

CRAY T3E™ C and C++ Optimization Guide

Document Number 004-2178-002

St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

Copyright © 1997, 1999 Cray Research, Inc. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Cray Research, Inc.

LIMITED AND RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the Rights in Data clause at FAR 52.227-14 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

Autotasking, CF77, CRAY, Cray Ada, CraySoft, CRAY Y-MP, CRAY-1, CRInform, CRI/*TurboKiva*, HSX, LibSci, MPP Apprentice, SSD, SUPERCLUSTER, UNICOS, and X-MP EA are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, CRAY APP, CRAY C90, CRAY C90D, Cray C++ Compiling System, CrayDoc, CRAY EL, CRAY J90, CRAY J90se, CrayLink, Cray NQS, Cray/REELlibrarian, CRAY S-MP, CRAY SSD-T90, CRAY SV1, CRAY T90, CRAY T3D, CRAY T3E, CrayTutor, CRAY X-MP, CRAY XMS, CRAY-2, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, GigaRing, HEXAR, IOS, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNETH, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, UNICOS MAX, and UNICOS/mk are trademarks of Cray Research, Inc., a wholly owned subsidiary of Silicon Graphics, Inc.

AMPEX and DST are trademarks of Ampex Corporation. DEC is a trademark of Digital Equipment Corporation. DLT is a trademark of Quantum Corporation. EXABYTE is a trademark of EXABYTE Corporation. IBM and Magstar are trademarks of International Business Machines Corporation. STK, TimberLine, RedWood, and WolfCreek are trademarks of Storage Technology Corporation. Silicon Graphics and the Silicon Graphics logo are registered trademarks of Silicon Graphics, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X/Open is a registered trademark, and the X device is a trademark, of X/Open Company Ltd.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

Record of Revision

<i>Version</i>	<i>Description</i>
3.0.1	August, 1997 Original printing.
3.2	January, 1999 Updated to support the compilers released with the Programming Environment 3.2 release.

Contents

	<i>Page</i>
Preface	xi
Related Publications	xi
Obtaining Publications	xi
Conventions	xii
Reader Comments	xiii
Background Information [1]	1
Message-passing Protocols	2
Parallel Virtual Machine (PVM)	2
SHMEM	2
Hardware	4
Memory	4
Procedure 1: Moving data from memory	9
Processing Element	15
Network and Peripherals	15
Disk Support	16
Tape Support	17
Network Protocols	18
Memory Performance Information	19
Measuring Performance	21
Parallel Virtual Machine (PVM) [2]	23
Setting the Size of a Message	24
Allocating Send Buffers	25
the Advantage of 32-bit Data	26
004-2178-002	iii

	<i>Page</i>
Example 1: Transferring 32-bit data	26
Sending and Receiving Stride-1 Data	28
Example 2: pvm_psend and pvm_preclv	29
Mixing Send and Receive Routines	30
Initializing and Packing Data	30
Working While You Wait	31
Avoiding Barriers	32
Using Broadcast or Multicast	33
Example 3: pvm_bcast	34
Example 4: pvm_mcast	35
Minimizing Synchronization Time When Receiving Data	37
Using the Reduction Functions	38
Example 5: A PvmSum example	39
Gathering and Scattering Data	40
Example 6: Gather operation	42
Example 7: Scatter operation	43
SHMEM [3]	45
Using shmem_get and shmem_put for data transfer	46
Example 8: Example of a shmem_put transfer	46
Optimizing Existing MPI and PVM Programs by Using SHMEM	51
Example 9: MPI version of the ring program	51
Optimizing by Using shmem_get	55
Example 10: shmem_get version of the ring program	55
Optimizing by Using shmem_put	58
Example 11: shmem_put version of the ring program	58
Passing 32-bit Data	59
Example 12: 32-bit version of ring program	60

	<i>Page</i>
Copying Strided Data	62
Example 13: Passing strided data using <code>shmem_iget</code>	62
Example 14: Passing strided data using <code>shmem_iput</code>	63
Gathering and Scattering Data	67
Example 15: <code>shmem_ixput</code> version of a reordered scatter	67
Broadcasting Data to Multiple PEs	71
Example 16: One-to-all broadcasting	71
Merging Arrays	73
Example 17: Merging arrays	74
Reading and Updating in One Operation	76
Example 18: Remote fetch and increment	77
Using Reduction functions	78
Example 19: Minimum value reduction routine	79
Example 20: Summation using a reduction routine	81
Single-PE Optimization [4]	85
Unrolling Loops	85
Software Pipelining	86
Optimizing a Program with Software Pipelining	86
Selecting the Level of Pipelining	87
Using the <code>Concurrent</code> and <code>ivdep</code> Directives	87
Identifying Loops for Pipelining	88
How Pipelining Works	89
Optimizing for Cache	92
Rearranging Array Dimensions for Cache Reuse	92
Example 21: Unoptimized code	92
Optimizing for Stream Buffers	95
Splitting Loops	96

	<i>Page</i>
Example 22: Original loop	97
Example 23: Splitting loops	97
Example 24: Stripmining	97
Example 25: Splitting loops across if statements	98
Example 26: Splitting individual statements	98
Changed Behavior from Loop Splitting	99
Maximizing Inner Loop Trip Count	100
Example 27: Rearranging array dimensions	100
Minimizing Stream Count	101
Example 28: Minimizing streams	101
Example 29: Reduced streams version	101
Grouping Statements That Use the Same Streams	102
Example 30: Original code	102
Example 31: Grouping statements within the loop	102
Example 32: Loop that will be split into four	102
Example 33: Loop that will be split into two	103
Enabling and Disabling Stream Buffers	103
Optimizing Division Operations	104
Example 34: Original code	105
Example 35: Modified code	106
Vectorization	107
Using the ivdep Directive	110
Bypassing Cache	110
Input/Output [5]	113
Strategies for I/O	113
Using a Single, Shared File	114
Using Multiple Files and Multiple PEs	116

	<i>Page</i>
Using a Single PE	118
Unformatted I/O	119
Sequential, Unformatted Requests	119
Formatted I/O	122
Reduce Formatted I/O	122
Make Large I/O Requests	122
FFIO	123
Memory-resident Data Files	123
Distributed I/O	125
Using the Cache Layer	127
Using Library Buffers	128
Random Access	128
Striping	128
Hardware Access [6]	131
Using E Registers	131
Basics	133
Usage Rules	133
Example of a Single-word Put	134
Example 36: Inline C function to put an <code>int</code> value to a remote processor's memory	135
Using Barrier and Eureka Synchronization Units	137
Example 37: Using BESUs in a C function	139
Using Distributed I/O	139
Global Application Team Termination	139
Glossary [7]	141
Index	153

Figures

Figure 1.	Data transfer comparison	3
Figure 2.	Position of E registers	5
Figure 3.	Flow of data on a CRAY T3E node	7
Figure 4.	Data flow on the EV5 microprocessor	8
Figure 5.	First value reaches the microprocessor	10
Figure 6.	Ninth value reaches the microprocessor	12
Figure 7.	Output stream	14
Figure 8.	An external GigaRing network	16
Figure 9.	Fan-out method used by broadcasting routines	34
Figure 10.	A PvmMax reduction	39
Figure 11.	The gather/scatter process	41
Figure 12.	shmem_put64 data transfer	48
Figure 13.	Identification of neighbors in the ring program.	54
Figure 14.	shmem_iget and shmem_iput transfers	66
Figure 15.	Reordering elements during a scatter operation	70
Figure 16.	shmem_broadcast operation	73
Figure 17.	Results of a shmem_fcollect	76
Figure 18.	Results of a shmem_double_min_to_all	81
Figure 19.	Overlapped iterations	90
Figure 20.	Pipelining a loop with multiplications	91
Figure 21.	Before and after array a has been optimized	94
Figure 22.	Multiple PEs using a single file	114
Figure 23.	Multiple PEs and multiple files	117
Figure 24.	I/O to and from a single PE	118
Figure 25.	Data paths between disk and an array	120

	<i>Page</i>
Figure 26. Data layout for distributed I/O	126
 Tables	
Table 1. Latencies and bandwidths for data cache access	20
Table 2. Latencies and bandwidths for access that does not hit cache	20
Table 3. Functional unit	92

Preface

This publication documents optimization options for the Cray C and C++ compilers running on CRAY T3E systems.

Related Publications

The following documents contain additional information that may be helpful:

- *Cray C/C++ Reference Manual*
Cray C/C++ Reference Manual
- *Cray C/C++ Ready Reference*
- *UNICOS/mk System Libraries Reference Manual*
- *Introducing the MPP Apprentice Tool*
- *Introducing the Cray TotalView Debugger*
- *Message Passing Toolkit: PVM Programmer's Manual*
- *Message Passing Toolkit: MPI Programmer's Manual*
- *CRAY T3E and CRAY T3D Programming Environment Differences*
- *UNICOS/mk System Calls Reference Manual*

Obtaining Publications

The *User Publications Catalog* describes the availability and content of all Cray Research hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, call +1 651 683 5907. Silicon Graphics employees may send electronic mail to orderdisk@sgi.com (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

<u>Convention</u>	<u>Meaning</u>																				
<code>command</code>	Denotes a command, library routine or function, system call, part of an application program, program output, or anything else that might appear on your screen.																				
<code>manpage(x)</code>	<p>Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers:</p> <table border="0"> <tr><td>1</td><td>User commands</td></tr> <tr><td>1B</td><td>User commands ported from BSD</td></tr> <tr><td>2</td><td>System calls</td></tr> <tr><td>3</td><td>Library routines, macros, and opdefs</td></tr> <tr><td>4</td><td>Devices (special files)</td></tr> <tr><td>4P</td><td>Protocols</td></tr> <tr><td>5</td><td>File formats</td></tr> <tr><td>7</td><td>Miscellaneous topics</td></tr> <tr><td>7D</td><td>DWB-related information</td></tr> <tr><td>8</td><td>Administrator commands</td></tr> </table> <p>Some internal routines (for example, the <code>_assign_asgcmd_info()</code> routine) do not have man pages associated with them.</p>	1	User commands	1B	User commands ported from BSD	2	System calls	3	Library routines, macros, and opdefs	4	Devices (special files)	4P	Protocols	5	File formats	7	Miscellaneous topics	7D	DWB-related information	8	Administrator commands
1	User commands																				
1B	User commands ported from BSD																				
2	System calls																				
3	Library routines, macros, and opdefs																				
4	Devices (special files)																				
4P	Protocols																				
5	File formats																				
7	Miscellaneous topics																				
7D	DWB-related information																				
8	Administrator commands																				
<i>variable</i>	Italic typeface denotes variable entries and words or concepts being defined.																				
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.																				
[]	Brackets enclose optional portions of a command or directive line.																				

(glossary, page *number*) References the glossary for a definition of the preceding term.

The following machine naming conventions are used throughout this document:

<u>Term</u>	<u>Definition</u>
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems.
Cray MPP systems	All configurations of the CRAY T3E series.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and part number of the document with your comments.

You can contact us in any of the following ways:

- Send electronic mail to the following address:

`techpubs@sgi.com`

- Send a facsimile to the attention of “Technical Publications” at fax number +1 650 932 0801.
- Use the Suggestion Box form on the Technical Publications Library World Wide Web page:

`http://techpubs.sgi.com/library/`

- Call the Technical Publications Group, through the Technical Assistance Center, using one of the following numbers:

For Silicon Graphics IRIX based operating systems: 1 800 800 4SGI

For UNICOS or UNICOS/mk based operating systems or CRAY Origin2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California 94043-1389

We value your comments and will respond to them promptly.

Background Information [1]

Welcome to CRAY T3E optimization. This chapter gives an overview of the optimization guide and background information on some of its major subjects. If you want to start optimizing your program right away, just select one of the following topics. You can always come back later.

- The Parallel Virtual Machine (PVM) is a portable message-passing protocol for programming the CRAY T3E system and other parallel systems. See Chapter 2, page 23.
- SHMEM stands for shared memory protocol. It is not as portable as PVM or the Message Passing Interface (MPI) but has potentially better performance. See Chapter 3, page 45.
- Single-PE optimizations concern getting the best performance out of each processing element (PE). See Chapter 4, page 85.
- Input/output (I/O) optimizations help you move data between external devices (such as disk) and memory. See Chapter 5, page 113.
- Access to the hardware lets you perform low-level operations that result in additional performance in some situations. See Chapter 6, page 131.

This publication contains a glossary with definitions of terms that might be unfamiliar to you. If you are reading this document online, you can link to the glossary as you encounter a term. Here is an example of a link that will point you to the glossary: [PE \(glossary, page 148\)](#). If you are reading a printed version of the document, you will see a page number in place of the hyperlink.

For background information, see the following topics in this chapter:

- An introduction to two message-passing protocols (see Section 1.1, page 2), including the following subtopics:
 - PVM, see Section 1.1.1, page 2.
 - SHMEM, see Section 1.1.2, page 2.
- A hardware overview (see Section 1.2, page 4), including the following subtopics:
 - Memory characteristics, see Section 1.2.1, page 4.
 - The processing element, or PE, see Section 1.2.2, page 15.

- The network and peripherals, see Section 1.2.3, page 15.
- Memory performance information, see Section 1.2.4, page 19.
- Measuring the performance of your code, see Section 1.3, page 21.

1.1 Message-passing Protocols

When you are optimizing a program on a CRAY T3E system, you may be faced with a number of decisions. One of the first will be which, if any, of the message-passing protocols you should use.

If you want to run the program on more than one vendor's MPP system, portability is a major concern, and you may want to choose PVM. (The Message Passing Interface (MPI) is also available and is widely portable.) If your only concern is the performance of the program, you may want to include shared memory access routines, known collectively as SHMEM.

1.1.1 Parallel Virtual Machine (PVM)

The PVM programming style offers a widely used, standardized method of programming a CRAY T3E system. PVM does not offer the performance of SHMEM, but it is more portable. PVM runs on both Cray massively parallel processing (MPP) systems and Cray parallel vector processing (PVP) systems, as well as on other parallel architectures. It is a *message-passing* system (glossary, page 146), meaning it exchanges explicit messages with other PEs. The messages often contain data, such as array elements.

PVM relieves the programmer of most synchronization concerns. By using explicit calls to send and receive routines, PVM handles its own synchronization in most cases.

For more introductory information on PVM, see the `pvm_intro(1)` man page.

1.1.2 SHMEM

SHMEM is a set of functions that pass data in a variety of ways, provide synchronization, and perform *reductions* (glossary, page 148). SHMEM functions are implemented on Cray MPP systems, multiprocessing Silicon Graphics systems, and Cray PVP systems but not on any other company's computers.

What SHMEM lacks in portability, it makes up for in performance. SHMEM is the fastest of the Cray MPP programming styles.

The reason for the speed is SHMEM's close-to-the-hardware approach. This demands more from the programmer in areas such as *synchronization* (glossary, page 150), which is provided automatically with some of the other programming styles. The following figure shows how the SHMEM functions enhance performance by dispensing with some of the processes followed by PVM.

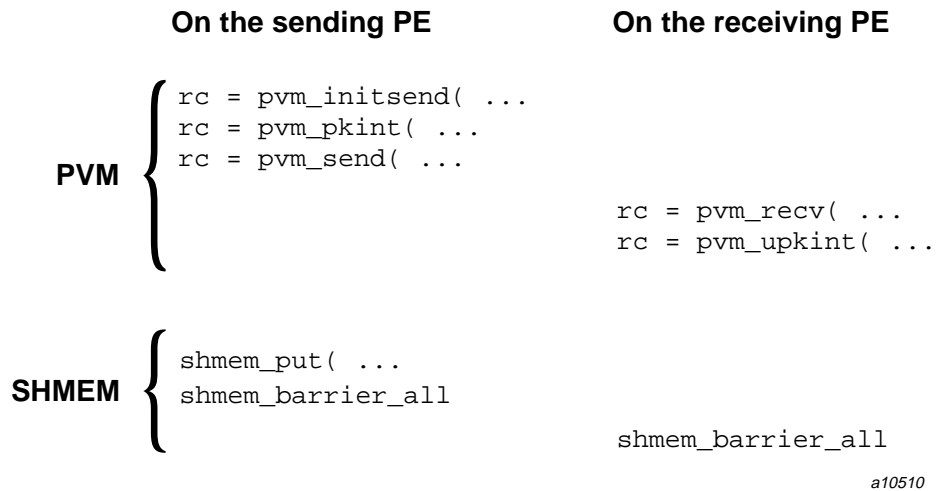


Figure 1. Data transfer comparison

In this example of typical data transfers, PVM requires five steps on the two PEs involved in the transfer: initialize a send buffer, pack the data, send the data, receive the data, and unpack the data. SHMEM requires only one step: send the data. However, one or more synchronization routines are almost always necessary when using SHMEM. You usually must ensure that the receiving PE does not try to use the data before it arrives.

SHMEM does a direct memory-to-memory copy, which is the fastest way to move data on a CRAY T3E system. Adding SHMEM functions to your code, or replacing the statements of another programming style with SHMEM functions, will almost always enhance the performance of your program. Replacing only the major data transfers with `shmem_put` or `shmem_get` can often give you a major speedup with minimal effort. For more information on the functionality available in SHMEM, see the `intro_shmem(3)` man page.

1.2 Hardware

The CRAY T3E hardware performs at a rate of two to three times that of CRAY T3D systems. The following sections contain an overview of the memory system, a brief description of the microprocessor, a look at the network and the system's peripherals, and statistics detailing where the increased performance comes from.

1.2.1 Memory

A memory operation from a PE takes one of two forms:

- A read from, or write to, the PE's own memory (called local memory). Each PE has between 64 Mbytes (8 64-bit Mwords) and 2 Gbytes (256 64-bit Mwords) of memory local to the processor.
- A read from, or write to, remote memory (the memory local to some other PE).

Note: A *word* in this document is assumed to be 64-bits in length, unless otherwise stated.

Operations between the memory of two PEs make use of E registers. E registers are special hardware components that let one PE read from and write to the memory of another PE. E registers are largely reserved for internal use, but users can customize access to them (see Section 6.1, page 131).

E registers are positioned between the PE and the network, as illustrated in the following figure. They are memory-mapped registers, which means they reside in noncached memory and have an address associated with them.

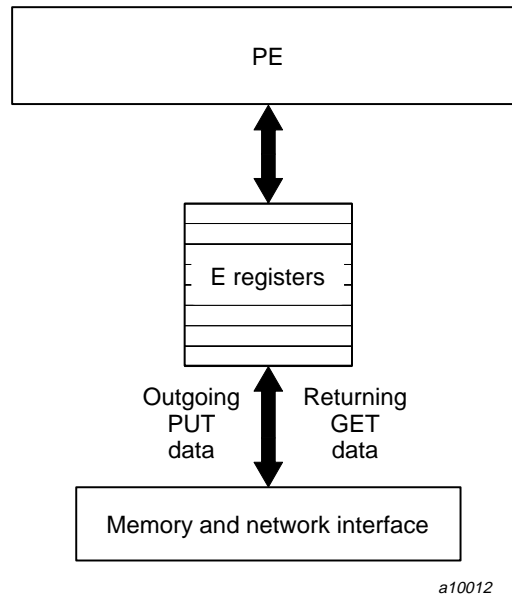


Figure 2. Position of E registers

Operations within a PE, between local memory and the microprocessor, are always faster than operations to or from remote memory. Data read from local memory is accessed through two levels of cache: a 96-Kbyte *secondary cache* (glossary, page 149) and a high-speed, 8-Kbyte *data cache* (glossary, page 143). Data written to local memory passes through a 6-entry *write buffer* (glossary, page 152) and secondary cache.

Cache coherence (glossary, page 142), which was a user concern on the CRAY T3D system, is performed automatically on the CRAY T3E system.

Cache is high-speed memory that helps move data quickly between local memory and the EV5 microprocessor registers. It is still an important part of MPP programming. The `cache_align` directive aligns each specified variable on a cache line boundary. This is useful for frequently referenced variables and for passing arrays in SHMEM (see Section 3.3, page 59). The `cache_align` directive can be used with all of the programming styles described in this guide.

Data cache is a *direct-mapped cache* (glossary, page 143), meaning each local memory location is mapped to one data cache location. When an array, for example, is larger than data cache, a location in data cache can have more than

one of the array addresses mapped to it. Each location is a single, 4-word (32-byte) *line* (glossary, page 146).

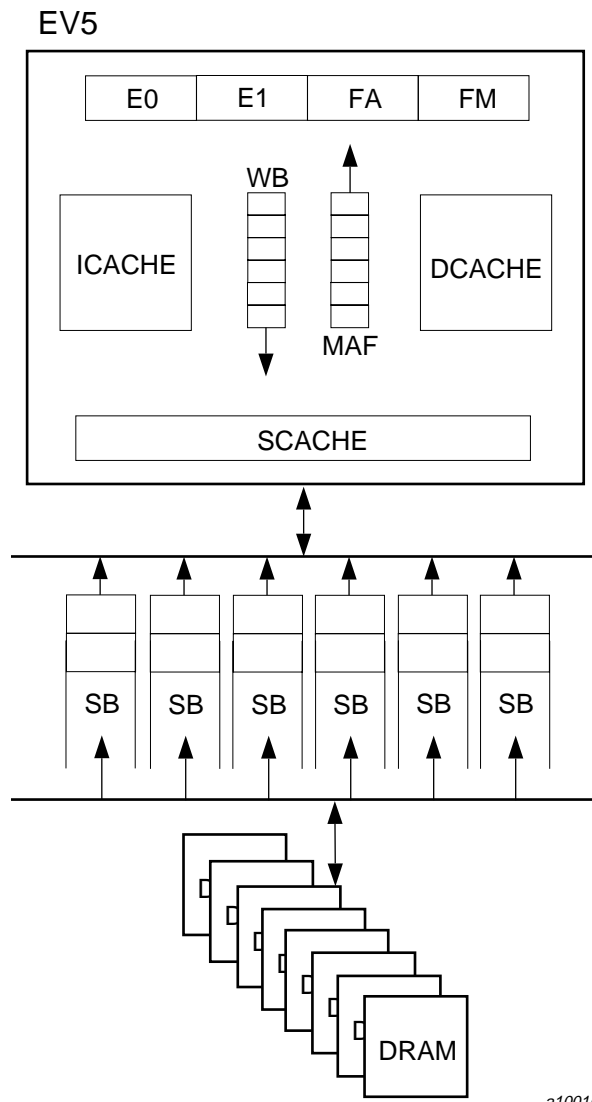
Secondary cache is three-way set associative, and lines are 8 (64-bit) words long, for a total of 64 bytes. In a three-way, *set-associative cache* (glossary, page 149), each memory location is associated with three lines in secondary cache. Which of the three lines to which the data is added is chosen at random. Any line can be selected.

For an example of how data cache and secondary cache work, see Procedure 1, page 9, which describes data movement between local memory and the microprocessor. For an illustration of the components of a PE, see Figure 3. The abbreviations on the figure have the following meanings. Many of these terms are also used in Chapter 4, page 85.

EV5	The RISC microprocessor
E0, E1	Integer functional units
FA, FM	Floating-point functional units
WB	Write buffer
MAF	Missed address file
ICACHE	Instruction cache (not relevant to this discussion)
DCACHE	Data cache
SCACHE	Secondary cache
SB	Stream buffer

DRAM

Local memory

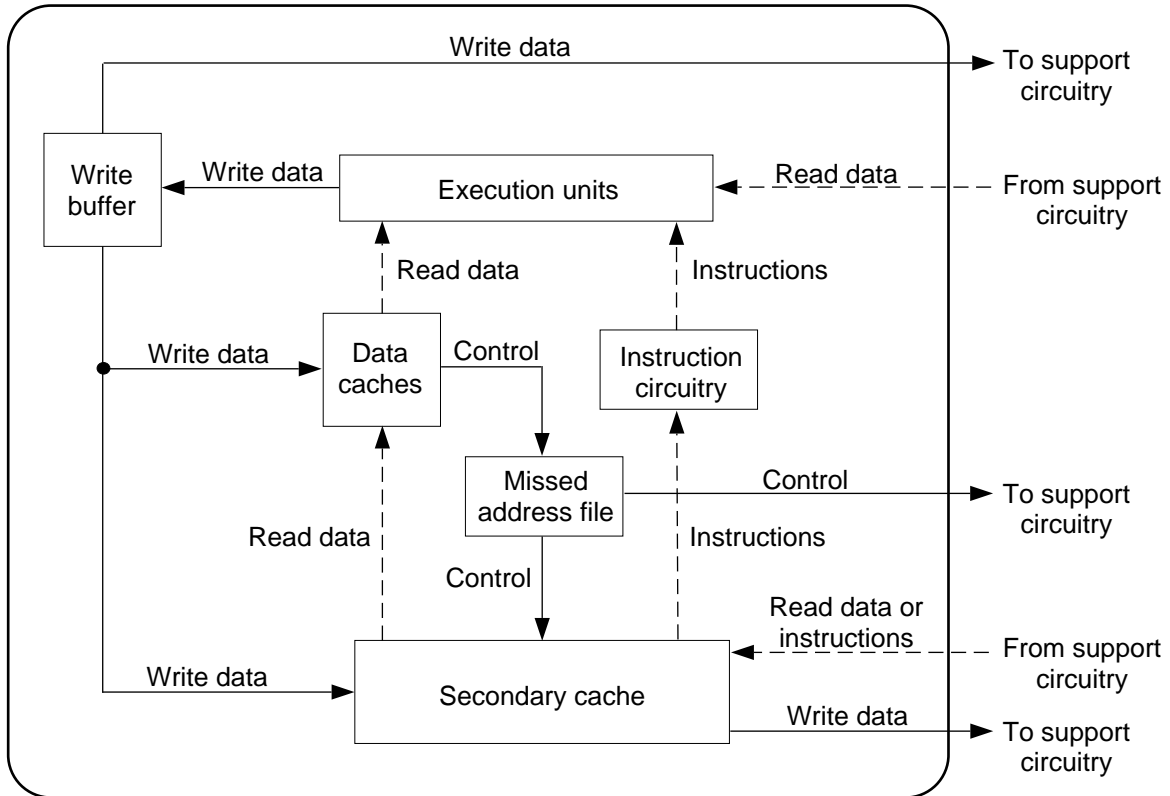


a10015

Figure 3. Flow of data on a CRAY T3E node

The size of a local memory page (marked as DRAM in the preceding figure) depends on the amount of memory in your machine. A memory size of 128

Mbytes, for example, has a page size of 16 Kbytes. The following figure shows more detail from the microprocessor part of the data flow.



a10014

Figure 4. Data flow on the EV5 microprocessor

Each PE has four functional units: two for floating-point operations and two for integer operations. It can handle six concurrent input and output data *streams* (glossary, page 150).

For the following loop, a PE will create streams between memory and the functional units for all of the input operands ($b[i]$, $c[i]$, and so on) and one stream between the functional units, through the write buffer, through secondary cache, and back to memory for the output operand ($a[i]$):

```
for (i=0; i<1000; i++) {  
    a[i] = b[i] + c[i] + d[i] + e[i] + f[i];  
}
```

By default, as soon as the PE detects two consecutive secondary cache-line misses, it begins to preload subsequent, consecutive locations and form a stream. Data streaming is a major optimization on the CRAY T3E system. For more information on creating streams, see Section 4.4.6, page 103.

The following procedure describes the process of moving data between memory and the microprocessor. It refers only to the key hardware elements:

- EV5 registers
- Write buffer
- Data cache
- Secondary cache
- Stream buffer
- Local memory

The example assumes the following loop:

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

Procedure 1: Moving data from memory

1. An EV5 register requests the value of $b[0]$ from data cache.
2. Data cache does not have $b[0]$. It requests $b[0]$ from secondary cache.
3. Secondary cache does not have $b[0]$. This is the first secondary cache miss. It retrieves a line (8 64-bit words for secondary cache) from local memory.
4. Data cache receives a line (4 64-bit words for data cache) from secondary cache.
5. The register receives $b[0]$ from data cache. The state of the data at this point is as illustrated in the following figure.

