively equal frequency, this provides the best average access times.

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 CONVEX Fortran ALLOCATABLE arrays and with the memory_class_malloc function in CONVEX C. 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, the memory class is specified in the declaration 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 or the C memory_class_malloc function. 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 *class_name*_POINTER (*allocatable-namelist*)

where *class_name* is one of THREAD_PRIVATE, NODE_PRIVATE, NEAR_SHARED, or FAR_SHARED and *allocatable-namelist* is a comma-delimited list of arrays previously declared to be ALLOCATABLE in the same procedure. *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. 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 which 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 `memory_class_malloc` function is very similar to standard `malloc` and has the following form:

memcls_ptr = `void *memory_class_malloc(size_t` *bytes*`,int` *class_name*`);`

where *memcls_ptr* is a previously-declared pointer to a variable of the desired memory class, *bytes* is the requested number of bytes, and *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`. Note that *memcls_ptr* need not be of the class indicated in *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` *class_name*`_POINTER` Fortran directive to allocate a pointer of one class to an array of a different class.

Not all combinations of pointer classes with data classes are supported, and not all make sense. The general rules are:

- `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.
- 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 *class_name*`_POINTER` directives is not used, dynamically allocated memory is accessed via a pointer of the same class as the allocated data.

Table 4 shows proper and improper pointer/data class combinations.

Table 4 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 Table 4, "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

Using standard `malloc` in a shared memory C program will allocate `far_shared` memory. Memory allocated by either standard `malloc` or `memory_class_malloc` can be deallocated using the `free` function.

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. Refer to Appendix C, "Fortran 90 compatibility," of the *Fortran Language Reference, 11th Edition*, for more information.

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 (i.e. within a C function, all data not assigned a storage class of `static` or `extern`; in Fortran, all data not declared in common blocks or `SAVE`d) is `near_shared` and resides on hypernode 0 by default. This default can be changed to `far_shared` using the `mpa(1)` utility. For more information, refer to the `mpa(1)` man page.

Using each dynamic memory class is explained in detail in the sections that follow.

thread_private

Since 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)
      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.

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;
.
.
.
#pragma _CNX loop_parallel(threads)
for(i=0;i<num_threads();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 regions, a shared pointer is needed. This is because when serial Fortran code running on 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 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, since 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

which are to be accessed in parallel code must be accessed with explicitly-declared `near_shared` or `far_shared` pointers. Since 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 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, 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 subcomplex, when the `ALLOCATE` statement executes, 8,000 bytes (1000 elements × 8 bytes per element) of virtual space are allocated for `XNP`. If the array is then accessed from a node-parallel region, 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 subcomplex accesses the array, it will occupy a total of 32,000 bytes of physical memory.

A similar 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);
```

This example allocates 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, since 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 which 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 loop or region.

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)
      DO I = 1, M
        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
        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
        DEALLOCATE(NODE_V)
          .
          .
          .
        ENDDO
```

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, which is described in Chapter 6, 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)
for(i=0; i<m; i++) {
  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];
  }
  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. Since all code outside of explicitly-defined node-parallel regions executes on hypernode 0, near_shared memory should be allocated from within explicitly-defined node-parallel regions. These regions are manually defined by the programmer using the LOOP_PARALLEL(NODES) directive and pragma, which are discussed in detail in Chapter 5.

Unconditionally allocating near_shared arrays in node-parallel regions 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 subcomplex.

```
      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
      DO I = 1, NUM_NODES()! IS ASSUMED
        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. This example allocates a `near_shared XNS(1000)` array for hypernode 0, and a `near_shared YNS(1000)` array for hypernode 1. When accessed, `XNS` is stored in hypernode-global physical memory on hypernode 0, and is therefore accessible from that hypernode with the least latency; similarly, `YNS` is physically stored on and quickly accessible from hypernode 1. The arrays' unique virtual addresses allow either to be accessed by the hypernode it doesn't 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 which allocates memory in the same fashion follows. Again we assume that this code will always run on a two-hypernode subcomplex.

```
static near_shared double *xns, *yns;
int node_id;
.
.
.
/* the following code must run node-parallel on a 2-node
   subcomplex:                                              */
#pragma _CNX loop_parallel(nodes, ivar = i)
for(i=0;i<num_nodes();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, the pointer class is assigned with the type declaration, and the data class is assigned by the `memory_class_malloc` function.

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). 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 region, 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 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 4 on page 133 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 region. 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))
        .
        .
        .
```

Here, thread-parallelism is achieved using the LOOP_PARALLEL(THREADS) directive, which is discussed further in Chapter 4. The hypernode on which the ALLOCATE statement executes allocates a near_shared copy of XNS for each thread. Each thread running on that hypernode can then reference its copy of XNS 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 XNS 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 will require sharing of the thread_private pointers.

An similar C example follows.

```
static thread_private double *zns;
.
.
.
/* the following code must run thread-parallel    */
#pragma _CNX loop_parallel(threads, ivar = i)
for(i=0;i<num_threads();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)
      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. Since this array is `near_shared`, it can also be accessed by any other hypernode, though that doesn't happen in this example. Using `LOOP_PARALLEL` to manually parallelize loops is further discussed in Chapter 6.

An analogous C example follows.

```
static near_shared double *node_scratch;
.
.
.
#pragma _CNX begin_tasks(nodes)
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

Since `far_shared` memory is distributed round-robin across all hypernodes in the subcomplex, it is best dynamically allocated in nonparallel regions. Allocating `far_shared` memory in a thread- or node-parallel region would create multiple copies of the requested `far_shared` array, one for each allocating thread or hypernode in the region. 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 REGION:
      ALLOCATE(XFS(10000))
```

Since 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 round-robin by 4KB pages to each hypernode in the subcomplex.

A similar C example follows.

```
static far_shared double *xfs;
.
.
.
/* the following code should run in a nonparallel
   region:                                          */
xfs = (double *)memory_class_malloc(sizeof(double)*1000,
                                    FAR_SHARED_MEM);
```

This example allocates memory in the same manner as the Fortran example.

`far_shared` is the default memory class for all data not otherwise assigned a class.

`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 which 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 hypernodes; 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 regions; in this case, the compiler will automatically distribute the data evenly across the hypernodes of the subcomplex. Allocating `block_shared` memory from parallel regions 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 REGION:
      ALLOCATE(XBS(10240), YBS(7680))
```

Here, 81920 bytes of virtual space is requested for `XBS`, and 61440 bytes is requested for `YBS`. Assuming this code is running on a 4-hypernode subcomplex, these `block_shared` arrays will have their physical pages divided equally, in contiguous chunks, among the 4 hypernodes. Recall that the page size is 4096 bytes, so `XBS` occupies 20 pages (81920/4096 = 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 hypernode 0 (yet still accessible via their virtual addresses from any other hypernode), the second 5 pages go to hypernode 1, the third 5 pages to hypernode 2, and the last 5 pages to hypernode 3.

The 61440 bytes of `YBS`, on the other hand, occupy 15 pages. 4 does not integrally divide 15, so the compiler automatically increases the size of `YBS` just enough to allow the number of hypernodes to integrally divide its pages. `YBS` becomes an 8192 element array; it now occupies 65536 bytes or 16 pages of memory. These 16 pages are divided up so that the first 4 pages map to hypernode 0 and so on, with the last 4 mapping to 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 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 example follows.

```
static block_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);
```

This example allocates and distributes `xbs` and `ybs` as `block_shared` arrays exactly as the Fortran example did, including the automatic resizing of `xbs` and `ybs`.

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.

Because `block_shared` data elements each have a unique virtual and physical address (there is no virtual address replication as with the `node_private` class), their size is limited by the 4 Gbyte virtual address space limit.Keep this in mind if your data set size scales with the number of available hypernodes. If your program needs to surpass this limit, it can do so using the `node_private` class as described in the "Advanced shared memory example" of Chapter 6.

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     ARRAY 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 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, and that the 4 Gbyte virtual address limit is not surpassed.

```
      REAL*8 XBS(:)
      ALLOCATABLE(XBS)
C$DIR BLOCK_SHARED(XBS)
      .
      .
      .
C **NO PARALLELISM**
      ALLOCATE(XBS(ARSZ))
      NN = NUM_NODES()
                        C$DIR LOOP_PARALLEL(NODES)
      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 hypernode-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 block_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)
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)
      DO I = 1, NUM_NODES()
        DO J = 1, N
          ABS(J,I) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

Here, `N` 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` insures that each parallel hypernode accesses its local elements automatically.

Advanced shared memory programming

6

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 which 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 dependencies were handled in Chapter 4.

This chapter is concerned with manual parallelizations that use the program-wide memory classes discussed in Chapter 5, and that handle multiple and ordered data dependencies.

Before we can discuss specific examples of such parallelization, however, we must introduce the remaining underlying concepts and available functions.

Parallel information functions

Several SPP1000 Series 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, and can appear in executable statements anywhere an integer expression is legal. The 8-byte versions are typically only used in Fortran programs in which the default data lengths have been changed using the `-cfc`, `-p8` or similar compiler options.

Note

All C code examples presented in this chapter assume that the line

```
#include <spp_prog_model.h>
```

appears above the 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 the process is using at runtime. They can be used to dimension automatic and adjustable arrays in Fortran, and may be used in C or Fortran to dynamically specify array dimensions and allocate storage.

In Fortran, these functions have the following forms:

```
INTEGER NUM_PROCS()
INTEGER*8 NUM_PROCS_8()
```

In C, they have the following forms:

```
int num_procs(void);
long long num_procs_8(void);
```

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 following forms:

```
INTEGER NUM_THREADS()
INTEGER*8 NUM_THREADS_8()
```

In C, they have the following forms:

```
int num_threads(void);
long long num_threads_8(void);
```

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 both C and Fortran to dynamically specify array dimensions and allocate storage.

In Fortran, these functions have the following forms:

```
INTEGER NUM_NODES()
INTEGER*8 NUM_NODES_8()
```

In C, they have the following forms:

```
int num_nodes(void);
long long num_nodes_8(void);
```

Number of threads on current hypernode

These functions return the number of threads running on the hypernode from which the function is called. This number can vary from one hypernode to another depending on subcomplex configurations, usage of manual parallelization directives, and the number of processors installed on each hypernode.

In Fortran, these functions have the following forms:

```
INTEGER NUM_NODE_THREADS()
INTEGER*8 NUM_NODE_THREADS_8()
```

In C, they have the following 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 following forms:

```
INTEGER MY_THREAD()
INTEGER*8 MY_THREAD_8()
```

In C, they have the following forms:

```
int my_thread(void);
long long my_thread_8(void);
```

Hypernode ID

These functions return the 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 following forms:

```
INTEGER MY_NODE()
INTEGER*8 MY_NODE_8()
```

In C, they have the following forms:

```
int my_node(void);
long long my_node_8(void);
```

Level of parallelism

These functions return a value representing the level of parallelism of the calling process.

In Fortran, these functions have the following forms:

```
INTEGER LEVEL_OF_PARALLELISM()
INTEGER*8 LEVEL_OF_PARALLELISM_8()
```

In C, they have the following forms:

```
int level_of_parallelism(void);
long long level_of_parallelism_8(void);
```

The return value is one or a sum of the values shown in Table 5. In C, these values are `#defined` as symbolic constants in `spp_prog_model.h`.

Table 5 Levels of parallelism

Function return value	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 D 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, but this can be changed via the `mpa(1)` utility.

In Fortran, these functions have the following forms:

```
INTEGER MEMORY_TYPE_OF_STACK()
INTEGER*8 MEMORY_TYPE_OF_STACK_8()
```

In C, they have the following 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 6.

Table 6 Stack type return values

Function return value	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, 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 subcomplex, even if they span hypernodes.

If you attempt to spawn thread-parallelism from within a thread-parallel construct, the compiler will ignore your directives, and your inner parallel construct will simply run serially.

Thread ID assignments

Chapter 3 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 region 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 region, 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 region 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 which 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 dependencies, but a rich set of directives, pragmas and data types are available to help you manually parallelize such loops by synchronizing (and, if necessary, ordering) access to the code containing the dependency. These directives can also be used to synchronize dependencies in parallel tasks. They allow you to efficiently exploit parallelism in loops and regions 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 "Synchronization functions" section, 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. They may be initialized and referenced only by passing them as arguments to the functions discussed in the following "Synchronization functions" section.

In C, gates and barriers are declared using the `gate_t`, `gate8_t`, `barrier_t` and `barrier8_t` data type statements, which have the following forms:

```
gate_t namelist
gate8_t namelist
barrier_t namelist
barrier8_t namelist
```

where *namelist* is a comma-delimited 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, 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 following forms:

```
C$DIR GATE(namelist)
C$DIR BARRIER(namelist)
```

where *namelist* is a comma-delimited 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.

In Fortran, gates and barriers can only appear in `COMMON` and `DIMENSION` statements, in preceding type statements, and as dummy and 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 must contain only gates or only barriers; it must not contain other data types. Arrays of gates or barriers must be dimensioned using `DIMENSION` statements.

Synchronization functions

The C and Fortran 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 which change the default data size to 8 bytes (e.g. `-cfc`, `-p8`, `-pd8`). 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 "Using gates and barriers" 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.

In C, the functions have the following 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.

Deallocation functions

These functions free the memory assigned to the specified gate or barrier.

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 gate or barrier variables, as appropriate.

In C, the functions have the following 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);
```

Where *gate_p* and *barrier_p* are pointers of the indicated type.

Always free your 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 doesn't require a wait. If the gate is acquired, the functions return 0; if not, they return -1.

The Fortran gate locking functions have the following 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, 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 which 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, the functions have the following 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, the functions have the following 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 insures that two such barriers are not simultaneously active. It is also the programmer's responsibility to insure that *nthr* reflects the correct number of threads.

loop_parallel(ordered)

The `loop_parallel(ordered)` directive and pragma was briefly introduced in Chapter 4. It is designed to be used with ordered sections, which are discussed in the next section, to execute loops with ordered dependencies 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 dependencies 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:

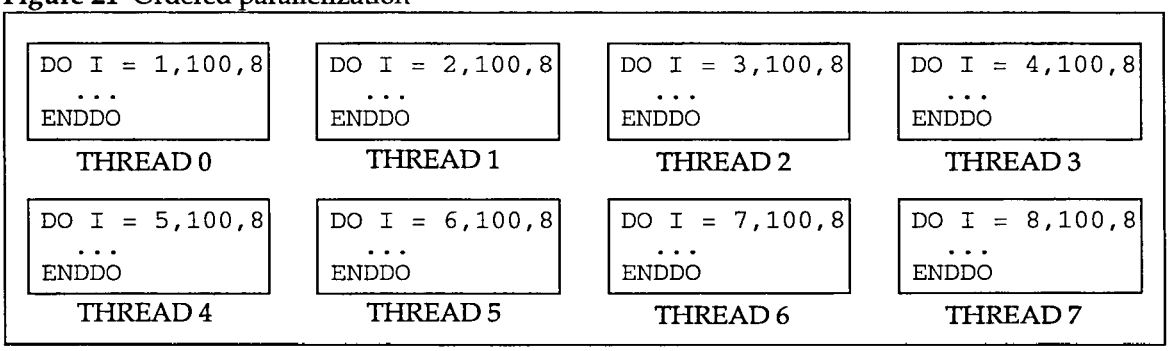
```
#pragma _CNX loop_parallel(ordered, ivar = i)
for(i=0;i<100;i++) {
  .
  . /* code containing ordered section */
  .
}
```

Assume that the body of this loop contains code that is parallelizable except for an ordered data dependency (otherwise there is no need to order the parallelization). This dependency 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
```

as shown in Figure 21.

Figure 21 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 dependencies.

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 following 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. The gate attribute is required when synchronizing access to a shared variable from multiple parallel tasks. When a gate is specified, it must be allocated using the `alloc_gate` intrinsic and should be initialized using the `unlock_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;

i.e. 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 dependencies 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 following 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. 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 dependency.