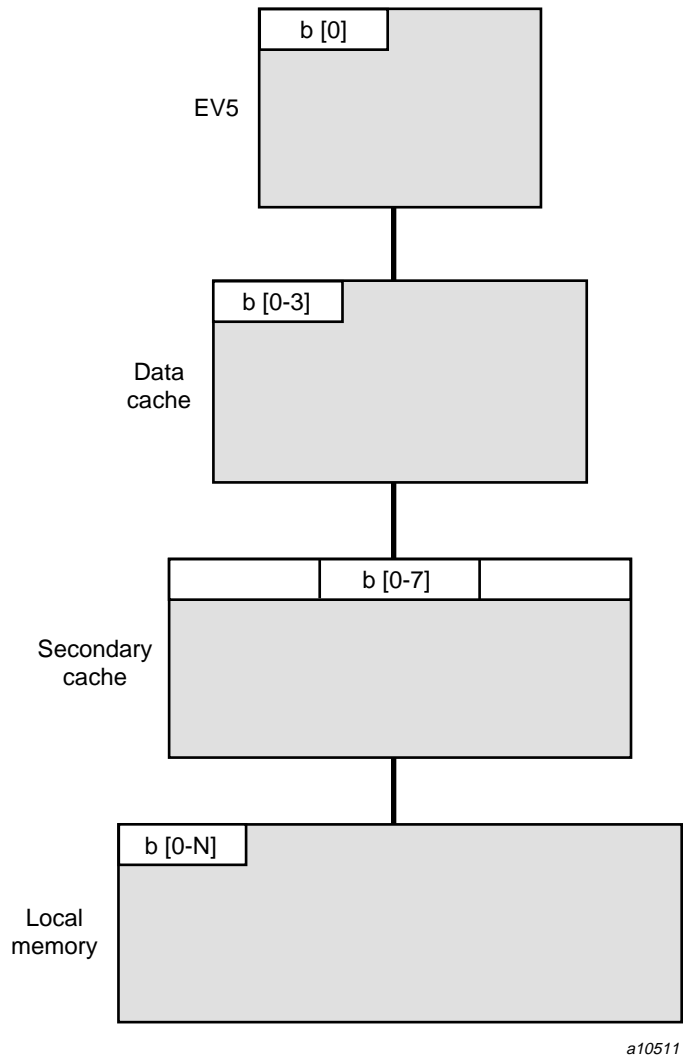


Figure 5. First value reaches the microprocessor

6. When other registers need `b[1]` through `b[3]`, they find them in data cache.
7. When a register needs `b[4]`, data cache does not have it.

8. Data cache requests $b[4]$ through $b[7]$ from secondary cache, which has them and passes them on.
9. Data cache passes $b[4]$ through $b[7]$ on to the appropriate registers as it gets requests for them. When the microprocessor finishes with them, it requests $b[8]$ from data cache.
10. Data cache requests a new line of data elements from secondary cache, which does not have them. This is the second secondary cache miss, and it is the signal to the system to begin streaming data.
11. Secondary cache requests another 8-word line from local memory and puts it into another of its three-line buckets. It may end up in any of the three lines, since the selection process is random.
12. A 4-word line is passed from secondary cache to data cache, and a single value is moved to a register. When the value of $b[8]$ gets to the register, the situation is as illustrated in the following figure.

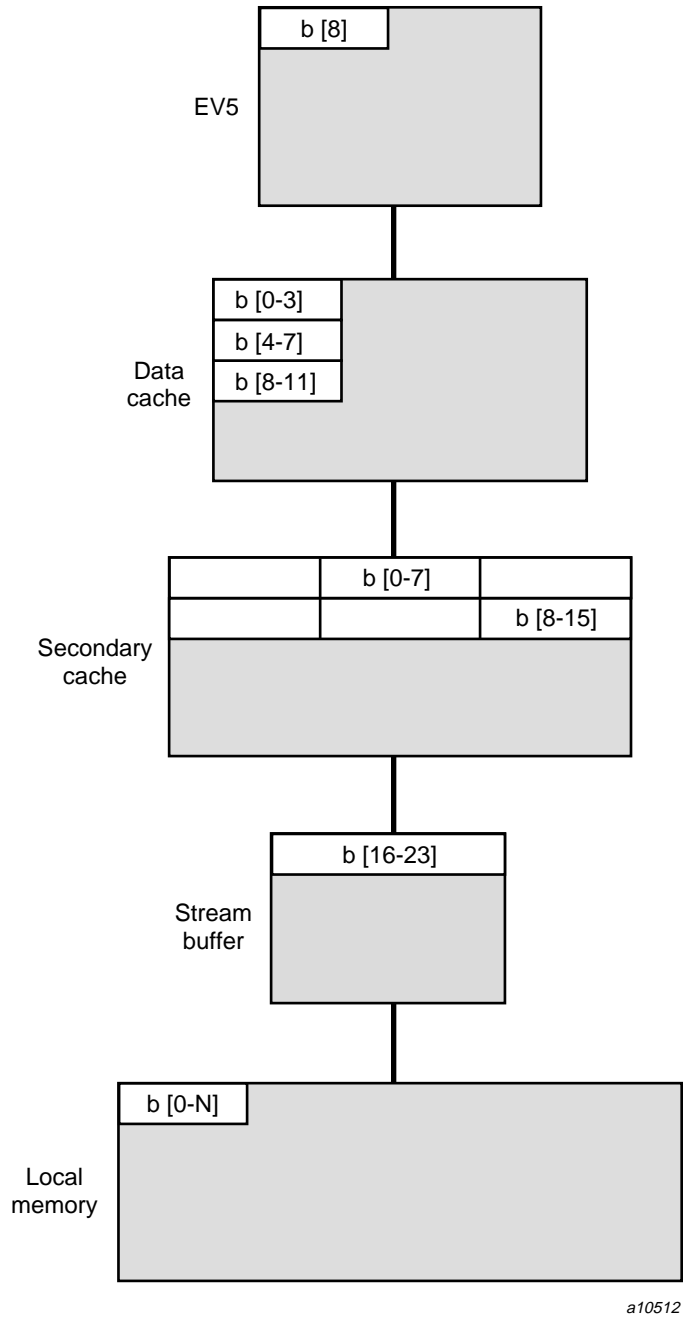
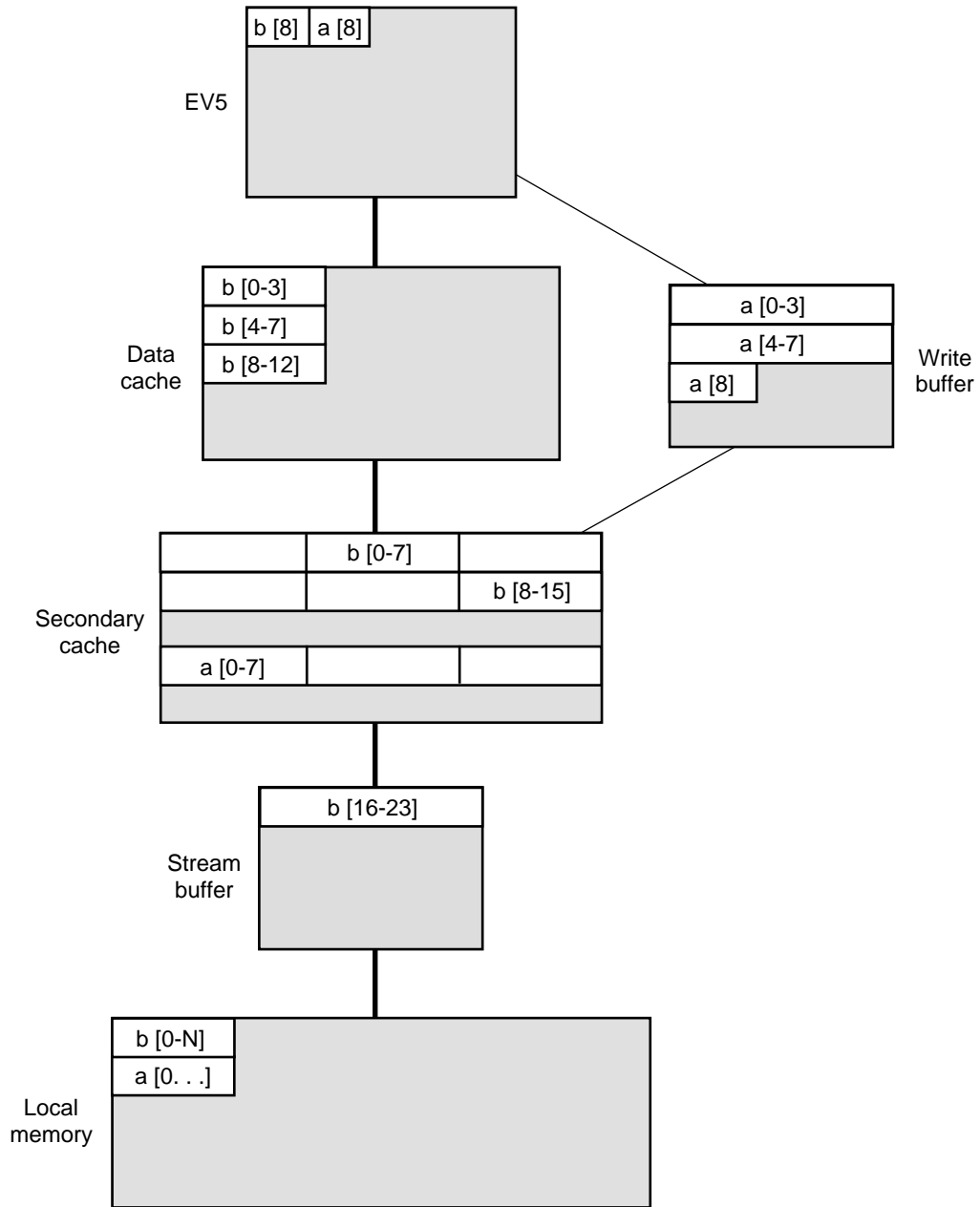


Figure 6. Ninth value reaches the microprocessor

13. Because streaming has begun, data is now prefetched. A stream buffer anticipates the microprocessor's continuing need for consecutive data, and begins retrieving $b[16]$ through $b[23]$ from memory before it is requested. As long as the microprocessor continues to request consecutive elements of b , the data will be ready with a minimum of delay.
14. The process of streaming data between local memory and the registers in the microprocessor continues until the loop is complete.

These steps describe the input stream only. The values of the a array pass through the write buffer and secondary cache, as illustrated in the following figure, on their way back to local memory. Values of a are written to local memory only when a line in secondary cache is dislodged by a write to the same line, or when values of a are requested by another PE.



a10513

Figure 7. Output stream

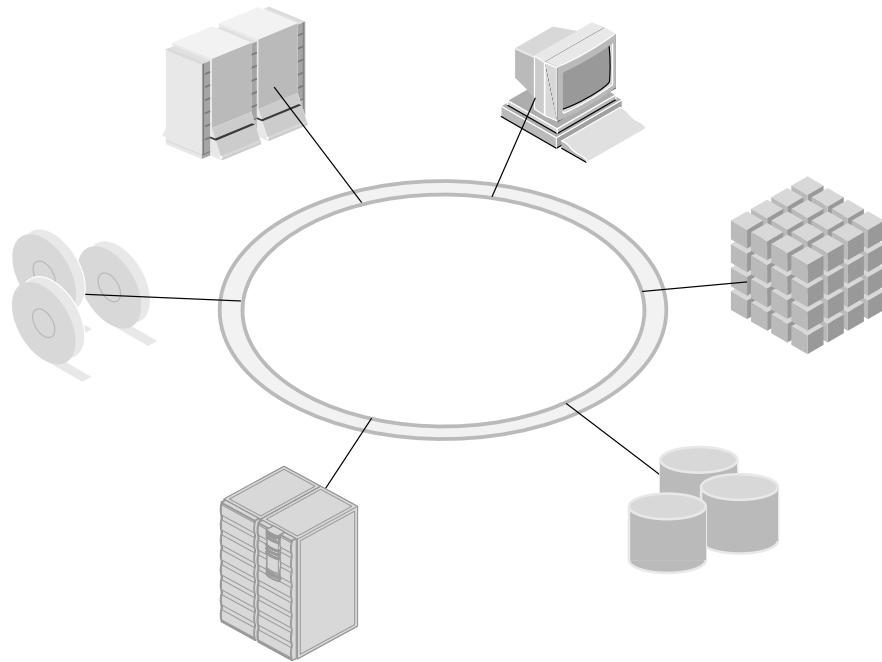
1.2.2 Processing Element

The CRAY T3E processing element contains an EV5 RISC microprocessor manufactured by Digital Equipment Corporation (DEC). It has the following characteristics:

- Either a 3.3 *nanosecond* (glossary, page 147), 300 megahertz clock period (CP) or a 2.2 ns (nanosecond), 450 megahertz CP, compared to the 6.6 ns clock in the CRAY T3D system.
- Thirty-two integer and thirty-two floating-point registers.
- Separate addition and multiplication functional units, including pipelines with a 4-CP (13.3 ns for 300 megahertz chip and 8.9 ns for 450 megahertz chip) execution time. The CRAY T3D system had a single functional unit for multiplication and addition, and the pipeline required 6 CPs (39.7 ns).
- An interface to the network for each PE.

1.2.3 Network and Peripherals

The network that operates between the CRAY T3E system and external systems is based on the *GigaRing* technology (glossary, page 145), which is the Standard Coherent Interface (SCI) with major Cray Research extensions. Physically, the *GigaRing* network is a double ring that passes messages between nodes. Figure 8, page 16, illustrates an external network that includes a CRAY T3E system. (Early CRAY T3E systems do not support multiple hosts on a single *GigaRing* channel. Cray PVP systems, such as CRAY T90 systems, connect to CRAY T3E systems through standard network interfaces, such as HIPPI.)



a10013

Figure 8. An external GigaRing network

The GigaRing channel supports the peripherals and networks described in the following sections.

1.2.3.1 Disk Support

CRAY T3E systems support the following disk drives. On all drives, users can define logical disks. *Disk striping* (glossary, page 144) and *disk mirroring* (glossary, page 144) are also supported.

- MPN-1 SCSI disks:
 - SCSI DD-314 disk drives
 - SCSI DD-318 disk drives
- FCN-1 Fiber Channel disks:
 - DD-308 disk drives.

- DD-308 disk drives, RAID-3.
- DD-308 disk drives, RAID-5. (Support is not yet available.)
- IPI-2 disks:
 - DD-60 disk drives and DA-60 disk arrays
 - DD-62 disk drives and DA-62 disk arrays
 - DD-301 disk drives and DA-301 disk arrays
 - DD-302 disk drives and DA-302 disk arrays
- 100 Mbyte/s HIPPI disks. (Support is not yet available.)
 - ND-12 network disk array
 - ND-14 network disk array
 - ND-30 network disk array
 - ND-40 network disk array

1.2.3.2 Tape Support

The following tape hardware is supported on CRAY T3E systems.

- The following models of SCSI tape drives:
 - SCSI STK 4781/4480 (18 track).
 - SCSI STK 4791/4490 (36 track).
 - SCSI DAT HP C1533-A.
 - SCSI STK 4890 (Twin Peaks).
 - SCSI STK 9490 (TimberLine).
 - SCSI STK SD-3 (RedWood).
 - SCSI IBM 3590 (Magstar).
 - IBM 3490E.
 - EXABYTE 8505.
 - DLT 4000.

- DLT 7000. (Support is not yet available.)
- AMPEX DST310. (Support is not yet available.)
- Block multiplexer tape drives:
 - IBM 3480
 - IBM 3490
 - STK 4480
 - STK 4490
 - 3420-compatible, 9-track reel tape drives with a 256-Kbyte block limit
- ESN-1 ESCON tapes:
 - IBM 3490E.
 - STK 9490 (TimberLine).
 - STK SD-3 (RedWood).
 - IBM 3590 (Magstar). (Support is not yet available.)
- Autoloaders.
 - STK 4400
 - STK WolfCreek
 - IBM 3494
 - STK 9710 (Panther)
 - IBM 3495

1.2.3.3 Network Protocols

The following networking protocols are supported over the GigaRing channel:

- MPN-1 Ethernet (ETN-10).
- MPN-1 Ethernet (ETN-11).
- MPN-1 FDDI (FDI 10).
- HPN-1 100 Mbyte/s HIPPI network, 2 HIPPIs per node.

- HPN-2 200 Mbyte/s HIPPI network, 1 HIPPI per node.
- MPN-1 ATM OC3.
- ATM OC12. (Support is not yet available.)

Network protocols, such as TCP/IP, that are supported on other Cray Research systems are also supported on CRAY T3E systems.

1.2.4 Memory Performance Information

This section presents some of the performance specifications of the CRAY T3E memory system. The times presented are theoretically the optimal times. In most cases, you cannot achieve these times in practice for one reason or another. You may decide at some point in the optimization process that the time required to approach the optimal times is not worth spending. In the tables, ns is nanoseconds, CP is *clock period* (glossary, page 143), and Mbyte/s is megabytes per second.

- Table 1 shows the time required for a CRAY T3E PE, with the microprocessor running at 300 megahertz, to load data from and store data to *data cache* (glossary, page 143). It compares those figures with the CRAY T3D system. For information on data cache, see Section 1.2.1, page 4.
- Table 2 shows performance when loads and stores miss in cache and must go to local memory. The statistics reflect a microprocessor running at 300 megahertz. In practice, the peak cacheable load and store bandwidths listed in this table are not likely to be achieved, because in about 50% of the cases, old data must be removed from cache before new data can be brought in. For more realistic peak numbers, reduce the bandwidths by approximately one-third.

Table 1. Latencies and bandwidths for data cache access

	CRAY T3D latency	CRAY T3D bandwidth	CRAY T3E latency	CRAY T3E bandwidth
Data cache load	20 ns (3 CP per load)	1200 Mbyte/s (1 word per CP)	6.67 ns (2 CP per load)	4800 Mbyte/s (2 words per CP)
Secondary cache load	N/A	N/A	26.67 ns (8 CP per load)	4800 Mbyte/s (2 words per CP)
Data or secondary cache store	N/A	N/A	N/A	2400 Mbyte/s (1 word per CP)

Table 2. Latencies and bandwidths for access that does not hit cache

	CRAY T3D latency	CRAY T3D bandwidth	CRAY T3E latency	CRAY T3E bandwidth
Cacheable load (stream/ read ahead buffer hit)	86 ns (13 CP per load)	320 Mbyte/s (.26 words per CP)	80 ns (24 CP per load)	960 Mbyte/s (.40 words per CP)
Cacheable store (stream/ read ahead buffer hit)				1,200 Mbyte/s (.50 words per CP)
Infinite vector A=B+C through cacheable loads and stores (stride 1, stream hit)				720 Mbyte/s (.30 words per CP)
Infinite vector Y-X*s+Y (SAXPY) through cacheable loads and stores (stride 1, stream hit)				900 Mbyte/s (.37 words per CP)
Cacheable load (local memory page hit)	147 ns	200 Mbyte/s (.16 words per CP)	283 ns	630 Mbyte/s (.30 words per CP)

	CRAY T3D latency	CRAY T3D bandwidth	CRAY T3E latency	CRAY T3E bandwidth
Cacheable store (local memory page hit)		533 Mbytes/s (.50 words per CP)		355 Mbyte/s (.15 words per CP)
Cacheable load (local memory page miss)	247 ns	123 Mbytes/s (.10 words per CP)	417 ns	430 Mbyte/s (.20 words per CP)
Cacheable store (local memory page miss)		209 Mbyte/s (.17 words per CP)		280 Mbyte/s (.10 words per CP)

1.3 Measuring Performance

You can use a variety of methods to time your code. The following are the most popular:

- The Cray MPP Apprentice tool is good for timing a complete program or a subroutine within a program. It will also give you information on how well you have parallelized your program and where you can make further improvement. Because it estimates its own overhead and subtracts that figure from its timings, MPP Apprentice can be counted on to be accurate to within at least 5% of the numbers it generates.
- The `_rtc` intrinsic function returns values from the real-time clock. It is good for timing blocks of code that are part of a subprogram.
- The performance analysis tool (PAT) runs only on the CRAY T3E system. It gives you information on load balancing across multiple PEs, generates and lets you view trace files, displays hardware performance counter information, estimates the amount of time spent in routines, and times individual calls to routines. See the `pat(1)` man page for more information.



Caution: Do not use both the Cray MPP Apprentice tool and `_rtc` on the same code at the same time. MPP Apprentice introduces a significant amount of overhead that will be included in the `_rtc` numbers but not in the numbers that MPP Apprentice itself reports. Distinguishing between the time used by your code and the overhead is difficult.

If your CRAY T3E system has PEs running at different clock rates (for instance, some at 300 megahertz and others at 450 megahertz), you will have to know

what each PE's clock rate is in order to time the program correctly. For information on how your mixed-speed PEs are configured, see your system administrator. The `grmview(1)` command shows you at what speed each *physical* PE runs, but, when you execute your program, physical PEs numbers are mapped to *logical* PE numbers, which are different.

Parallel Virtual Machine (PVM) [2]

The Parallel Virtual Machine (PVM) message-passing library passes messages between PEs to distribute data and to perform other functions necessary for running programs. (The network version of PVM, which enables message passing between computer systems, is not described in this publication.) For background information on PVM, see Section 1.1.1, page 2.

The differences between PVM on a CRAY T3D system and PVM on a CRAY T3E system are few. The major difference is that the channels feature is not implemented on the CRAY T3E system. But optimizations that worked on the CRAY T3D system should still work on the CRAY T3E system.

This chapter describes the following methods of speeding up your PVM program:

- Saving extra transfers by setting the size of a message properly (see Section 2.1, page 24).
- Allocating the most efficient send buffers, depending on the nature of your message (see Section 2.2, page 25).
- Realizing the performance advantage of 32-bit data (see Section 2.3, page 26).
- Using functions that are optimized for sending and receiving stride-1 data (see Section 2.4, page 28).
- Making quick improvements by mixing optimized send and receive functions (see Section 2.5, page 30).
- Avoiding performance pitfalls when initializing and packing data (see Section 2.6, page 30).
- Accomplishing work while you wait for messages (see Section 2.7, page 31).
- Minimizing wait time by avoiding barriers (see Section 2.8, page 32).
- Using broadcast rather than multicast when sending data to multiple PEs (see Section 2.9, page 33).
- Minimizing synchronization time and maximizing work time when receiving data (see Section 2.10, page 37).
- Using the reduction functions to execute an operation on multiple PEs (see Section 2.11, page 38).

- Distributing data from one PE to multiple PEs and gathering data from multiple PEs to a single PE (see Section 2.12, page 40).

2.1 Setting the Size of a Message

Setting the size of a message properly can save you extra transfers and, consequently, message-passing overhead. You can control the size of a message by setting the `PVM_DATA_MAX` environment variable. The default size for the first message sent is 4,096 bytes, or 512 64-bit words, which should be large enough for most messages. If the data in a message is larger than the value of `PVM_DATA_MAX`, however, the data will be divided up into parts, and the parts will be sent in separate messages until all of it has been delivered.

To find the current value of `PVM_DATA_MAX` within your program, use the `pvm_getopt(3)` function, as follows. The variable `max` will hold the maximum message size value, in bytes.

```
max = pvm_getopt(PVM_DATA_MAX);
```

You cannot, however, change the value within the program by using the `pvm_setopt(3)` function. You must reset the value outside of your program, as follows. Specify the new value for `PVM_DATA_MAX` in bytes.

```
% setenv PVM_DATA_MAX 8192
% ./a.out
```

This example changes the value of the maximum message size to 8,192 bytes (or 1,024 64-bit words) for the entire program. The second line executes the program.

Increasing the size of `PVM_DATA_MAX` is not always the best solution. If you have one or two large transfers in your program, but a number of smaller transfers, you may not want to increase the size of all messages. Adjusting the size of `PVM_DATA_MAX` may not help your overall performance. It takes away from the memory available to the application, and a large message is not always transferred quickly, especially when it is broadcast to multiple PEs.

Breaking the large messages up into smaller messages may be faster in some cases. Whether this proves to be faster in your program depends upon the application. You may have to time the program to find out. For information on timing your code, see Section 1.3, page 21.

PVM does not handle large amounts of data in the same way as small amounts. For large transfers (greater than the value of `PVM_DATA_MAX`), the message

contains the first chunk of data and the address of the data block on the sending PE. After the receiving PE unpacks, it uses remote loads to get the remainder of the data.

Often, remote stores used for short messages can occur at the same time as computation on the receiving PE. But with large messages, remote loads require the receiving PE to wait until the loads complete. If the same data is being sent to several PEs, those PEs may all try to do remote loads at the same time, creating a slowdown as they share the limited memory bandwidth.

2.2 Allocating Send Buffers

The `pvm_initsend(3)` function lets you choose what PVM will do with the data it sends. Each of the following three choices can be used to advantage in certain circumstances:

- `PvmDataRaw`
- `PvmDataInPlace`
- `PvmDataDefault`

Assuming your application is running only on the CRAY T3E system, the fastest of the three choices is usually `PvmDataInPlace`, as specified in the following example:

```
bufid = pvm_initsend(PvmDataInPlace);
```

The `PvmDataInPlace` specification has the following advantages and disadvantages:

- It does not copy the data into a send buffer, which is the primary reason for its speed, unless the data streams feature is turned on. If data streams are turned on, `PvmDataInPlace` is the same as `PvmDataRaw`. For more information on data streams, see Section 4.4.6, page 103.
- It requires you to wait until the transfer is complete before accessing the data, which can slow the program down at times.
- You must either provide your own synchronization or send a short message from the receiving PE to let the sending PE know the transfer is complete.
- It is optimized for contiguous (stride-1) data. You lose any performance benefit if your data is not contiguous.

Although it is not always the fastest, the `PvmDataRaw` specification is often considered the most useful of the three for the following reasons:

- It does not convert the data into another format, thereby saving on encoding costs.
- It ensures that the data is copied into send buffers, meaning the original data can be reused (for example, changed) immediately.

If you are sending integer data and the data does not need more than 32 bits of accuracy, you could see a performance benefit using `PvmDataDefault`. Because this form of packing copies only the low-order 32 bits of integer data, you can get twice as much data into the same block packed using `PvmDataRaw`. This can offer some performance benefit with 32-bit data.

Use of the `PvmDataRaw` method is recommended for most transfers, but, as is often the case, which method is best depends on your application.

2.3 the Advantage of 32-bit Data

Passing 32-bit data can be faster than sending each data item in its own 64-bit word. If, however, your data is not aligned on a 64-bit word and you are using `PvmDataInPlace` for packing, your code will probably slow down.

In the following example, data is sent in the `float` format (32 bits on CRAY T3E systems):

Example 1: Transferring 32-bit data

```
1.  /* 32-bit data transfer */
2.  #include <stdio.h>
3.  #include <pvm3.h>
4.  #define N 1000;
5.  main()
6.  {
7.      int me;
8.      int sender=0, receiver=1, mtag=2;
9.      int istat, pkstat, sstat, rstat, upkstat, mytid, i;
10.     float d_send[1000], d_recv[1000];
11.
12.     /* Get PE information */
13.     mytid = pvm_mytid();
14.     me = pvm_get_PE(mytid);
15.
```

```
16.  /* Initialize data */
17.  for (i=0; i <N; i++) {
18.      d_send[i] = i * 1.0;
19.  }
20.
21.  /* Send data from PE 0 */
22.  if (me == sender) {
23.      istat = pvm_initsend(PvmDataRow);
24.      pkstat = pvm_pkfloat(d_send, N, 1);
25.      sstat = pvm_send(receiver, mtag);
26.  } /* End of if */
27.
28.  /* Receive data */
29.  else {
30.      if (me == receiver) {
31.          rstat = pvm_rcv(sender, mtag);
32.          upkstat = pvm_upkfloat(d_recv, N, 1);
33.      }
34.
35.      /* Print results */
36.      printf("Receiver = %d\n", me);
37.      for (i=0; i !=8; i++)
38.          printf("D_recv[%d] = %f\n", i, d_recv[i]);
39.  } /* End of else */
40. }
```

The program in this example produces the following output:

```
Receiver=1
d_recv[0] = 0.
d_recv[1] = 1.
d_recv[2] = 2.
d_recv[3] = 3.
d_recv[4] = 4.
d_recv[5] = 5.
d_recv[6] = 6.
d_recv[7] = 7.
```

To move on to the next optimization topic, go to Section 2.4, page 28. For a brief description of the above program, continue with this section.

Line 3 references the PVM header file. See your system administrator for its actual location on your system. You can specify the location of the header file with the `-I` option on the `cc(1)` or `CC(1)` command line.

```
3.  #include <pvm3.h>
```

Line 8 defines the sending PE as PE 0 and the receiving PE as PE 1.

```
8.      int sender=0, receiver=1, mtag=2;
```

Line 10 declares the sending and receiving arrays as type `float`, meaning they contain 32-bit data on CRAY T3E systems.

```
10.     float d_send[1000], d_recv[1000];
```

The program uses the fastest of the data encoding arguments to `pvm_initsend`, which is `PvmDataRaw`. If the program were transferring 32-bit integer data, `PvmDataDefault` would be faster. Because the sending and receiving arrays both begin with the first element and use a stride-1 increment, the 32-bit values will be packed and unpacked two data items per 64-bit word.

Lines 23 through 25 initialize the send buffer, pack the 32-bit data, and send the data:

```
23.     istat = pvm_initsend(PvmDataRaw);
24.     pkstat = pvm_pkfloat(d_send, N, 1);
25.     sstat = pvm_send(receiver, mtag);
```

Lines 31 and 32 receive and unpack the 32-bit data:

```
31.     rstat = pvm_recv(sender, mtag);
32.     upkstat = pvm_upkfloat(d_recv, N, 1);
```

2.4 Sending and Receiving Stride-1 Data

The `pvm_psend(3)` and `pvm_precv(3)` functions transfer either a single data item or stride-1 data between two PEs. You do not have to initialize a send buffer or pack and unpack the data when you use `pvm_psend` and `pvm_precv`. For short messages, they run faster than the traditional send and receive functions, `pvm_send` and `pvm_recv`.

The trade-off for the increase in speed is reduced flexibility. Using `pvm_psend`, you are limited to a single block of contiguous data, and it can be sent to just one other PE.

You are also limited to receiving a single block of contiguous data with `pvm_precv`. But, after `pvm_precv` completes, it is done with the message. Using `pvm_recv`, one or more unpack calls may follow the receive call, and information about the message must be kept around in case the user calls

`pvm_bufinfo(3)`. In both the send and the receive, the `pvm_psend` and `pvm_precv` functions offer much simpler and faster code.

The speedups from using `pvm_psend` and `pvm_precv` are most noticeable for small messages, meaning less than the value of the `PVM_DATA_MAX` environment variable (see Section 2.1, page 24). For large messages (greater than `PVM_DATA_MAX`), the performance benefits over `pvm_send` and `pvm_recv` are not significant.

The following example shows a program that passes data by using the `pvm_psend` and `pvm_precv` functions. The source PE (`src` in the program) passes data to the destination PE (`dest`), which in turn passes it back to the `src` PE.

Example 2: `pvm_psend` and `pvm_precv`

```
#include <stdio.h>
#include <pvm3.h>
#include <intrinsics.h>
main()
{
    int src=0, dest=1, len=10, back_and_forth=1000, tag=99;
    int me, i, retsend, retrec, atid, atag, alen;
    long array[10], *parray;

    me = _my_pe();
    parray = &array[0];

    /* Initialize data */
    if (me == src)
        for (i=0; i < len; i++)
            array[i] = i * 1.0;

    /* Send and receive data back_and_forth
       times */
    for ( i=0; i < back_and_forth; i++) {
        if (me == src) {

            /* Send data to dest PE */
            retsend = pvm_psend (dest, tag, parray,
                                len, PVM_LONG);

            /* Receive data from dest PE */
            retrec = pvm_precv (dest, tag, parray,
```

```
        len, PVM_LONG, &atid, &atag, &alen);
    }
    else if (me == dest) {

        /* Receive data from src PE */
        retrec = pvm_precv (src, tag, parray,
            len, PVM_LONG, &atid, &atag, &alen);

        /* Send data to src PE */
        retsend = pvm_psend (src, tag, parray,
            len, PVM_LONG);
    } /* End of else */
} /* End of for loop */
}
```

2.5 Mixing Send and Receive Routines

The `pvm_psend(3)` and `pvm_precv(3)` functions operate together with the `pvm_send` and `pvm_recv` send and receive functions. You can gradually change your code to use them as appropriate, with some benefit accruing with each change. For example, if you find a place that is sending a one-word message using `pvm_send(3)`, you can change it to use `pvm_psend` without worrying about finding and changing the matching `pvm_recv` call. You can also continue to use `pvm_bcast(3)` to broadcast a one-word message to all other PEs, something that can be very efficient (see Section 2.9, page 33), but change the receives to use the `pvm_precv` routine.

2.6 Initializing and Packing Data

You can use PVM in several different ways to send the same message to multiple targets. Some performance benefit is available by using special techniques.

If possible, avoid the following code construct, which initializes and packs the data buffer for each send.

```
for (i=0; i < numpes; i++) {
    bufid = pvm_initsend(PvmDataRow);
    retpk = pvm_pklong (parray, n, 1);
    retsend = pvm_send (i, mtag);
}
```

In the simplest case, you can remove the initialization and packing steps from the loop, as follows:

```
bufid = pvm_initsend(PvmDataRaw);
retpk = pvm_pklong (parray, n, 1);
for (i=0; i < numpes; i++) {
    retsend = pvm_send (i, mtag);
}
```

This is more efficient because you pack the data only once. This means you need only one extra data block (for `PvmDataRaw` or `PvmDataDefault` packing) and one memory copy. Other PVM functions, such as `pvm_bcast(3)` and `pvm_mcast(3)` (see Section 2.9, page 33), provide alternatives to what remains of the `for` loop.

Some programs may include this inefficient code construct but hide it, as in the following example:

```
for (i=0; i < numpes; i++) {
    retmy = myown_send(i, parray, n, 1);
}
```

In this example, `myown_send` is a message-passing envelope that contains PVM code that might look something like the following:

```
int myown_send(int pe, long *pdata, int size, int incr)
{
    int pe, size, incr, rc, bufid, retpk;

    bufid = pvm_initsend (PvmDataRaw);
    retpk = pvm_pklong (pdata, size, incr);
    rc = pvm_send (pe, 99);
    return rc;
}
```

In this case, you are gaining portability but sacrificing performance. You may want to consider using PVM directly or writing a routine that runs more efficiently.

2.7 Working While You Wait

Sometimes you will be able to get work done while waiting for a message to arrive. In such cases, the `pvm_nrecv(3)` function is a good substitute for `pvm_recv(3)`. The `pvm_nrecv` function does a *nonblocking receive* (glossary

page 147), meaning it does not wait until the message arrives but rather returns immediately if there is no message. By checking with `pvm_nrecv` periodically, your program can monitor the arrival of a message and execute other statements while it waits.

The following example outlines one way in which you can make use of `pvm_nrecv`:

```
arrived = pvm_nrecv(-1, 4);
if (arrived == 0) {
    /* Do something else */
}
else {
    /* Process data in message */
}
```

2.8 Avoiding Barriers

Barrier synchronization is appealing because it provides a clear definition of the status of each PE. Although the hardware barrier mechanism on CRAY T3E systems is fast, the waiting time may be long. A barrier requires that all PEs involved arrive at the barrier before any can proceed, so the true speed of a barrier is the speed of the slowest PE. If your application is not well balanced, waiting can slow it down dramatically.

PVM provides a simple form of synchronization. When a PE uses a *blocking receive* (glossary, page 142) to receive a message from another PE, you know that the receiving PE will not go beyond that blocking receive until the sending PE has completed its send. (The `PVMFRCV` routine, for instance, uses a blocking receive.) Yet synchronization is accomplished without involving the other PEs and is combined with the transfer of data. Further synchronization, such as using barriers, is usually not needed.

The follow-on to avoiding barriers is to avoid synchronization of any sort, if possible. Synchronization creates idle PEs, especially when some PEs have more work than others. Although synchronous communication may be easier to understand, asynchronous communication provides better performance.

Unlike some message-passing systems, PVM does not have an *asynchronous receive* (glossary, page 141), in which a receive is issued in parallel with the send and the application later checks the status of this receive. Instead, it provides a nonblocking receive, the `pvm_nrecv` function (see Section 2.7, page 31). On the CRAY T3E system, `pvm_nrecv` provides comparable performance to the `pvm_recv` function. If possible, write your code in such a way that it can use a

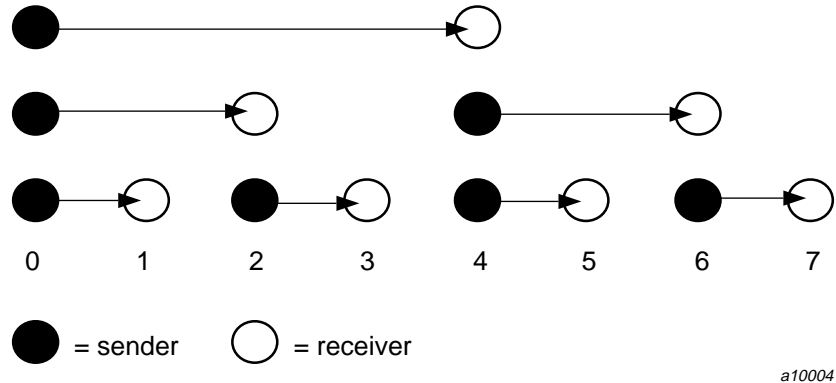
nonblocking receive, so that if the message has not arrived, the code can do other work. See the example in the preceding subsection.

2.9 Using Broadcast or Multicast

The `pvm_bcast(3)` and `pvm_mcast(3)` functions offer two methods of sending messages to multiple PEs in a single call. The *broadcast* (glossary, page 142) function, `pvm_bcast`, sends to all PEs in a group, whether that group consists of all PEs involved in the job or a predefined subset of all PEs. The *multicast* (glossary, page 147) function, `pvm_mcast`, sends to all PEs with PE numbers that appear in an array that you define.

Although `pvm_mcast` provides more flexibility concerning which PEs to send to, `pvm_bcast` is usually faster. If the group name you give to `pvm_bcast` is the global name, specified by a nil point or a pointer to a null string, PVM uses an optimized method to transfer the data. Instead of the broadcasting PE sending directly to all other PEs, it sends to half the PEs. When these PEs receive the message, they each forward it to half the remaining PEs, and so on. (For an illustration, see Figure 9.) This provides better and more scalable performance in the following situations:

- If the number of PEs is approximately 32 or larger. There is usually extra time involved in forwarding such messages, meaning the forwarding method may not be as efficient with a smaller number of PEs.
- If the data packets are small (less than or equal to `PVM_DATA_MAX`). If they are larger, the forwarding method is abandoned, and all the receiving PEs try to do remote loads from the sending PE at more or less the same time.



a10004

Figure 9. Fan-out method used by broadcasting routines

If you use a group name representing a subset of the PEs, there is no special optimization. PVM simply goes through the list of PEs in the group and sends to each PE.

The `pvm_mcast` function does not offer special optimizations. PVM goes through the specified array of PE numbers and sends to each PE.

The following two examples use `pvm_bcast` and `pvm_mcast`, respectively, to transfer an array of 10 64-bit elements to all other PEs attached to the job:

Example 3: `pvm_bcast`

```

#include <pvm3.h>
#include <stdio.h>

main()
{
    int len = 10, mtag = 99;
    int mytid, me, npes, bufid, i;
    int retpk, retbc, retrecv, retupk;
    double arr[10];

    /* Use PVM method of obtaining task id, */
    /* PE number, and number of PEs */
    mytid = pvm_mytid();
    me = pvm_get_PE(mytid);
    npes = pvm_gsize(NULL);

```

```
/* PE 0 initializes, packs, and sends */
/* the array of 10 elements */

if (me == 0) {

    /* Initialize array */
    for (i=0; i<len; i++)
        arr[i] = (i+1) /2.0;

    /* Send array to all PEs */
    bufid = pvm_initsend(PvmDataRow);
    retpk = pvm_pkdouble(arr, len, 1);
    retbc = pvm_bcast(NULL, mtag);

    /* All other PEs receive it */
}
else {
    retrecv = pvm_recv(0, mtag);
    retupk = pvm_upkdouble(arr, len, 1);

    /* A representative PE prints the array */
    if (me == 2) {
        printf("The array values are: ");
        for (i=0; i<len; i++)
            printf("%f ", arr[i]);
    } /* End of inner if */
} /* End of else */
} /* End of main */
```

Example 4: pvm_mcast

```
#include <pvm3.h>
#include <stdio.h>

main()
{
    int len = 10, mtag = 99;
    int mytid, me, npes, bufid, i;
    int retpk, retmc, retrecv, retupk;
    int pe_arr[10];
    double arr[10];
```

```
/* Use PVM method of obtaining task id, */
/* PE number, and number of PEs */
mytid = pvm_mytid();
me = pvm_get_PE(mytid);
npes = pvm_gsize(NULL);

/* Set up an array of PE numbers */

for(i=0; i<len; i++)
    pe_arr[i] = i+1;

/* PE 0 initializes, packs, and sends */
/* the array of 10 elements */

if (me == 0) {

    /* Initialize array */
    for (i=0; i<len; i++)
        arr[i] = (i+1) /2.0;

    /* Send array to PEs */
    bufid = pvm_initsend(PvmDataRaw);
    retpk = pvm_pkdouble(arr, len, 1);
    retmc = pvm_mcast(pe_arr, npes-1, mtag);
}

/* All other PEs receive it */
else {
    retrecv = pvm_recv(0, mtag);
    retupk = pvm_upkdouble(arr, len, 1);

    /* A representative PE prints the array */
    if (me == 2) {
        printf("The array values are: ");
        for (i=0; i!=len; i++)
            printf("%f ", arr[i]);
    } /* End of inner if */
} /* End of else */
} /* End of main */
```

The output from both programs is as follows:

```
The array values are: 0.5 1. 1.5 2. 2.5 3. 3.5 4. 4.5 5.
```

Because the efficient message-passing system used by `pvm_bcast` becomes more of a factor as the number of PEs increases, the advantage in using `pvm_bcast` is most apparent when more PEs are involved in the job. But even when using as few as eight PEs, `pvm_bcast` still has better performance than `pvm_mcast`.

2.10 Minimizing Synchronization Time When Receiving Data

When a single PE receives data from multiple PEs, use `-1` as the task identifier argument to the `pvm_recv(3)` function to save time receiving the data. The `-1` value allows a PE to receive data from any PE. The following example forces data to be received in the numeric order of the PEs. It assumes that PE 0 is receiving the data. The first argument to the `pvm_recv` function specifies the PE number from which the data is received.

```
offset = 1;
for(i=0; i<npes; i++) {
    istat = pvm_recv(i, msgtag);
    istat = pvm_upklong(&x[msgtag-1], length, 1);
    offset = offset + length;
}
```

In the preceding example, regardless of which PE gets its data there first, PE 0 will wait until the data from PE 1 arrives and is received before it can receive data from any other PE. The following example receives whatever data arrives first:

```
for (i=0; i<npes; i++) {
    istat = pvm_recv(-1, msgtag);
    istat = pvm_upklong(&x[msgtag-1] * length, 1);
}
```

Unless the data in this example can be put into the `x` array in random order, you must check the message tag to find out which PE sent a given message. The example assumes the sending PE has sent its PE number in the message tag. Remember, the data is likely to arrive in a different order for different executions of the program.

A loop such as the following offers a second way to order the arriving data in the receiving array:

```
for(i=0; i<npes; i++) {
    istat = pvm_rcv(-1, msgtag);
    istat = pvm_upkint(source, 1, 1);
    istat = pvm_upklong(&x[source * length], length, 1);
}
```

This example assumes that each PE has sent its identifier in `source`, which is the first part of the message.

The following example assumes the sending PEs did not include their identifiers in the message. Instead, a call to the `pvm_bufinfo` function retrieves the value of the task identifier (the fourth argument), converts it to a PE number using `pvm_get_PE(3)`, and places it in the variable `nextpe`. `nextpe` is then used in the `pvm_upklong` function to provide the element number in the array `x`.

```
for(i=0; i<npes; i++) {
    istat = pvm_rcv(-1, msgtag);
    istat = pvm_bufinfo(bufid, &bytes, &msgtag, &source);
    nextpe = pvm_get_PE(source);
    istat = pvm_upklong(&x[(nextpe-1) * length], length, 1);
}
```

The three preceding methods are approximately equivalent in terms of performance.

2.11 Using the Reduction Functions

A single interface, `pvm_reduce`, performs common reduction functions across multiple PEs, returning the results to a single PE. For instance, if each PE contains an array of integers, `pvm_reduce` can find the largest value in any of the arrays at each location and return those answers to an array on a PE that you specify. By executing the following call, you will end up with an array of the largest values on PE 0, as illustrated in Figure 10:

```
istat = pvm_reduce(PvmMax, ARR, 10, PVM_INT, mtag, "squad1", 0);
```

	PE0	PE1	PE2	PE3
	arr	arr	arr	arr
0	31	6	13	31
1	11	9	7	11
2	91	14	91	8
3	18	3	18	5
4	5	1	3	5
5	4	0	4	4
6	36	36	0	14
7	17	11	17	15
8	20	20	12	18
9	19	19	7	1

a11385

Figure 10. A PvmMax reduction

Reduction functions are faster than other PVM methods of finding the same answers. The following example adds the values at each array position for each instance of the array and returns the sum to that array position on PE 0:

Example 5: A PvmSum example

```
#include <pvm3.h>
#include <stdio.h>

main()
{
    int mytid, me, npes, istat, len=10;
    int arr[10], results[10];
    int i, mtag;

    mytid = pvm_mytid();
    mtag = 99;
    me = pvm_get_PE(mytid);
    npes = pvm_gsize(NULL);
```

```
/* Everyone sums */

/* Initialize the array */
for(i=0; i<len; i++)
    arr[i] = me * i;

/* Get the summers organized */
pvm_barrier(NULL,npes);

/* Find the sums */
istat = pvm_reduce(PvmSum, arr, len, PVM_INT,
                  mtag, NULL, 0);

/* Write the answers from PE 0 */
if(me == 0) {
    printf("The array sums are: ");
    for(i=0; i<len; i++)
        printf(" %d ", arr[i]);
} /* End of if */
} /* End of program */
```

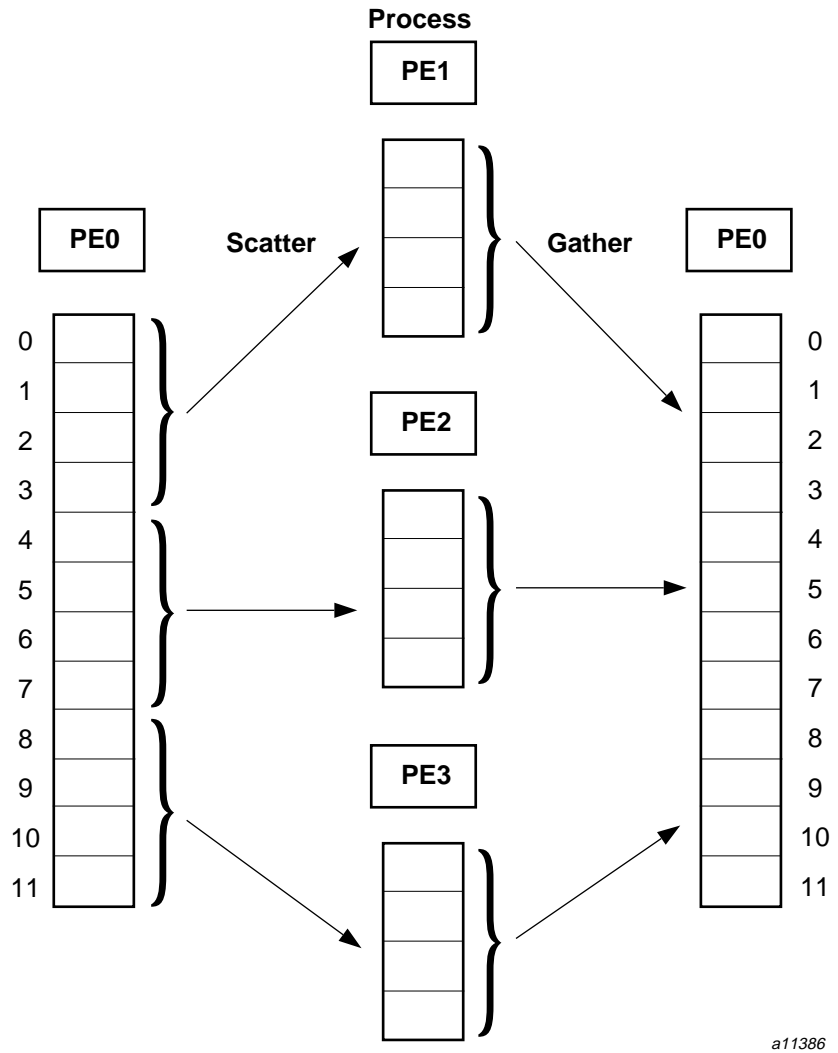
Running the program on eight PEs, the output is as follows:

```
The array sums are: 0 28 56 84 112 140 168 196 224 252
```

2.12 Gathering and Scattering Data

An efficient way to process a large array using PVM is to divide its elements among multiple PEs, process those elements, and reassemble the array on a single PE. The `pvm_gather(3)` and `pvm_scatter(3)` functions are well suited to do just that.

The `pvm_scatter` function distributes sections of an array among a group of PEs, and `pvm_gather` combines arrays from multiple PEs into a single array. Because they are designed for this purpose, they are faster than other PVM functions at gathering and scattering. The process is illustrated in Figure 11.



a11386

Figure 11. The gather/scatter process

The following example collects each PE's `small_c` array into the `big_c` array on the root PE:

Example 6: Gather operation

```
#include <pvm3.h>
#include <stdio.h>
main() {
    int root_pe = 0, ixdim = 64, iydim = 128, msgtag = 9;
    int istat, mype, i, mytid, j, npes;
    double a[64][128], b[128];
    /* Assume 4 PEs, one of which is root_pe */
    double small_c[64], big_c[64*4];

    mytid = pvm_mytid();
    mype = pvm_get_PE(mytid);
    npes = pvm_gsize(0);

    /* Initialize arrays */
    for(i=0; i<ixdim; i++)
        for(j=0; j<iydim; j++)
            a[i,j] = mype;

    for(i=0; i<iydim; i+=2)
        b[i] = -1.2;

    for(i=1; i<iydim; i+=2)
        b[i] = 1;

    for(i=0; i<ixdim; i++)
        small_c[i] = 0.0;

    /* Perform matrix-vector multiplication on each PE */
    for(i=0; i<ixdim; i++)
        for(j=0; j<iydim; j++)
            small_c[i] = small_c[i] + a[i][j] * b[j];

    /* Gather each small_c into root_pe's big_c */
    istat = pvm_gather(big_c, small_c, ixdim, PVM_DOUBLE,
                       msgtag, 0, root_pe);

    if(mype == root_pe)
        for(i=0; i<ixdim*4; i++)
            printf("%f", big_c[i]);
}
```

The following example scatters the `big_x` array into the smaller `small_x` arrays.

Example 7: Scatter operation

```
#include <pvm3.h>
#include <stdio.h>

main() {

    /* Assume 4 PEs, any of which is root_pe */
    int root_pe = 0, iy = 128, ix = iy/4, mtag = 11;
    int istat, mype, mytid, i;
    double small_x[32], big_x[128];

    mytid = pvm_mytid();
    mype = pvm_get_PE(mytid);

    /* Initialize data on root_pe */
    if(my_pe == root_pe)
        for(i=0; i<iy; i++)
            big_x[i] = 1.0 * i;

    istat = pvm_scatter(small_x, big_x, ix, PVM_DOUBLE,
                       mtag, 0, root_pe);

    /* Print from any PE */
    if(mype == 1) {
        printf("For PE %d: \n", mype);
        for(i=0; i<ix; i++)
            printf("Small_x %d is %f\n", i, small_x[i]);
    }
}
```


You can either use shared memory (SHMEM) routines alone or mix them into a program that primarily uses *PVM* (glossary, page 148) or *MPI* (glossary, page 146), thereby offering opportunities for optimizations beyond what the message-passing protocols can provide. Be aware, however, that SHMEM is not a standard protocol and will not be available on machines developed by companies other than Silicon Graphics and Cray Research. SHMEM is supported on Cray PVP systems, Cray MPP systems, and on Silicon Graphics systems.

For background information on SHMEM, see Section 1.1.2, page 2. For an introduction to the SHMEM routines, see the `shmem_intro(3)` man page.

This chapter describes the following optimization techniques:

- Improving data transfer rates in any CRAY T3E program by using SHMEM get and put routines (see Section 3.1, page 46). This section provides an introduction to data transfer, which is the most important capability that SHMEM offers.
- Improving the performance of a PVM or MPI program by adding SHMEM data manipulation routines (see Section 3.2, page 51).
- Avoiding performance pitfalls when passing 32-bit data rather than 64-bit data (see Section 3.3, page 59).
- Copying *strided* (glossary, page 150) data while maintaining maximum performance. The strided data routines enable you, for example, to divide the elements of an array among a set of processing elements (PEs) or pull elements from arrays on multiple PEs into a single array on one PE (see Section 3.4, page 62).
- Gathering and scattering data and reordering it in the process (see Section 3.5, page 67).
- *Broadcasting* (glossary, page 142) data from one PE to all PEs (see Section 3.6, page 71).
- Merging arrays from each PE into a single array on all PEs (see Section 3.7, page 73).
- Executing an *atomic* memory operation (glossary, page 141) to read and update a remote value in a single process (see Section 3.8, page 76).

- Using reduction (glossary, page 148) functions to execute a variety of operations across multiple PEs (see Section 3.9, page 78).

3.1 Using `shmem_get` and `shmem_put` for data transfer

In general, avoiding communications between PEs (including data transfer) improves performance. The fewer the number of communications, the faster your program can execute. Data transfer is, however, often necessary. Finding the fastest method of passing data is an important optimization, and the SHMEM routines are usually the fastest method available.

The `shmem_put` and `shmem_get` functions avoid the extra *overhead* (glossary, page 147) sometimes associated with message passing routines by moving data directly between the user-specified memory locations on local and remote PEs.

The `shmem_put` and `shmem_get` functions perform at virtually the same rates. For large transfers, `shmem_put` offers a different kind of performance advantage by letting the calling PE perform other work while the data is in the network. Because `shmem_put` is asynchronous, it may allow statements that follow it to execute while the data is in the process of being copied to the memory of the receiving PE. The `shmem_get` function forces the calling PE to wait until the data is in *local memory* (glossary, page 146), meaning that no overlapping of data transfer and early work can be done.

Passing data in large chunks is always faster than passing it in small chunks because it saves overhead. Whenever possible, put all of your data (such as an array) into a single `shmem_put` or `shmem_get` call rather than calling the function more than once. The same applies to other SHMEM functions.

In the following example, eight 64-bit words are transferred from PE 1 to PE 0 by using `shmem_put`. PE numbering always begins with 0.

Example 8: Example of a `shmem_put` transfer

```
1.  #include <stdio.h>
2.  #include <mpp/shmem.h>
3.  #include <intrinsics.h>
4.
5.
6.  int me, npes, i;
7.  int source[8], dest[8];
8.  main()
9.  {
10.     /* Get PE information */
11.     me = _my_pe();
```

```
12.  npes = _num_pes();
13.
14.  /* Initialize and send on PE 1 */
15.  if(me == 1) {
16.      for(i=0; i<8; i++)
17.          source[i] = i+1;
18.      shmem_put64(dest, source, 8*sizeof(dest[0])/8, 0);
19.  }
20.
21.  /* Make sure the transfer is complete */
22.  shmem_barrier_all();
23.
24.  /* Print from the receiving PE */
25.  if(me == 0) {
26.      _shmem_udcflush();
27.      printf(" DEST ON PE 0:");
28.      for(i=0; i<8; i++)
29.          printf(" %d%c", dest[i],
30.              (i<7) ? ',' : '\n');
31.  }
32. } /* End of main */
```

See the following figure for an illustration of the transfer.

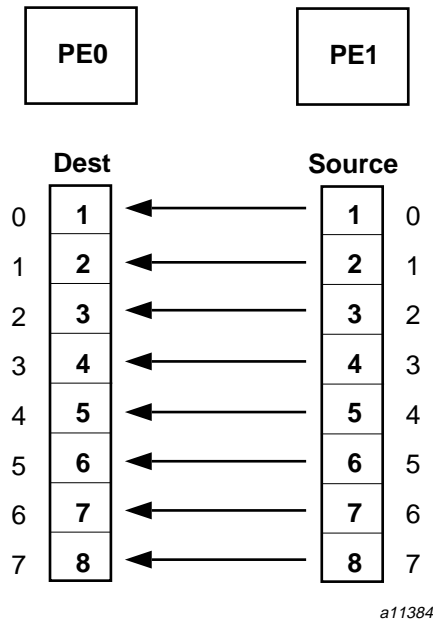


Figure 12. shmem_put64 data transfer

The output from the example is as follows:

```
DEST ON PE 0:  1,  2,  3,  4,  5,  6,  7,  8
```

Defining the number of PEs in a program and the number in an *active set* (glossary, page 141) as powers of 2 (that is, 2, 4, 8, 16, 32, and so on) helped performance on CRAY T3D systems. Also, declaring arrays as powers of 2 was necessary if you were using Cray Research Adaptive Fortran (CRAFT) on CRAY T3D systems. Both have changed as follows on CRAY T3E systems:

- Declaring arrays such as `source` and `dest` as multiples of 8 helps SHMEM speed things up somewhat, since 8 is the vector length of a key component of the PE remote data transfer hardware. Declaring the number of elements as a power of 2 does not affect performance unless that number is also a multiple of 8.
- Defining the number of PEs, whether you are referring to all PEs in a program or to the number involved in an active set, as a power of 2 does not usually enhance performance in a significant way on the CRAY T3E system. Some SHMEM routines, notably `SHMEM_BROADCAST`, still do benefit somewhat from defining the number of PEs as a power of 2.