Synchronizing code

Code containing dependencies can be parallelized by synchronizing the way the parallel tasks access the dependency. 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 of Chapter 4 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 dependencies and dependencies 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 insure 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. Since 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 dependencies between

the critical sections. If no dependencies 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(...)
      .
      .
      .
      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 insure 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 do the work of locking and unlocking a specified gate to isolate a section of code in a loop. However, they also insure 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 = 1, N
   . ! PARALLELIZABLE CODE...
   .
   .
   A(I) = A(I-1) + B(I)
   . ! MORE PARALLELIZABLE CODE...
   .
   .
ENDDO
```

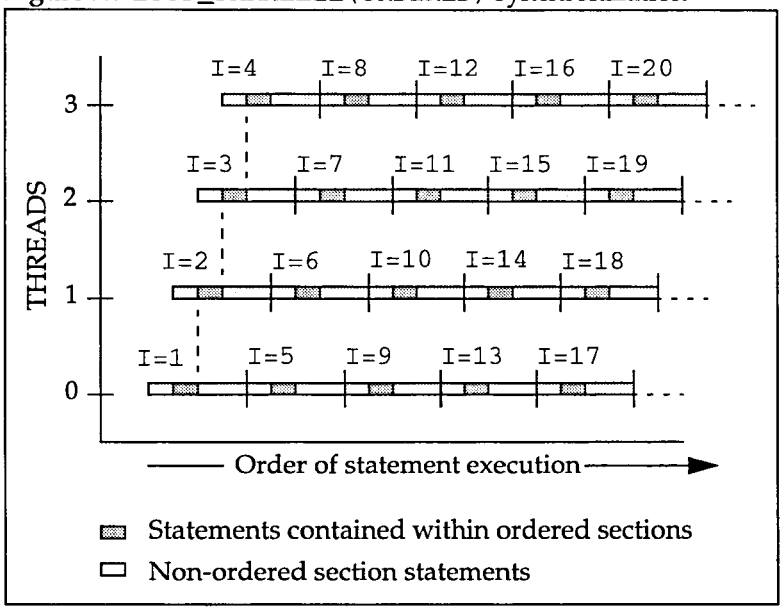
To simplify illustration, we will stick to Fortran, but an analogous C example could similarly be parallelized.

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 = 1, 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 Figure 21 on page 164, and the ordered section containing the A(I) assignment will execute in iteration order, insuring 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 22.

Figure 22 `LOOP_PARALLEL(ORDERED)` synchronization



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 once and only 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'S 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<ni++) {
  .
  .
  .
  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's 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 can't 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
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; here, 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:

```
#pragma _CNX loop_parallel(ordered,ivar = i)
for(j=0;j<99;j++) {
  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 dependencies in a structured, semiautomatic manner. The same isolation can

be accomplished manually using the functions discussed in the "Synchronization functions" section on page 160.

Recall our simple critical section example from Chapter 4:

```
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$DIR NEXT_TASK    ! TASK FOUR:
      DO M = 1, N                   ! NO DEPENDENCIES IN THIS TASK
        Y(M) = X(M) * Y(M) / Z(M)   ! Y CAN BE COMPUTED WITH
      ENDDO                         ! A, B, P, D
C$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 dependency 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.

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 regions 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, since 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 doesn't 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.

An 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 dependencies 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 regions is the ability to conditionally lock them. This allows the task that wishes to execute the region 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.

Advanced shared memory example

We will now consider a more detailed Fortran example which is manually parallelized in two dimensions and makes use of barrier synchronization and `block_shared` arrays, two topics which are difficult to illustrate in brief examples.

Consider the following Fortran code.

```
REAL*8 A(800,800,800), B(800,800,800)
   .
   .
   .
DO K = 1,800
  DO J = 1, 800
    DO I = 1, 800
      B(I,J,K) = ...
       .
       .
       .
    ENDDO
  ENDDO
ENDDO
   .
   .
   .
DO K = 2,799
  DO J = 2, 799
    DO I = 2, 799
      A(I,J,K)=A(I,J,K)+1./6.*(B(I,J+1,K)+B(I,J-1,K)+
>      B(I+1,J,K)+B(I-1,J,K)+B(I,J,K+1)+B(I,J,K-1))
       .
       .
       .
    ENDDO
  ENDDO
ENDDO
```

While this code contains several opportunities for parallelization, each of its arrays occupies nearly 4 Gbytes. The two arrays together with any additional arrays or variables required by the program therefore require far more virtual memory than is available to the process.

To get around this, we must somehow increase the virtual address space available to the process. We can do this by requiring that the process run on a minimum number of hypernodes, and allocating sections of `A` and `B` in `node_private` memory on the individual hypernodes which

will compute them. Since these sections will only be accessible to the hypernodes on which they reside, any elements that must be shared must be copied into shared memory. We will use `block_shared` memory for this function. Recall that `block_shared` arrays are distributed in chunks across hypernodes, allowing fastest access from the hypernode on which the section resides (like `node_private`), but also allowing other hypernodes to access the data. This will provide optimal array-access efficiency.

The following implementation uses manually-partitioned arrays and two-dimensional manual parallelization to solve this example as efficiently as possible. It assumes that performance testing indicates that a minimum of 4 hypernodes is required to adequately provide the necessary amounts of virtual memory.

```
      REAL*8 A, B
C EACH NODE WILL ALLOCATE A 1/NUM_NODES-SIZE SECTION OF EACH ARRAY:
      ALLOCATABLE A(:,:,:),B(:,:,:)
C$DIR NODE_PRIVATE(A,B)
C     REQUIRED FOR DYNAMIC NODE_PRIVATE ALLOCATED BY A SINGLE THRD:
C$DIR FAR_SHARED_POINTER(A,B)
      REAL*8 BLEFT, BRIGHT ! EACH NODE COMMUNICATES ITS ENDPLANES
      ALLOCATABLE BLEFT(:,:,:),BRIGHT(:,:,:)! TO OTHER NODES VIA
C$DIR BLOCK_SHARED(BLEFT, BRIGHT)           ! BLEFT/BRIGHT
      INTEGER NODE_ID, nodeNS     ! EACH NODE'S ID AND ARRAY SIZE
      INTEGER K, J, I, KS, KE     ! INDUCTION VARIABLES
C$DIR NODE_PRIVATE(NODE_ID, nodeNS, KS, KE)
C$DIR THREAD_PRIVATE(K, J, I, LBAR2)
C$DIR BARRIER(BARRALL) ! TO PREVENT USING B BEFORE IT IS ASSIGNED
      INTEGER CEILING
      CEILING(X) = AINT(1.0+X)-AINT(1.0+AINT(X)-X)
      .
      .
      .
      PRINT*, "ENTER NUMBER OF HYPERNODES:"
      READ*, NN
      IF (NN .GE. 4) THEN           ! ENOUGH MEMORY TO SOLVE
C       NS IS THE MAX ARRAY SIZE NEEDED BY AT LEAST 1 NODE:
        NS = CEILING(800./FLOAT(NN)) ! SIZE OF A,B ARRAY SECTIONS
      ELSE
        STOP 'NOT ENOUGH MEMORY' ! STOP IF NOT ENOUGH MEMORY
      ENDIF
      .
      .
      .
```

```
C      ALLOCATE EACH NODE'S ARRAYS;
C      B'S 3RD DIMENSION HAS ROOM FOR NEIGHBOR VALUES:
       ALLOCATE(A(800,800,NS),B(800,800,0:NS+1))
C      ALLOCATE BLOCK_SHARED BLEFT AND BRIGHT TO ALLOW EACH NODE
C      TO WRITE ENDPLANES LOCALLY, SHARE NEIGHBORS GLOBALLY:
       ALLOCATE(BLEFT(800,800,NN),BRIGHT(800,800,NN))
       LBAR1 = ALLOC_BARRIER(BARRALL) ! ALLOCATE BARRIER
                        C$DIR LOOP_PARALLEL(NODES)
       DO NODE = 1, NN      ! DO ON EACH NODE:
         NODE_ID=MY_NODE()+1 ! INDEX INTO SHARED ARRAY.
C        FIND EXACT SIZE OF ARRAYS TO BE COMPUTED ON EACH NODE:
         IF (NODE_ID .GT. 800 - NN * (NS-1)) THEN
           nodeNS=NS-1 ! HIGH NUMBERED NODES MIGHT DO ONE LESS.
         ELSE
           nodeNS = NS ! ALL OTHER NODES DO THE SAME SIZE.
         ENDIF
C        LOOP NEST 1:
C$DIR    LOOP_PARALLEL(THREADS) !IN THRD-PARALLEL LOOP, EACH NODE
         DO K=1,nodeNS          !COMPUTES ITS PIECE OF THE B ARRAY
           DO J = 1, 800
             DO I = 1, 800
               B(I,J,K) = ...
               .
               .
               .
             ENDDO
           ENDDO
         ENDDO ! END LOOP NEST 1
C        LOOP NEST 2:
C        NOW PUT EACH NODE'S B ENDPLANES IN SHARED ARRAYS:
         DO J = 1, 800
           DO I = 1, 800
C            CURRENT NODE'S RIGHT PLANE IS NEIGHBOR NODE'S LEFT:
             BLEFT(I,J,NODE_ID)=B(I,J,nodeNS)
C            CURRENT NODE'S LEFT IS NEIGHBOR NODE'S RIGHT:
             BRIGHT(I,J,NODE_ID)=B(I,J,1)
           ENDDO
         ENDDO ! END LOOP NEST 2
C        BARRIER INSURES ALL ENDPLANES ARE COPIED BEFORE USED:
         LBAR2=WAIT_BARRIER(BARRALL,NN)!ONE THREAD PER NODE CALLS
C        LOOP NEST 3:
         IF (NODE_ID .GT. 1) THEN !THIS NODE IS INTERIOR OR RTMOST
           DO J = 1, 800
             DO I = 1, 800
               B(I,J,0)=BLEFT(I,J,NODE_ID-1) !GET LEFT ENDPLANE
             ENDDO
           ENDDO
         ENDIF
```

```
      IF(NODE_ID .LT. NN) THEN !THIS NODE IS INTERIOR OR LFTMOST
        DO J = 1, 800
          DO I = 1, 800
            B(I,J,nodeNS+1)=BRIGHT(I,J,NODE_ID+1) !GET RT ENDPLANE
          ENDDO
        ENDDO
      ENDIF
C     GOT THE ENDPLANES, NOW COMPUTE ARRAY BOUNDS:
      IF (NODE_ID .EQ. 1) THEN ! THIS NODE IS LEFTMOST
        KS = 2
        KE = nodeNS
      ELSEIF (NODE_ID .EQ. NN) THEN ! THIS NODE IS RIGHTMOST
        KS = 1
        KE = nodeNS-1
      ELSE                              ! THIS NODE IS INTERIOR
        KS = 1
        KE = nodeNS
      ENDIF
C     NOW COMPUTE ARRAY A:
C     LOOP NEST 4:
C$DIR LOOP_PARALLEL(THREADS) ! GO THREAD-PARALLEL ON EACH NODE
      DO K = KS, KE
        DO J = 2, 1023
          DO I = 2, 1023
            A(I,J,K)=A(I,J,K)+1./6.*(B(I,J+1,K)+B(I,J-1,K)+
     >        B(I+1,J,K)+B(I-1,J,K)+B(I,J,K+1)+B(I,J,K-1))
            .
            .
            .
          ENDDO
        ENDDO
      ENDDO ! END LOOP NEST 4
    LBAR1 = FREE_BARRIER(BARRALL) ! DEALLOCATE BARRIER
    ENDDO  ! END NODE PARALLEL CODE
```

Here, the A and B arrays, which are used in the bulk of the compute-intensive code and occupy the bulk of the process's virtual memory, are assigned the `node_private` class. They are allocated in serial code, so that when they are accessed physical copies are created on each hypernode, and each hypernode accesses its copy using the same array names. `far_shared` pointers are used to access A and B, as described in Chapter 5 on page 136.

If enough hypernodes aren't available to handle the virtual memory requirements of A and B at allocation time, the program stops.

Note that B is allocated with an extra K dimension plane at each end so that the hypernodes can satisfy computational dependencies involving these endplanes by sharing them.

B is computed in loop nest 1, and is then used in loop nest 4 to compute A. B must be entirely computed before computation of A can begin. Because the computation of A contains both positive and negative offset indexing of B, each hypernode must share its B endplanes using the extra endplanes allocated before computation of A can commence.

This sharing is accomplished through the BLEFT and BRIGHT block_shared arrays. Loop nest 2 copies the endplanes from B into these arrays, and, after the BARRALL barrier insures that this copying is complete, loop nest 3 copies the endplanes from BLEFT and BRIGHT into the appropriate extended dimensions of B on the appropriate hypernodes. Note that BLEFT and BRIGHT are allocated based on the number of hypernodes, and indexed in their third dimension by NODE_ID; this insures that the elements most often used on a hypernode will physically reside on that hypernode. This means copying the endplanes of B into these arrays can be done with minimal access latency. Copying the endplanes into neighboring hypernodes' copies of B in loop nest 3 requires more costly interhypernode memory accesses. However, if the problem is to run on multiple hypernodes, which it must to obtain the virtual memory it needs, some interhypernode communication is inevitable, and this code minimizes it to the extent possible.

After this copying takes place, each hypernode determines its index range in the third dimension of A and B, and A is computed in loop nest 4. Here, all accesses are to node_private memory, so access latency is minimal.

This problem can be solved in parallel in a number of ways, which include splitting A and B up across threads in thread_private memory, or placing BLEFT and BRIGHT in far_shared memory. The solution presented above is more versatile and efficient.

This problem can also be solved using message passing, as explained in Chapter 7.

Message passing programming

7

This chapter presents a high level overview of the message passing paradigm, in which all parallelism and data distribution must be explicitly handled by the programmer. Tasks are coordinated and data is shared among processes by passing messages using the functions of the ConvexPVM library, which are accessible from both C and Fortran. This chapter assumes a familiarity with ConvexPVM and is therefore primarily concerned with providing Exemplar-specific PVM coding tips and examples. For more information on using ConvexPVM, refer to the *ConvexPVM User's Guide*. For more information on specific ConvexPVM functions, including code examples, refer to the man page for the function in question.

The message passing paradigm is useful for porting message passing programs written for other systems, or for those programmers already familiar with programming in this style.

A brief comparison of the message passing and shared memory paradigms is given in Chapter 1.

Basic operation

In this programming style, the user writes an application to be run as a collection of single-threaded processes which can be run on one or more processors. Typically each process determines its actions at runtime based on its assigned task identifier (tid), which resembles the UNIX process-identifier (pid) but includes encoded information regarding the location of the process. The tid is available through a ConvexPVM function. The user also specifies the communication of data values among the processes using message passing.

Message passing programs are inherently parallel, and unless explicitly coordinated by message-waiting, 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.

Message passing library

The message passing library is ConvexPVM (Parallel Virtual Machine). ConvexPVM functions are written in C, but are accessible from both C and Fortran programs. The Fortran versions automatically perform appropriate argument translations and use the correct calling conventions for calling C.

Exemplar-specific configuration features

ConvexPVM for SPP Series machines includes extensions to the hostfile syntax, new default system filenames, performance tuning functions, and a new architecture specifier.

New hostfile syntax

The following ConvexPVM hostfile options are provided for use on Exemplar systems:

- `mt=shm`—use shared memory for message passing within subcomplexes. This is the default.
- `mt=sock`—use sockets for message passing, both within and between subcomplexes. This will impede performance in multi-processor subcomplexes.
- `nf=`*n*—*n* is the number of 4 kbyte message segments to allocate in shared memory. When `mt=shm` is specified, this memory is used for all intertask communication within a subcomplex. The default is 4000 segments, or nearly 8 Mbytes of communication buffer memory.
- `np=`*m*—*m* is the maximum number of ConvexPVM tasks allowed on the virtual machine. Your program cannot spawn more than *m-1* tasks (your program counts as one task). *m* should be at least as large as the number of processors you want your program to run on in the host subcomplex plus one; if your program spawns substantially more tasks than available processors, performance may be inhibited. 6 kbytes of shared memory is allocated to hold incoming message pointers for each task at ConvexPVM startup, so memory can be conserved by choosing *m* carefully. The default is 50.

When multiple subcomplexes are used in a virtual machine, they communicate via sockets and must therefore appear in separate entries in the hostfile. Use the following form to specify a subcomplex in the hostfile:

myhostname : *mysubcomplex*

Where *myhostname* is the name of the machine and *mysubcomplex* is the name or numeric subcomplex ID of the subcomplex.

System filenames

On SPP Series machines, default names for PVM system files include subcomplex IDs. These files use the following form:

`/tmp/pvmd.`*uid*`.`*scid*

Where *uid* is the user ID and *scid* is the subcomplex ID of the task which generated the file.

Performance tuning functions

ConvexPVM for SPP Series machines includes the `pvm_setopt(int` *option*`, int` *value*`)` and `pvmfsetopt(`*option*`,` *value*`,` *oldvalue*`)` performance tuning functions. Refer to the `setopt(3pvm)` man page for more information.

New architecture specifier

The `pvm_spawn` and `pvmfspawn` functions now recognize `CSPP` as the architecture specifier for SPP Series machines.

Exemplar as part of a PVM cluster

If you are running PVM applications across a cluster of machines that includes an Exemplar, you must start the ConvexPVM daemon, `pvmd3`, on the Exemplar. The first daemon started is the master daemon, and when an Exemplar is in the cluster, it must run the master in order to handle Exemplar-specific requirements.

Exemplar Intertask communication

ConvexPVM automatically routes messages between tasks running in one subcomplex on an Exemplar system via shared memory and the CTIrings. This greatly decreases message latency over traditional PVM socket-based communication. The amount of shared memory to reserve for communication is

determined by the `nf=n` hostfile option as described in the "New hostfile syntax" section.

Messages to other machines or to other subcomplexes are routed via sockets.

ConvexPVM buffering on Exemplar

On traditional, distributed-memory networks of machines running ConvexPVM, messages are packed into a send buffer in local memory by the sending task, sent over the network to the receiving task by the ConvexPVM daemon (unless a direct connection exists, in which case the daemon is circumvented), and placed in a receive buffer at the receiving end, from which they are unpacked by the receiving task.

The use of shared memory for intrasubcomplex communication on Exemplar systems greatly speeds this process. Rather than each task allocating send and receive buffers in private memory, the sending task packs its data into a reserved area of shared memory (the size of which is specified with the hostfile `nf=n` option). The receiving task can then unpack the data from shared memory. The ConvexPVM daemon never gets involved, and since shared memory is used to transfer the data, the actual transfer time is negligible compared to network-based message passing. In fact, packing and unpacking message data is the most time-consuming part of message passing on Exemplar systems.

Another advantage of shared memory message buffering is the larger buffer size available on Exemplar. Because the message buffer size is user-definable and limited only by the amount of shared memory available to the subcomplex, potentially many more messages can be pipelined by the sending task, allowing the task to continue with other work while the messages are being unpacked by the receiving tasks. On other systems, limited network bandwidth can sometimes block the sending task from doing parallel work while it waits to send data across the network.

When calling the `pvm_initsend`, `PVMFINITSEND`, `pvm_mkbuf`, or `PVMFMKBUF` functions in preparation for packing a message buffer, it is most efficient on Exemplar systems to specify no encoding in the `encoding` argument. In Fortran, use the constant `PVMRAW` to do this; in C, use `PvmDataRaw`. These values will enhance performance when sending messages between Exemplar and HP systems also.

Message passing approaches to parallelism

ConvexPVM programs generally take one of two approaches to parallelism: the master/slave approach, or the single program multiple data (SPMD) approach.

In the master/slave approach, a set of computational slave processes perform work for one or more master processes. This approach is generally used when little synchronization is required between tasks.

In the more common SPMD approach, the program spawns several identical tasks which perform the same work independently on different data sets. In this approach, synchronization is often required between parallel tasks. Exemplar systems running ConvexPVM are especially suited to this model because of their fast shared memory communication, which minimizes synchronization delays.

The following sections present both master/slave and SPMD examples. In both cases, the number of processors is hard coded at 4. These examples are presented in Fortran to simplify explanation; they can also be implemented in C.

Master/slave example

The example presented here uses the master/slave paradigm to implement the following matrix multiply algorithm:

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

To do this, the master program spawns four slave programs and broadcasts the same column of `B` to each. The slaves receive this column into a single dimensional array, and each uses it, along with a section of `A`, to simultaneously compute a disjoint, single-dimensional section of `C`. These sections of `C` are passed back to the master as they are computed; when the master receives sections of C from all four slaves, it broadcasts new columns of `B` back to them, and the process is repeated until all of `C` has been computed.

The master program follows.