For information on optimizing an existing MPI program using `shmem_get` and `shmem_put`, see Example 8, page 46. For a complete description of the MPP-specific statements in the preceding example, continue on with this section.

In the `shmem_put` example, line 2 imports the SHMEM include file, which defines parameters needed by many of the routines. The location of the file may be different on your system. Check with your system administrator if you do not know the correct path. If you are using the `module` command and have loaded `craytools`, you do not need this line in your program. Subsequent examples will use the `#include` method.

```
2.  #include <mpp/shmem.h>
```

Line 3 includes the `intrinsic` include file, `intrinsic.h`, which is required if you are using the C++ compiler and are accessing intrinsic functions in your program. This program accesses the `_my_pe` and `_num_pes` intrinsics in lines 11 and 12. If you are using the C compiler, you do not need to include `intrinsic.h`, but it does no harm to do so.

```
3.  #include <intrinsic.h>
```

Note: Most of the examples in this publication use the intrinsic functions `_my_pe` and `_num_pes` because they are at least as fast as their message-passing system equivalents, such as the `shmem_my_pe` and `shmem_n_pes` functions. Both are available on Cray PVP systems as well as Cray MPP systems.

In lines 11 and 12, the `_my_pe` and `_num_pes` functions are called on each PE. Their values are assigned to `me` and `npes`, respectively. Any subsequent reference to the variable `me` returns the number of the calling PE.

```
10.  /* Get PE information */
11.  me = _my_pe();
12.  npes = _num_pes();
```

The value of `me` is tested in line 15. Because the test returns a value of true on PE 1, PE 1 initializes the `source` array in line 17.

```
15.  if(me == 1) {
16.      for(i=0; i<8; i++)
17.          source[i] = i+1;
```

In line 18, PE 1 executes the `shmem_put64` function call that sends the data. It sends eight array elements from its `source` array to the `dest` array on PE 0.

```
18.      shmem_put64(dest, source, 8*sizeof(dest[0])/8, 0);
```

Line 22 is a *barrier* (glossary, page 141), which provides a *synchronization* point (glossary, page 150). No PE proceeds beyond this point in the program until all PEs have arrived. The effect in this case is to wait until the transfer has finished. Without the barrier, PE 0 might print the `dest` array before receiving the data. Calling `shmem_barrier_all` is as fast as calling the `barrier` function directly.

```
22.      shmem_barrier_all();
```

Line 25 selects PE 0, which is passively receiving the data. PVM requires their receiving *tasks* (glossary, page 150) to call routines that receive and unpack the data, but `shmem_put64` places the data directly into PE 0's local memory. PE 0 is not involved in the transfer operation. After being released from the barrier in line 22, PE 0 prints `dest`, and the program exits. The `shmem_udc_flush` function in line 26 is a no-op on CRAY T3E systems, but keeping it in your code makes the program more portable.

```
24.  /* Print from the receiving PE */
25.  if(me == 0) {
26.      _shmem_udc_flush();
```

```
27.     printf(" DEST ON PE 0:");
28.     for(i=0; i<8; i++)
29.         printf("%d", dest[i];
30.         (i<7) ? ',' : '\n');
31.     }
32. } /* End of main */
```

3.2 Optimizing Existing MPI and PVM Programs by Using SHMEM

The following sections show two methods of optimizing an MPI program by using SHMEM routine calls. SHMEM data-transfer routines have lower *latencies* (glossary, page 146) and higher *bandwidth* (glossary, page 141) than comparable message-passing routines. The MPI version of the program is as follows.

Note: The following programs implement a global summation around a ring. They are intended to compare equivalent SHMEM and MPI versions of a program, not to provide an optimal implementation of a global summation. For a faster version of a global summation, see the reduction routines on page 81.

Example 9: MPI version of the ring program

```
1.  #include <stdio.h>
2.  #include <mpi.h>
3.  #include <intrinsics.h>
4.  MPI_Status istat;
5.  int me, npes, total, send, recv, i;
6.  int next, prev, expect, bufid, istat;
7.  main(int *argc, char **argv)
8.  {
9.
10.     MPI_Init(argc,&argv);
11.     /* Get PE values */
12.     me = _my_pe();
13.     npes = _num_pes()
14.
15.     /* Define the ring */
16.     next = me + 1;
17.     if(next >= npes)
18.         next = next - npes;
19.     prev = me -1;
20.     if(prev < 0)
21.         prev = prev + npes;
```

```
22.
23.  /* Initialize data */
24.  send = me;
25.  total = me;
26.
27.  for(i=2; i<=npes; i++) {
28.
29.      /* Send data to next PE */
30.      ierr=MPI_Send(&send,1,MPI_INT,next,99,MPI_COMM_WORLD);
31.
32.      /* Receive data from previous PE */
33.      ierr=MPI_Recv(&recv,1,MPI_INT,prev,99,MPI_COMM_WORLD,
34.                  &istat);
35.      /* Do work */
36.      total = total + recv;
37.      send = recv;
38.  } /* End of for loop */
39.
40.  printf("PE= %d Result= %d Expect= %d", me, total,
41.         (int)((npes-1)*npes*.5));
42. }
```

This program simply passes messages around a ring of PEs. All of the PEs execute all of the statements. Each passes its PE number around and adds the number it receives to the variable `total`. When each PE has seen the PE number of every other PE, each prints out its own PE number and the total it has calculated.

The output from the program, reflecting the random order in which the PEs finish, is as follows:

```
PE = 1 Result = 28 Expect = 28
PE = 5 Result = 28 Expect = 28
PE = 3 Result = 28 Expect = 28
PE = 7 Result = 28 Expect = 28
PE = 2 Result = 28 Expect = 28
PE = 4 Result = 28 Expect = 28
PE = 6 Result = 28 Expect = 28
PE = 0 Result = 28 Expect = 28
```

For the `shmem_get` version of the same program, see Section 3.2.1, page 55; otherwise, see the remainder of this section for a more detailed description of the MPI version.

Line 2 references the MPI header file. If you use the `module(1)` package to define your environment and have loaded `mpt` as one of the modules, this line is not necessary. Line 3 references the C intrinsics file.

```
2.  #include <mpi.h>
3.  #include <intrinsics.h>
```

Line 10 initializes MPI.

```
10.  MPI_Init(argc,&argv);
```

Lines 12 and 13 use functions declared in `intrinsics.h` to determine each PE's number and how many other PEs are involved in the program. Line 12 returns the PE number for each PE and puts the value into the variable `me`. Line 13 determines the number of PEs in the program and puts the value into the `npes` variable. The intrinsic functions `_my_pe` and `_num_pes` are faster than other methods of gaining the same information, and they are implemented on Cray PVP systems. They are not, however, available on systems other than Cray Research systems:

```
11.  /* Get PE values */
12.  me = _my_pe();
13.  npes = _num_pes();
```

The next few statements define a PE's neighbors. The variables `next` and `prev`, defined in lines 16 and 19, specify which of its neighbors most of the PEs will be passing to and which they will be receiving from. Lines 17, 18, 20, and 21 define neighbors for the PEs on the end, namely PE 0 and PE 7 in an 8-PE configuration. Lines 17 and 18 cause PE 7 to pass to PE 0, and lines 20 and 21 cause PE 0 to receive from PE 7.

```
15.  /* Define the ring */
16.  next = me + 1;
17.  if(next >= npes)
18.      next = next - npes;
19.  prev = me - 1;
20.  if(prev < 0)
21.      prev = prev + npes;
```

The values for `next` and `prev` in each PE are as illustrated in the following figure.

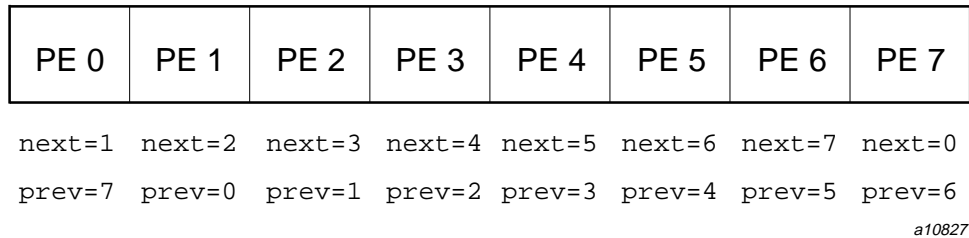


Figure 13. Identification of neighbors in the ring program.

Line 23 initializes the variable that each PE will pass on to `next`, and line 24 initializes the variable that will hold the running total in each PE. Both variables at first contain the number of the respective PE.

```

23.  /* Initialize data */
24.  send = me;
25.  total = me;
    
```

Next comes the `for` loop, within which the MPI statements pass and receive the data. It executes once for every PE except `me`, since the value of `me` is already in the running total. In line 30, the `MPI_Send` function sends the data (`send`) to the next PE. All of the MPI functions are described in more detail on their man pages.

```

29.  /* Send data to next PE */
30.  ierr=MPI_Send(&send,1,MPI_INT,next,99,MPI_COMM_WORLD);
    
```

Line 33 receives the data from the PE that is defined in `prev` and puts it into the variable `recv`.

```

32.  /* Receive data from previous PE */
33.  ierr=MPI_Recv(&recv,1,MPI_INT,prev,99,MPI_COMM_WORLD,
34.  &istat);
    
```

At the end of the `for` loop, each PE updates its running total and moves the number it received into the `send` variable, preparing to pass it on to `next` in the next iteration of the loop.

```

35.  /* Do work */
36.  total = total + recv;
37.  send = recv;
    
```

3.2.1 Optimizing by Using `shmem_get`

To optimize the MPI version of the ring program shown in Example 9, page 51, you can replace the MPI message-passing statements with SHMEM statements. You also need explicit synchronization points, in the form of barriers, to replace the implicit synchronization provided by the MPI send and receive routines. The optimization described in this section and shown in the following example uses the `shmem_get` function; in Section 3.2.2, page 58, MPI is replaced by `shmem_put`.

Instead of passing a single value, the SHMEM versions of the program pass an array of 10 elements. Each of the array elements contains the same value, a PE number.

Example 10: `shmem_get` version of the ring program

```
1.  #include <stdio.h>
2.  #include <mpp/shmem.h>
3.  #define N 10
4.  void work(int n,double *d_total,double *d_send,double *d_recv);
5.  double d_send[N];
6.  main()
7.  {
8.      double d_recv[N], d_total[N];
9.      double d_me;
10.     int i, me, npes, next, prev;
11.     /*
12.     Get PE info
13.     */
14.     me = _my_pe();
15.     npes = _npes();
16.     /*
17.     Define the ring
18.     */
19.     next = me + 1;
20.     if (next >= npes) next = next - npes;
21.     prev = me - 1;
22.     if (prev < 0) prev = prev + npes;
23.     /*
24.     Initialize data
25.     */
26.     d_me = me;
27.     for (i=0; i < N; i++){
28.         d_send[i] = d_me;
```

```
29.         d_total[i] = d_me;
30.     }
31.     for (i=1; i < npes; i++) {
32.
33. /* Synchronize - Make sure data is ready on other PEs */
34.     shmем_barrier_all();
35.
36. /* Get data from previous PE */
37.     shmем_double_get(d_recv, d_send, N, prev);
38.
39. /* Synchronize - Make sure data is ready on other PEs */
40.     shmем_barrier_all();
41. /*
42.  Perform work
43. */
44.     work(N, d_total, d_send, d_recv);
45.     }
46.
47.     printf(" PE = %d  result = %f  expect = %f\n",
48.           me, d_total[0], (npes-1)*npes*.5);
49. }
50. void work(int n,double *d_total,double *d_send, double *d_recv)
51. {
52.     int i;
53.     for (i=0; i < n; i++){
54.         d_total[i] += d_recv[i];
55.         d_send[i] = d_recv[i];
56.     }
57. }
```

Lines 2 and 3 reference the `stdio` and `SHMEM` `#include` files. Line 3 declares the size of the remote and local arrays to be 10.

```
1.  #include <stdio.h>
2.  #include <mpp/shmem.h>
3.  #define N 10
```

Lines 14 and 15 use the same intrinsic functions and get the same information (the number of the calling PE and the number of PEs involved in the job) as the MPI calls in lines 11 and 12 of the MPI version of the program (see Example 9, page 51).

```
14.     me = _my_pe();
15.     npes = _npes();
```

The SHMEM version of the program defines the ring, initializes the data, updates the running total, and writes the output exactly as in the MPI version of the program. Only the method of passing data differs.

Lines 34 and 40, which set barriers, are necessary when using the `shmem_get` function. Synchronization is implicit in the MPI version because of the MPI mode of operation: each send is matched by a receive. You must provide your own synchronization when using SHMEM. The `shmem_barrier_all` function takes advantage of the fast hardware barrier mechanism, making these calls relatively inexpensive. This can replace the implicit synchronization in MPI by this faster synchronization method when you are converting between MPI and SHMEM.

```
33. /* Synchronize - Make sure data is ready on other PEs */
34.     shmem_barrier_all();
35.
36. /* Get data from previous PE */
37.     shmem_double_get(d_recv, d_send, N, prev);
38.
39. /* Synchronize - Make sure data is ready on other PEs */
40.     shmem_barrier_all();
```

Other performance improvements that you will see when converting from MPI to SHMEM data passing are as follows. They apply whether you are using `shmem_put` or `shmem_get`.

- SHMEM does not require separate calls to routines to initialize, to send the data, and to receive the data.
- SHMEM does not require the remote PE to be involved while doing transfers. That means the remote PE is free to do other work, although it does not do so in this example.

If you have written programs for the CRAY T3D system or for Cray PVP systems, you are probably accustomed to *flushing data cache* (glossary, page 145) on the PE receiving the data in order to preserve *cache coherence* (glossary, page 142). That is no longer necessary on the CRAY T3E system. For portability purposes, however, you can leave cache flushing routine calls in your program. They are essentially ignored on CRAY T3E systems, so they do not affect performance, but they are required by CRAY T90 systems and may be required by future systems.

3.2.2 Optimizing by Using `shmem_put`

The `shmem_put` function can deliver the answers as fast as `shmem_get`. The following example shows the `shmem_put` version of the ring program.

Example 11: `shmem_put` version of the ring program

```
1.  /*
2.     program ring_nsum_3
3.     Summing around a ring
4.     shmem_put version
5.  */
6.  #include <stdio.h>
7.  #include <mpp/shmem.h>
8.  #define N 10
9.  void work(int n,double *d_total,double *d_send,double *d_rcv);
10. double d_rcv[N];
11. main()
12. {
13.     double d_send[N], d_total[N];
14.     double d_me;
15.     int i, me, npes, next, prev;
16.  /*
17.     Get PE info
18.  */
19.     me = _my_pe();
20.     npes = _npes();
21.  /*
22.     Define the ring
23.  */
24.     next = me + 1;
25.     if (next >= npes) next = next - npes;
26.     prev = me - 1;
27.     if (prev < 0) prev = prev + npes;
28.  /*
29.     Initialize data
30.  */
31.     d_me = me;
32.     for (i=0; i < N; i++){
33.         d_send[i] = d_me;
34.         d_total[i] = d_me;
35.     }
36.     for (i=1; i < npes; i++) {
37.
```

```

38.  /* Send data to next PE */
39.      shmem_double_put(d_rcv, d_send, N, next);
40.
41.  /* Synchronize -
42.      Wait for data to arrive at other PE (implicit SHMEM_QUIET)
43.      and make sure everyone has data */
44.      shmem_barrier_all();
45.
46.  /* Perform work */
47.      work(N, d_total, d_send, d_rcv);
48.
49.  /* Synchronize - Make sure everyone is ready to continue */
50.      shmem_barrier_all();
51.  }
52.
53.      printf(" PE = %d  result = %f  expect = %f\n",
54.             me, d_total[0], (npes-1)*npes*.5);
55.  }
56. void work(int n,double *d_total,double *d_send,double *d_rcv)
57. {
58.     int i;
59.     for (i=0; i<N; i++) {
60.         d_total[i] += d_rcv[i];
61.         d_send[i] = d_rcv[i];
62.     }
63. }

```

The first half of the program is the same as the `shmem_get` version (see Example 10, page 55), except that the remote variable is now `d_rcv`, which is the target of the data transfer.

In line 39, each PE passes the data to the next PE.

```

38.  /* Send data to next PE */
39.      shmem_double_put(d_rcv, d_send, N, next);

```

3.3 Passing 32-bit Data

Two variants of `shmem_get` and `shmem_put`, `shmem_get32` and `shmem_put32`, are designed and optimized specifically for passing 32-bit data. When used properly, they can pass 32-bit data faster than `shmem_get64` and `shmem_put64`.

When you use the 32-bit routines, try to align either both or neither of the destination array and the source array on a 64-bit boundary. Performance slips significantly when the two are not so aligned, as in the following call:

```
shmem_put32(dest[0], source[5], nlong, pe);
```

Instead, cache-align the two arrays and begin the transfer either on two even-numbered or two odd-numbered array elements. The C and C++ directive `cache_align` serves the purpose of aligning cache.

```
#pragma _CRI cache_align dest, source
shmem_put32(dest, source, nlong, pe);
```

The following 32-bit version of the ring program uses the `shmem_put32` function:

Example 12: 32-bit version of ring program

```
1.  /*
2.     program ring_nsum_3_4
3.     Summing around a ring
4.     shmem_put version
5.  */
6.  #include <stdio.h>
7.  #include <mpp/shmem.h>
8.  #define N 100000 /* Size of transfer */
9.  void work(int n,float *d_total,float *d_send,float *d_recv);
10. float d_recv[N];
11. main()
12. {
13.     float d_send[N], d_total[N];
14.     float d_me;
15.     int i, me, npes, next, prev;
16.  /*
17.     Get PE info
18.  */
19.     me = _my_pe();
20.     npes = _n_pes();
21.  /*
22.     Define the ring
23.  */
24.     next = me + 1;
25.     if (next >= npes) next = next - npes;
26.     prev = me - 1;
```

```
27.         if (prev < 0) prev = prev + npes;
28.  /*
29.   Initialize data
30.  */
31.         d_me = me;
32.         for (i=0; i < N; i++){
33.             d_send[i] = d_me;
34.             d_total[i] = d_me;
35.         }
36.         for (i=1; i < npes; i++) {
37.
38.  /* Send data to next PE */
39.             shmem_put32(d_rcv, d_send, N, next);
40.
41.  /* Synchronize -
42.   Wait for data to arrive at other PE (implicit SHMEM_QUIET)
43.   and make sure everyone has data */
44.             shmem_barrier_all();
45.
46.  /* Make sure cache is updated */
47.             shmem_udcflush();
48.
49.  /* Perform work */
50.             work(N, d_total, d_send, d_rcv);
51.
52.  /* Synchronize - Make sure everyone is ready to continue */
53.             shmem_barrier_all();
54.         }
55.
56.         printf(" PE = %d  result = %f  expect = %f\n",
57.                me, d_total[0], (npes-1)*npes*.5);
58.     }
59. void work(int n,float *d_total,float *d_send,float *d_rcv)
60. {
61.     int i;
62.     for (i=0; i < N; i++) {
63.         d_total[i] += d_rcv[i];
64.         d_send[i] = d_rcv[i];
65.     }
66. }
```

The output from the program is as follows:

```
PE = 0 Result = 28. Expect = 28.
PE = 7 Result = 28. Expect = 28.
PE = 3 Result = 28. Expect = 28.
PE = 1 Result = 28. Expect = 28.
PE = 6 Result = 28. Expect = 28.
PE = 2 Result = 28. Expect = 28.
PE = 4 Result = 28. Expect = 28.
PE = 5 Result = 28. Expect = 28.
```

3.4 Copying Strided Data

The *strided*-data (glossary, page 150) copy operation, using the `shmem_iput` and `shmem_iget` functions, takes data from an array on one PE and delivers it to an array on another PE. You control the stride for both the source and target arrays through arguments in the function calls. The SHMEM versions of the strided copy operation offer both speed and the ability to reorder the elements. (For information on reordering data, see Section 3.5, page 67.)

The following examples take elements from an array on PE 0 and move them to an array on PE 1. Both examples, one using `shmem_double_iget` and the other `shmem_double_iput`, use strides other than 1. These examples are assumed to be 2-PE programs.

Example 13: Passing strided data using `shmem_iget`

```
1.  /*
2.     shmem_iget version
3.     The sending array is accessed with stride 2
4.     The receiving array is accessed with stride 3
5.  */
6.  #include <stdio.h>
7.  #include <mpp/shmem.h>
8.
9.  #define N 100
10. #define SENDER 0
11. #define RECEIVER 1
12.
13. double d_send[2*N];
14.
15. main()
16. {
17.     int me, i;
18.     double d_recv[3*N];
```

```

19.
20. /* Get PE info */
21.     me = shmem_my_pe();
22.
23.     if (me == SENDER) {
24.
25. /* Initialize data */
26.     for (i=0; i < 2*N; i++) {
27.         d_send[i] = i + me + 1.0;
28.     }
29.     for (i=0; i < 3*N; i++) {
30.         d_rcv[i] = 0.0;
31.     }
32.
33. /* Synchronize - Make sure data is ready */
34.     shmem_barrier_all();
35. /* Note: Sender does nothing but synchronize,
36.     Receiver does all the work */
37.     }
38.     else if (me == RECEIVER) {
39.
40. /* Synchronize - Make sure data is ready on other PE */
41.     shmem_barrier_all();
42.
43. /* Get data */
44.     shmem_double_iget(d_rcv,d_send,3,2,N,SENDER);
45.
46. /* Print results */
47.     printf("Receiver=%d d_rcv=%f %f %f %f %f %f %f\n",
48.         me, d_rcv[0],d_rcv[1],d_rcv[2],d_rcv[3],
49.         d_rcv[4],d_rcv[5],d_rcv[6]);
50.     }
51. }

```

Example 14: Passing strided data using `shmem_iput`

```

1. /*
2.     shmem_iput version
3.     The sending array is accessed with stride 2
4.     The receiving array is accessed with stride 3
5. */
6. #include <stdio.h>
7. #include <mpp/shmem.h>

```

```
8.  #define N 100
9.  #define SENDER 0
10. #define RECEIVER 1
11.
12. double d_recv[3*N];
13.
14. main()
15. {
16.     int me, i;
17.     double d_send[2*N];
18.
19.     /* Get PE info */
20.     me = shmem_my_pe();
21.
22.     if (me == SENDER) {
23.
24.     /* Initialize data */
25.
26.         for (i=0; i < 2*N; i++)
27.             d_send[i] = i + me + 1.0;
28.
29.         for (i=0; i < 3*N; i++)
30.             d_recv[i] = 0.0;
31.
32.     /* Send data */
33.         shmem_iput(d_recv, d_send, 3, 2, N, RECEIVER);
34.
35.     /* Synchronize - Make sure data has arrived */
36.         shmem_barrier_all();
37.     }
38.     else if (me == RECEIVER) {
39.
40.     /* Synchronize - Make sure data has arrived */
41.         shmem_barrier_all();
42.
43.     /* Make sure cache is up to date */
44.         shmem_udcflush();
45.
46.     /* Print results */
```

```

47.         printf("Receiver=%d d_recv=%f %f %f %f %f %f\n",
48.             me, d_recv[0],d_recv[1],d_recv[2],d_recv[3],
49.             d_recv[4],d_recv[5],d_recv[6]);
50.     }
51. }

```

Whether `shmem_iput` or `shmem_iget` is faster depends on the stride, but `shmem_iput` is usually the best choice. For one thing, `shmem_iput` returns before all the data is delivered to the remote PE, but `shmem_iget` does not return until the data is delivered to the local PE.

The `shmem_iget` and `shmem_iput` functions are faster than the `shmem_ixget` and `shmem_ixput` functions, but `shmem_ixget` and `shmem_ixput` let you reorder the array. To provide the same functionality, use non-unit strides with the PVM packing and unpacking routine. The relevant lines from a PVM strided copy are as follows:

```

/* Send data */
    bufid = pvm_initsend(PvmDataRaw);
    istat = pvm_pkdouble(d_send, N, 2);
    istat = pvm_send(receiver, mtag);

/* Receive data */
    bufid = pvm_rcv(sender, mtag);
    istat = pvm_upkdouble(d_recv, N, 3);

```

For a description of how to efficiently collect data from all PEs and distribute it to all PEs, see Section 3.5, page 67. Or continue on with the remainder of this section for a brief description of the two SHMEM strided data programs.

The data is transferred from within an `if` statement, beginning on line 38 in the `shmem_iget` version and line 33 in the `shmem_iput` version.

The structures of the two `if` statements are identical in that the sender (PE 0) executes the `if` clause and the receiver (PE 1) executes the `else` clause, but the placement of the data transfer functions differs. The `shmem_double_iget` function, executing on PE 1, retrieves the data from PE 0. The `shmem_double_iput` function, executing on PE 0, copies the data to PE 1.

`shmem_iget` version:

```

44.         shmem_double_iget(d_recv,d_send,3,2,N,SENDER);

```

`shmem_iput` version:

```
33.          shmem_double_iput(d_recv, d_send, 3, 2, N, RECEIVER);
```

As the third and fourth arguments of the function calls specify, both functions take every second array element from the source array (`d_send`) and place them in every third element of the target array (`d_recv`). The arguments to the two functions are the same. See the following figure for an illustration of the transfers:

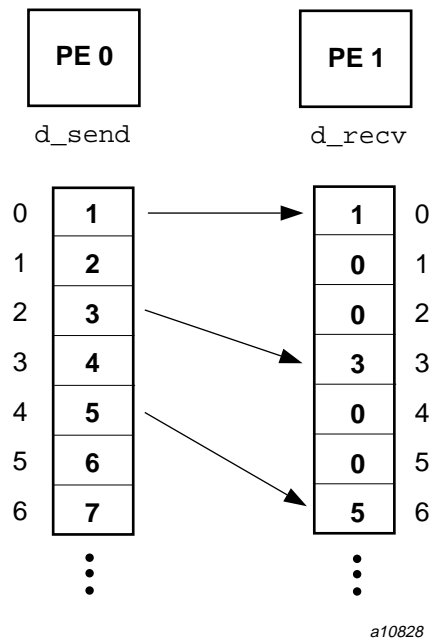


Figure 14. `shmem_iget` and `shmem_iput` transfers

As mentioned earlier (see page 57), the following `shmem_udcflush` call is ignored on CRAY T3E systems. It is left in the `shmem_iput` program for compatibility reasons.

```
43.  /* Make sure cache is up to date */
44.          shmem_udcflush();
```

The output, which is the same for both programs, identifies the PE that received the data and the values of the first seven elements of the `d_recv` array:

```
Receiver=1 d_recv=1. 0. 0. 3. 0. 0. 5.
```

3.5 Gathering and Scattering Data

The functions `shmem_ixget` and `shmem_ixput` copy data (such as arrays) from one PE to another. An index array, specified as an argument, gives the SHMEM functions the additional capability of reordering the array elements that are being passed. SHMEM is faster than PVM because, in order to provide equivalent functionality, PVM must reorder the elements during the pack (for gather) or unpack (for scatter).

The following example shows the SHMEM version of a reordered scatter operation using `shmem_ixput`. It assumes a 2-PE job.

Example 15: `shmem_ixput` version of a reordered scatter

```
1.  /*
2.     shmem_ixput version
3.     Use the index array for the receiving array (scatter)
4.  */
5.  #include <stdio.h>
6.  #include <mpp/shmem.h>
7.  #define NN 100
8.  #define N 10
9.  #define SENDER 0
10. #define RECEIVER 1
11.
12. double d_recv[NN];
13. long index[N] = { 99, 19, 28, 91, 82, 37, 73, 46, 64, 55 };
14.
15. main() {
16.
17.     double d_send[N];
18.     int me, i;
19.
20.     /* Get PE info */
21.     me = _my_pe();
22.
23.     /* Initialize data */
24.     for(i=0; i < N; i++) {
25.         d_send[i] = i + me + 1.0;
26.     }
27.     for(i=0; i < NN; i++) {
```

```
28.         d_recv[i] = 0.0;
29.     }
30.
31.     /* Synchronize - Make sure data arrays are initialized */
32.     shmем_barrier_all();
33.     if (me == SENDER) {
34.
35.     /* Send data */
36.         shmем_ixput(d_recv, d_send, index, N, RECEIVER);
37.
38.     /* Synchronize - Make sure data has arrived */
39.         shmем_barrier_all();
40.     }
41.     else if (me == RECEIVER) {
42.
43.     /* Synchronize - Make sure data has arrived */
44.         shmем_barrier_all();
45.
46.     /* Make sure cache is up to date */
47.         shmем_udcflush();
48.
49.     /* Print results */
50.         printf("Receiver=%d d_recv= %f %f %f %f %f %f\n",
51.             me, d_recv[99], d_recv[19],
52.             d_recv[28],d_recv[91],d_recv[82],d_recv[37]);
53.     }
54. }
```

The following lines reorder the data in a PVM version of the same program:

```
for(i=0; i<N; i++)
    iupk = pvm_upkdouble(d_recv[1+index[I]], 1, 1);
```

In the following lines of the SHMEM program, the index array (defined in line 13) is referenced in the call to the shmем_ixput function, which reorders the array itself.

```
13. long index[N] = { 99, 19, 28, 91, 82, 37, 73, 46, 64, 55 };
36.         shmем_ixput(d_recv, d_send, index, N, RECEIVER);
```

The output from the SHMEM version on a 2-PE configuration is as follows:

```
Receiver=1 d_rcv=1.  2.  3.  4.  5.  6.
```

For an illustration of the three arrays involved in the `shmem_ixput` program, see the following figure.

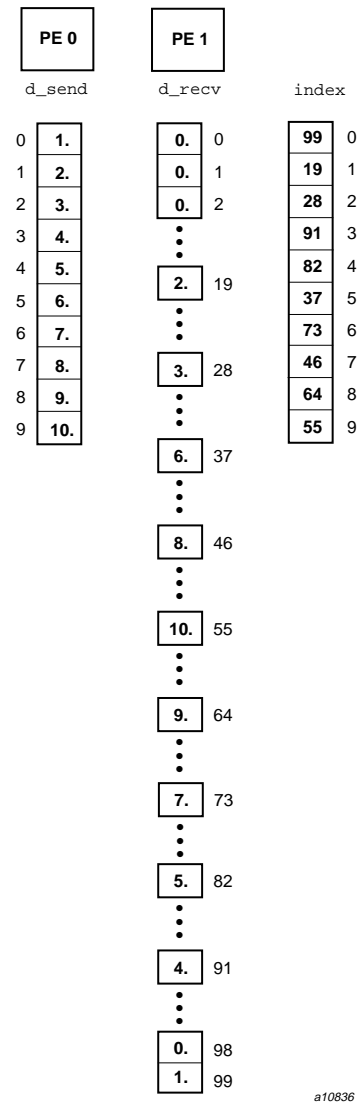


Figure 15. Reordering elements during a scatter operation

A program using the `shmem_ixget` routine would be almost identical to the preceding `shmem_ixput` program, except that the RECEIVER would call the following data transfer statement:

```
shmem_ixget(d_recv, d_send, index, N, SENDER)
```

3.6 Broadcasting Data to Multiple PEs

The `shmem_broadcast(3)` function copies an array on a single PE to a target array on each of the other PEs. It uses the *fan-out* (glossary, page 144) method of copying the data, which makes it the most efficient function for copying from one PE to all other PEs. With 32 or more PEs and relatively little data, the fan-out process will enhance program performance. For an illustration of how fan-out operates on an 8-PE configuration, see Figure 9, page 34. `shmem_broadcast` gives you significant performance advantages when compared to the `pvm_bcast(3)` function.

The following program copies an 8-element array named `source` on PE 0 to an array named `dest` on the three other PEs in a 4-PE *active set* (glossary, page 141). Be sure to use the `pSync` array for synchronization when you use `shmem_broadcast`.

Example 16: One-to-all broadcasting

```
#include <stdio.h>
#include <mpp/shmem.h>

int dest[8], source[8];
long pSync[_SHMEM_BCAST_SYNC_SIZE];

main()
{
    int i, j;

    for (i=0; i < _SHMEM_BCAST_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
    }
    shmem_barrier_all(); /* Wait for PEs to initialize pSync */

    /* Initialize the source array */
    if (_my_pe() == 0) {
        printf("The original array values are: ");
        for (i=0; i < 8; i++) {
            source[i] = (i+1) * (i+1);
            printf(" %d", source[i]);
        }
        printf(" \n");
    }

    /* Broadcast an 8-element array from PE 0 */
```

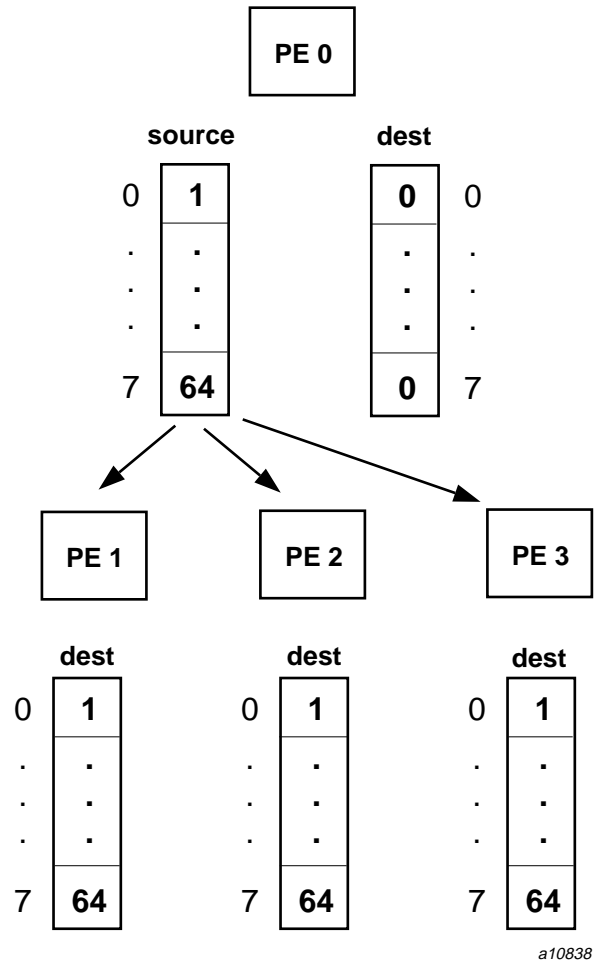
```
shmem_broadcast(dest, source, 8, 0, 0, 0, 4, pSync);

/* Print the dest array on all PEs */
for (i=0; i < 4; i++) {
    if (my_pe() == i) {
        printf("PE %d has ", i);
        for (j=0; j < 8; j++)
            printf("%d", dest[j]);
    } /*End of if */
} /* End of for */
} /* End of program */
```

The output from this program is as follows. Notice that the broadcast did not include the dest array on PE 0.

```
The original array values are: 1  4  9  16  25  36  49  64
PE 0 has 8*0
PE 1 has 1  4  9  16  25  36  49  64
PE 2 has 1  4  9  16  25  36  49  64
PE 3 has 1  4  9  16  25  36  49  64
```

The following figure illustrates the contents of the arrays after the `shmem_broadcast` program has executed.

Figure 16. `shmem_broadcast` operation

3.7 Merging Arrays

The `shmem_fcollect(3)` function quickly combines blocks of data, such as arrays, from multiple PEs into a single array on all PEs. You can consider it a many-to-many broadcast. The following example merges four copies of the array `myvals` into a single array, `allvals`, which is present on all of the PEs. As in the previous example, be sure to include the `pSync` array.

Example 17: Merging arrays

```
#include <stdio.h>
#include <shmem.h>

int myvals[4], allvals[16];
long pSync[_SHMEM_BCAST_SYNC_SIZE];

main()
{
    int i, j, n;

    for (i=0; i < _SHMEM_BCAST_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
    }

    /* Assume 4 PEs */
    npes = _num_pes();
    me = _my_pe();

    /* The values to be passed will be based on PE numbers */
    n = npes * me;
    for(i=0; i < npes; i++) {
        n = n + 1;
        myvals[i] = n;
    }

    /* Wait until all PEs are initialized */
    shmem_barrier_all();

    shmem_fcollect(allvals, myvals, 4, 0, 0, npes, pSync);

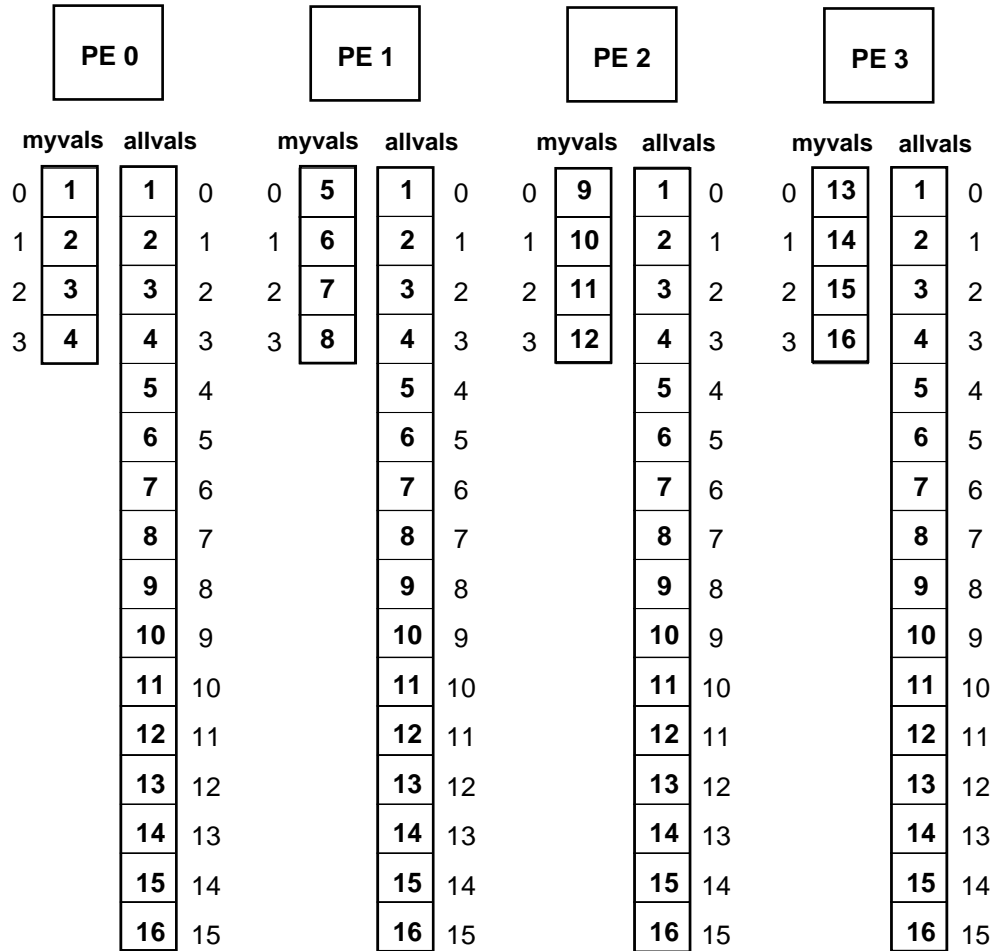
    /* Print allvals on each PE */
    printf("PE %d has ", me);
    for (j=0; j < npes*4; j++)
        printf("%d", allvals[j]);

} /* End of program */
```

The output from the program is as follows. Notice that, unlike `shmem_broadcast`, `shmem_fcollect` sends its data to all of the PEs, including itself.

```
PE 3 has 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
PE 0 has 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
PE 1 has 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
PE 2 has 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

The following figure illustrates the contents of both the `myvals` and the `allvals` arrays on each PE after the example is run. A call to `pvm_gather` followed by a call to `pvm_bcast` would produce the same result as one call to `shmem_fcollect`, but the call to `shmem_fcollect` is faster.



a10839

Figure 17. Results of a shmem_fcollect

3.8 Reading and Updating in One Operation

The shmem_swap(3), shmem_short_finc(3), and shmem_short_fadd(3) functions give you the ability to read and change a remote memory address in a single, *atomic operation* (glossary, page 141). An atomic operation cannot be interrupted by another operation. No other PE can access the same data location while an atomic operation is accessing it. In the following example, the shmem_short_finc function reads and toggles a value. One possible use of

the `shmem_short_finc` function is to implement your own locks, as shown in the following example.

Example 18: Remote fetch and increment

The lock function:

```
1.  #include <mpp/shmem.h>
2.  #define _LOCK          0
3.  #define _SERVING      1
4.  void lock(short *lck) {
5.      short myno, now_serving;
6.
7.      /* Take the next number. */
8.      myno = shmem_short_finc(lck+_LOCK, 0);
9.
10.     /* Wait until my number comes up. */
11.     shmem_short_get(&now_serving, lck+_SERVING, 1, 0);
12.     while(now_serving != myno)
13.         shmem_short_get(&now_serving, lck+_SERVING, 1, 0);
14. }
```

The unlock function:

```
1.  #include <mpp/shmem.h>
2.  #define _SERVING      1
3.  void unlock(short *lck) {
4.
5.      /* Increment the serving number */
6.      (void) shmem_short_finc(lck+_SERVING, 0);
7.  }
```

The following code uses the two functions:

```
1.  short *pvint;                                /* Shared variable */
2.  int mype = _my_pe();
3.
4.  /* Allocate on symmetric
heap */
5.  pvint = shmalloc(sizeof(short)*2);
6.  for (i = 0; i < 2; ++i) {
7.      pvint[i] = 0;
8.  }
9.  shmem_barrier_all();
10.
```

```
11.  if (mype != 0) {
12.    lock(pvint);
13.    printf("PE %d owns the lock \n",mype);
14.    unlock(pvint);
15.  }
16.
17.  shmem_barrier_all();
```

Using the `shmem_barrier_all` function is faster than writing your own synchronization function on CRAY T3E systems, but there are instances in which you may prefer an atomic read and update function. Synchronizing between two PEs is just one example.

3.9 Using Reduction functions

The *reduction* (glossary, page 148) functions combine array elements from each active PE to yield an array of results, which are distributed to all PEs. For instance, one function adds the values at each array location for an array spread across multiple PEs and distributes an array of those sums to each PE. The result is that each PE has an array of answers when the function completes. In its simplest form, a reduction function is a *collective* (glossary, page 143) operation that involves all PEs.

Once again, the SHMEM reduction functions are usually faster than those of PVM and MPI because of the difference in overhead, and they are faster than other means of executing operations across PEs (such as `shmem_put`, `shmem_get`, and standard compiler or library operations).

The reduction routines perform the following operations on arrays:

- Return the smallest value for each array location (see Example 19, page 79)
- Return the largest value for each array location
- Return the product of each array location
- Return the sum of each array location (see Example 20, page 81)
- Return the logical OR of each array location
- Return the logical exclusive OR (XOR) of each array location
- Return the logical product (AND) of each array location

The following example finds the smallest value at each position in four arrays, sends those values to arrays on all PEs, and has PE 0 print the values from its array.

Example 19: Minimum value reduction routine

```
#include <mpp/shmem.h>
#include <stdio.h>
#include <intrinsics.h>
#define NR 4

main() {
    int me, i;
    long pSync[SHMEM_REDUCE_SYNC_SIZE];
    double foo[NR], foomin[NR], pWrk[SHMEM_REDUCE_MIN_WRKDATA_SIZE];

    for(i=0; i < SHMEM_REDUCE_SYNC_SIZE; i++)
        pSync[i] = SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE;

    me = _my_pe();

    srand48();

    for(i=0; i < 4; i++) {
        foo[i] = drand48() * 1000.0;
    }

    /* Print the preliminary numbers on each PE */

    printf("At first, PE %d has: ", me);
    printf("%f %f %f %f\n", foo[0], foo[1], foo[2], foo[3]);

    shmem_barrier_all();

    shmem_double_min_to_all(foomin, foo, NR, 0, 0, 4, pWrk, pSync);

    /* All the values should be the same */
    for(i=0; i<NR; i++) {
        printf("Result on PE %d : ", me);
        printf("%f %f %f %f\n", foomin[0], foomin[1],
                foomin[2], foomin[3]);
    }
}
```

The output from the program is as follows:

```
At first, PE 0 has 158.04 422.40 267.21 272.92
At first, PE 3 has 790.22 318.94 692.47 859.08
At first, PE 1 has 474.13 54.58 847.66 801.64
At first, PE 2 has 474.13 686.76 112.02 330.36
```

```
Result on PE 0 : 158.04 54.58 112.02 272.92
Result on PE 1 : 158.04 54.58 112.02 272.92
Result on PE 2 : 158.04 54.58 112.02 272.92
Result on PE 3 : 158.04 54.58 112.02 272.92
```

The following figure illustrates the contents of the two arrays on each PE at the end of the program.

		PE 0		PE 1	
		foo	foomin	foo	foomin
0		158.04	158.04	474.13	158.04
1		422.40	54.58	54.58	54.58
2		267.21	112.02	847.66	112.02
3		272.92	272.92	801.64	272.92

		PE 2		PE 3	
		foo	foomin	foo	foomin
0		474.13	158.04	790.22	158.04
1		686.76	54.58	318.92	54.58
2		112.02	112.02	692.47	112.02
3		330.36	272.92	859.08	272.92

a10840

Figure 18. Results of a `shmem_double_min_to_all`

In the next example, the summation performed by the ring program is implemented by using the `shmem_int_sum_to_all(3)` reduction function. The reduction is faster than using either `shmem_get` or `shmem_put`, which would pass data 15 times on a 16-PE configuration. It is a valid optimization technique that replaces a slower algorithm with a more efficient algorithm.

Example 20: Summation using a reduction routine

```
#include <mpp/shmem.h>
#include <stdio>
#include <intrinsics.h>
#define N 100000
main() {
```

```
int me, npes, next, prev, istat, i;
int total[N], send[N];
double pWrk(max(N/2+1, _SHMEM_REDUCE_MIN_WRKDATA_SIZE));
long pSync[_SHMEM_REDUCE_SYNC_SIZE];

for(i=0; i<_SHMEM_REDUCE_SYNC_SIZE; i++)
    pSync[i] = _SHMEM_REDUCE_SYNC_SIZE*_SHMEM_SYNC_VALUE;

/* Get PE information */
me = _my_pe();
npes = _num_pes();

/* Initialize data */
for(i=0; i<N; i++)
    send[i] = me;

/* Synchronize, make sure data is on all PEs */
shmem_barrier_all();

/* Perform the work */
shmem_int_sum_to_all(total, send, N, 0, 0, npes, pWrk, pSync);

printf("PE = %d Result = %d Expect = %f", me,
        total[0], (npes - 1) * npes * .5);
}
```

The output from running the program on 16 PEs is as follows. As with most programs involving output from multiple PEs, the order in which the PEs finish is random.

```
PE = 5 Result = 120 Expect = 120.
PE = 12 Result = 120 Expect = 120.
PE = 13 Result = 120 Expect = 120.
PE = 2 Result = 120 Expect = 120.
PE = 11 Result = 120 Expect = 120.
PE = 10 Result = 120 Expect = 120.
PE = 14 Result = 120 Expect = 120.
PE = 8 Result = 120 Expect = 120.
PE = 7 Result = 120 Expect = 120.
PE = 15 Result = 120 Expect = 120.
PE = 6 Result = 120 Expect = 120.
PE = 9 Result = 120 Expect = 120.
```

```
PE = 0 Result = 120 Expect = 120.  
PE = 3 Result = 120 Expect = 120.  
PE = 4 Result = 120 Expect = 120.  
PE = 1 Result = 120 Expect = 120.
```

Comparing this to the PVM, `shmem_get`, and `shmem_put` versions of the ring program, `shmem_int_sum_to_all` delivers the best performance.

Single-PE Optimization [4]

Some of the most significant improvements you can make to your program are not linked to parallelism. They fall into the category of single-PE optimizations. This chapter describes what you can do to get each processing element (PE) running at as close to peak performance as possible.

This chapter makes frequent reference to the hardware, especially the path between local memory and the functional units. For background information on CRAY T3E hardware, see Section 1.2.1, page 4.

To identify the parts on your program that take the most time and to get feedback on performance improvements, use a performance analyzer such as `pat(1)` or the MPP Apprentice tool. For more information, see Section 1.3, page 21.

This chapter addresses the following optimization topics:

- Unrolling loops (see Section 4.1, page 85).
- Using pipelining for loop optimization (see Section 4.2, page 86).
- Making the best use of cache (see Section 4.3, page 92).
- Optimizing stream buffers, which are key to many of the single-PE optimizations (see Section 4.4, page 95).
- Optimizing division operations (see Section 4.5, page 104).
- Vectorizing some math operations within a loop (see Section 4.6, page 107).
- Bypassing cache (see Section 4.7, page 110).

4.1 Unrolling Loops

Loop unrolling is a technique that is beneficial on many computer systems, not just the CRAY T3E system. It can provide the following performance benefits:

- Increasing the basic block size, thus increasing the potential for instruction-level parallelism and covering the latency of memory references.
- Reducing loop overhead, thus potentially increasing the instruction issue rate.
- Eliminating redundant memory references.

- Increasing merging in the missed address file (MAF) of the EV5 processor. For an illustration of where the MAF fits in, see Figure 3, page 7, and Figure 4, page 8.

Although the compiler does unroll loops for you, unrolling is not done by default on CRAY T3E systems. You can enable the unrolling of all loops generated by the compiler by including the `-h unroll` option on the `cc(1)` or `CC(1)` command line. You can direct the compiler to unroll select loops by setting the command line option to `-h nounroll` and placing the `unroll` directive immediately in front of a loop, as follows:

```
#pragma _CRI unroll
for(i=0; i<n; i++) {
```

You can direct the compiler not to unroll a specific loop by placing the `nounroll` directive in front of it.

The `unroll` directive can be applied to any loop of a loop nest, not just the innermost loop. For a loop that is not the innermost loop, the compiler performs a technique called unroll and jam. A loop must meet special criteria, however, to ensure that correct behavior is maintained. In particular, the loop must have no data *dependencies* (glossary, page 143) across its iterations. Also, the compiler performs unroll and jam only on nests in which each loop (except the innermost) contains only one loop. If these criteria are not met, the compiler does not take the risk of performing the optimization.

4.2 Software Pipelining

Software pipelining is an advanced scheduling technique performed by the compiler that overlaps the execution of successive loop iterations in an attempt to optimize utilization of the processor's scheduled resources (such as floating-point functional units, integer functional units, and cache bandwidth).

Software pipelining applies to innermost `for` loops, `do` loops, and `while` loops, provided the loops contain no conditional code or subroutine calls. Software pipelining is often effective when used in conjunction with the `-h vector3` option to vectorize intrinsic functions. For information on vectorization, see Section 4.6, page 107.

4.2.1 Optimizing a Program with Software Pipelining

Before you run your program you can select the level of automatic pipelining with a command-line option. Currently, pipelining can only be turned on or off,

but more options are planned for future releases. Along with the command-line option, you can provide the compiler with additional information on selected loops with a directive.

4.2.1.1 Selecting the Level of Pipelining

The `-h pipelinen` option specifies the level of pipelining in effect for the program. The levels are as follows:

<code>pipeline0</code>	Disables pipelining (the default).
<code>pipeline1</code>	Enables standard pipelining. Numeric results obtained at this level do not differ from results obtained at the <code>pipeline0</code> level.
<code>pipeline2</code>	Currently equivalent to <code>pipeline1</code> .
<code>pipeline3</code>	Currently equivalent to <code>pipeline1</code> .

Enabling pipelining increases compile times, but execution times are often shorter. Timing your program will tell you if the payoff in faster execution is worth the slower compile time.

4.2.1.2 Using the `concurrent` and `ivdep` Directives

The `concurrent` directive lets you provide information about array dependencies. Sometimes the compiler cannot understand ambiguous array references, forcing it to assume dependencies exist (for safety reasons) when there are none. Using the `concurrent` directive, you can tell the compiler that no dependencies exist. The pipeliner uses that information to more aggressively schedule memory references into preceding iterations.

Using the `concurrent` directive with the optional `safe_distance=n` argument allows the compiler to assume that no dependencies exist between the current iteration of the loop and n preceding or subsequent iterations.

The directive immediately precedes the loop that benefits from it, as in the following example:

```
#pragma _CRI concurrent safe_distance=3
for(i=p+1; i<=n; i++)
    x[i] = a[i] + x[i-p]
```

The `concurrent` directive in this example informs the software pipeliner that the relationship $p > 3$ is true. This allows the pipeliner, for example, to safely

load any of the array references $x[i-p]$, $x[i-p+1]$, $x[i-p+2]$, and $x[i-p+3]$ during the i -th loop iteration.

The `ivdep` directive can also be used on the CRAY T3E system to communicate the absence of data dependence to the pipeliner, but the information it provides is more limited than that of the `concurrent` directive, and thus it does not allow as much optimization. The `ivdep` directive only indicates the absence of vector dependencies, which are data dependencies between a memory reference and those memory references that lexically precede it in the loop. For more information on the `ivdep` directive, see Section 4.6.1, page 110.

4.2.2 Identifying Loops for Pipelining

Theoretically, the software pipeliner is guaranteed to optimize utilization for one of the functional units only when there are no *recurrences* (glossary, page 148) in the loop. (Updates of induction variables do not count as recurrences.) This means that parallel loops and vector loops should provide the best candidates for pipelining.

In practice, traditional instruction scheduling will already optimize the use of functional units whenever the loop body contains enough parallel instructions. But if either of the following cases applies to your loop, it is likely that pipelining will significantly increase the performance of a parallel or vector loop:

- The loop body is not too large. (An approximate definition for *large* is a loop that translates to more than 64 assembly instructions.) On large loop bodies with many parallel instructions, the pipeliner will exhaust the available registers sooner than the default scheduler.
- The loop body is not memory bound. Since most memory events are difficult to predict at compile time, the pipeliner cannot manage the memory bandwidth resource accurately.

Sometimes unrolling loops, either manually or by letting the compiler do it, results in loops with large bodies and too many parallel instructions. Pipelining such unrolled loops yields minimal, if any, performance improvement.

Because it relies on overlapping loop iterations to increase performance, anything that decreases the amount of overlap makes it harder for the pipeliner to increase performance. One or more recurrences within the loop can fall into that category. However, you can still get significant performance increases if one or more of the following conditions are satisfied:

- The recurrence can be ignored because it applies to iterations that are too distant to be overlapped. In the following example, if p has a value larger

than about 3, the loop will be translated into a high-performance software pipeline, provided the compiler knows about the lower bound of p . That information can be provided by using the `concurrent` directive.

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

If p is a constant known at compile time, the compiler will eliminate the load instructions to carry the value across iterations in registers. In that case the `concurrent` directive is no longer required or useful.

- The loop body contains enough instructions that are not involved in a recurrence cycle. In that case, the loop initiation interval is probably constrained more by functional unit availability than by the recurrence itself. A typical example might look like the following:

```
for(i=2, i<=n; i++) {
    x[i] = a[i] +x[i-1];           /* Recurrence */
    y[i] = b[i+1] * r +b[i-1] * s /*Unrelated work */
    z[i] = z[i] * y[i] * c[i]     /*Unrelated work */
}
```

Combining parallel and vector loops with recurrent loops before pipelining is worthwhile provided the resulting loop body does not grow too large and does not become memory bound.

4.2.3 How Pipelining Works

The way pipelining attempts to optimize utilization of the scheduled processor resources is best understood through an example:

```
for(i=0; i<n; i++)
    y[i] = x[i];
```

When this loop is compiled, it is translated into assembly instructions, which are then block-scheduled. The resulting code appears as pseudo code in the next example, in which each line corresponds to one clock period (CP). The `t` identifier represents a register.

```
i = 0;
for {
    t = x[i];

    y[i] = t;
    i = i+1;
```

```
    if (i==n) exit;
}
```

Without software pipelining, the processor issues an average of less than one instruction per CP. The execution of successive loop iterations is sequential, as opposed to overlapped. A given iteration does not begin until the previous iteration completes. This loop takes 5 CPs per iteration (assuming hits in data cache, which has a latency of 2 CPs).

However, by overlapping the execution of successive iterations and by creating a new loop body, pipelining produces an average throughput of one iteration every 2 CPs. The initiation interval (the time required to start each iteration) of 2 CPs, along with the fact that every iteration now takes 6 CPs to complete, proves that an overlap of 3 ($6 \div 2$) has been achieved.

After the transformation, the new loop has parts of multiple iterations executing at the same time (see the following figure), has multiple exits, uses twice as much register space, and reorders the update to the loop induction variable ($i = i+1$;) relative to its use in the store to y . But the throughput has increased by a factor of 2.5, and the two integer functional units of the EV5 processor are kept busy within the loop.