```
      PROGRAM MASTER
      REAL*4 C(40,40),B(40,40)
      INTEGER TIDS(4),NPROCS,N,MY_NUM,J,I
      CHARACTER*4 ARCH
      CHARACTER*24 SLAVENAME
      CHARACTER*24 OUTFILE
      INCLUDE '/USR/INCLUDE/FPVM3.H'

      N = 40

C ENROLL UNDER PVM:
      CALL PVMFMYTID(MY_NUM)
      CALL INIT_OUTFILE

      NPROCS = 4
      SLAVENAME = "SLAVE"  !NAME OF SLAVE PROGRAM

C SPAWN SLAVE PROCESSES:
      CALL PVMFSPAWN(SLAVENAME,PVMDEFAULT,'*',NPROCS,TIDS,NUMT)
      IF(NUMT .LT. NPROCS) THEN     ! INDICATES SPAWN FAILURE
        WRITE(6,*) 'PVMFSPAWN FAILURE; NUMT =',NUMT
       ENDIF

C INITIALIZE B ARRAY:
      CALL INIT_ARRAY(B)

C CLEAR DEFAULT SEND BUFFER AND SPECIFY NO MESSAGE ENCODING:
      CALL PVMFINITSEND(PVMRAW,IBUFID)

      MSGTYPE = 1 ! INIT MSG WILL BE RECEIVED BY ALL SLAVES;
                  ! FOR MSGTYPE >= 2, EVEN MSGTYPE VALUES
                  ! INDICATE RAW DATA BEING SENT/RECEIVED; ODD
                  ! VALUES INDICATE RESULTS ARE BEING SENT/RECVD.

C PACK ARRAY DIMENSION SIZE:
      CALL PVMFPACK(INTEGER4,N,1,1,INFO)

C PACK NUMBER OF PROCESSORS:
      CALL PVMFPACK(INTEGER4,NPROCS,1,1,INFO)

C PACK TIDS ARRAY:
      CALL PVMFPACK(INTEGER4,TIDS,NPROCS,1,INFO)

C ALL INFO ON CURRENT MESSAGE BUFFER IS INIT INFO FOR ALL SLAVES;
C MULTICAST MESSAGE BUFFER TO ALL THREADS:
      CALL PVMFMCAST(NPROCS,TIDS,MSGTYPE,INFO)
```

```
C LOOP OVER ARRAY COLUMNS:
      DO J = 1,N
        MSGTYPE = MSGTYPE + 1 ! INCREMENT FOR THREAD-SPECIFIC MSGS
        CALL PVMFINITSEND(PVMRAW,IBUFID)

C       PACK COLUMN NUMER:
        CALL PVMFPACK(INTEGER4,J,1,1,INFO)

C       PACK CURRENT COLUMN OF B:
        CALL PVMFPACK(REAL4,B(1,J),N,1,INFO)

C       MULTICAST ARRAY COL; MSGTYPE HERE IS ALWAYS EVEN, INDICATING
C       RAW DATA:
        CALL PVMFMCAST(NPROCS,TIDS,MSGTYPE,INFO)

        MSGTYPE = MSGTYPE + 1   ! MSGTYPE BECOMES ODD
        NR = N/NPROCS    ! NUMBER OF COLS OF C TO RECEIVE BELOW
C LOOP TO RECEIVE RESULTS FROM EVERY PROCESSOR:
        DO I = 1,NPROCS

C         RECEIVE RESULTS FROM SLAVES:
          CALL PVMFRECV(-1,MSGTYPE,IBUFID)

C UNPACK ROW NUMBER:
          CALL PVMFUNPACK(INTEGER4,IROW,1,1,INFO)

C UNPACK COLUMN NUMBER:
          CALL PVMFUNPACK(INTEGER4,ICOL,1,1,INFO)

C UNPACK NR ARRAY DATA ELEMENTS STARTING AT IROW,ICOL:
          CALL PVMFUNPACK(REAL4,C(IROW,ICOL),NR,1,INFO)
        ENDDO
      ENDDO

      CALL OUTPUT(C,N)

      CALL PVMFEXIT(INFX)
      END
```

Here, the J loop loops over columns of B, broadcasting them to all slave processes. The I loop then loops over the number of slaves, insuring that all results are received before allowing J to continue by broadcasting another column of B.

Now consider the slave program:

```
      PROGRAM SLAVE
      DIMENSION A(10,40)
      DIMENSION B(40),CIJ(10)
      DIMENSION TIDS(4)
      INTEGER  MASTERNUM, IBUFID, NUMBYTES
      CHARACTER*17 OUTFILE
      INCLUDE '/USR/INCLUDE/FPVM3.H'

C ENROLL IN PVM AND GET TID:
      CALL PVMFMYTID(MY_NUM)

C CREATE OUTPUT FILE
      CALL INIT_OUTFILE

      MSGTYPE = 1   ! SET MSGTYPE TO RECEIVE INITIALILIZATION MSG

C ALL SLAVES RECEIVE INITIALIZATION MESSAGE:
      CALL PVMFRECV(-1,MSGTYPE,IBUFID)
      CALL PVMFBUFINFO (IBUFID, NUMBYTES, MSGTYPE, MASTERNUM, INFO)

C UNPACK ARRAY DIMENSION:
      CALL PVMFUNPACK(INTEGER4,N,1,1,INFO)

C UNPACK NUMBER OF PROCESSORS:
      CALL PVMFUNPACK(INTEGER4,NPROCS,1,1,INFO)

C UNPACK TIDS ARRAY:
      CALL PVMFUNPACK(INTEGER4,TIDS,NPROCS,1,INFO)

C DETERMINE NUMBER OF ROWS RECEIVED:
      NR = N / NPROCS

      CALL INIT_ARRAY(A)

C EACH SLAVE DETERMINES THE ROW NUMBER OF ITS DATA:
      DO I = 1,NPROCS  ! LOOP OVER ALL PROCESSORS
        IF(MY_NUM .EQ. TIDS(I)) THEN
          INUM = I - 1   ! SET ROW NUMBER ACCORDING TO TID
        ENDIF
      ENDDO

      IROW = INUM  * NR  ! DETERMINE UPPER LIMIT FOR ROWS
```

```
C LOOP OVER COLUMNS; EACH SLAVE RECEIVES, PROCESSES AND RETURNS DATA
C IN THIS LOOP:
      DO J = 1,N
        MSGTYPE = MSGTYPE + 1 !MSGTYPE BECOMES EVEN FOR RAW DATA

C     RECEIVE DATA FROM MASTER:
        CALL PVMFRECV(MASTERNUM,MSGTYPE,IBUFID)

C     UNPACK COLUMN NUMBER:
        CALL PVMFUNPACK(INTEGER4,ICOL,1,1,INFO)

C     UNPACK N DATA ELEMENTS OF B; NOTE COLUMNS ARE HANDLED AS
C     SINGLE-DIMENSIONAL ARRAYS IN SLAVE PROCESSES:
        CALL PVMFUNPACK(REAL4,B(1),N,1,INFO)

C     PERFORM MATRIX MULTIPLY:
        DO I = 1,NR
          CIJ(I) = 0.0
        ENDDO
        DO K = 1,N
          DO I = 1,NR
            CIJ(I) = CIJ(I) + A(I,K) * B(K)
          ENDDO
        ENDDO

C     RETURN RESULTS FOR THIS COLUMN TO MASTER:
        MSGTYPE = MSGTYPE + 1
        CALL PVMFINITSEND(PVMRAW,IBUFID)

C     PACK ROW NUMBER:
        CALL PVMFPACK(INTEGER4,IROW+1,1,1,INFO)

C     PACK COLUMN NUMBER:
        CALL PVMFPACK(INTEGER4,ICOL,1,1,INFO)

C     PACK NR COMPUTED ELEMENTS OF C:
        CALL PVMFPACK(REAL4,CIJ(1),NR,1,INFO)

C     SEND BUFFER TO MASTER PROGRAM:
        CALL PVMFSEND(MASTERNUM,MSGTYPE,INFO)
      ENDDO

      CALL PVMFEXIT(INFX)
      END
```

Here, the J loop loops over the same space as the J loop in the master program. Within the loop, the slave receives columns of B into a single-dimensional array which it uses in the

computation of an NR-row section of C. When these sections are computed, the slaves send them back, get another column of B, and the process is repeated.

SPMD example

The example presented here uses the SPMD paradigm to perform a matrix addition. In it, the main program spawns a process for each processor, then the spawning process splits up the arrays into chunks and distribute them to the child processes. Both parent and child processes are controlled by the same program; the correct code is executed based on the presence or absence of a parent process, since only child processes have parents.

The main program follows.

```
      PROGRAM SPMD
      IMPLICIT NONE
      INCLUDE "/USR/INCLUDE/FPVM3.H"

      INTEGER NPROC, MAXROWS, MAXCOLS, MAXPROCS

      PARAMETER (NPROC = 4, MAXROWS = 20, MAXCOLS = 20)
      INTEGER A(MAXROWS,MAXCOLS), B(MAXROWS,MAXCOLS)
      INTEGER C(MAXROWS, MAXCOLS)
      COMMON / ARRAYBLK / A, B, C
      CHARACTER*128 ERRMSG
      CHARACTER*8 SPMD_EXE
      INTEGER MYTID, NTASKS, TIDS(0:NPROC-1), I, J, TID
      INTEGER STATUS, MYPROC, CHUNKID, BUFID, NUMBYTES

C ENROLL THE UNDER PVM:
      CALL PVMFMYTID (MYTID)

C DETERMINE IF RUNNING PROCESS IS PARENT OR CHILD:
      CALL PVMFPARENT(TIDS(0)) ! ONLY SPAWNED PROCESSES HAVE PARENTS
      IF(TIDS(0) .EQ. PVMNOPARENT) THEN ! THIS PROCESS IS PARENT
        TIDS(0) = MYTID     ! SO PUT ITS ID INTO TIDS(0)
        MYPROC = 0

C     SPAWN A COPY FOR EACH PROCESSOR:
        SPMD_EXE = 'SPMD.EXE'   ! NAME OF THIS PROGRAM
        CALL PVMFSPAWN(SPMD_EXE,PVMDEFAULT,'*',NPROC-1,TIDS(1),
     1                 NTASKS)
```

```
C     INITIALIZE ARRAYS A, B, AND C:
        CALL INIT_ARRAYS()

C     LOOP OVER SPAWNED TASKS:
        DO I = 2, NPROC
C       INITIALIZE SEND BUFFER:
          CALL PVMFINITSEND(PVMDEFAULT, STATUS)
C       PACK TIDS ARRAY ON SEND BUFFER:
          CALL PVMFPACK (INTEGER4, TIDS, NPROC, 1, STATUS)

C       NOTE: 5X20 SECTIONS OF A AND B ARE PACKED FOR EACH CHILD:
C       PACK A ARRAY SECTION ON SEND BUFFER:
          CALL PVMFPACK (INTEGER4, A(1, (I - 1) * 5 + 1),
     1                   MAXROWS * MAXCOLS / NPROC, 1, STATUS)

C       PACK B ARRAY SECTION ON SEND BUFFER:
          CALL PVMFPACK (INTEGER4, B(1, (I - 1) * 5 + 1),
     1                   MAXROWS * MAXCOLS / NPROC, 1, STATUS)

C       SET CHUNKID TO NUMBER OF CURRENT ITERATION (CHUNKID IS USED
C       BY SENDS AND RECEIVES TO DETERMINE WHERE A GIVEN ARRAY
C       CHUNK FITS INTO WHOLE ARRAY):
          CHUNKID = I

C       SEND MESSAGE BUFFER TO THE SPAWNED PROCESSES:
          CALL PVMFSEND (TIDS(I-1), CHUNKID, STATUS)
        END DO
      ELSE      ! THIS PROCESS IS A CHILD
C     NOTE NO LOOP REQ'D; EVERY CHILD RUNS THIS BRANCH IN PARALLEL
C     RECEIVE THE MESSAGE FROM THE PARENT PROCESS AND UNPACK IT:
        CALL PVMFRECV (TIDS(0), -1, BUFID)

C     GET INFORMATION ABOUT THE MESSAGE BUFFER RECEIVED
C     (EACH CHILD PROCESS RECEIVES A UNIQUE BUFFER)
        CALL PVMFBUFINFO (BUFID, NUMBYTES, CHUNKID, TID, STATUS)
        IF (STATUS .LT. 0) THEN
          ERRMSG = 'CHILD:PVMBUFINFO FAILED'
          WRITE (6,*) 'STATUS = ', STATUS
          CALL PROGRAM_EXIT (ERRMSG, MYTID, TIDS)
        ELSE IF ((CHUNKID .LT. 2) .OR. (CHUNKID .GT. NPROC)) THEN
C     MAKE SURE CHUNKID IS VALID FOR CHILD PROCESS
          ERRMSG = 'CHILD:PVMBUFINFO BOGUS MESSAGE TYPE'
          CALL PROGRAM_EXIT (ERRMSG, MYTID, TIDS)
        END IF
```

```
C     MESSAGE BUFFER IS OKAY; UNPACK:
C       UNPACK TIDS ARRAY:
        CALL PVMFUNPACK(INTEGER4,TIDS,NPROC,1,STATUS)

C       UNPACK ARRAY A:
        CALL PVMFUNPACK(INTEGER4,A,(MAXROWS*MAXCOLS)/NPROC,1,STATUS)

C       UNPACK ARRAY B:
        CALL PVMFUNPACK(INTEGER4,B,(MAXROWS*MAXCOLS)/NPROC,1,STATUS)

C       FIND CURRENT PROCESS'S TID IN THE TID ARRAY:
        DO I = 1, NPROC-1
          IF (MYTID .EQ. TIDS(I)) THEN
            MYPROC = I   ! ASSIGN PROCESS NUMBER FOR CURRENT PROCESS
            GOTO 100     ! AND EXIT LOOP
          END IF
        END DO

100     IF (MYPROC .LT. 1) THEN
          ERRMSG = 'CHILD:BOGUS PROCESS NUMBER'
          CALL PROGRAM_EXIT (ERRMSG, MYTID, TIDS)
        END IF

      ENDIF

C ALL 'NPROC' PROCESSES ARE EQUAL NOW AND CAN BE ADDRESSED BY
C TIDS(0) THRU TIDS(NPROC-1)

C CALL COMPUTATION ROUTINE:
      CALL DOWORK(MYPROC, TIDS, CHUNKID, NPROC)

C TERMINATE THIS PROGRAM.
      CALL PVMFEXIT (STATUS)
      CALL EXIT (0)
      END
```

Here, after spawning the specified number of child processes, the main program sends chunks of the A and B arrays to the appropriate children, and keeps a chunk of each for itself. The main program also contains code run by both parent and children which unpacks the array chunks and calls the DOWORK routine to perform the matrix addition.

The DOWORK subroutine is shown below.

```
      SUBROUTINE DOWORK(MYPROC, TIDS, SENT_CHUNKID, NPROC)

C THIS ROUTINE IS CALLED BY EACH PROCESS (INCLUDING PARENT) TO DO
C MATRIX ADDITION ON THE PROCESS'S CHUNKS OF ARRAYS A, B AND C.
C AFTER CALCULATION, THE PARENT COLLECTS & OUTPUT RESULTS

      IMPLICIT NONE
      INCLUDE "/USR/INCLUDE/FPVM3.H"
      INTEGER NPROC, MAXROWS, MAXCOLS
      PARAMETER (MAXROWS = 20, MAXCOLS = 20)
      INTEGER A(MAXROWS, MAXCOLS), B(MAXROWS, MAXCOLS)
      INTEGER C(MAXROWS, MAXCOLS)
      COMMON / ARRAYBLK / A, B, C
      CHARACTER*128 ERRMSG
      INTEGER TIDS(0:NPROC-1), I, J, TID
      INTEGER STATUS,MYPROC,CHUNKID,BUFID,NUMBYTES,SENT_CHUNKID

C PERFORM MATRIX ADDITION USING CURRENT PROCESS'S SECTIONS:
      DO J = 1, MAXCOLS / NPROC
        DO I = 1, MAXROWS
          C(I, J) = A(I, J) + B(I, J)
        END DO
      END DO

C NOW CHILDREN SEND ARRAY SECTIONS TO PARENT, WHICH COLLECTS THEM:
      IF (MYPROC .EQ. 0) THEN ! CURRENT PROCESS IS PARENT
        DO I = 1, NPROC-1  ! LOOP OVER ALL PROCESS NUMBERS

C       MESSAGES BEING RETURNED FROM THE CHILD TASKS WILL HAVE
C       THE VALUE (SENT_CHUNKID + NPROC):
          CALL PVMFRECV (-1, -1, BUFID)

C         GET INFORMATION ABOUT THE LAST MESSAGE BUFFER RECEIVED:
          CALL PVMFBUFINFO (BUFID, NUMBYTES, CHUNKID, TID, STATUS)
          IF (STATUS .LT. 0) THEN
            ERRMSG = 'PARENT:PVMFBUFINFO ERROR'
            CALL PROGRAM_EXIT (ERRMSG, TIDS(MYPROC), TIDS)
          ELSE IF(NUMBYTES .NE. (MAXROWS*MAXCOLS*4)/NPROC) THEN
C           CHECK THE LENGTH OF THE MESSAGE AND THE MESSAGE ID:
            ERRMSG = 'PARENT:PVMFBUFINFO ERROR'
            CALL PROGRAM_EXIT (ERRMSG, TIDS(MYPROC), TIDS)
            ELSE IF((CHUNKID.LT.6).OR.(CHUNKID.GT.2*NPROC)) THEN
C             MAKE SURE CHUNKID IS VALID FOR CHILD PROCESS:
              ERRMSG = 'PARENT:PVMFBUFINFO ERROR'
              CALL PROGRAM_EXIT (ERRMSG, TIDS(MYPROC), TIDS)
            END IF
```

```
C       EXTRACT ARRAY C (RESULT OF MATRIX ADDITION) FROM BUFFER;
C       NOTE CHUNKID AND NPROC ARE USED TO DETERMINE WHAT SECTION OF
C       ARRAY IS BEING UNPACKED:
            CALL PVMFUNPACK(INTEGER4,C(1,(CHUNKID-NPROC-1)*5+1),
     1                      (MAXROWS*MAXCOLS)/NPROC,1,STATUS)
          END DO
        ELSE    ! CURRENT PROCESS IS CHILD

C       SEND ARRAY C (RESULT ARRAY) SECTION BACK TO PARENT:
          CALL PVMFINITSEND(PVMDEFAULT,STATUS) !INITIALIZE SEND BUFFER

C       PACK C ARRAY SECTION ON SEND BUFFER:
          CALL PVMFPACK(INTEGER4,C,(MAXROWS*MAXCOLS)/NPROC,1,STATUS)

C       ENCODE ARRAY'S CHUNKID SO THAT PARENT'S UNPACK CAN DETERMINE
C       WHICH CHUNK IT'S RECEIVING:
          CHUNKID = SENT_CHUNKID + NPROC

C       SEND BUFFER TO PARENT PROCESS:
          CALL PVMFSEND (TIDS(0), CHUNKID, STATUS)
        END IF

C PARENT OUTPUTS RESULTS:
        IF (MYPROC .EQ. 0) CALL OUTPUT()
          RETURN
        END
```

In DOWORK, the matrix addition is performed; then, in the ELSE clause of the following IF construct, the child processes send their chunks of the array C back to the parent. In the IF clause, the parent unpacks them and reassembles the entire C array. After this is done, the parent outputs the results.

Limits of optimization

8

This chapter discusses common optimization problems you may encounter when developing programs for SPP Series machines, 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:

- Erroneous (nonstandard) code
- Floating-point imprecision (roundoff error)
- Misused directives and options
- Misused memory classes
- Compiler limitations

Note

The 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.

Erroneous code

The most common cause of answers that change with optimization is erroneous code in the form of aliases and invalid subscripts.

Aliases

As described in the "Inhibitors of localization" section of Chapter 3, 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 -O2 and above. However, code motion can also cause aliasing problems, at optimization levels -O1 and above.

Consider the following Fortran program.

```
PROGRAM ALIAS
INTEGER I
COMMON /DATA/I
I = 666
CALL CONFUSED(I)
END

SUBROUTINE CONFUSED(N1)
DO I = 1, 2
   N2 = 3 * (N1 + 1)
   CALL CALC(N2)
   WRITE(*,*)'Iteration:', I, ', n1 = ', N1
   WRITE(*,*)'Iteration:', I, ', n2 = ', N2
ENDDO
RETURN
END

SUBROUTINE CALC(N)
INTEGER K, N
COMMON /DATA/ K
K = N + 1
RETURN
END
```

In the subroutine CONFUSED, the compiler assumes that N1 is invariant. This is not true, as N1 is in the DATA common block, where it is manipulated by the CALC subroutine. The right side of the assignment to N2 appears to be invariant, so the compiler moves the assignment to N2 out of the loop at optimization levels

-O1 and higher. When compiled at -O1 or above, the program produces incorrect answers.

The results of this program compiled and run at optimization levels -O0 and -O1 are shown below. Note that the answers are changed at optimization level -O1.

```
% fc -O0 alias.f -o O0.out
% O0.out
Iteration: 1, n1 = 2002
Iteration: 1, n2 = 2001
Iteration: 2, n1 = 6010
Iteration: 2, n2 = 6009

% fc -O1 alias.f -o O1.out
% O1.out
Iteration: 1, n1 = 2002
Iteration: 1, n2 = 2001
Iteration: 2, n1 = 2002
Iteration: 2, n2 = 2001
```

Aliases in C

Because they frequently use pointers, C programs are especially susceptible to aliasing problems. The C compiler has two different algorithms for detecting potential aliases: a "worst-case" algorithm, which is used in backward-compatible (-pcc) mode, and an ANSI-C aliasing algorithm, which is used in the ANSI-C compatibility modes (-ext, and -std).

The worst-case aliasing algorithm views all pointer references as subscripts into a giant array that encompasses all of memory. This mythical array (known as *MEM* in the compiler's optimization report) includes all global variables and local variables whose address is taken using the address (&) operator. This results in the following conditions:

- Every pointer reference is a potential alias for every other pointer reference.
- Every pointer reference is a potential alias for every global variable (and vice versa).
- Every pointer reference is a potential alias for every local variable that is used with &.

ANSI C provides stricter type-checking. This allows the C compiler to use a stricter algorithm that eliminates many potential aliases found by the worst-case algorithm. Instead of one giant *MEM* array, the ANSI-C algorithm uses a model with separate arrays for each base type (such as int, float, or

double). Pointers and variables cannot alias with pointers or variables of a different base type.

The ANSI aliasing algorithm permits the compiler to parallelize the code shown below; the worst-case algorithm does not.

```
void alex1(a, ib, n)
float *a;
int *ib, n;
{
   int i;
   for ( i=0; i<n; i++ )
      a[i] = ib[i] + n;
}
```

Pointers `a` and `ib` point to different base types. The worst-case algorithm considers them to be aliased through `*MEM*`, but under ANSI C rules, no potential alias exists, and the loop is parallelized.

The ANSI C aliasing algorithm may not be safe if your program is not ANSI compliant. The non-ANSI-compliant code shown below allows two pointers of different base types to become aliased with one another. (The assignment to `fptr` makes the code non-ANSI C compliant.)

```
int array[100];
int *dummy;
float *fptr;

*dummy = array;    /* illegal pointer/integer
                      combination. */
fptr = dummy;      /* operands of = point to
                      incompatible types. */
```

Compiling this code generates the warning messages shown in the comments. (Compiling with `-d arg_ptr_ref=e` converts these warnings into error messages.)

Because of the ANSI standard violation, this code does not compile safely under the ANSI aliasing algorithm.

In the following example, function `caller` passes two `long int` pointers to `foo`, which expects one pointer to `long int` and one pointer to `short int`. Because `ia` and `ib` have different base types, the ANSI C aliasing algorithm assumes that no alias between `*ia` and `*ib` can exist.

At optimization level -O3 in ANSI mode, the compiler compiles this code (with a warning) and parallelizes the loop even though aliasing and a recurrence do exist. In -pcc mode, the code will not compile.

```
void foo(long int *ia, short int *fb)
{
   int i;
   for (i=0; i<500; i++ ) {
      ia[i]=1;
      fb[i] = ia[i];
   }
}
caller()
{
   long int arra[600];
   foo(&arra[0], &arra[100]);
}
```

To specify an aliasing mode, use one of the following options on the cc command line:

- -alias cautious
- -alias standard
- -alias worst

In cautious mode, the compiler uses the ANSI C aliasing algorithm. If the compiler finds constructs that could cause hidden aliasing, it switches to the worst-case aliasing algorithm. If you do not specify an aliasing mode, the compiler uses cautious mode by default.

In standard mode, the compiler always uses the ANSI-C aliasing algorithm.

In worst mode, the compiler uses the worst-case aliasing algorithm.

These and other C aliasing options are further discussed in Appendix B, "Optimization options."

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.

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 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
      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$DIR NEXT_TASK
      DO I = 1, 10000
        X = C(I) + D(I)
      ENDDO
C$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
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 = 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, or `REAL*8` rather than `REAL*4` in Fortran). It is always poor practice to test floating point numbers for exact equality.

Disabling underflow traps

By default, PA-RISC processor hardware traps floating point underflow when a floating point result 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 *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*). This practice is extremely costly in terms of execution time and seldom provides any benefit. You can disable this behavior by passing the `+FPD` flag to the loader. This is done using the `-W` compiler option on either the `cc` or `fc` command line. The following example shows such an `fc` command line.

```
fc -Wl,+FPD prog.f
```

This command line compiles the program `prog.f` and instructs the loader to disable floating point underflow. For more information on the `-W` option, refer to the *Fortran User's Guide* or *C User's Guide*.

Misused directives, pragmas and options

Misused directives and pragmas are a common cause of wrong answers. For example, forcing parallelization of a loop containing a call is safe only if the called routine contains no dependencies.

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 dependency (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 "Inhibitors of localization" section of Chapter 3.

You cannot safely parallelize a loop that contains any kind of LCD. This is discussed further in the "Inhibitors of parallelization" section of Chapter 3.

The MAIN section of the Fortran program below initializes A, calls CALC, and outputs the new array values. In subroutine CALC, the indirect index used in A(IN(I)) introduces a potential dependency 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
```

The analogous C code looks like this:

```
float arra[1025];

void calc(int IN[])
{
  int i,j;

    for(i = 0; i <= 1024; i++)
      arra[i] = arra[IN[i]];
}
main()
{
  int i,j,IN[1024];

  for(i = 0; i <= 1024; 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 at optimization level `-O3`.

```
      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 <= 1024; i++)
        arra[i] = arra[IN[i]]
}
```

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)
      DO I = 1, NUM_THREADS()
        DO J = 1, N
          WRONGTP(J) = ... ! ONLY EXISTS FOR
          .                ! THREAD 0
          .
          .
        ENDDO
```

And the analogous C example:

```
/* 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)
for(I=0;I<num_threads();I++) {
  for(J=0;J<N;J++) {
    WRONGTP[J] = ... /* only exists for thread 0 */
    .
    .
    .
  }
}
```

Here, the array `WRONGTP` is allocated, but since 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 shown in the example on page 134.