```
i=0;
t1=x[i];
i=i+1;

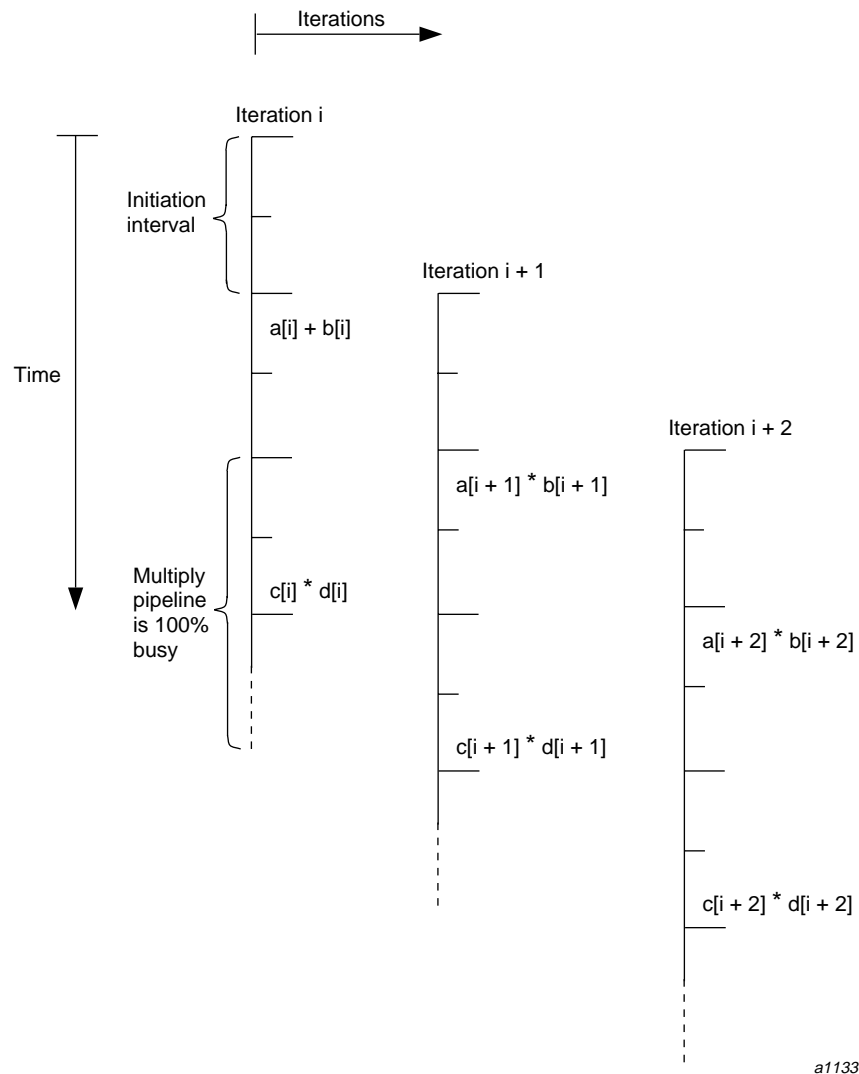
                                t2=x[i];

for {
y[i-1]=t1;                        i=i+1;
if(i==n) exit;                    t1=x[i];
                                y[i-1]=t2;        i=i+1;
                                if(i==n) exit;        t2=x[i];
}
}
```

Figure 19. Overlapped iterations

Figure 20, page 91, illustrates the overlap of iterations for the following loop:

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



a11336

Figure 20. Pipelining a loop with multiplications

The multiplication functional unit has nine stages, each requiring 1 CP to complete its task. But stages 0 through 3 of each pipeline are for instruction decode and issue, and bypasses between the pipelines give you latencies less than the total pipeline length once the instruction is issued. A snapshot of the functional unit when it is completely busy (see the following table) shows

multiplication operations from three separate iterations of the loop, each at a different stage.

Table 3. Functional unit

$c[i+1] * d[i+1]$
$a[i+2] * b[i+2]$
$c[i] * d[i]$
$a[i+1] * b[i+1]$

4.3 Optimizing for Cache

Like CRAY T3D systems, the CRAY T3E system has an 8-Kbyte primary data cache that is *direct mapped* (glossary, page 143). But it also has a 96-Kbyte secondary cache that is three-way *set associative* (glossary, page 149). Although this makes optimizing for cache use less critical on the CRAY T3E system, programming for cache is still an important source of potential performance improvement.

The following section describes how to rearrange array dimensions. For background information on how data cache and secondary cache work, see Procedure 1, page 9, and the associated figures.

4.3.1 Rearranging Array Dimensions for Cache Reuse

You can decrease the execution time for your code by increasing the number of times a piece of data is used while it is resident in cache. One way to increase cache reuse is by making reused array dimensions the fastest moving, or rightmost, dimensions of an array.

The array dimensions in the following example can be rearranged to increase reuse.

Example 21: Unoptimized code

```
double a[3][3][n], b[3][3][n], c[3][3][n]

for(i=0; i<3; i++) {
  for(k=0; k<3; k++) {
    for(j=0; j<n; j++) {
```

```

        c[k][i][j] = a[1][i][j] * b[k][1][j]
                   + a[2][i][j] * b[k][2][j]
                   + a[3][i][j] * b[k][3][j];
    }
}
}

```

One way to detect potential cache reuse in a loop nest is by looking for array references that do not contain all of the loop nest's loop control variables. Dimensions that have references without a loop control variable, or that have loop control variables that differ from reference to reference, are generally candidates for reuse.

In the preceding example, the first and second dimensions have reuse potential because they each have references in which the subscript is not a loop control variable. These dimensions should be made the fastest running, rightmost dimensions. The dimension declarations can be changed as follows:

```
double a[n][3][3], b[n][3][3], c[n][3][3];
```

Rearranging a dimension should be accompanied by changing the order of the nested `for` loops. This optimization maximizes the number of *loop invariant* (glossary, page 146) references in the inner loop. Maximizing the proportion of stride-1 reference streams is also important and often easier to do. In the loop from the preceding example, the number of invariant references is the same for both the `k` and `i` loops. Both the `k` and `i` loops have more invariant references than the `j` loop. The `i` loop gives more stride-1 references, thus the loop nest should be changed in the following way:

```

for(j=0; j<n; j++) {
    for(k=0; k<3; k++) {
        for(i=0; i<3; i++) {
            c[j][k][i] = a[j][1][i] * b[j][k][1]
                       + a[j][2][i] * b[j][k][2]
                       + a[j][3][i] * b[j][k][3];
        }
    }
}

```

For an illustration of how the internal ordering of array `a` changes as a result of the optimization, see the following figure. The array is now processed in the same order it is stored. The illustration assumes that `n` has a value of 6.

a [5][2][2]		a [2][2][5]	
0	a [0][0][0]	0	a [0][0][0]
1	a [1][0][0]	1	a [1][0][0]
2	a [2][0][0]	2	a [2][0][0]
3	a [3][0][0]	3	a [0][1][0]
4	a [4][0][0]	4	a [1][1][0]
5	a [5][0][0]	5	a [2][1][0]
6	a [0][1][0]	6	a [0][2][0]
7	a [1][1][0]	7	a [1][2][0]
8	a [2][1][0]	8	a [2][2][0]
9	a [3][1][0]	9	a [0][0][1]
10	a [4][1][0]	10	a [1][0][1]
11	a [5][1][0]	11	a [2][0][1]
12	a [0][2][0]	12	a [0][1][1]
13	a [1][2][0]	13	a [1][1][1]
14	a [2][2][0]	14	a [2][1][1]
15	a [3][2][0]	15	a [0][2][1]
16	a [4][2][0]	16	a [1][2][1]
17	a [5][2][0]	17	a [2][2][1]
18	a [0][0][1]	18	a [0][0][2]
19	a [1][0][1]	19	a [1][0][2]
20	a [2][0][1]	20	a [2][0][2]
21	a [3][0][1]	21	a [0][1][2]
22	a [4][0][1]	22	a [1][1][2]
23	a [5][0][1]	23	a [2][1][2]
24	a [0][1][1]	24	a [0][2][2]
25	a [1][1][1]	25	a [1][2][2]
26	a [2][1][1]	26	a [2][2][2]
27	a [3][1][1]	27	a [0][0][3]
28	a [4][1][1]	28	a [1][0][3]
29	a [5][1][1]	29	a [2][0][3]
30	a [0][2][1]	30	a [0][1][3]
31	a [1][2][1]	31	a [1][1][3]
32	a [2][2][1]	32	a [2][1][3]
33	a [3][2][1]	33	a [0][2][3]
34	a [4][2][1]	34	a [1][2][3]
35	a [5][2][1]	35	a [2][2][3]
36	a [0][0][2]	36	a [0][0][4]
37	a [1][0][2]	37	a [1][0][4]
38	a [2][0][2]	38	a [2][0][4]
39	a [3][0][2]	39	a [0][1][4]
40	a [4][0][2]	40	a [1][1][4]
41	a [5][0][2]	41	a [2][1][4]
42	a [0][1][2]	42	a [0][2][4]
43	a [1][1][2]	43	a [1][2][4]
44	a [2][1][2]	44	a [2][2][4]
45	a [3][1][2]	45	a [0][0][5]
46	a [4][1][2]	46	a [1][0][5]
47	a [5][1][2]	47	a [2][0][5]
48	a [0][2][2]	48	a [0][1][5]
49	a [1][2][2]	49	a [1][1][5]
50	a [2][2][2]	50	a [2][1][5]
51	a [3][2][2]	51	a [0][2][5]
52	a [4][2][2]	52	a [1][2][5]
53	a [5][2][2]	53	a [2][2][5]

a11152

Figure 21. Before and after array a has been optimized

Of course, loop interchange can only be done if it does not violate data dependencies. The compiler will usually perform interchange under default

optimization if it detects that doing so is both profitable in terms of performance and does not cause incorrect behavior.

Cache reuse in loops such as the one in the preceding example can be further increased by *tiling* (glossary, page 151). Tiling involves further *stripmining* (glossary, page 150) the inner loops and interchanging loops. This is especially profitable when the fastest-moving dimensions are large and less likely to stay entirely in cache. Tiling is currently not performed by the compiler. You must do it manually. For an example of stripmining, see Example 24, page 97.

4.4 Optimizing for Stream Buffers

Each CRAY T3E PE has six *stream* (glossary, page 150) buffers located between secondary cache and local memory (see Figure 3, page 7). When allocated, the stream buffers prefetch data from local memory before it is actually requested, increasing memory *bandwidth* (glossary, page 141) and decreasing *latency* (glossary, page 146). A new stream buffer is allocated when the hardware detects two *secondary cache* (glossary, page 149) line misses that are consecutive in memory. (For an example, see Procedure 1, page 9.)

If an inner loop contains references that allocate more than six different streams, stream buffer *thrashing* (glossary, page 151) occurs, and the stream buffers are ineffective. The following programming techniques will let your program make the best use of stream buffers.

- Letting the Cray Research compiler enhance the performance of your program by splitting loops (see the following section).
- Avoiding undesirable side effects from loop splitting (see Section 4.4.2, page 99).
- Getting the most out of a stream by maximizing the inner loop trip count (see Section 4.4.3, page 100).
- Rearranging the dimensions of an array in order to cut down on the number of streams (see Section 4.4.4, page 101).
- Reducing overhead by grouping statements that use the same stream (see Section 4.4.5, page 102).
- Enabling or disabling stream buffers to get the most benefit from the hardware (see Section 4.4.6, page 103).

4.4.1 Splitting Loops

Splitting an inner loop that allocates more than six stream buffers into a sequence of smaller inner loops that each allocate six or fewer stream buffers is a profitable optimization in many cases. It eliminates stream buffer thrashing and can reduce the execution time of a loop that is making stride-1 references to an array in local memory by up to 40%.

Because splitting loops by hand is tedious and will not always be a performance improvement on other systems, the Cray Research C and C++ compilers provide command-line options (`-h split` and `-h nosplit`) and a source directive (`split`) to help you split loops. The compiler does not split loops by default, because it can degrade performance in some cases, and the compiler cannot currently detect all such cases. For more information on problems with loop splitting and how to detect them, see Section 4.4.2, page 99.

Place the `split` compiler directive immediately before the `for` or `while` statement of the loop to be split. Good candidates for loop splitting are loops with all of the following characteristics:

- Trip counts are higher than 24.
- Performance is bound by memory bandwidth or latency.
- Most references in the loop cause sequences of consecutive cache lines to be read from local memory.
- They have few, if any, `if` statements. Loops with `if` statements cannot be split as profitably as those without `if` statements.

When you place the `split` directive in front of a loop, you are telling the compiler only that the performance of the loop will profit from splitting. You are not telling it that the loop is safe for splitting. The compiler decides on its own if the loop can be safely split. The compiler splits the loop only if it can be done without changing the results of the computation. Therefore, it will not cause incorrect behavior in codes that conform to the International Standards Organization (ISO) and the American National Standards Institute standards for C and C++. Usually, a loop is safe to split under the same conditions that a loop is vectorizable for Cray PVP systems. The compiler splits only inner loops.

The `split` directive also causes the original loop to be *stripmined* (glossary, page 150). Stripmining increases the potential for cache hits between the resulting smaller loops. On the negative side, stripmining can reduce the average stream length for a loop nest, thus reducing the effectiveness of the streams. A strip length of 256 represents a good balance between cache reuse and stream effectiveness.

The following examples show optimizations based on splitting loops:

Example 22: Original loop

```
#pragma _CRI split
for(i=0; i<1000; i++) {
    a[i] = b[i] * c[i];
    t = d[i] * a[i];
    e[i] = f[i] + t * g[i];
    h[i] = h[i] + e[i];
}
```

First, the compiler generates the following loops. Notice the expansion of the scalar temporary `t` into the compiler temporary array `ta` in the following example.

Example 23: Splitting loops

```
for(i=0; i<1000; i++) {
    a[i] = b[i] * c[i];
    ta[i] = d[i] * a[i];
}
for(i=0; <1000; i++) {
    e[i] = f[i] + ta[i] * g[i];
    h[i] = h[i] + e[i];
}
```

Finally, in the following example, the compiler stripmines the loops by 256 to increase the potential for cache hits and reduce the size of arrays created for scalar expansion. Stripmining itself does not provide a performance benefit, but in combination with other optimizations (especially loop splitting, unrolling, and vectorization), it can speed up your program.

Example 24: Stripmining

```
for(i1=0; i1<1000; i1+=256) {
    if(i1 + 256 < 1000)
        i2 = i1 + 256;
    else
        i2 = 1000;

    for(i=i1; i<i2; i++) {
        a[i] = b[i] * c[i];
        ta[i-i1+1] = d[i] * a[i];
    }
}
```

```
for(i=i1; i<i2; i++) {
    e[i] = f[i] + ta[i-i1+1] * g[i];
    h[i] = h[i] + e[i];
}
}
```

In the following example, the compiler splits a loop that includes an `if` statement. The result is two loops, each with an `if` statement.

Example 25: Splitting loops across `if` statements

```
#pragma _CRI split
for(i=0; i<1000; i++) {
    if(a[i] < 0.0)
        b[i] = c[i] * d[i];
    else
        e[i] = f[i] * g[i];
}
```

The preceding loop is split as follows:

```
for(i=0; i<1000; i++) {
    l[i] = a[i] < 0.0;
    if(l[i] != 0)
        b[i] = c[i] * d[i];
}
for(i=0; i<1000; i++) {
    if(l[i] == 0)
        e[i] = f[i] * g[i];
}
```

The compiler does not split up `if` statements that are nested within other `if` statements. Nested `if` statements remain intact at the end of the splitting process.

The compiler also splits individual statements that would allocate more than six stream buffers, as in the following example.

Example 26: Splitting individual statements

```
#pragma _CRI split
for(i=0; i<1000; i++)
    a[i] = b[i] * c[i] + d[i] * e[i] + f[i] * g[i];
```

The preceding loop would be split as follows and then stripmined:

```
for(i=0; i<1000; i++)
    t[i] = b[i] * c[i] + d[i] * e[i];
for(i=0; i<1000; i++)
    a[i] = t[i] + f[i] * g[i];
```

Statements such as those in the preceding example are split only on add, subtract, and multiply operations.

The `-h split` command-line option directs the compiler to try to split all loops in a file. The `-h nosplit` option, which is the default, splits only loops preceded by the `split` directive.

Note: There is potential for increasing the execution time of certain loops by splitting them. Loop splitting also increases compile time, especially when loop unrolling is also enabled. The `split` compiler directive and the `-h nosplit` command-line specification let you select only those loops that will benefit from splitting. Timing a loop both ways can help you determine whether splitting it will enhance performance.

For more information on loop splitting options and directives, see the *Cray C/C++ Reference Manual*, publication SR-2179.

4.4.2 Changed Behavior from Loop Splitting

Enabling loop splitting on CRAY T3E systems may cause previously working, but nonstandard, codes to change behavior.

Loop splitting assumes that all arrays are referenced with subscripts that are within the array's declared bounds. If a code indexes an array with a subscript that is outside of the declared bounds (a nonstandard practice), loop splitting may change the behavior of the code. The change in behavior may result in different numerical results, or it might cause an exception, such as an operand range error or a floating-point exception.

To find the source of the problem, first try to isolate the loop that is causing the change in behavior. Try adding the `-h overindex` option to the compiler command line. It disables some assumptions made by the compiler that *overindexing* (glossary, page 147) is not done. Depending on the nature of the overindexing, this option may cause the program to behave correctly and still give most of the benefit of loop splitting.

4.4.3 Maximizing Inner Loop Trip Count

Once a stream is allocated, it is usually most advantageous to maximize the number of references that go through the stream in order to recover the startup time and increase performance. Reducing the number of streams also reduces the number of stream startups.

One technique for increasing the references to a stream is to rearrange array dimensions in order to maximize inner loop trip counts. The technique is also used on Cray PVP systems to increase performance through increased vector length. Because split loops are stripmined by 256, 256 is the largest inner loop count possible for loops that are split by the compiler.

Currently, this technique is not performed automatically by the compiler. You must do it manually.

The loop in the following example processes the arrays in streams of 32 elements:

Example 27: Rearranging array dimensions

```
double x[1000][32], y[1000][32], z[1000][32];
double s[1000][32], t[1000][32], w[1000][32], u[1000][32];

for(j=0; j<n1000; j++) {
  for(i=0; i<n32; i++) {
    x[j][i] = y[j][i] + z[j][i];
    w[j][i] = x[j][i] * t[j][i] - s[j][i] * u[j][i];
  }
}
```

By switching dimensions, as in the following code, the loop is processed in streams of 256 elements (the stripmine length), which reduces the aggregate stream startup count by 88%:

```
double x[32][1000], y[32][1000], z[32][1000];
double s[32][1000], t[32][1000], w[32][1000], u[32][1000];

for(j=0; j<32; j++) {
  for(i=0; i<1000; i++) {
    x[j][i] = y[j][i] + z[j][i];
    w[j][i] = x[j][i] * t[j][i] - s[j][i] * u[j][i];
  }
}
```

4.4.4 Minimizing Stream Count

Minimizing the number of different streams in a loop reduces the amount of loop splitting required.

One technique is to rearrange the dimensions of an array to make short dimensions (usually between 2 and 20 elements) the fastest running (or rightmost) dimension. This is the opposite of the technique described in Section 4.4.3, page 100, and it is only profitable if all the array's references are unwound along the short dimension and grouped within the loop so as to allocate only one stream.

Currently, this technique is not performed automatically by the compiler. You must do it manually.

Although the loop in the following example makes the best use of vectors on Cray PVP systems, it will allocate 12 streams on CRAY T3E systems:

Example 28: Minimizing streams

```
double x[6][1000], y[6][1000];
for(i=0; i<1000; i++) {
    x[0][i] = y[0][i];
    x[1][i] = y[1][i];
    x[2][i] = y[2][i];
    x[3][i] = y[3][i];
    x[4][i] = y[4][i];
    x[5][i] = y[5][i];
}
```

The loop in the following example allocates only two streams on CRAY T3E systems:

Example 29: Reduced streams version

```
double x[1000][6], y[1000][6];
for(i=0; i<1000; i++) {
    x[i][0] = y[i][0];
    x[i][1] = y[i][1];
    x[i][2] = y[i][2];
    x[i][3] = y[i][3];
    x[i][4] = y[i][4];
    x[i][5] = y[i][5];
}
```

4.4.5 Grouping Statements That Use the Same Streams

Although loop splitting is beneficial to program performance, it does introduce some overhead. The fewer times a loop is split, the less overhead you will have. As the compiler splits loops, it creates new loops by processing statements in the order in which they occur in the original loop. It is beneficial to group statements that use the same streams.

Currently, this technique is not performed automatically by the compiler. You must do it manually.

The compiler would split the loop in the following example into four loops:

Example 30: Original code

```
double x[1000][2], y[1000][2], z[1000][2];
double s[1000][2], t[1000][2], w[1000][2], u[1000][2];

for(i=0; i<1000; i++) {
    x[i][0] = y[i][0] + z[i][0];
    w[i][0] = x[i][0] * t[i][0] - s[i][0] * u[i][0];
    x[i][1] = y[i][1] + z[i][1];
    w[i][1] = x[i][1] * t[i][1] - s[i][1] * u[i][1];
}
```

If the statements using the same streams are grouped together, the loop only needs to be split into two loops, as shown in the following example:

Example 31: Grouping statements within the loop

```
for(i=0; i<1000; i++) {
    x[i][0] = y[i][0] + z[i][0];
    x[i][1] = y[i][1] + z[i][1];
    w[i][0] = x[i][0] * t[i][0] - s[i][0] * u[i][0];
    w[i][1] = x[i][1] * t[i][1] - s[i][1] * u[i][1];
}
```

The situation in the preceding example might be found in codes that contain loops that have been unrolled manually. These loops should either have their statements grouped, or the loops should be rerolled. The following loop, unrolled manually, will be split into four different loops:

Example 32: Loop that will be split into four

```
double x[1000], y[1000], z[1000];
double s[1000], t[1000], w[1000], u[1000];
```

```

for(i=0; i<999; i+=2) {
  x[i] = y[i] + z[i];
  w[i] = x[i] * t[i] - s[i] * u[i];
  x[i+1] = y[i+1] + z[i+1];
  w[i+1] = x[i+1] * t[i+1] - s[i+1] * u[i+1];
}

```

As in Example 30, page 102, and Example 31, page 102, the first and third lines of the loop access the same streams, as do the second and fourth lines. By grouping the first and third lines and second and fourth lines, the loop in the following example, which was unrolled manually, will be split into two loops. (Rerolling this loop would probably be best, but the intent of the example is to demonstrate grouping.)

Example 33: Loop that will be split into two

```

for(i=0; i<999; i+=2) {
  x[i] = y[i] + z[i];
  x[i+1] = y[i+1] + z[i+1];
  w[i] = x[i] * t[i] - s[i] * u[i];
  w[i+1] = x[i+1] * t[i+1] - s[i+1] * u[i+1];
}

```

4.4.6 Enabling and Disabling Stream Buffers

There are two levels for the stream buffers, and 1. Level 1 enables stream buffers and is usually the default.

At level 1, stream buffers are allocated when two secondary cache misses on consecutive local memory locations are detected. The allocation occurs only if the second miss occurs within eight memory locations of the first miss. Once a stream is activated, it reads memory in paging mode. This causes memory reads to be done in blocks of four cache lines, optimizing memory bandwidth. For an example of starting a stream, see Procedure 1, page 9.

At level 0, no stream buffers are allocated. Level 0 may be needed for certain rare loops with references that activate stream buffers but never access them, causing unused memory traffic.

Currently, the stream buffer level is not adjusted for an individual loop by the compiler. You must change the level manually before each loop if you want a setting other than the default of 1. The stream buffer level can be set at run

time by using the `set_d_stream(3)` library function. The stream buffer level is changed to 0 in the following example:

```
#include <mpp/rastream.h>
set_d_stream(0);
```

The `get_d_stream(3)` function saves the current stream buffer level for the purpose of restoring it later.

For certain versions of the CRAY T3E system, stream buffers are disabled by default for the following classes of programs:

- A program that calls subroutines from the SHMEM library.
- A program that uses the `cache_bypass` directive (see Section 4.7, page 110).

The document *CRAY T3E Programming with Coherent Memory Streams* outlines conditions under which you can safely enable streams for programs in these categories. The document is available online at the following URL:

<http://www.sgi.com/t3e/guidelines.html>

Once you ensure that your program is safe, you can enable streams by using the `set_d_stream(3)` library function or by setting the `SCACHE_D_STREAMS` environment variable to 1.

4.5 Optimizing Division Operations

The division operation is relatively expensive in terms of performance. Depending on how many bits need to be generated (that is, 1.0 divided by 2.0 is quicker than 1.0 divided by 3.0), the operation varies between 22 and 60 CPs for a 64-bit divide and between 15 and 31 CPs for a 32-bit divide. Division operations are not pipelined, so a second divide cannot be issued while the first is in progress.

The best strategy for division is to avoid it whenever possible. Changing a division operation into a reciprocal multiplication operation, which the C and C++ compilers do by default, can improve performance dramatically. In the following example, `a`, `b`, `c`, and `d` are all cache-resident arrays:

```
#pragma_CRI unroll
for(i=0; i<256; i++)
    a[i] = (b[i] + 2.0 * c[i] + d[i]) / x;
```

Because the divide is loop invariant (glossary, page 146), the divide can be changed to a multiply by the reciprocal, as shown in the following example.