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
```

And the analogous C example:

```
/* 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);
  .
  .
  .
}
```

Recall from Chapter 5 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.

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, 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, as explained in Chapter 5, when accessing node_private arrays from node-parallel code, far_shared pointers must be used. Failing to do this will render the copies of the arrays on all but hypernode 0 inaccessible. Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!!
      REAL*8 WRONGNP(:)
C$DIR NODE_PRIVATE(N0NP)
      ALLOCATABLE(N0NP)
      .
      .
      .
      ALLOCATE(N0NP(M))
      N = NUM_NODES
C$DIR LOOP_PARALLEL(NODES)
      DO I = 1, N
        DO J = 1, M
          N0NP(J) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

And the analogous C example:

```
/* 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)
for(i=0;i<N;i++) {
  for(J=0;j<M;J++) {
    N0NP[J] = ...
    .
    .
    .
  }
}
```

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 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 J loop, which is running in parallel on several hypernodes, these other hypernodes attempt to access N0NP, they will do so using the garbage contents of their uninitialized pointers, typically causing a runtime error.

Chapter 5 covers correct pointer/data combinations, and explains the situations in which non-default pointers should be used. To avoid uninitialized pointer problems such as the one described above, follow the recommendations of Chapter 5 carefully.

Hidden dependencies

Improperly accessing a shared variable from parallel threads can create a dependency that the compiler cannot find. 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 compiler would find the dependency and avoid generating parallel code for the tasks. But the compiler cannot find the dependency on `HOLD` since it is in another procedure, so the results of this code are indeterminate. 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 task_private float x,y;
  #pragma _CNX begin_tasks
  x = ...;
  .
  .
  .
  addhold(&hold,x);
  #pragma _CNX next_task
  y = ...;
  .
  .
  .
  addhold(&hold,y);
  #pragma _CNX end_tasks
}
```

Compiler limitations

Compiler limitations can produce faulty optimized code when the source code contains:

- Reductions
- Different possible evaluation orderings
- Iterations by zero
- Nondeterminism of parallel execution
- Replaceable loop test variables
- Trip counts greater than $2^{31} - 1$ at optimization levels `-O2` and `-O3`
- Hidden ordered sections

Reductions

Reductions are a special class of dependency 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 dependency between the references to `A(I)` on line 4

and the assignment to A(JA(J)) on line 5. 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.

```
DATA JA /11,12,13,14,15,16,17,18,19,20/
DO I = 1, 10
   DO J = I, 10
      A(I) = A(I) + B(J) * C(J)    !line 4
      A(JA(J)) = B(J) + C(J)       !line 5
   ENDDO
ENDDO
```

In the following C example, the apparent dependency is between the reference to A[I] on line 4 and A[JA[J]] on line 5.

```
JA[] = {11,12,13,14,15,16,17,18,19,20};
for (I=0; I<10; I++)
   for (J=0; J<10; J++) {
      A[I] += B[J] * C[J];     /* line 4 */
      A[JA[J]] = B[J] + C[J]; /* line 5 */
   }
```

Note

In both these examples as well as the examples that follow, the apparent dependence becomes real if any of the values of the elements of JA are equal 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 4; the apparent dependence on line 5 is properly handled because of the directive. The resulting code runs fast but produces incorrect answers.

To solve this problem, distribute the J loop, isolating the reduction from the other statements, as shown in the following Fortran example.

```
DATA JA/11,12,13,14,15,16,17,18,19,20/
DO I = 1, 10
  DO J = I, 10
    A(I) = A(I) + B(J) * C(J)
  ENDDO
ENDDO
```

```
C$DIR NO_LOOP_DEPENDENCE(A)
      DO I = 1, 10
        DO J = I, 10
          A(JA(J)) = B(J) + C(J)
        ENDDO
      ENDDO
```

And in C:

```
for (I=0; I<10; I++)
  for (J=I; J<10; J++)
    A[I] += B[J] * C[J];

#pragma _CNX no_loop_dependence(A)
for (I=0; I<10; I++)
  for (J=I; J<10; J++)
    A[JA[J]] = B[J] + C[J];
```

The apparent dependency is removed, and both loops can be optimized.

This problem occurs only if the reduction and the apparent dependency involve the same variable or array element. If the reduction and the apparent dependency involve different variables or array elements, as in the following Fortran example, both reduction and dependency are handled correctly without your intervention.

```
      DATA JD /6, 7, 8, 9, 10/
      DO I = 1, 5
C$DIR NO_LOOP_DEPENDENCE(D)
         DO J = I, 5
            A(I) = A(I) + B(J) * C(J)
            D(JD(J)) = D(I) + B(J) + C(J)
         ENDDO
      ENDDO
```

In C:

```
jd[] = {6,7,8,9,10};
for ( i=0; i<5; i++ )
#  pragma _CNX no_loop_dependence(D)
   for ( j=0; j<5; j++ ){
      A[i] += B[j] * C[j];
      D[jd[j]] = D[i] + B[j] + C[j];
   }
```

Evaluation order

Assumptions the compiler makes about reordering code can sometimes cause answers to change at higher optimization levels. If this happens, use parentheses to force a specific order of evaluation.

Incrementing by 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 previously described symptoms occur. The following Fortran example shows three loops that increment by zero.

```
CALL SUB1(0)
.
.
.
SUBROUTINE SUB1(IZR)
DIMENSION A(100), B(100), C(100)

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

DO I = 1, N, IZR    ! INCREMENT VALUE OF 0 IS
                    ! NON-STANDARD
   A(I) = B(I)
ENDDO

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

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++ ){
    b[i] = a[j];
    a[j] = c[i];
    j += IZR;
  }
  for(i=0; i<N; i+=IZR)
    a[i] = b[i];
  for(i=0; i<N;i++) {
    j = j + IZR;
    b[i] = a[j];
    a[j] = c[i];
  }
}

main()
{
  SUB1(0);
}
```

Because `IZR` is an argument passed to `SUB1`, the compiler does not detect that `IZR` has been set to zero. All three loops parallelize at `-O3`, but because of the zero increments, their runtime behavior cannot be reliably predicted. All three loops compile at `-O1`, but the second loop, which specifies the step as part of the `DO` statement (or as part of the `for` statement in C), will cause a runtime error. Runtime behavior of the other two loops cannot be predicted at `-O1`.

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 dependency 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 dependency on `A(I)`. This dependency 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 insure 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 dependencies can sometimes be manually parallelized using the `LOOP_PARALLEL(ORDERED)` directive as described in Chapter 6. Otherwise, unless you are sure that no loop carried dependency exists, it is safest to let the compiler choose which loops to parallelize.

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 number of times the loop executes is called the *trip count* (`N` in the example), and the initial value of the induction variable is the *start* value.

The following C function also contains an induction variable that is 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` is used as the induction variable rather than `ind`, again producing a stride of 2^{22}.

Test replacement, a standard optimization at levels `-O1` 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 default (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.

When the trip count is a constant, the compiler can check *start+((trip-1)*stride)* for overflow at compile time and catch this error. However, if the trip count is a variable, no compile-time checking is done, and so large trip and stride combinations can cause the loop to terminate prematurely.

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:

$$start + ((trip - 1) * stride) \quad \leq 2^{31}$$

The start value is 1, so *trip* can be solved for as follows:

$$\begin{aligned} start + ((trip - 1) * stride) &\leq 2^{31} \\ 1 + (trip - 1) * 2^{22} &\leq 2^{31} \\ (trip - 1) * 2^{22} &\leq 2^{31} - 1 \\ trip - 1 &\leq 2^{9} - 2^{-22} \\ trip &\leq 2^{9} - 2^{-22} + 1 \\ trip &\leq 512 \end{aligned}$$

The maximum value for `n` in the given loop, then, is 512.

If you find that certain loops give wrong answers at optimization levels `-O1` or higher, the problem may be test replacement. If you still want to optimize these loops at `-O1` or above, restructure them to force the compiler to choose a different induction variable.

Large trip counts at `-O1` and above

When a loop is optimized at level `-O1` 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). If the compiler can determine that the trip count is larger than this at compile time, it will issue a warning. 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 `-O1` and above.

Loops with trip counts that overflow 32 bits can be optimized by manually strip mining the loop.

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$DIR BEGIN_TASKS
      CALL SUB1(A)
      .
      .
      .
C$DIR NEXT_TASK
      CALL SUBN
      .
      .
      .
C$DIR END_TASKS
      .
      .
      .
      END

      SUBROUTINE SUB1(A)
      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
```

Here, the tasks in the main program go thread parallel by default, so when the loop in SUB1 is reached it cannot go parallel. However, because parallelism exists in the main program, the ordered section in SUB2 expects that it will be executed by all parallel threads. Only thread 0 is executing SUB1, since only the first ordered 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 causes the program to deadlock in the ordered section.

The analogous C code follows:

```
void sub1(float *a) {
  static gate_t lock;
  int lk;
  lk = alloc_gate(lock);
#pragma _CNX loop_parallel(ordered, ivar=1)
  for(i=0;i<100;i++)
    sub2(lock,a,i);
  lk = free_gate(lock);
}

void sub2(gate_t lock, float *a, int i) {
#pragma ordered_section(lock)
  a[i] = a[i-1];
#pragma end_ordered_section
}
```

```
main() {
  float a[100];
  .
  .
  .
#pragma _CNX begin_tasks
  sub1(a[]);
  .
  .
  .
#pragma _CNX next_task
  subn();
  .
  .
  .
#pragma _CNX end_tasks
  .
  .
  .
}
```

If you encounter this kind of problem, try moving the ordered section into the same routine as its parent loop, or, if possible, add another dimension of parallelism so that the loop running the ordered section is running in parallel.

Potentially unsafe optimizations

9

This chapter describes optimizations that can potentially generate incorrect results. By default, the SPP Series Fortran and C compilers avoid performing these optimizations. You can enable potentially unsafe optimizations by specifying the `-uo` compiler option.

The `-uo` option enables the compiler to perform these optimizations:

- Simple strength reductions
- Code motion
- Elimination of type conversions

Simple strength reduction

Chapter 3, "Compiler optimizations," describes how the compiler replaces slow operations with faster ones on the assumption that arithmetically equivalent expressions always yield the same results. However, reducing an expression such as `X/C` to `(1/C)*X` can be unsafe because it can increase roundoff error.

When you use the `-uo` option, the compiler replaces division operations with multiplication. If a possibility of overflow exists, however, the compiler does not perform this optimization.

Code motion

The compiler normally moves an invariant expression out of a loop only if the expression is located on a path to all loop exits. When you use `-uo`, the compiler can move an invariant

expression out of a loop if the expression does not lie on a path to all loop exits.

In the following example, the invariant expression `A=B/X` is relocated only when the program is compiled with the `-uo` option.

```
DO I = 1, 100
   IF (X .NE. 0) THEN
      A= B/X
      AR(I) = A*C
   ELSE
      AR(I) = D*C
   ENDIF
ENDDO
```

Conversion elimination

Type conversions are costly in terms of machine cycles, and they can inhibit optimization. When you use the `-uo` option, the compiler eliminates costly type conversions by creating `REAL` induction variables that it then increments concurrently with the loop's `INTEGER` induction variables. Consider the following Fortran loop.

```
REAL A(100000)
.
.
.
DO I = 1, 100000
   A(I) = I
ENDDO
```

And its C equivalent:

```
float a[100000];

int foo()
{
   int I;

   for( I=0; I<100000; I++ )
      a[I] = I;
}
```

Here, in absence of the `-uo` option, `I` must be converted to floating point on every iteration of the loop. With the `-uo` option the compiler avoids this costly operation by copying `I` into a `REAL` (or `float`) induction variable before entering the loop, then

incrementing this variable by `1.0` on every iteration of the loop. At optimization level `-O3`, the compiler can then parallelize the following optimized Fortran loop.

```
      REAL_I = 1.0
      I = 1
10    A(I) = REAL_I
      REAL_I = REAL_I + 1.0
      I = I + 1
      IF (I .LE. 100000) GOTO 10
```

In C:

```
float a[100000];

int foo()
{
    int I;
    float REAL_I;

    for(I=0,REAL_I=0.0;I<100000;I++,REAL_I++ )
        a[I] = REAL_I;
}
```

This optimization is considered potentially unsafe because the internal representation of real numbers is inexact, and this can lead to a significant accumulated error when `REAL_I` is incremented over the course of the loop.

Compiler directives and pragmas

A

This appendix presents an alphabetical list of all the Fortran directives and C pragmas that are supported by the CONVEX SPP Series compilers.

This appendix is intended to provide only a brief overview of the available directives and pragmas. More specific information and examples can be found elsewhere in this guide. The Fortran directives not supported as C pragmas are expressed as storage class extensions (`thread_private`, etc.) or as typedefs (`gate_t`, `barrier_t`, etc.), and are described in chapters 5 and 6.

The form of an SPP Series Fortran compiler directive is:

`C$DIR [CSERIES|SPP]` *directive-specification*

The form of an SPP Series C pragma is:

`#pragma _CNX [CSERIES|SPP]` *directive-specification*

Where *directive-specification* is one of the directives/pragmas described in this chapter. `CSERIES` or `SPP` can optionally be included to indicate the target machine for the directive; this is useful if you are writing code that may be compiled on either C Series or Exemplar architectures, and wish to include directives or pragmas for both without having the compiler issue warnings for those that do not apply to the current machine.

Directive names are presented here in lower case; they may be specified in either case in both languages, but "`#pragma`" must always appear in lower case in C. In the sections that follow, *namelist* represents a comma delimited 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 lower-case *n* is used to indicate a compile-time integer constant expression; values not determinable at compile time cannot be used. Occurrences of *gate_var* are for variables that have been, or are being, defined as gates. Any parameters that appear within square brackets ([and]) are optional.

barrier(*namelist*)

This Fortran directive is used to denote 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. Note that in C, barrier is a typedef, rather than a pragma. For more information, refer to Chapter 6.

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;

- (`max_threads=`*m*)
- (`ordered`)
- (`nodes`)
- (`threads`)
- (`ordered,nodes`)
- (`ordered,threads`)
- (`ordered,max_threads=`*m*)
- (`nodes,max_threads=`*m*)
- (`threads,max_threads=`*m*)
- (`ordered,nodes,max_threads=`*m*)
- (`ordered,threads,max_threads=`*m*)

Refer to chapters 4 and 6 for a complete discussion of parallel tasking

block_loop[(block_factor=*n*)]

This directive or pragma is used to indicate a specific loop to block, and optionally, the block factor that will be used in the compiler's internal computation of loop nest based data reuse. In absence of the `block_factor` argument, this directive is useful for indicating which loop in a nest the compiler should block. Refer to Chapter 3 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 subcomplex. If the user specified size is not an integral multiple of page size × `num_nodes()` then the compiler will automatically round it up to meet this criterion. Refer to Chapter 5 for more information on memory classes.

critical_section[(*gate_var*)]

This 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 chapters 4 and 6 for more information.

end_critical_section

This directive or pragma is used to define 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 chapters 4 and 6 for more information.

end_ordered_section

This directive or pragma is used to define 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_tasks`

This is used to terminate 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 chapters 4 and 6 for more information.

`far_shared(`*namelist*`)`

This Fortran directive causes the compiler to place the data objects in *namelist* (i.e., 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 subcomplex. The `far_shared` data objects of a process are addressable by all threads of that process. Refer to Chapter 5 for more information on memory classes.

`far_shared_pointer(`*alloc_var_name*`)`

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `far_shared` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5 for more information on memory classes.

`gate(`*namelist*`)`

This Fortran directive defines a gate variable that will be subsequently used in a critical section, ordered section, or passed as an argument to the synchronization intrinsics. Note that in C, `gate` is a typedef, rather than a pragma. Refer to Chapter 6, "Advanced shared memory programming," for more information.

`loop_parallel[(`*attribute_list*`)]`

This is an explicit instruction to the compiler to parallelize the immediately following loop. The loop iterations will be run in an undetermined order unless the optional `ordered` attribute

appears. The user is responsible for any required data privatization and loop synchronization, as described in chapters 4 and 6. The optional *attribute_list* can be any of the following combinations:

- `(chunk_size=`*n*`)`
- `(max_threads=`*m*`)`
- `(ordered)`
- `(nodes)`
- `(threads)`
- `(ordered,nodes)`
- `(ordered,threads)`
- `(nodes,chunk_size=`*n*`)`
- `(threads,chunk_size=`*n*`)`
- `(chunk_size=`*n*`,max_threads=`*m*`)`
- `(ordered,max_threads=`*m*`)`
- `(nodes,max_threads=`*m*`)`
- `(threads,max_threads=`*m*`)`
- `(ordered,nodes,max_threads=`*m*`)`
- `(ordered,threads,max_threads=`*m*`)`
- `(nodes,chunk_size=`*n*`,max_threads=`*m*`)`
- `(threads,chunk_size=`*n*`,max_threads=`*m*`)`
- `(ivar =` *indvar*`)`—Required for all loops in C and for `DO WHILE` and hand rolled loops in Fortran; optional for Fortran `DO` loops; can be specified with any other attribute.

Attributes may be listed in any order. Any attribute combinations other than those listed above will be flagged with a fatal error by the compilers. Refer to Chapter 6, "Advanced shared memory programming," for more information.

`loop_private(`*namelist*`)`

This is used to declare a list of variables and/or arrays private to the immediately following loop. To be loop private, the variables and/or arrays must be assigned before they are used on any 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. No values may be carried into the loop by `loop_private` variables. Values assigned to `loop_private`

variables on the final iteration 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 chapters 4 and 6 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 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 subcomplex but it is "closer" (in terms of access latency) to the threads on the hypernode that allocates the data. Refer to Chapter 5 for more information on memory classes.

`near_shared_pointer` (*alloc_var_name*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `near_shared` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5 for more information on memory classes.

`next_task`

This 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 chapters 4 and 6 for more information.

`no_block_loop`

Informs the compiler to perform no blocking on the immediately following loop. Generally the compiler will attempt to automatically block a nested loop in order to achieve optimal

cache data reuse between loop iterations. Refer to Chapter 3 for more information on loop blocking.

no_loop_dependence (*namelist*)

Informs the compiler that the arrays in *namelist* do not have any dependencies for iterations of the immediately following loop. Use `no_loop_dependence` for arrays only; use `loop_private` to indicate dependence-free scalar variables.

This will cause the compiler to ignore any dependencies that it perceives to exist. This can enhance the compiler's ability to optimize the loop, including the possibility of parallelization.

Refer to chapters 3 and 8 for more information on this directive.

no_parallel

This directive or pragma is used to prevent the compiler from generating parallel code for the immediately following loop. Refer to Chapter 3, "Compiler optimizations," for more information.

no_peel

This will indicate to the compiler to perform no peeling on the immediately following loop. Loop peeling involves removing tests that appear for the first and/or last iterations of a loop to outside the loop body. Refer to Chapter 3, "Compiler optimizations," for more information.

no_promote_test

When this directive or pragma is used the compiler will not attempt any test promotion on 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.

no_unroll_and_jam

Disables unrolling and jamming of loops. Refer to the "Loop unroll and jam" section of Chapter 3 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. Process threads within a hypernode all share access to the copy on their hypernode and cannot access the copies on other hypernodes. Refer to Chapter 5, "Memory classes," for more information.

node_private_pointer(*alloc_var_name*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `node_private` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, "Memory classes," for more information.

ordered_section(*gate_var*)

An ordered section is the same as a critical section with the additional restriction that the threads must pass through the ordered section in iteration order. Ordered sections must appear within the control flow of a `loop_parallel` or `prefer_parallel` loop with the `ordered` attribute specified. Refer to Chapter 6, "Advanced shared memory programming," for more information.

peel_all

This has the same effect as the `peel` directive or pragma except that it permits the compiler to replicate code without bound. Normally peeling will only replicate a certain amount of code. This can slow the compile phase and might even cause internal

compiler tables to overflow. Refer to Chapter 3, "Compiler optimizations," for more information.

peel

This directive or pragma allows the compiler to peel the loop immediately following the directive, expanding the code beyond the default conservative limit, but not without bound. Refer to Chapter 3 for more information on peeling. Refer to Chapter 3, "Compiler optimizations," for more information.

prefer_parallel

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 determinable at runtime before loop invocation, and contains no LCDs, procedure calls, or I/O operations. Refer to Chapter 4, "Basic shared memory programming," for more information. Refer to Chapter 4, "Basic shared memory programming," for more information.

promote_test_all

Instructs the compiler to promote tests out of the immediately following loop. The amount of code replication is not constrained in any way. Refer to Chapter 3, "Compiler optimizations," for more information.

promote_test

Instructs the compiler to promote tests out of the immediately following loop, expanding code beyond the default conservative limit but not without bound. Refer to Chapter 3, "Compiler optimizations," for more information.

row_wise(*array_namelist*)

In Fortran, the normal convention of storing arrays in column major order (left-most index varies fastest) can be switched for the arrays specified in *array_namelist*. Specifying `row_wise` for an array causes it to be stored in row major order which is consistent with the C language storage ordering.

save_last

This specifies that all variables named in an associated `loop_private(`*namelist*`)` or `task_private(`*namelist*`)` must have their last value saved into the "shared" variable of the same name, at loop or task 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 parallelize or data-localize a loop on which this directive appears.

task_private(*namelist*)

This will privatize the variables and arrays specified in *namelist* for each task specified in the immediately following `begin_tasks/end_tasks` block. The privatized variables and arrays will not carry their values beyond the `end_tasks` directive or pragma. Refer to chapters 4 and 6 for more information.

thread_private(*namelist*)

This Fortran directive will cause the variables and arrays specified in *namelist* to be treated (by software convention) as being `thread_private`. `thread_private` data objects map to unique `node_private` addresses for each thread of a process. Refer to Chapter 5, "Memory classes," for more information.

thread_private_pointer(*alloc_var_name*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `thread_private` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, "Memory classes," for more information.

unroll[(unroll_factor=*n*)]

This causes the immediately following loop to have its body replicated (*n* times if `unroll_factor` is specified) and unrolled in order to reduce loop overhead.

This only takes place if:

- the loop is a scalar loop
- there is no internal branching
- the specified loop is innermost.

Complete unrolling occurs if the loop count is less than 5. Otherwise only partial unrolling occurs. When used on a loop nest, this directive or pragma must be placed on the loop that ends up being innermost after compilation, or it will have no effect.

Refer to the "Loop unrolling" section of Chapter 3 for more information.

unroll_and_jam[(unroll_factor=*n*)]

This causes one or more non-innermost 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 non-innermost after any compiler-initiated interchanges.

Refer to the "Loop unroll and jam" section of Chapter 3 for more information.

Optimization options

B

This appendix lists the optimization options available for use with CONVEX SPP Series Fortran and C compilers and briefly describes each option. A complete list of all Fortran compiler options is available in Chapter 1 of the *CONVEX Fortran User's Guide*. A complete list of all C compiler options is available in Chapter 2 of the *CONVEX C User's Guide*.

Optimization level options

The options listed in this section specify the level of optimization allowed.

`-no`

Machine instruction level scalar optimization. This option is the default.

`-O0`

Basic block level scalar optimization.

`-O1`

`-O0` optimizations plus program unit level scalar optimization and global register allocation.

`-O2`

`-O1` plus data localization.

`-O3`

`-O2` plus parallelization.

`-or` *table*

Specifies the contents of the optimization report. Values of *table* and the optimization reports they produce are shown in Table 7.

Table 7 Optimization report contents

table value	Report contents
`all`	Loop table, privatization table and array table
`loop`	Loop table only (default)
`array`	Array table only
`private`	Loop table and privatization table
`none`	No report

For more information about the optimization report, refer to Appendix C, "Optimization report."

Cross compilation options

The options listed in this section allow a program to be optimized for a machine configuration that differs from the configuration of the machine the program is being compiled on.

`-tm` *target*

Specifies the target machine architecture for which compilation is to be performed. *target* can presently only take the value `spp1` on SPP Series machines; this is also the default in absence of the `-tm` option. This option may be useful when cross-compiling for future SPP Series architectures.

`-cache` *n*

Instructs the compiler to assume a per-processor direct-mapped cache size of *n* kbytes. The compiler uses this information when determining a loop's blocking factor. Cache size defaults to the size of the cache on which the program is being compiled. This option may be useful when cross-compiling for future SPP Series machines which may have different cache sizes.

Loop replication options

The options listed in this section control loop replication optimizations.

`-rl`

Causes the compiler to automatically select loops and replicate them by unrolling or dynamic selection. This option is available only at optimization level `-O2` or `-O3`.

`-ur`

Causes the compiler to automatically find unrollable loops and unroll them. Loops with iteration counts determinable at compile time to be less than 5 are unrolled completely.

Those with indeterminate iteration counts, or determinate counts of 5 or more, are partially unrolled. Only innermost loops can be unrolled. This option is the default, and is available only at optimization levels -O2 or -O3.

-urn *n*

Enables loop unrolling with an unroll factor of *n*; *n* is the number of times to replicate the body of the loop.

-nur

Disables loop unrolling.

-uj

Enables the unroll and jam optimization. This optimization unrolls one or more non-innermost loops in a nest, then jams the resulting loops back together, greatly improving register usage. This option is the default, and is available only at optimization levels -O2 and -O3. For more information, refer to the "Loop unroll and jam" section of Chapter 3.

-ujn *n*

Enables unroll and jam with an unroll factor of *n*.

-nuj

Disables unroll and jam.

Loop blocking options

The options listed in this section control the loop blocking optimization.

-blockloop *n*

Instructs the compiler to use a block factor of *n* and to automatically select which loops to block. In a loop nest containing *m* loops, *m*-1 loops will be blocked. If the -blockloop option is not used, the compiler will attempt to choose an optimal blocking factor based on the cache size and the amount of data being manipulated by the loop.Loop blocking is provided at optimization levels -O2 and -O3 unless the -noblock option is specified.

-noblock

Disables loop blocking for the sources being compiled. Because loop blocking occurs at optimization levels -O2 and -O3, the -noblock option is effective only at these levels.

IF-DO and if-for optimization options

The options listed in this section control the degree of loop peeling and test promotion allowed.

`-nopeel`

Disallows loop boundary value peeling, which is enabled by default at optimization levels `-O2` and `-O3`. Refer to the `-peel` and `-peelall` options described in this section.

`-noptst`

Disallows test promotion, which is enabled by default at optimization levels `-O2` and `-O3`. Refer to `-ptst` and `-ptstall` below.

`-peel`

Removes the first and/or last iterations of a loop when doing so removes conditional tests from the loop. This is done when the loop contains a test involving an explicit reference to the loop index variable that always evaluates to `.TRUE.` or `.FALSE.` for the first and/or last iteration. By default, the compiler peels boundary values and expands code up to a predetermined conservative limit. With the `-peel` option, this limit is increased and code expansion may become significant. `-peel` must be used with the `-O2` or `-O3` optimization options.

`-peelall`

Same as `-peel`, but allows code expansion without bound. For code containing large numbers of boundary value operations, this can greatly lengthen compiler time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables. `-peelall` must be used with the `-O2` or `-O3` optimization options.

`-ptst`

Causes a test to be promoted out of the loop that encloses it by replicating the containing loop for each branch of the test. By default, the compiler replicates code up to a predetermined conservative limit. The `-ptst` option increases this limit and can cause a noticeable increase in compile time. `-ptst` must be used with the `-O2` or `-O3` optimization options.

`-ptstall`

Same as `-ptst`, but allows code replication without bound. For loops containing large numbers of tests, this can significantly increase compile time and can increase the size of

the code enough to exceed the limits of some of the compiler's internal tables. `-ptstall` must be used with the `-O2` or `-O3` optimization options.

Register use options

The options listed in this section control the extent to which registers are exploited.

`-nga`

Disable global register allocation for arguments passed by reference. This is useful for programs that violate the ANSI standard by passing a constant as a procedure argument when the procedure may potentially write to the dummy argument. Refer to the "Global register allocation" section of Chapter 3 for more information.

`-ngs`

Disable global register allocation for shared memory variables which are visible to multiple threads. This option may help if a variable shared among parallel threads is causing wrong answers. Refer to the "Global register allocation" section of Chapter 3 for more information.

C aliasing options

Potential aliases occur more frequently in C than Fortran because of C's typically heavy use of pointers. This section covers the options supported by the CONVEX C compiler to help you deal with aliasing problems.

`-alias array_args`

Causes the compiler to assume that formal array parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but can allow greater optimization to occur. This option cannot be used if the formal parameter itself is assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to the elements of the formal parameter (for example, `formalParameter[10] = x[10]`).

The following source code illustrates a situation in which the `-alias array_args` option is useful.

```
void vaf( char [] );
char global = 'A';

int main()
{
  char b[10];

  vaf(b);
  return(1);
}

void vaf(char array[])
{
  int i;

  for(i=0; i<10; i++)
    array[i] += global;
}
```

The loop in the `vaf` function can be parallelized when the `-alias array_args` option is specified because the compiler can assume that the address of `global` is not the address of an element of `array`.

`-alias standard`

Performs aliasing based on assumptions permitted in ANSI C; pointers of one object type can only reference objects of that same type. For example, a pointer to an object of type float cannot reference an object of type `int`. Such assumptions permit additional optimizations. If this option is specified when such conditions do not exist, the resulting program may not function correctly.

`-alias cautious`

Tries to compile with `-alias standard`, but if inappropriate constructs are found, compiles with `-alias worst` instead. This is the default in the ANSI C-compatible modes.

`-alias worst`

Performs pointer aliasing based on the assumption that pointers can modify objects of any type. This is the default in the backward-compatible (`-pcc`) mode.

The default aliasing algorithm for ANSI C assumes that pointers of one type do not point to objects of another type. For example, `int` pointers cannot point to objects with type `float`.

`-alias [no_global | global]`

When `-alias no_global` is specified, the compiler assumes that globals (external variables) are not accessed via pointers. The default is `global`.

Compiling the following code with `-alias no_global` would be an *incorrect* use of the option because the value of the global is accessed through the pointer `gptr` as well as the variable `glob`.