```
xinv = 1.0/x;
for(i=0; i<256; i++)
    a[i] = (b[i] + 2.0 * c[i] + d[i]) * xinv;
```

By default, the C and C++ compilers change a divide into a reciprocal multiply for you. You do not have to change the code at all. Unless you disable it by specifying the `-h nofastfpdivide` option on the `cc` or `CC` command line, the compiler will use the faster reciprocal multiply at every opportunity.

Other operations can proceed when a divide operation is in progress. If moving a divide operation outside of a loop is not possible, you can sometimes preschedule it from within your source code. The following inner loop has a divide operation in line 7 that causes a wait of about 60 CPs, and the result is immediately used.

Example 34: Original code

```
1.  for(jn=0; jn<lprr; jn++) {
2.      j = iar2[jn + lpair];
3.      ic = ico[iaci_iac[j]];
4.      xw1 = tmp1 - x[1][j];
5.      xw2 = tmp2 - x[2][j];
6.      rwtmp = tmp3 - x[3][j];
7.      r2inv = 1.0/((xw1 * 100) + (xw2 * 100) + (rwtmp * 100));
8.      /* Line 7 is the problem. The result of
9.      the divide is used in the next calculation. */
10.     df2 = cgi * cg[j] * r2inv;
11.     eelt = eelt + df2;
12.     r6 = r2inv * 1000;
13.     f1 = cn12[1][ic] * (r6 * r6);
14.     f2 = cn12[2][ic] * r6;
15.     enbt = enbt + (f2 - f1);
16.     df = (df2 + 6.0 * ((f2 - f1) - f1)) * r2inv;
17.     fw1 = xw1 * df;
18.     fw2 = xw2 * df;
19.     fw3 = rwtmp * df;
20.     f[1][j] += fw1;
21.     f[2][j] += fw2;
22.     f[3][j] += fw3;
23.
24.     tmp4 -= fw1;
```

```
25.     tmp5 -= fw2;
26.     tmp6 -= fw3;
27. }
```

Using a technique similar to *bottom loading* (glossary, page 142), the division required for the next iteration of the loop is computed in advance. The divide operation itself is in line 14 of the following example. The result of the divide is not needed until the next pass of the loop, so the floating-point operations following the divide can overlap with the 60 CPs, assuming a 64-bit divide. This kind of division is unconventional, but it increases the performance of the code. The compiler does not make the following changes automatically.

Example 35: Modified code

```
1.  /* First divide computed */
2.  j = iar2[1 + lpair];
3.  xw1 = tmp1 - x[1][j];
4.  xw2 = tmp2 - x[2][j];
5.  rwtmp = tmp3 - x[3][j];
6.  r2inv = 1.0/((xw1 * 100) + (xw2 * 100) + (rwtmp * 100));
7.
8.  for(jn=0; jn<lpr; jn++) {
9.      /* Compute the divide for next pass */
10.     j_next = iar2[(jn + 1) + lpair];
11.     xw1_next = tmp1 - x[1][j_next];
12.     xw2_next = tmp2 - x[2][j_next];
13.     rwtmp_next = tmp3 - x[3][j_next];
14.     rnxt=1.0/((xw1_next*100)+(xw2_next*100)+(rwtmp_next*100));
15.     ic = ico[iaci + iac[j]];
16.     df2 = cgi * cg[j] * r2inv;
17.     eelt = eelt + df2;
18.     r6 = r2inv * 1000;
19.     f1 = cn12[1][ic] * (r6 * r6);
20.     f2 = cn12[2][ic] * r6;
21.     enbt = enbt + (f2 - f1);
22.     df = (df2 + 6.0 * ((f2 - f1) - f1)) * r2inv;
23.     fw1 = xw1 * df;
24.     fw2 = xw2 * df;
25.     fw3 = rwtmp * df;
26.     f[1][j] += fw1;
27.     f[2][j] += fw2;
28.     f[3][j] += fw3;
29. }
```

```
30.     tmp4 -= fw1;
31.     tmp5 -= fw2;
32.     tmp6 -= fw3;
33.     /* Juggle the values for the next pass */
34.     j = j_next;
35.     xw1 = xw1_next;
36.     xw2 = xw2_next;
37.     rwtmp = rwtmp_next;
38.
39.     /*The result of the divide is not
40.        needed until here. All the work
41.        above this can proceed concurrently
42.        with the divide. */
43.     r2inv = rnxt;
44. }
```

The early divide for the last iteration in the preceding example is potentially unsafe because it may go out of bounds. You may have to provide a special case for the last iteration.

4.6 Vectorization

The CRAY T3E compiler offers a method to *vectorize* select math operations inside loops. This is not the same kind of vectorization available on a Cray PVP systems. On a CRAY T3E system, the compiler restructures loops containing scalar operations and generates calls to specially coded vector versions of the underlying math routines. The vector versions are between two and four times faster than the scalar versions. The compiler uses the following process:

1. Stripmine the loop. (For more information on stripmining, see Example 24, page 97.)
2. Split vectorizable operations into separate loops, if necessary. (For more information on loop splitting, see Example 24, page 97.)
3. Replace loops containing vectorizable operations with calls to vectorized intrinsics.

Vectorizing reduces execution time in the following ways:

- By reducing aggregate call overhead, including the subroutine linkage and the latency to bring scalar values into registers needed by the intrinsic routine.

- By improving functional unit utilization. It provides better instruction scheduling by processing a vector of operands rather than a single operand.
- By producing loops that can be pipelined by the software. (For more information on pipelining, see Section 4.2, page 86.)

The programming environment also offers the `libmfastv` library of faster, but less accurate, vector versions of the `libc` math functions. These routines deliver results that are generally 1 to 2 bits less accurate than the results given by the `libc` math functions. Less accurate scalar versions of the library functions are also used to provide identical results between vector and non-vector invocations within the same program. The `libmfastv` routines reduce execution time spent in math intrinsics by between 50% and 70%.

Because the vector functions may not provide a performance improvement in all cases (due to necessary loop splitting), vectorization is not on by default. It is enabled through the compiler command-line option `-h vector3`. The default is `vector2`, which currently does no intrinsic vectorization. The `vector1` and `vector0` options have their own meanings on Cray PVP systems, but on the CRAY T3E system, they are the same as `vector2`: they turn vectorization off.

If you have selected `-h vector3`, you can further control vectorization using the following:

- The vectorization directive `novector`. It lets you turn vectorization off for selected loops in your program.
- In the case of ambiguous data dependences within the loop, you can express a loop's vectorization potential with the `ivdep` pragma directive. `ivdep` tells the compiler to ignore vector dependencies in the following loop. For more information, see Section 4.6.1, page 110.
- Access to the `libmfastv` routines can be controlled with the `-l` compiler option. For example, the following command line links in the faster, less accurate math routines rather than the slower, default routines in `libc`:

```
% cc -hvector3 -lmfastv test.c
```

Transformation from scalar to vector is implemented by splitting loops. This may cause extra memory traffic due to the expansion of scalars into arrays and reduce the opportunity for other scalar optimizations. This could negatively impact the profitability of the vectorization.

The less accurate version offered through `libmfastv` varies from default `libc` results generally within 1 to 2 bits, although some results could differ by larger amounts. Exceptions may also differ from the `libc` versions, where some calls

to `libmfastv` may generate only a NaN for a particular operand rather than an exception, causing exceptions further down the line. A NaN is a value that is not a number but rather a symbolic entity encoded in floating-point format.

Vectorization is only performed on loops that the compiler judges to be vectorizable. This determination is based on perceived data dependencies and the regularity of the loop control. These loops will likely be a significant subset of those seen as vectorizable by the Cray PVP compiler. Vectorization of conditionally executed operators is deferred. Vectorization of loops that contain potentially early exits from the loop is also deferred.

Vectorization will be performed on the following intrinsics and operators. The first set supports both 32-bit and 64-bit floating-point data:

`sqrt(3C)`

`1/sqrt` (replaced by a call to `sqrtinv(3C)`)

`log(3C)`

`exp(3C)`

`sin(3C)`

`cos(3C)`

`cos(3C)` (replaced by a combined call to `sin` and `cos`)

The following support 64-bit floating point only:

`pow(3C)`

`rand(3c)`

The vector intrinsic routines are designed to read an arbitrary number of operands from memory and write their results to memory. They can also handle operands and results that do not have a stride of one.

The compiler stripmines and splits (if necessary) any loop for which intrinsic vectorization is indicated by the programmer. The stripmine factor is currently 256. The loop is stripmined to limit the size of scalar expansion arrays and to decrease the likelihood of cache conflict.

The following example illustrates the kind of loop to which the vectorization optimization can be applied:

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

4.6.1 Using the `ivdep` Directive

Use the `ivdep` directive to tell the compiler that the following loop can be vectorized, despite the presence of what might appear to be vector dependencies.

Placing the `ivdep` directive in front of a loop also enables the compiler to perform the following optimizations:

- Loop splitting.
- Using the vectorizing intrinsic functions listed in Section 4.6, page 107.
- Using the `cache_bypass` directive (see Section 4.7, page 110).
- Improving pipelining (see Section 4.2, page 86).

Vector dependencies prevent the compiler from analyzing a loop because of the unknown value of variables at the time the program is executed. In the following example, the first statement in the loop may read a value that was written by the second statement in an earlier iteration:

```
for(i=0; i<=n; i++) {  
    x = a[i-k];  
    a[i] = b[i] * c[i] - w;  
}
```

In this example, there may be a dependence, depending on the value of `k`. If `k` is positive, there is a dependence, because the first statement is trying to access a value written by a later statement. If `k` is negative, there is no dependence. The compiler cannot know the value of `k` ahead of time. It will not vectorize the loop unless you place an `ivdep` directive immediately before it.

4.7 Bypassing Cache

The `cache_bypass` directive offers some applications a semi-automatic way to speed up memory references. The execution time for some loops can be reduced by up to 45%.

Local memory references prefaced by the `cache_bypass` directive are routed through E registers rather than through cache. Because E registers offer a finer granularity of access to local memory (see Section 6.1, page 131), they give you higher bandwidth for sparse index array accesses, such as gather/scatter and large stride accesses that do not take advantage of the spatial locality of cache references.

The `cache_bypass` directive can also be used to initialize large arrays if the contents are not immediately needed in cache, avoiding unnecessary reads into cache and improving the memory bandwidth.

The directive precedes a `for`, `while`, `do while`, or `if ... goto` loop and affects all of the named arrays whose base data types are 64 bits. (Support for 32-bit and complex data types is not yet implemented.) In the following loop, arrays `a` and `b` will be accessed through E registers rather than through cache:

```
#pragma _CRI cache_bypass a, b
for(i=0; i<n; i++)
    x[i] = a[i] * b[i];
```

Note: The `cache_bypass` directive makes no guarantees about the state of cache before or after the specified loop. The directive is strictly a performance hint to the compiler. In particular:

- It does not guarantee that the specified variables are not in cache before the loop.
- It does not guarantee that the specified variables are not in cache after the loop.
- It does not invalidate cache.
- It does not affect program results in any way.

Even if you include a `cache_bypass` directive before a loop, the compiler ignores it if it determines it cannot generate code efficiently. The loop must meet the following requirements before the compiler uses E registers:

- The loop must be an inner loop, if it is nested in another loop.
- The loop must be vectorizable. Use the `ivdep` directive in cases where ambiguous data dependencies are preventing the loop from vectorizing. (For more information on the `ivdep` directive, see Section 4.6.1, page 110. For more information on vectorization, see Section 4.6, page 107.)

You will probably have to enable loop unrolling to realize the full benefit from this feature. For information on the unrolling command-line option and directive, see Section 4.1, page 85.

The benefit is greater the more random the index stream, however, benefit has been seen from index streams with secondary cache hit rates as high as 50%.

Bypassing cache does generate more code for candidate loops, potentially increasing the compile time slightly. It also increases the latency of memory

references in return for greater bandwidth. Applying the `cache_bypass` directive may increase the execution time for loops that would otherwise benefit from cache references.



Caution: For some CRAY T3E installations, this feature causes the stream buffer hardware feature to be disabled by default for the entire application. Streaming is disabled if the compiler cannot guarantee correctness in the interaction of the stream buffers and the E register operations generated by this feature. Disabling stream buffers can cause considerable performance degradation for other sections of the program. The stream buffer feature can be reenabled using the `set_d_stream(3)` library routine. Consult your system administrator to determine if your CRAY T3E installation falls into this category.

For background information on streaming, see Figure 4, page 8, and the example that follows. See the document *CRAY T3E Programming with Coherent Memory Streams* for details on how and when streams can be safely reenabled in the presence of E register operations. For the online address of the document, plus information on enabling and disabling stream buffers, see Section 4.4.6, page 103.

Optimizing I/O on the CRAY T3E system is not, for the most part, very different from optimizing I/O on Cray PVP systems. If you are already acquainted with Cray PVP I/O, much of this chapter should be familiar to you.

As on other Cray Research systems, there are a few optimizations that apply regardless of the kind of I/O your program is performing. For instance, using binary (or unformatted) data rather than ASCII data is a good idea that should be used whenever possible. To reduce redundancy, it is not listed as an optimization in every section of this chapter.

The following optimization topics are covered:

- Choosing a strategy for doing I/O in a parallel programming environment (see Section 5.1, page 113).
- Using unformatted I/O whenever possible (see Section 5.2, page 119).
- Coping with formatted I/O when necessary (see Section 5.3, page 122).
- Making use of the performance-enhancing FFIO layers in your program (see Section 5.4, page 123).
- Optimizing random access I/O (see Section 5.5, page 128).
- Striping a file over disk partitions (see Section 5.6, page 128).

5.1 Strategies for I/O

One of the first questions to answer when optimizing on a parallel system is what your I/O strategy will be. Should you do all of your I/O from a single PE? Should each PE perform its own I/O? Should you work with a single data file or multiple files?

If you want to write data from multiple PEs to one file, you must choose one of the following methods:

- Have all of the PEs open one or more shared files. A shared file can be on a single disk or *striped* (glossary, page 144) over many disks. This method requires you to synchronize carefully when you are writing to the file (see Section 5.1.1, page 114).

- Have each PE involved in I/O, perhaps all of them, open a separate file. This method often requires you to divide data up into multiple files before reading and to merge files after writing. You can do the dividing and merging outside the scope of the program. Each file can be read from and written to different disks (see Section 5.1.2, page 116).
- Have one PE do all of the I/O. The PE performing the read shares the data with the rest of the PEs and collects it again before writing the output. This method can be very fast when you make use of disk striping (see Section 5.1.3, page 118).

The following sections describe the performance benefits and detriments of these three methods.

5.1.1 Using a Single, Shared File

PEs can read the same file at the same time. They can also, if you are careful, write to the same file at the same time. For an illustration of this process, see the following figure.

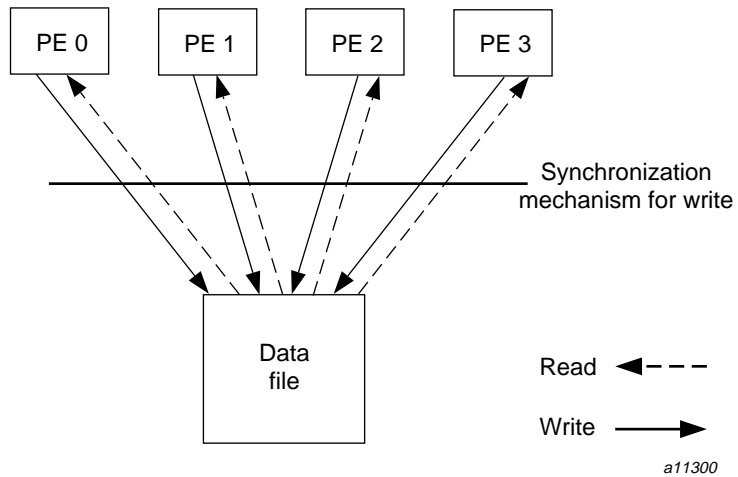


Figure 22. Multiple PEs using a single file

Having many PEs reading from the same file can cause a slowdown due to I/O contention. Reading or writing to a single file is most effective under the following circumstances:

- Your program is doing distributed I/O, which automatically spreads an input file across the memory of multiple PEs. This is a supported and optimized method of doing shared file I/O, and it is the recommended approach. For more information on this method, see Section 5.4.2, page 125.
- Your program is doing random access I/O.
- The number of PEs involved is not greater than the number of I/O channels on your CRAY T3E system. This need not be a restriction unless your program is I/O bound. If your program is not I/O bound, you can overlap computation with I/O. Choosing an asynchronous I/O mechanism, such as either the `bufa` or `cachea` FFIO layer, lets you continue to execute statements in your program while I/O is taking place. For information on asynchronous I/O, see Section 5.4.3, page 127, and Section 5.4.4, page 128.
- Your file is striped over multiple disk drives. For information on disk striping, see Section 5.6, page 128.

If each PE reads its own part of the input file, use an offset into the file based on each PE's number. If all PEs need the same data, reading from one PE and broadcasting the data to the others might be faster than having each PE read from the same file, especially if a large number of PEs are involved in the job.

The following example demonstrates how multiple PEs can share the job of reading and writing a single random-access file. In the example, the function `findnext` returns a record number based on the PE that invokes it and the number of PEs participating in the I/O. This example will run fast whether the shared file is on a single disk or striped over many disks.

```
#include <stdio.h>
#include <stdlib.h>
#include <ffio.h>
#include <fcntl.h>

main()
{
    int me, i, next, workers, size=100000;
    int num_elements, fd, j, ret, fret;
    int array[size];

    me = _my_pe();
    workers = _num_pes;
    size = 100000;
    num_elements = size/workers;
```

```
fd = fopen("datafile", O_RDWR);
for(i=0; i<num_elements; i++) {
    next=findnext(next, me, num_elements);
    fret = fseek(fd, next, 0);
    if (fret < 0) exit();

    ret = fread(fd, array[i], 8);
    if (ret < 0) abort();

/* Perform calculations */

    ret = fwrite(fd, array[i], 8);
}
}
```

When executing this program, pipe the input and output data files as follows:

```
mpprun -n 10 a.out <infile >outfile
```

5.1.2 Using Multiple Files and Multiple PEs

Reading from and writing to multiple files may be the easiest and fastest I/O strategy, given a number of PEs less than or equal to the number of data streams on your CRAY T3E system. The I/O library routines, the I/O hardware paths between local memory and disk, and the disk devices themselves can all operate in parallel.

Use conditional `fopen(3C)` functions, as in the following example, to read from and write to four files in a 4-PE program. You may be able to optimize this example further by performing I/O to different file systems located on different GigaRings. See your system administrator for information on how file systems are partitioned across GigaRings.

```
#include <ffio.h>
#include <fcntl.h>
#include <cs_os_def.h>

main()
{
    int fd, errno;
    char inbuf[8*4096], outbuf[8*4096];

    if(_my_pe() == 0)
```

```

    fd=ffopen("file0",O_RDWR|O_RAW);
if(_my_pe() == 1)
    fd=ffopen("file1",O_RDWR|O_RAW);
if(_my_pe() == 2)
    fd=ffopen("file2",O_RDWR|O_RAW);
if(_my_pe() == 3)
    fd=ffopen("file3",O_RDWR|O_RAW);

ffread(fd, inbuf, 8*4096, errno, FULL, 0);

/* Work on data */

ffwrite(fd, outbuf, 8*4096);
}

```

In this example, each PE reads from and writes to its own file using the FFIO routines `ffread(3C)` and `ffwrite(3C)`. For more information on FFIO, see Section 5.4, page 123. The external file for each PE is identified by its own copy of `fd`. The number `8*4096` is assumed to be eight disk sectors.

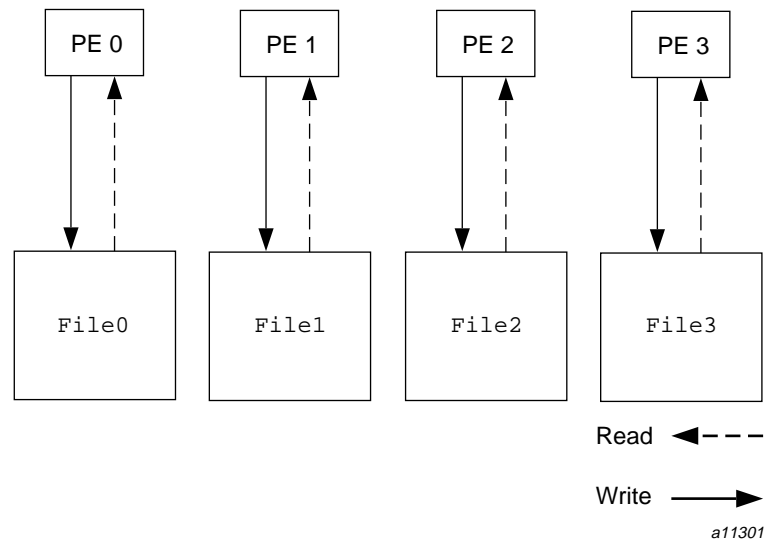


Figure 23. Multiple PEs and multiple files

5.1.3 Using a Single PE

When a single PE performs the I/O, you will usually have to share the data with other PEs. For an illustration, see the following figure.

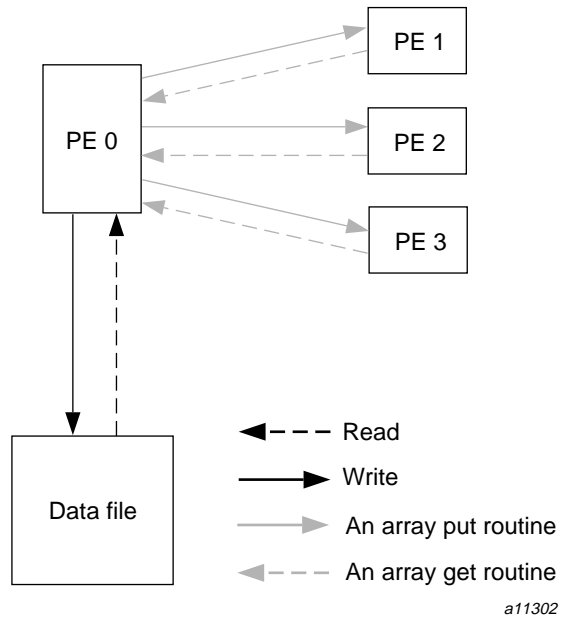


Figure 24. I/O to and from a single PE

You can share the data using one of the following methods:

- You can use one of the Message Passing Toolkit (MPT) products to pass the data on. PVM, MPI, and SHMEM all have broadcast routines that pass data to other PEs. If every PE needs all the data, the `shmem_broadcast` routine is the fastest of the three, but it is not portable to other vendors' systems. For examples of `shmem_broadcast`, see Section 3.6, page 71. For information on `pvm_bcast(3)`, see Section 2.9, page 33.
- If each PE only needs part of the data, use get and put functions. Again, the SHMEM put and get routines are the fastest. For information on using `shmem_put64` and `shmem_get64`, see Section 3.1, page 46. If portability is a concern, PVM and MPI have put and get functions, but they are slower than SHMEM.

5.2 Unformatted I/O

Because formatted I/O requires data conversion, it will add overhead to your program. Avoid formatted I/O whenever practical.

For example, if you are moving data between machines, you can send the binary (unformatted) version instead of ASCII. You can best convert the data as follows:

- Use the `assign(1)` command before running your program to convert to other formats automatically. The following command converts to CRAY T3E format:

```
% assign -N t3e f:myfile1
```

- Convert the `scanf(3)` and `printf(3)` function calls in C and C++, and the `cout`, `cin`, `<<`, and `>>` C++ functions to unformatted functions. Use the `ffread(3)` and `ffwrite(3)` functions, along with the `assign` command before executing your program, as follows:

```
% cc -X4 myprog.c
% assign -s bin f:outfile
% ./a.out
```

By default, the system always accesses the next record automatically during a read or a write. That means sequential I/O will be fast. When you are also manipulating unformatted data, you have the potential for very fast data transfers. The following section describes how to get the most out of a good combination.

5.2.1 Sequential, Unformatted Requests

This section describes how best to optimize I/O when you are reading or writing sequential, unformatted data.

A permanent file exists on an external device, such as a disk. Instead of reading from the disk every time, you can save time by moving your file into memory and reading from there. The file in memory is temporary, since you will probably write it out at some point in the program.

Reading from or writing to a disk involves accessing system calls, which move the data between disk and system buffers. Once data has been read into the system buffer, the I/O library moves it from there to its own set of buffers or into cache, both of which are in the memory of another PE on a CRAY T3E system.

If the whole file does not fit in memory, you can read in parts of it at a time, possibly getting work done on the current data while waiting for the next chunk to be read. The following figure illustrates the data flow for an array named *A* in PE 4.

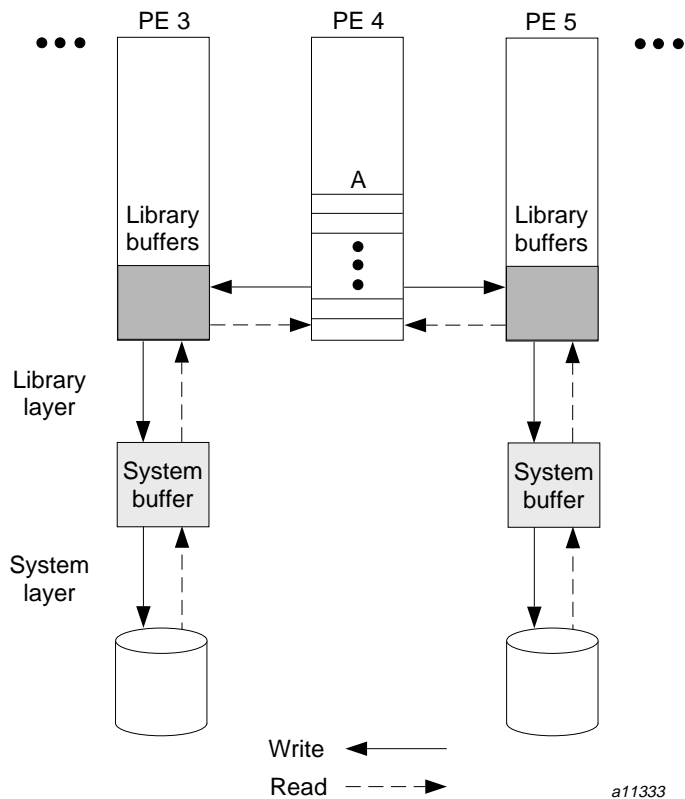


Figure 25. Data paths between disk and an array

To move the data between disk and the system buffers, use the following optimizations:

- Choose system call I/O. System call I/O is specified on an `assign` command either explicitly, by selecting the `system` or `syscall` FFIO layer, or implicitly; if it is not specified, it is added automatically. (For more information on FFIO, see Section 5.4, page 123.) The following example selects `syscall`:

```
% assign -F syscall
```

- Make I/O requests that begin and end on disk sector boundaries. Most disk sectors are the same as a block size, 512 words (or 4,096 bytes). Check with your system administrator to make sure of the size of a disk sector on your CRAY T3E system.

The following command preallocates an area that is 80 512-word blocks in size:

```
% assign -n 80
```

- If you are using the `read(2)` and `write(2)` system calls directly, switch to `ffread(3C)` and `ffwrite(3C)`. Doing so will reduce the number of calls to the system, and performance should be at least as good as using the system calls.

Optimizations such as double buffering (or even triple and quadruple buffering) and disk striping are performed by the operating system. User striping can still gain you performance improvements, but it is more labor intensive than other optimizations described in this chapter. (For an example of user striping, see Section 5.6, page 128.)

By adjusting the arguments to the `cachea` and `bufa` layers of FFIO, you can have double buffering done automatically. The following `assign` command creates two buffers, each 50 blocks in size:

```
& assign -F bufa:50:2
```

To optimize the process of moving data between the system buffers and an array in your program, use the following techniques:

- Access the FFIO libraries using the `ffread(3)` and `ffwrite(3)` functions.
- Take advantage of asynchronous I/O if you can accomplish other work while the I/O is taking place. If sequential, unformatted I/O requests take most of your program's time, you can probably improve performance by combining computation with the inherent asynchronous capability of I/O. First, select an asynchronous FFIO layer by running `assign` commands such as the following before executing your program:

```
% assign -F cachea:80:7 f:indata
% assign -F bufa:100:12 f:indata
```

The preceding examples take advantage of library caching and buffering, respectively. The `bufa` and `cachea` layers have *read-ahead* (glossary, page

148) and *write-behind* (glossary, page 152) capabilities that can improve performance significantly.

- Use the `setvbuf(3)` function to specify the size of the library buffer. The following sets the buffer size to 48 512-word blocks, which is the default:

```
setvbuf (FILE stream-file, *NULL, _IOFBF, 48*512)
```

- If the file is small enough to fit entirely into the memory of a PE, or if a certain part of the file is heavily accessed, use the memory-resident layer in FFIO. The memory-resident layer involves less overhead than, for example, the *cachea* layer. For more information, see Section 5.4.1, page 123.

5.3 Formatted I/O

If using formatted I/O cannot be avoided, you can use some of the optimization techniques described in the following sections to speed it up.

5.3.1 Reduce Formatted I/O

You can create incremental speedups to your program by reducing the amount of the formatted I/O as follows:

- If you are not going to use all of the output, print only a sample.
- Do not format intermediate results.

5.3.2 Make Large I/O Requests

Rather than reading an array an element at a time, read multiple data items in a single statement. This will reduce overhead and speed up your program.

For example, the following codes (the first reads in C and the second writes in C++) make many small I/O requests by reading an array within a loop, element-by-element:

```
float a[isize];
for(i=0, i<isize; i++)
    scanf("%f", a[i]);
```

```
float a[isize];
for(i=0; i<isize;) {
    myfile << a[i] << ' ';
```

```
    ++i;
    if(i%5 == 0) myfile << '\n';
}
```

The following example reads five elements in a single I/O request. It makes 80% fewer I/O calls and helps the program to execute faster:

```
float a[i];
for(i=0; i<isize; i+=5)
    scanf("%f %f %f %f %f", a[i], a[i+1], a[i+2], a[i+3], a[i+4]);
```

5.4 FFIO

The `bufa` and `cachea` layers of flexible file I/O (FFIO) do asynchronous buffering and caching internally. Those two, along with `global`, which distributes a data file across multiple PEs, and `mr`, which stores part or all of a data file in the memory of a single PE, are the high performance FFIO layers. The following sections describe how to improve the performance of your program by using them.

5.4.1 Memory-resident Data Files

You can keep part or all of a data file in the local memory of the PE performing the I/O by using the memory-resident layer of FFIO. This technique can speed up data access dramatically for a small, frequently accessed file or for a large file with most of the I/O activity occurring at the beginning of the file.

Memory is a more precious commodity on a CRAY T3E PE than on a Cray PVP system. Creating a memory-resident file that is too large could take up memory space necessary to your application. To determine how much space you have for data, you first must know how much memory is available on each PE. A CRAY T3E system comes with between 64 Mbytes and 2 Gbytes of local memory per PE. If you do not know how much your system has, you can find out by entering the `grmview -l` command. The following example shows the first part of the output, with several columns that are irrelevant to this subject removed:

PE Map: 20 PEs configured

Type	PE	Ap. Size		Number		x	y	z	Clock	UsrMem	FreMem
		min	max	running	Aps. limit						
+ APP	0	2	10	0	1	0	0	0	300	119	119
+ APP	1	2	10	0	1	1	0	0	300	119	119
+ APP	2	2	10	1	1	0	1	0	300	119	116
+ APP	3	2	10	1	1	1	1	0	300	119	116
+ APP	4	2	10	1	1	0	2	0	300	119	116
+ APP	5	2	10	1	1	1	2	0	300	119	116
+ APP	6	2	10	1	1	0	3	0	300	119	117
+ APP	7	2	10	1	1	1	3	0	300	119	116
+ APP	8	2	10	0	1	1	0	1	300	119	119
+ APP	9	2	10	0	1	0	1	1	300	119	119
+ CMD	10	1	1	4	unlim	1	1	1	300	114	52
+ CMD	11	1	1	5	unlim	0	2	1	300	111	78
+ CMD	12	1	1	2	unlim	1	2	1	300	113	57
+ CMD	13	1	1	2	unlim	0	3	1	300	115	57
+ CMD	14	1	1	2	unlim	1	3	1	300	113	52
+ CMD	15	1	1	3	unlim	0	0	2	300	115	55
+ CMD	16	1	1	2	unlim	1	0	2	300	106	50
+ CMD	17	1	1	1	unlim	0	1	2	300	107	45
+ OS	18	0	0	0	0	1	1	2	300	95	67
+ OS	19	0	0	0	0	0	0	1	300	75	0

The application PEs (APP under the Type column) are the ones to look at. The UsrMem column shows 119 Mbytes available to a user program, meaning each PE probably has 128 Mbytes of local memory. If the combined size of your data and your executable file do not approach 119 Mbytes, you may be able to move the entire data file into the memory of a single PE. To enable the memory-resident layer of FFIO, enter an assign command such as the following before executing your program. This example allocates 10 512-word blocks (about .4 Mbytes) of memory for the data coming from the file myfile.