```
int glob; /* wrong answers result if
             compiled with -alias
             no_global!!!            */
int *gptr;

bar()
{
  gptr = &glob;
}

main()
{
  int tmp;

  bar();
  glob = 1;
  *gptr = 2;
  tmp = glob;
  printf("%d\n", tmp);
}
```

The program may print 1 when compiled with `-alias no_global`.

`-alias [no_addr | addr]`

`-alias no_addr` indicates that address operands do not introduce aliases. This is useful when your program passes data by reference and you are sure that no aliases are produced. The default is `-alias addr`.

`-alias ptr_args`

Causes the compiler to assume that the variables identified by formal pointer parameters do not overlap each other or any external variable that is used in the function (unless all uses

are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but allows greater optimization to occur. This option cannot be used if the formal parameter itself is assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to variables identified by the formal parameter (for example, `formalParameter[10] = x[10]`).

`-alias restrict_args`

Causes the compiler to treat the code as though a `restrict` qualifier was applied to each pointer parameter. The `restrict` qualifier tells the compiler that a pointer provides exclusive access to the data object at a given memory location. When you use the `restrict` qualifier, you must access the object pointed to by a `restrict` pointer only through that pointer. If you reference the object by some other means, the compiler may incorrectly optimize code.

Other optimization options

This section lists optimization options that cannot be otherwise categorized.

`-align cseries`

Causes the Fortran compiler to store `COMMON` blocks using "tight" packing (the ANSI standard). Tight `COMMON` block packing is the default on CONVEX C Series machines.

Rather than padding `COMMON` block data items to their natural boundaries (the default on SPP Series machines), this method stores them in memory contiguously, so partial-word items may cause other data items to align on unnatural boundaries. For instance, a `REAL*8` item may align on a 2– or 4-byte boundary instead of an 8-byte boundary.

`-align spp` is the opposite of `-align cseries` and is the default on SPP Series machines.

For more information, refer to the "`COMMON` block packing" section of the *CONVEX Fortran Language Reference* Chapter 5.

`-il`

Instructs the Fortran compiler to prepare an intermediate language (`.fil`) file for a subprogram that is to be used for inline substitution. The `-il` option cannot be used with the `-c`, `-cs`, or `-S` options. Optimization levels are ignored. Refer to the *CONVEX Fortran User's Guide,* Chapter 1, "Compiling programs."

`-is` *directory*

Instructs the Fortran compiler to attempt inline substitution of each subprogram for which there exists a `.fil` (intermediate-language file) file in the specified *directory*. This option must be repeated for each directory containing `.fil` files to be used for inline substitution. Refer to the *CONVEX Fortran User's Guide*, Chapter 1, "Compiling programs."

`-sr`

Enable scalar replacement. This is the default at optimization level `-O2` and above. Refer to the "Scalar replacement" section of Chapter 3.

`-nsr`

Disable scalar replacement.

`-nore`

SPP Series compilers compile all procedures for reentrancy by default (refer to the `-re` option). Fortran procedures can be compiled using the `-nore` compiler option, which causes nonreentrant compilation. Procedures compiled in this manner cannot be parallelized or specified in a `NO_SIDE_EFFECTS` directive or pragma.

Many Fortran compilers, including the CONVEX C Series Fortran compiler, save local procedure variables by default. If your program is expecting this behavior, default reentrant compilation may cause wrong answers. Compiling the offending procedures with `-nore` should correct the problem.

`-re`

Causes reentrant compilation, which is the default. Reentrant procedures store all local variables on the stack; no values can be carried from one invocation of the procedure to the next. This allows parallel procedure calls.

`-uo`

Performs potentially unsafe optimizations, for example, moving the evaluation of common subexpressions or invariant code from within conditionally executed code. This moved code may be executed unconditionally. Refer to Chapter 9, "Potentially unsafe optimizations."

Optimization report

C

This appendix provides a complete description of the optimization report produced by the CONVEX SPP Series C and Fortran compilers. When you compile a program with the `-O2` or `-O3` option, the compiler generates an optimization report for each program unit. The `-or` *table* option determines the report's contents based on the value of *table*, as shown in Table 8.

Table 8 Optimization report contents

table value	Report contents
`all`	Loop table, privatization table and array table
`loop`	Loop table only (default)
`array`	Array table only
`private`	Loop table and privatization table
`none`	No report

Loop table

The loop table lists the optimizations that were performed on each loop and, if appropriate, the 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 table follows.

`Line Num.`

Specifies the source line of the beginning of the loop, or of the loop from which it was derived. If the line number has two parts separated by a hyphen, the second part is the distributed number (due to loop distribution).

`Id Num.`

Specifies a unique ID number for every loop. 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 Loops` column as described further on. 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.

`Iter. Var.`

Specifies the name of the iteration variable controlling the loop. If the variable is compiler-generated, its name is listed as `*VAR*`; if there is no iteration variable, it is listed as `*NONE*`. If the iteration variable has two parts separated by a colon, the second part is the inline substitution instance of that variable. 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 table, analysis table, and test table in the report.

`Reordering Transformation`

Indicates which reordering transformations were performed. Reordering transformations are performed on loops 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 9.

Table 9 Reordering transformations reported in opt. report

Value	Explanation
`Serial`	No reordering transformation was performed.
`PARALLEL`	The loop runs in parallel mode.
`Interchange`	Loop interchange was performed. The new loop order may be indicated under the `Optimizing/Special Transformation` column, as shown in Table 10.
`Dist`	Loop distribution was performed.
`DynSel`	Dynamic selection was performed. The numbers in the `New Loops` column correspond to the loops created; for reentrant parallel loops, these generally include a `PARALLEL` and a `Serial` version.
`Peel`	Loop peeling was performed. In addition to cases in which loops are peeled to remove invariant boundary-iteration assignments, this may appear for automatically parallelized loops when it is necessary to peel the last iteration in order to assign automatically privatized variables their last-iteration value.
`Promote`	Test promotion was performed.
*	Appears at left of loop-producing transformation optimizations (distribution, dynamic selection, peeling, interchange, promotion).

`New Loops`

Specifies the loop ID number(s) for loops 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 they represent were not present in the original source code.

`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 10.

Table 10 Optimizing/special transformations in opt. report

Value	Explanation
`Unroll`	The loop was completely or partially unrolled.
`Reduction`	The compiler recognized a reduction in the loop.
`Pattern`	The compiler recognized a special pattern in the loop.
`Removed`	The compiler removed the loop.
`StripMine`	The loop was strip mined.
`Blocked`	The loop was blocked.
(*oldorder*) -> (*neworder*)	This may appear when `Interchange` is reported under `Reordering Transformation`. *oldorder* indicates the order of loops in the original nest; *neworder* indicates the new order. *oldorder* and *neworder* consist of user iteration variables presented in outermost to innermost order; if user variables are not available, line numbers are used. If the multiple nests begin on the same line, as with compiler-generated loops associated with Fortran 90 array assignments, this information is not reported.

Supplemental tables

The tables described in this section are included if necessary to provide information supplemental to the loop table.

Analysis table

If necessary, an analysis table is included in the optimization report to further elaborate on optimizations reported in the loop table. A description of each column of the analysis 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.

Iter. Var.

Specifies the name of the iteration variable controlling the loop, *VAR*, or *NONE*, as described in the "Loop table" section of this appendix.

Analysis

Indicates why a transformation or optimization was not performed, or additional information on what was done.

Test table

If any test promotion or removal optimizations were performed, a test table is included in the optimization report. A description of each column in the test table follows.

Line Num.

Specifies the source line number of the beginning of the IF test.

Col. Num.

Specifies the source column number of the beginning of the IF test.

Test Transformation

Specifies the transformation performed: either TEST PROMOTED or TEST REMOVED.

Analysis

Presents a further explanation of the transformation performed, including the source line number of the original loop from which the test was transformed, and (if applicable), in parentheses, the loop ID number of the compiler-generated loop from which the test was transformed. This ID will be missing if the loop was removed.

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 table 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.

`Iter. Var.`

Specifies the name of the iteration variable controlling the loop, `*VAR*`, or `*NONE*`, as described in the "Loop table" section of this appendix.

`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 privatization performed. For example, this column may indicate that a variable was automatically privatized by the compiler and that its last assignment was peeled from the loop so that its final value could be saved for later use.

Variable name footnote table

Variable names that are too long to fit in the `Iter. Var.` 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 Iter. Var.`

Specifies the truncated variable name and its footnote number.

`User Variable Name`

Specifies the actual name of the variable as given by the user in the source code.

Array table

The array table lists array references that prevented optimization or array references on which special optimizations were performed. The array table contains the following information.

`Line Num.`

Specifies the source line on which the reference occurs.

`Var. Name`

Specifies the name of the array being referenced.

`Optimization`

Describes the optimizations, if any, performed on the array in question.

`Dependencies`

If an array or memory dependency prevented optimization, this column shows the names of variables in the recurrence, in the form *name@linenumber*. If the reference could be to any memory location, it is in the form `*MEM*@`*linenumber*. If the reference is to a subprogram call, it is in the form `*CALL*@`*linenumber*.

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 code would produce similar optimization reports.

Example 1

Consider the following loop (line numbers are provided for reference):

```
1     PROGRAM EXAMPLE1
2     REAL A(500,500), B(500,500), C(500)

3     DO ILOOPINDEX = 1, 500
4       C(ILOOPINDEX) = ILOOPINDEX
5       DO JLOOPINDEX = 1, 500
6         A(ILOOPINDEX,JLOOPINDEX) = B(JLOOPINDEX,ILOOPINDEX) +
      ^       C(ILOOPINDEX)
7       ENDDO
8     ENDDO

9     END
```

Figure 23 shows the optimization report generated by compiling the program `EXAMPLE1` at optimization level `-O3`.

Figure 23 Optimization report for Example 1

```
% fc -O3 -or all example1.f
            Optimization for EXAMPLE1

Line      Id    Iter.      Reordering          New       Optimizing / Special
Num.      Num.  Var.       Transformation      Loops     Transformation
-------------------------------------------------------------------------------
    3       1   ILOOPI:1 *Dist               (2-3)
    3-1     2   ILOOPI:1 *DynSel             (4-5)
    3       4   ILOOPI:1  PARALLEL

    3       5   ILOOPI:1  Serial

    3-2     3   ILOOPI:1 *Interchange        (6)       (ILOOPI:1 JLOOPI:2
                                                       JLOOPI:2) -> (JLOOPI:2
                                                       ILOOPI:1 JLOOPI:2)
    5-2     6   JLOOPI:2 *DynSel             (7-8)     Blocked
    5       7   JLOOPI:2  PARALLEL
    3       9   ILOOPI:1  Serial
    5      10   JLOOPI:2  Serial

    5       8   JLOOPI:2  Serial
    3      11   ILOOPI:1  Serial
    5      12   JLOOPI:2  Serial

Line      Id    Iter.   Analysis
Num.      Num.  Var.
-------------------------------------------------------------------------------
    5       7   JLOOPI:2Loop blocked by 224 iterations
    5       7   JLOOPI:2Parallel outer strip mine loop

Footnoted   User Variable
Iter. Var.  Name
-------------------------------------------------------------------------------
ILOOPI:1    ILOOPINDEX
JLOOPI:2    JLOOPINDEX
```

Loop number 1 is the loop that appears on line 3 of the source. It iterates over the variable `ILOOPINDEX`. It was distributed and two new loops, numbers 2 and 3, were created.

Note that according to the variable name footnote table, `ILOOPINDEX` is abbreviated as `ILOOPI:1`; similarly, `JLOOPINDEX` is abbreviated as `JLOOPI:2`. These names are used throughout the report to refer to these iteration variables.

3-1, the line number for loop number 2, tells us that it came from the loop at line 3 of the source, and that it is in the first distributed part of that distribution (this is indicated by the -1). Two loops, 4 and 5, are created to facilitate dynamic selection of

loop 2. Loop 4 is the parallel version, and loop 5 is the serial version.

The line number 3-2 for loop 3 tells us that it is the second part of the distribution of the loop appearing on line 3 of the source (loop 2). The `Optimizing/Special Transformation` column shows the loop nest order before and after interchange. There are two `JLOOPINDEX` loops before interchange as a result of strip mining, which is performed as part of the distribution of loop 1 and not explicitly reported. The innermost `JLOOPINDEX` loop is interchanged to outside the `ILOOPINDEX` loop, and this creates loop 6.

Dynamic selection transforms loop nest 6 into loop nests 7 and 8.

Loop 7 is then the parallel version of loop 6; the analysis table tells us that it is an outer strip mine loop. Loop 8 is the serial version. Loop 7 is also the blocked version of loop 6, by 224 iterations according to the analysis table.

Loops 9 and 10 are contained within the strip mine loop 7; these loops are listed as `Serial` because they run serially across the available processors under the control of parallel loop 7 (the outer strip mine loop).

Loops 11 and 12 are similarly subordinate to loop 8, which is the final serial version of loop 3.

Example 2

The following Fortran code provides an example of other transformations the compiler performs. (Line numbers are listed for reference.)

```
1   SUBROUTINE EXAMPLE2(A,N,ZERO,NEGATE,SUM)
2   REAL A(N), SUM
3   LOGICAL ZERO, NEGATE
4
5   SUM = 0.0
6   DO I = 1, N
7     SUM = SUM + A(I)
8     IF (ZERO) THEN
9       A(I) = 0
10    ELSE IF (NEGATE) THEN
11      A(I) = -A(I)
12    ENDIF
13    IF (I .EQ. 1 .OR. I .EQ. N) THEN
14      A(I) = -1
15    ENDIF
16  ENDDO
17  END
```

Figure 24 shows the optimization report generated by compiling the subroutine `EXAMPLE2` above for parallelization. The `-c` option suppresses loading because no main program is included.

Figure 24 Optimization report for Example 2

```
%fc -O3 -c example2.f
           Optimization for EXAMPLE2

Line      Id    Iter.     Reordering          New       Optimizing / Special
Num.      Num.  Var.      Transformation      Loops     Transformation
-------------------------------------------------------------------------------
    6       1   I         *Promote            (2-3)
    6       2   I         *Promote            (4-5)
    6       4   I         *Peel               (6)
    6       6   I         *DynSel             (7-8)
    6       7   I         *Peel               (9)
    6       9   I          PARALLEL

    6       8   I          Serial

    6       5   I         *Peel               (10)
    6      10   I         *DynSel             (11-12)
    6      11   I         *Peel               (13)
    6      13   I          PARALLEL

    6      12   I          Serial

    6       3   I         *Peel               (14)
    6      14   I         *DynSel             (15-16)
    6      15   I         *Peel               (17)
    6      17   I          PARALLEL
    .
    6      16   I          Serial

Line     Col.   Test              Analysis
Num.     Num.   Transformation
-------------------------------------------------------------------------------
    8     14    TEST PROMOTED     Test promoted out of loop on 6 (9)
   10     19    TEST PROMOTED     Test promoted out of loop on 6 (9)
   13     16    TEST REMOVED      Peeled first iteration of loop on 6 (17)
    .
    .
    .

Line      Id    Iter.    Priv.     Privatization Information
Num.      Num.  Var.     Var.      for Parallel Loops
-------------------------------------------------------------------------------
    6      17   I        *NONE*    Loop peeled to save last values
    6      17   I        SUM       Scalar reduction privatized, value saved
    6      13   I        *NONE*    Loop peeled to save last values
    6      13   I        SUM       Scalar reduction privatized, value saved
    6       9   I        *NONE*    Loop peeled to save last values
    6       9   I        SUM       Scalar reduction privatized, value saved
```

There is only one loop in this example, and it appears at line 6 in the source. Loops 2 and 3, which are noted under `New Loops` on loop 1's line, are the result of promoting tests from the original loop; loops 4 and 5 are the result of promoting tests from loop 2. Loop 6 is created when loop 4 is peeled. Two loops, 7 and 8, are then created to facilitate dynamic selection of loop 6. Loop 7 is peeled to facilitate privatization of the variable `SUM`. Loop 9 is the parallel version of loop 7 after peeling, so the peeling is reported in the privatization table as being for loop 9.

Loop 8 is the serial version of loop 6.

Loops 3 and 5 were both peeled, and the resulting loops were replicated into parallel and serial versions by dynamic selection. Loop 5 is peeled to create loop 10, which is transformed by dynamic selection into loops 11 and 12. Loop 11 is the parallel version, which is transformed by peeling into loop 13; this peeling is reported in the privatization table. Loop 12 is the serial version.

Loop 3 is peeled to create loop 14, which produces loops 15 and 16; loop 15 is peeled into the parallel loop 17. Loop 16 is the serial version.

After all the transformations are completed, six loops (ID numbers 8, 9, 12, 13, 16, and 17) remain in the program. These remaining loops can be easily spotted under the `Reordering Transformation` column, as they are the loops that are not marked with the "*" transformation indicator. Loops marked with this symbol no longer exist because they are replaced by the new loops indicated in the `New Loops` column.

The test table of this report gives details of the test promotions and peelings that are mentioned in the loop table. For brevity, several of the peeling explanations that appeared at the end of the test table were omitted from Figure 24.

Note the difference between the peelings reported in the test table and in the privatization table. Those reported in the test table were performed to remove tests from the loop; those reported in the privatization table were, as indicated in the table, performed to facilitate saving final loop values in parallelized loops. For each loop, the table contains an entry indicating that peeling was done to save last values, and a second entry indicating the privatization of a scalar reduction (involving the variable `SUM` in this case). This indicates that `SUM` was accumulated separately in a private copy in each parallel thread, and these copies were automatically added together after the threads joined.

Compiler Parallel Support Library

D

Introduction

The CONVEX Compiler Parallel Support Library (CPSlib) is a library of thread management and synchronization routines which can be used to control parallelism on Exemplar systems. Most programs can fully exploit their parallelism via higher-level devices such as automatic parallelization, compiler directives, and ConvexPVM; CPSlib is provided for those few cases in which a lower-level interface is required. 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 of the routines 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

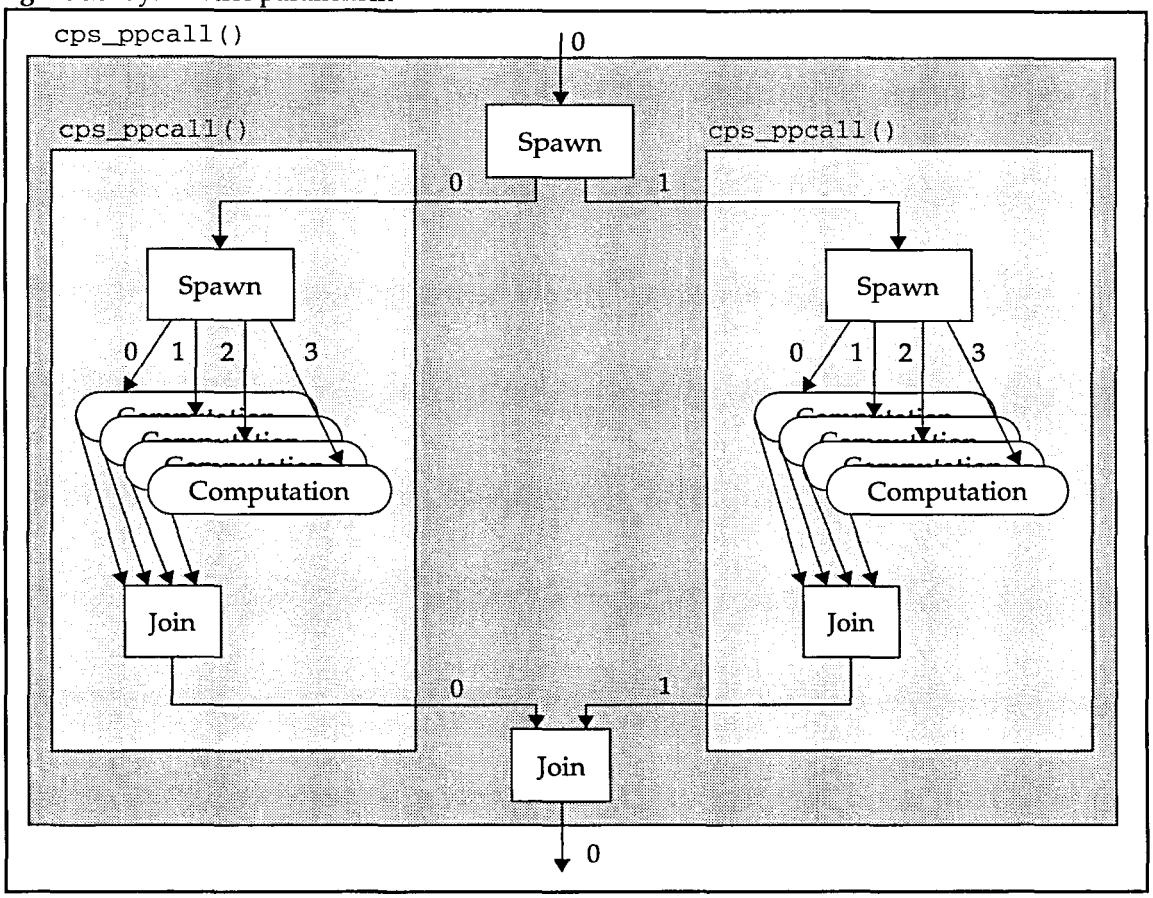
In symmetric parallelism, several threads execute the same instruction stream. Symmetric parallelism is typically used by the compilers to parallelize a loop; the description of parallelism given in Chapter 3 is a description of symmetric parallelism.

Symmetric parallel threads are created 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 25 shows a `cps_ppcall()` that creates 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 "Thread ID assignments" section of Chapter 6.

Figure 25 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 created 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 created by the last spawn have also joined. The join operation contains an implicit barrier synchronization, 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.

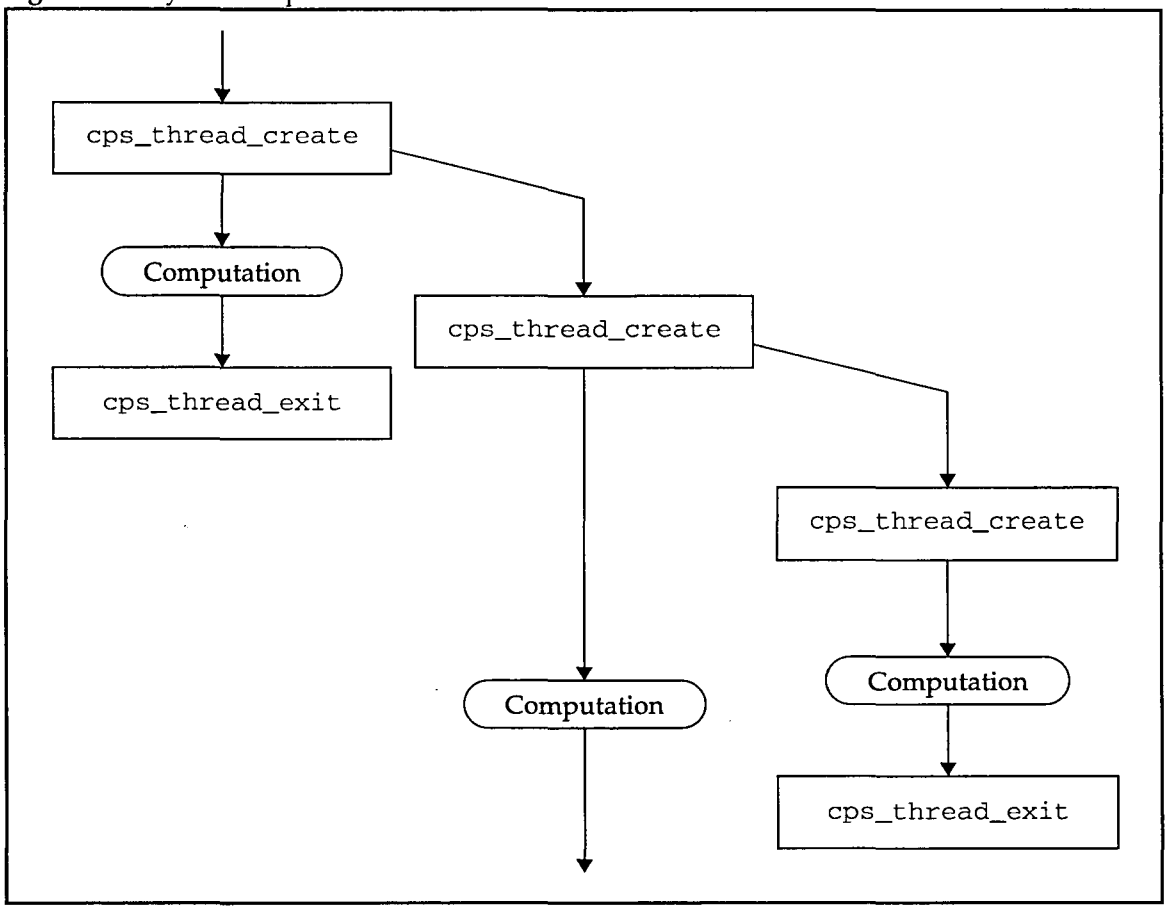
Asymmetric Parallelism

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 create additional symmetric or asymmetric threads.

Asymmetric threads are created using the `cps_thread_create()` function and terminated using the `cps_thread_exit()` function. Asymmetric threads can not 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 26 shows an asymmetric thread tree. Asymmetrically created 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 created the child thread.

Figure 26 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 programs which use CPSlib functions must include the header file `cps.h`, as shown:

```
#include <cps.h>
```

To access `errno` symbolic constant values in C, you must also include the header file `errno.h`:

```
#include <errno.h>
```

C and Fortran share a common interface to CPSlib; the same library, libcps.a, allows access to the CPSlib functions from either language. This library is automatically linked when you compile your program with the CONVEX Exemplar `fc` or `cc` compiler drivers.

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 four byte values and take four byte arguments. Eight byte versions are also available; the names of these functions are suffixed with `_8` (e.g. 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 examples presented here assume that `cps.h` is included in the program. Note that the NULL value in C is equivalent to 0 (zero) in Fortran.

Note

CPSlib routines are generally incompatible with system functions of the form `cnx_*`; mixing the two may cause wrong answers, deadlock or runtime errors. Mixing CPSlib routines and compiler directives may cause wrong answers at `-O1` or higher; if this happens, lower the optimization level or refer to the "Global register allocation" section of Chapter 3.

Thread management functions

These functions allow you to create, terminate, and get information about threads.

Spawn symmetric threads

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
```

In 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 hypernode ID of the hypernode on which to allocate the threads, or they can take one of the values shown in Table 11.

Table 11 `params->node/PARAMS(1)` values

C symbolic constant name	Value	Meaning
`CPS_ANY_NODE`	-1	Allocate requested threads on any hypernode.
`CPS_SAME_NODE`	-2	Allocate threads on same hypernode as calling thread
`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 12 shows acceptable values for these parameters. Except where noted, the logical `or` of two or more of these values can be used.

Table 12 `params->threadscope/PARAMS(4)` values

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_ppcall` creates 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 created will be assigned increasing spawn thread IDs ranging from 1 to *m*-1, where *m* is the number of threads created.

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 created including the parent. If an error occurs, they return -1 and, in C, `errno` is set as shown in Table 13.

Table 13 `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 8 Mbytes. If `func` declares local variables that occupy more than 8 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 SPP-UX; you can modify it via the mpa(1) utility.

For more information, refer to the `cps_ppcall(3)` man page.

Asymmetric thread functions

These functions allow you to create, terminate, and wait for asymmetric threads.

`cps_thread_create`

This function allows you to spawn asymmetrically parallel threads. In Fortran, this function has the following form:

```
INTEGER FUNCTION CPS_THREAD_CREATE(NODE, FUNC, ARG)
INTEGER NODE
EXTERNAL FUNC
```

In C:

```
int cps_thread_create(const int *node, void (*func) (void *),
                      void *arg);
```

This function creates 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 values as the `node` argument to `cps_ppcall`, which is described in the "Spawn symmetric threads" section.

If successful, `cps_thread_create` returns the kernel thread ID of the newly created thread. No assumptions can be made about kernel thread IDs except that they are unique. Asymmetric threads do not have spawn thread IDs.

Caution

cps_thread_create returns 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 but before func exits, you must insure that this manipulation is synchronized between the parent and func, or the value of arg will be indeterminate.

If unsuccessful, cps_thread_create returns -1 and, in C, sets errno as shown in Table 14.

Table 14 errno values for cps_thread_create

errno value	Meaning
EAGAIN	The asymmetric thread could not be allocated.
EINVAL	node is not a valid hypernode ID in the program-assigned subcomplex

Nested cps_thread_create calls 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:

```
int cps_thread_exit(void);
```

Upon successful completion of func (specified in cps_thread_create), asymmetric threads created 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, 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:

```
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 which 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, 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 functions

These functions provide information on active parallel threads and the subcomplex configuration. 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:

```
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:

```
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` which 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:

```
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:

```
int cps_plevel(void);
```

The return value is a bit mask. The possible return values are a sum of those shown in Table 15.

Table 15 `cps_plevel` return values

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:

```
int cps_node_id(void);
```

Logical hypernode IDs range from 0..*n*-1, where *n* is the number of available hypernodes in the subcomplex.

cps_node_cpus
This function returns the number of CPUs 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:

```
int cps_node_cpus(void);
```

Note that the return value represents the number of CPUs visible to the calling application, and does not necessarily indicate the total number of physical CPUs 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:

```
int cps_node_nthreads(void);
```

Active threads include threads created by `cps_ppcall`, `cps_ppcalln`, or `cps_thread_create`, as well as threads created 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:

```
int cps_is_parallel(void);
```

Note that the `mpa` utility can be used to modify attributes of the executable such as whether or not it is parallel. Refer to the mpa(1) man page for more information.

cps_complex_cpus
This function returns the total number of CPUs available to the application. In Fortran, it has the following form:

```
INTEGER CPS_COMPLEX_CPUS()
```

In 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 subcomplex on which the call is executing. In Fortran, it has the following form:

```
INTEGER CPS_COMPLEX_NTHREADS()
```

In C:

```
int cps_complex_nthreads(void);
```

Active threads include threads created by `cps_ppcall`, `cps_ppcalln`, or `cps_thread_create`, as well as threads created 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 subcomplex. In Fortran it has the following form:

```
INTEGER CPS_COMPLEX_NODES()
```

In C:

```
int cps_complex_nodes(void);
```

High-level synchronization functions

These routines manipulate the barriers and mutexes which 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. 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:

```
int cps_barrier_alloc(barrier_t *barr);
```

If this function fails when called from 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:

```
int cps_barrier_free(barrier_t *barr);
```

If this function fails when called from 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:

```
int cps_barrier(barrier_t *barr, const int *nthreads);
```

If this function fails when called from 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:

```
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, 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:

```
int cps_mutex_free(cps_mutex_t *mutx);
```

If this function is successful when called from C, `mutx` is set to NULL. In Fortran, `MUTX` is undefined.

If unsuccessful when called from 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:

```
int cps_mutex_lock(cps_mutex_t *mutx);
```

If the calling thread has already acquired mutx this function returns -1 and, in 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:

```
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:

```
int cps_mutex_trylock(cps_mutex_t *mutx);
```

If the calling thread has already acquired mutx this function returns -1 and, in 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:

```
int c_init32(mem_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:

```
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:

```
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 FUNCTOIN C_UNLOCK(CS)
INTEGER CS
```

In 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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
int m_fetch_and_clear32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

Examples

The examples presented here demonstrate various constructs that can be programmed using the CPSlib functions described in the previous sections.

You can compile a program that uses CPSlib to achieve parallelism at any optimization level, however, you must pass appropriate flags to the linker to indicate that the program is parallel. This is done using the `-Wl,` option, which indicates that the comma-delimited options following it up to the next space are to be passed to the linker. You can pass either the `+min` and `+max` options to indicate the minimum and maximum number of CPUs on which to run the program, or the `+parallel` option to indicate that the program is parallel and that any available CPUs should be used.

The following example `fc` command line compiles the program `cpspar.f` at optimization level `-O2` and indicates to the linker that the program should run on a minimum of 2 and a maximum of 8 processors:

```
fc -Wl,+min,2,+max,8 -O2 cpspar.f
```

The compiler does not recognize CPSlib parallelism, so while this command line would generate an optimization report, the report would only detail compiler optimizations; it would not report manually-coded parallelism. To see if your executable is parallel, use the `file` command.

You can also use the `mpa` utility to modify many attributes, including parallelism, of an executable file after compilation. Refer to the mpa(1) man page for more information.

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 "CPS library functions" section.

Block parallelism

Block parallelism is the most commonly used form of parallelism; it is the form generated by default by CONVEX SPP Series compilers. It involves splitting up the iterations of a loop into iteration blocks of similar size, and running each block on a separate processor. Block parallelism is illustrated in Chapter 3 starting on page 84.

A simple Fortran example which 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
      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
      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 DON'T HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+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 processors, 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 Chapter 6 on page 163.

A simple Fortran example which 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
      EXTERNAL PARCYC ! PARCYC IS AN ARGUMENT
      READ *, NPROCS
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
      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 insures 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
      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
      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 on page 291 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
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
      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 DON'T HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+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 regions of code they delimit to one thread at a time, allowing you to construct critical sections similar to those discussed in Chapter 4.

In the following Fortran example, the routine SUMMER performs the same task it did in 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
      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
      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 DON'T HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+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
      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
      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 DON'T HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+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.

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
      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
      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 insures 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.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

U

V

W

Z

ORDER NUMBER
DSW-067

DOCUMENT NUMBER
710-030230-002