```
% assign -F mr:10 f:myfile
```

The data is automatically read into the memory-resident area when the file myfile is opened and written back out when myfile is closed. If the data area proves to be too small, the data file is split automatically between local memory and disk.

5.4.2 Distributed I/O

The distributed FFIO layer distributes input data across the PEs involved in a job. One PE makes an I/O request that reads the data and automatically divides it among the local memories of the PEs involved in the job.

When executing your program, a read retrieves the data from the right PE, using the `shmem_get(3)` function. You do not need to know where the data is stored or how it is retrieved. The performance is not as fast as using memory-resident (`mr`) data, but it is approximately equivalent to using `bufa`. A sample `assign` command looks as follows:

```
% assign -F global:5:1 f:filea
```

This example allocates 50 blocks for each page, with each block capable of containing 512 64-bit words of data. For a 10-PE program, it allocates 1 page for each PE, meaning there are 50 blocks on each of the 10 PEs, for a total of 500 blocks. The distribution for an array of 500 blocks would place the first 50 on the first PE to access a file page, the second 50 on the next PE to access a file page, and so on. The following figure represents the layout of the words of data on each PE. It does not reflect the random order in which the PEs access the file.

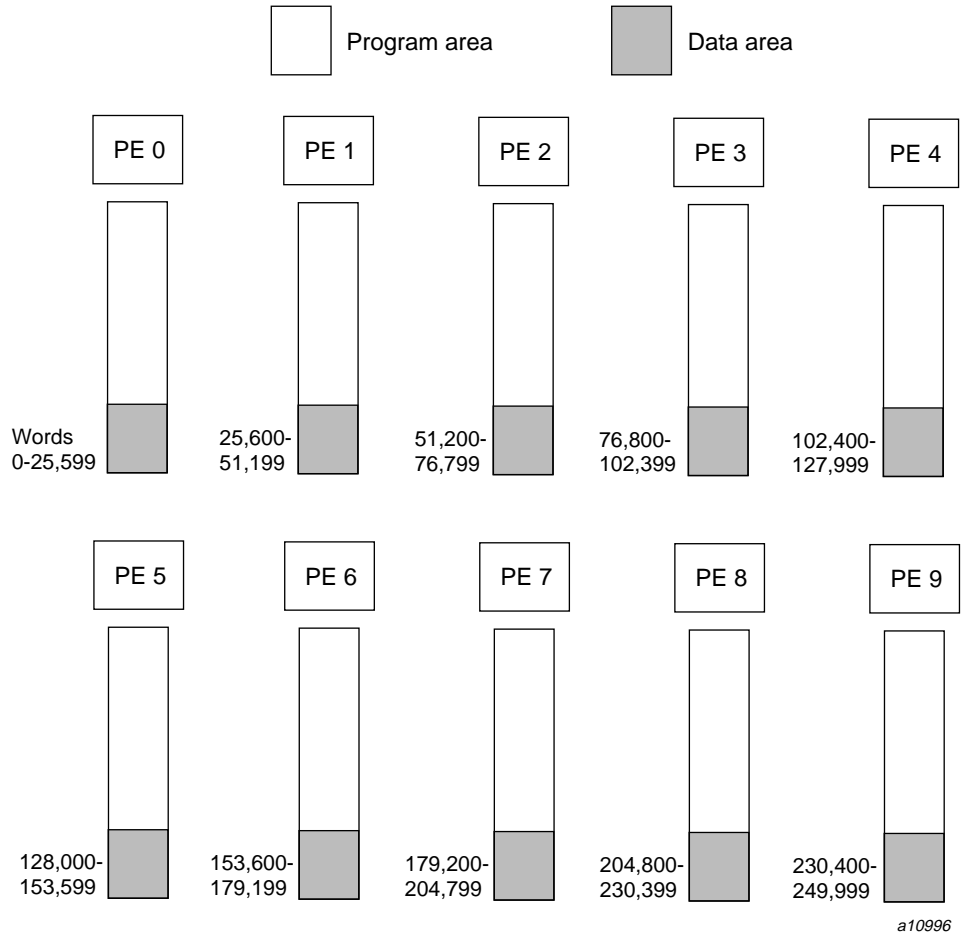


Figure 26. Data layout for distributed I/O

If you use distributed I/O during an operation in which all of the PEs were involved, each PE would hold data for every other PE. Although this might seem like a confusing arrangement, it is a good use of memory for most applications.

The advantages of using distributed I/O are as follows:

- You get more buffer space without severely impacting the memory of any single PE.

- The data becomes essentially a globally accessible file. You do not have to know on which PE any particular data element is stored.

The following are disadvantages:

- You are using memory that might be needed by a PE.
- PEs might need a large amount of data residing in the memory of other PEs, creating many remote transfers.

5.4.3 Using the Cache Layer

Using the cache layer is especially productive on random access files when many requests are made for data that has already been read into cache, though it is also effective on sequential data.

Be sure to choose the asynchronous FFIO cache layer (*cachea*) for your I/O. The following command, executed before you run your program, will set up a cache in the local memory of all PEs involved in the job:

```
% assign -F cachea:100:40:2 f:indata
```

This example sets up a cache of 40 pages, each page of which is 100 blocks of 512 64-bit words (51,200 words). If the I/O libraries detect sequential access, they perform either asynchronous read-ahead or asynchronous write-behind. The third parameter to *cachea*, which is 2 in the example, tells the libraries how many pages you want to be read ahead. Setting the third parameter is important to the performance for a program using sequential I/O, since the default is no read ahead.

Using cache is similar to using library buffers (see the following section); both have an asynchronous capability and both are stored in the memory of PEs. There are differences between the two, however.

Cache contains an indexing system for the data in an active cache. You can choose any indexed data in cache and quickly move it into a register.

A buffer does not have an indexing scheme; it knows only the file position at the top of the buffer. A buffer is designed for sequential access. If you reposition within a buffer, the current buffer is flushed and a new set of data is read from disk.

5.4.4 Using Library Buffers

When you are dealing with sequential access data, create large buffers. Set the buffer size to the size of the entire data file if possible, or to a fraction of the data size that keeps transfers to a minimum. Use the `assign(1)` statement as follows:

```
% assign -F bufa:40:1 f:file1
```

This allocates one buffer of 40 512-word blocks, or about 164 Kbytes. The buffers are allocated during program execution. Each PE opening a file receives a buffer.

5.5 Random Access

Random access (also called direct access) does not read or write files in sequential order. Because there is no pattern, the I/O libraries cannot optimize random I/O in the same way as sequential I/O. But there are some things you can do to speed up random access:

- Use binary files that bypass the system cache by entering the `assign -l none` command before executing your program. Because random access usually involves small reads and writes, going through system cache creates extra overhead. Also, name the array in a `cache_align` directive to align the array in secondary cache and further speed up processing. If you do have large reads or writes, system cache may benefit your program.
- Do not use formatted or blocked file formats. Converting or unblocking data takes extra time.
- If your data file fits in local memory, consider using the memory-resident layer of FFIO. For more information, see Section 5.4.1, page 123.
- If it is practical, rearrange your data so that you can process it sequentially.

5.6 Striping

Striping a file over disk partitions adds a level of parallel processing to the slowest part of I/O: the physical reading data from and writing data to disk. You can specify automatic striping by entering an `assign(1)` statement such as the following before executing your program:

```
% assign -p 0-3 -n 8400 -q 21 -s u f:mydata
```

This command stripes over four partitions (0, 1, 2, and 3), putting 21 sectors on each partition.

Hardware Access [6]

To provide a convenient and efficient way to run distributed programs, the UNICOS/mk operating system manages special groups of processes called *application teams* (glossary, page 141). An application team is a group of one or more processes, running on one or more PEs, that have capabilities that are extensions to standard UNIX. Application teams provide support for *multithreaded* programs (glossary, page 147) on CRAY T3E systems.

The special capabilities of an application team include the following hardware-level operations:

- Communication using E registers, which are a special set of hardware registers that provide the ability to directly read and write to the memory of another process within an application team. See Section 6.1.
- Hardware barrier and eureka synchronization units, which permit very fast barrier and *eureka* (glossary, page 144) capabilities within an application team. See Section 6.2, page 137.
- Distributed I/O. One process may make an I/O request to or from memory associated with another process in the application team. See Section 6.3, page 139.
- Global stop, exit, and abort processing. One process can terminate the execution of all processes in an application team. See Section 6.4, page 139.

On CRAY T3E systems, application teams provide a user environment based on UNIX process semantics. On CRAY T3D systems, PEs within an application team had some characteristics of a process (separate address spaces) and some characteristics of threads (every PE in an application team has the same process identification number (PID)). On CRAY T3E systems, threads packages, like Pthreads and DCE threads, are implemented within a process on a single PE. The CRAY T3E system does not lend itself to running multiple threads for the same process across multiple PEs.

6.1 Using E Registers

E registers provide the following special capabilities, in addition to the buffer space that holds data being transferred to or from a remote processor:

- Memory reads or writes to any process within the application team.

- Atomic memory operations, including masked atomic swap, compare-and-swap, fetch-and-increment, and fetch-and-add. These operations can be executed on any data address in any process within the application team.
- Population count and leading zero count operations.

The easiest and most portable method of accessing E registers on CRAY T3E systems is by using SHMEM routines from `libshma`. SHMEM routines offer a consistent interface on both Cray MPP systems and Cray PVP systems. (For more information on SHMEM, see Chapter 3, page 45.)

If you want to customize your access to E registers, you may want to perform low-level, direct programming.



Caution: The use of E registers documented in this section represents the conventions followed by CRAY T3E system software at initial release. The compiler, library, and header file interfaces described in this section are not formally supported and are subject to change at major software release levels. If you are manipulating low-level CRAY T3E hardware features, watch new software release notices for any changes you need to make to affected user code.

For more information on the interface to E registers, see one of the following hardware publications:

- *System Architecture Kit (CRAY T3E System)*
- *Commands Quick Reference - CRAY T3E Series Systems*

Note: You must sign a nondisclosure agreement with Cray Research to order either of these manuals.

You can find a convenient source of E register opcodes and bit field definitions in the `mpp/mpphw_t3e.h` header file, available in the CrayLibs package. It has both physical and virtual addresses for the E register commands. The physical addresses correspond to the values listed in the *Commands* booklet, but you must use the virtual addresses in user and library code. The operating system maps the virtual addresses to the physical address when an E register command is issued.

The header file has examples of inline functions that use E registers. Inline functions are also available in the CrayLibs package for CRAY T3E systems.

The following sections include an overview of material in the hardware reference documents and add information about related support for E register manipulation in the compilers, loaders, and operating system.

6.1.1 Basics

The `stq`, `ldq`, `stl`, and `ldl` load/store hardware instructions trigger E register operations when accompanied by special addresses. The opcodes and operands for the E register commands are packaged in the following locations:

- The address of the load/store instruction.
- The data argument in the load/store instruction.
- The more operands block (MOBE) of E registers. A bit field in the data argument points to up to 4 E registers.
- The source or destination (SADE) E register, which contains data to be transferred. This E register is specified in a bit field within the address for the load/store instruction.

6.1.2 Usage Rules

The operating system, the compilers, and the system libraries also use E registers. You must follow certain conventions when using E registers to avoid breaking the system software. The following conventions are analogous to Alpha register use conventions that must be observed by assembly language programmers:

- Use context 0. The E register commands allow two contexts. Context 1 is reserved for the operating system. User-level code must use context 0. The context number is packed into the E register command address.
- Select E registers from the correct pool. The `mpp/mpphw_t3e.h` header file documents the four pools of E registers. The starting offsets within the E registers of these pools are identified by the following constants:
 - `_MPC_E_REG_STRIDE1`
 - `_MPC_E_REG_STRIDE_LW`
 - `_MPC_E_REG_SAV_MOBE`
 - `_MPC_E_REG_SCR_MOBE`
 - `_MPC_E_REG_SADE`

- Use a compiler directive to indicate E register usage. The following directives allocate E registers:

- `#pragma _CRI uses_eregs`
- `#pragma _CRI sade n and #pragma _CRI mobe n`

Because future C compiler releases might utilize E registers to perform single-processor optimizations, it would be advantageous for user code to explicitly specify the number of scratch source-and-destination E registers (SADE) and more operands block E registers (MOBE) needed. Also, the loader sets a special bit in the `a.out(5)` header if any of the modules linked into a program use E registers.

- Use `MB` and `WMB` instructions where needed. The memory barrier (`MB`) and write memory barrier (`WMB`) instructions are used when issuing E register commands. The `MB` instruction enforces the ordering of loads and stores, and the `WMB` instruction enforces the ordering of stores. Ordering is needed to ensure, for example, that previous stores to set up MOBE or SADE E registers have completed before issuing an E register command through a subsequent `stq` store instruction. The C and C++ compilers provide the `_memory_barrier()` and `_write_memory_barrier()` intrinsic functions, which generate `MB` and `WMB` instructions.
- Use volatile qualifiers to force loads and stores to memory. The C and C++ compilers ensure that a dereference of a volatile pointer always accesses memory. To operate the E registers, set up pseudo addresses in volatile pointers, and then load or store to the pseudo address to trigger the E register operation.

6.1.3 Example of a Single-word Put

This section contains an example of a C inlined function that stores an `int` value to an address on a remote processor. A remote update like this is called a put operation. The `_I_shmem_int_p` function shown here is present in the CRAY T3E version of the `mpp/shmem.h` header file.

Of course, E register operations can be coded in assembly code as well. The `mpp/mpphw_t3e.h` header file can be included in assembler code to import the same helpful E register opcode and bit-field definitions.

Example 36: Inline C function to put an int value to a remote processor's memory

```

1. #include <mpp/mpphw_t3e.h>
2.
3. static void
4. _I_shmem_int_p(int *addr, int value, int pe)
5. {
6.     volatile long * const Ecmd =
7.         (volatile long *)(_PUT(_MPC_E_REG_SADE));
8.     volatile int * const sade =
9.         ((volatile int*)_MPC_E_REG_BASE) + _MPC_E_REG_SADE;
10. #pragma _CRI sade 1
11.     *sade = value;           /* value to be transferred */
12.     _write_memory_barrier(); /* wait for SADE store */
13.     *Ecmd =                  /* issue the PUT command */
14.         (_MPC_E_REG_STRIDE1 << _MPC_BS_EDATA_MOBE) |
15.         (                (pe) << _MPC_BS_DFLTCENTPE) |
16.         (                (long)(addr)                );
17.
18.     _memory_barrier(        /* PUT must precede load */
19.                       /* of EREG_PENDING */
20. );
21. #pragma _CRI inline _I_shmem_int_p

```

For a line-by-line explanation of this example, continue on with this section. To move on to information on how to use the barrier and eureka synchronization units, see Section 6.2, page 137.

Line 1 includes the header file that defines bit fields and opcodes.

```
1. #include <mpp/mpphw_t3e.h>
```

Lines 3 and 4 declare the function to be static so that it can be included in a header file and be declared an inline function. If a function is inlined, there is no need for external linkage. If you do not want to create an inlined function, omit the static qualifier.

```
3. static void
4. _I_shmem_int_p(int *addr, int value, int pe)
```

The Ecmd pointer in line 6 is a pseudo-pointer that contains the E register put command code and a pointer to the E register (`_MPC_E_REG_SADE`) that will contain the data to be transferred to the remote processor. The `volatile`

keyword in the declaration of `Ecmd` ensures that the pseudo-store to `*Ecmd` that follows will be translated by the compiler into an actual store instruction.

```
6.     volatile long * const Ecmd =
7.     (volatile long *)(_PUT(_MPC_E_REG_SADE));
```

The `sade` pointer in line 8 is assigned the memory-mapped address of the E register that will hold the value being transferred by the put command. Again, it is declared `volatile` so that the store to `*sade` will be translated by the compiler into a store instruction.

```
8.     volatile int * const sade =
9.     ((volatile int*)_MPC_E_REG_BASE) + _MPC_E_REG_SADE;
```

The `#pragma` directive in line 10 tells the C compiler that the user is accessing the first source-and-destination E register (SADE). The compiler will not itself generate code that uses this SADE behind the scenes to do its own optimizations. It will use a different SADE, if needed.

```
10. #pragma _CRI sade 1
```

Line 11 stores the value to be transferred into the SADE.

```
11.     *sade = value;           /* value to be transferred */
```

The `_write_memory_barrier` intrinsic function is translated by the compiler into a WMB instruction, ensuring that the store to the SADE in line 11 precedes the E register command, which is issued in line 13.

```
12.     _write_memory_barrier(); /* wait for SADE store */
```

The store to `*Ecmd` in lines 13 through 16 is translated by the compiler into a `stq` instruction, which stores a value that contains encoding for the following:

- Specification of a MOBE (`_MPC_E_REG_STRIDE1`), which is pre-initialized in library program startup processing.
- Specification of a PE number. This is a 0-based index that selects the process within the application team that will receive the data.
- The address on the remote PE to which data will be delivered.

```
13.     *Ecmd =           /* issue the PUT command */
14.     (_MPC_E_REG_STRIDE1 << _MPC_BS_EDATA_MOBE) |
15.     (                    (pe) << _MPC_BS_DFLTCENTPE) |
16.     (                    (long)(addr)                    );
```

The `_memory_barrier` intrinsic function in line 18 is translated by the compiler into an MB instruction. An MB instruction ensures that all prior loads and stores are processed before all loads and stores that follow. In this case, the put command triggered by the prior pseudo-store to `*Ecmd` must be processed before any succeeding loads of the `_MPP_EREG_PENDING` memory-mapped register (MMR). This MMR is loaded by the barrier routine when waiting for prior remote put commands to complete.

```
18.     _memory_barrier(    /* PUT must precede load */
19.                               /* of EREG_PENDING */
```

The `#pragma _CRI inline` directive in line 21 inlines the function just defined. It is placed wherever it is called in the C source file. If you do not want to inline the function, omit this directive.

```
21. #pragma _CRI inline _I_shmem_int_p
```

6.2 Using Barrier and Eureka Synchronization Units

An application team can access up to 27 barrier/eureka synchronization units (BESUs). Each BESU can be used to implement a barrier or eureka event flag. A barrier flag changes state after all processes in an application team store to it. A eureka event flag changes state when one or more processes in an application team stores to it.

A barrier flag is most often used in same program, multiple data (SPMD) programs, in which all processes must synchronize around some type of communication. A eureka event flag is most often used to indicate the success of a searching algorithm by one process when all processes have been conducting similar searches in parallel.

The most portable way to use barrier and eureka event flags is to use library routines that interface to the hardware BESUs. Barrier routines to choose from include the following:

- `barrier(3)`
- `pvm_barrier(3)`
- `shmem_barrier_all(3)`

The eureka event routines include the following:

- `set_event(3)`, callable from Fortran, C, and C++
- `wait_event(3)`, callable from Fortran, C, and C++

- `test_event(3)`, callable from Fortran, C, and C++
- `clear_event(3)`, callable from Fortran, C, and C++

For more information on accessing BESUs, see *Barrier and Eureka Synchronization (CRAY T3E Systems)*, publication HMM-141-0. (A nondisclosure agreement must be signed with Cray Research before you can obtain this document.) For a convenient source of BESU state codes, see header file `mpp/mpphw_t3e.h`.

As is the case with E registers, BESUs are used internally by the libraries. To avoid introducing conflicts with system software, observe the following conventions:

- Use the following compiler directive to allocate BESUs:

```
#pragma _CRI besu n
```

This directive indicates that *n* BESUs should be allocated for use in the compilation unit. The sum of the BESU counts specified with directives in a program is recorded at link time and placed in the `a.out(5)` header. The operating system allocates the specified number to the application team at program startup. As a special case, the operating system does not allocate a BESU if the BESU count in the `a.out` header is 1 and the program uses one PE.

Note: The `besu` directive is not required when accessing BESUs through the standard barrier and event library routines mentioned in this section. It is only required when programming BESUs directly through techniques described in the *Barrier and Eureka Synchronization (CRAY T3E Systems)*.

- Use the `besu_alloc()` function to obtain the correct memory-mapped address. The synopsis for `besu_alloc` is as follows:

```
#include <mpp/besu.h>
besu_t *besu_alloc(void);
```

The `besu_alloc` function should be called during program execution exactly as many times as there are BESUs allocated by compiler directives. Furthermore, all PEs should call `besu_alloc` as a concurrent, collective operation so that all PEs obtain the same memory-mapped address for each BESU.

The following is an example of a barrier routine that illustrates correct program style for the use of BESUs:

Example 37: Using BESUs in a C function

```

#include <mpp/mpphw_t3e.h>
#include <mpp/besu.h>
#pragma _CRI besu 1

void mybarrier(void)
{
    static int firstcall = 1;
    static besu_t *besuptr;
    if (firstcall) {
        firstcall = 0;
        besuptr = besu_alloc();
    }
    /* Arm the barrier, indicating this PE's arrival */
    *besuptr = _MPC_OP_BAR;
    /* Wait for all other PEs to arrive at the barrier */
    while (*besuptr != _MPC_S_BAR) ;
}

```

6.3 Using Distributed I/O

The `listio(2)` system call is extended on CRAY T3E systems to provide the ability to issue read and write I/O requests for memory locations on a different PE in the application team.

To make a remote `listio` request, set the `LF_REMOTE` bit in the `li_flags` field and the PE number in the `li_remote_vpe` field of the `listreq` structure. The `iosw` structure, which flags I/O completion, must reside on the PE issuing the I/O request, not on the remote PE.

For more information on distributed I/O, see Section 5.4.2, page 125.

6.4 Global Application Team Termination

Pure process semantics apply to the following forms of termination processing:

- `exit()`
- `_exit()`

Only the process calling one of the above terminates. The other processes in the application team continue running.

When a process in an application team calls the `abort` function from C or C++, or when a program receives an exception signal (operand range error and floating-point errors are two examples), the entire application team is terminated. The following library extensions to C and C++ terminate the whole application team:

- `globalexit()`. This C-callable function causes all processes in the application team to go through normal exit processing. This function takes a single `int` argument to indicate exit status:

```
void globalexit(int status);
```

- The `C_APTEAM` category can be passed to the `killm(2)` system call to cause the signal to be sent to all processes in an application team.

active set

In SHMEM, a set of PEs defined to participate in a collective operation, such as sending a value from one PE to multiple PEs. See also *group* for the PVM equivalent, page 145, and *communicator* for the MPI equivalent, page 143.

application team

A group of one or more processes, running on one or more PEs, that have capabilities that are extensions to standard UNIX. Members of a team can work on the same or different parts of a program in parallel.

asynchronous receive

A receive operation that proceeds in parallel with other operations on the receiving PE. The send process must check later to find out if the receive has completed.

atomic operation

An operation that cannot be interrupted.

atomic swap

An atomic read-and-update operation on a remote or local data object. The value read is guaranteed to be the value of the data object at the time of the update.

bandwidth

The amount of data that can be moved from one place to another in a given period of time; it is usually expressed in megabytes per second.

barrier

A location in a program at which all PEs (or tasks) must stop until the final PE arrives. A barrier synchronizes the PEs and prevents situations such as having one PE reading a memory location that does not have the correct data yet.

blocking receive

A message receiving operation that waits until a message has arrived. Only after the message is received will the next instruction be executed.

bottom loading

A single-PE optimization technique used on some scalar loops in which operands are prefetched during each loop iteration for use in the next iteration. The operand is available as soon as the first loop instruction executes. A prefetch is performed even during the final loop iteration, before the loop's final jump test has been performed.

broadcast operation

Sending data from a single PE to all other PEs.

buffer

A block of memory used to store data temporarily before transferring it somewhere else.

cache-aligned data

Data stored at the beginning of a cache line. The C and C++ `cache_align` directive aligns data in cache.

cache coherence

All processors see the same value for any memory location, regardless of which cache the actual data is in, or which processor most recently changed the data. On the CRAY T3E system, only local memory references can be cached (all remote memory references use external E registers). Hardware on each CRAY T3E processor maintains cache coherence with the local memory, including when data is modified by a remote processor.

cache hit

A memory reference to a data object already in primary or secondary cache. Such references are closer and faster than references to data objects in local memory.

clock period

The time it takes an instruction to complete (sometimes called the cycle time) on the EV5 microprocessor. This is the smallest measurable unit of time used by the computer hardware. The length of a clock period varies from machine to machine. On the CRAY T3E system, the clock period is 3.3 nanoseconds (or 3.3 billionths of a second), and on a CRAY T3E-900 system, the clock period is 2.2 nanoseconds.

collective routine

A SHMEM routine that must be called by all PEs simultaneously. Such a routine requires the cooperation of all participating PEs. See also *individual routine*, page 145.

commit

In MPI, saving the formal description of a newly defined data type so that it can be used again.

communicator

In MPI, a group of processes that can send messages to each other. See also *group*, page 145, for the PVM equivalent and *active set*, page 141, for the SHMEM equivalent.

data cache

In each PE, a high-speed, random-access memory that temporarily stores frequently or recently accessed data. For an illustration showing where data cache fits in, see Figure 4, page 8. See also *secondary cache*, page 149.

dependency

When data from one section of code relies on a value from an earlier section of code.

direct-mapped cache

Method of relating areas of local memory to areas of data cache. Each line in memory is mapped to a bucket, which is a specific area in data cache. If a line in local memory is read, the data is moved into its bucket in data cache. Each bucket holds just one line. See also *set-associative cache*, page 149.

disk mirroring

A logical disk device composed of two or more physical disk slices. Mirroring is used to provide data redundancy when data integrity is important. It is implemented by using two to eight slices, usually on as many different physical disks, each of the same size. A write operation to a mirrored device causes separate write operations to be performed on each of the components. A read operation can be performed on any of the component devices.

disk striping

Splitting a disk file across two or more disk drives to enhance I/O performance. The performance gain is a function of the number of drives and channels used.

envelope

In MPI, message information used to identify and selectively receive messages. The four parts of the envelope are as follows:

- The rank of the receiver
- The rank of the sender
- A message tag
- A communicator

eureka

Hardware search mechanism. A eureka is like a barrier. When all of the PEs are searching for something, the one that finds it posts an eureka that is visible to all of the other PEs. The posting of the eureka stops the search.

fan-out

An optimized method of passing messages when a single PE is sending a message to multiple PEs. Rather than the sending PE sending the message to every PE, it sends the message to a subset of the PEs. Then the PEs that received the message in turn send it to a subset of the remaining PEs, and so on. See Figure 9, page 34, for an illustration of the process.

flushing cache

Clearing cache of its contents and storing the data of a write-back cache, such as data cache. On CRAY T3E systems, cache is automatically flushed.

gather operation

Collecting arrays from multiple remote PEs into a single array on the local PE. Also, collecting scattered elements of one array on one PE to consecutive elements on another PE. See also *scatter operation*, page 149.

GigaRing

The networking protocol that connects the CRAY T3E system to other resources (such as I/O devices and other computer systems). For an example of a GigaRing, see Figure 8, page 16.

group

In PVM, a defined set of tasks (or PEs) that participate in the same synchronization and communication processes. A group can either be all the tasks defined for a job or a user-defined subset of all the tasks. See also *active set* for the SHMEM equivalent, page 141, and *communicator* for the MPI equivalent, page 143.

GSEG

A global segment is used by the operating system to map a remote virtual address to a remote physical address.

individual routine

A SHMEM routine that must be called by a single PE, regardless of how many PEs are involved in the same operation. See also *collective routine*, page 143.

instance number

In PVM, if you specify the global group, the instance number is the same as the PE number. If you specify a group other than the global group, you can find the instance number from the `pvm_getinst(3)` function or from the return value of the `pvm_joingroup(3)` function.

latency of memory

The startup time. The period of time between when a PE requests data and when it can use the data.

line

A division of cache memory. Every location in local memory is mapped to an area of both data cache and secondary cache. This area in data cache contains one line, which is 4 64-bit words long. Each area in secondary cache contains three lines, each of which are 8 words long.

local memory

The memory available to a microprocessor on its own PE. Although any PE can access the memory of any other PE, the most efficient method is always a PE accessing its own memory. In some publications, memory is also called DRAM (for direct random access memory).

loop invariant

A value that does not change between iterations of a loop. In the following loop, 2.0, val, c[j], and j are loop invariant:

```
for (i=0; i<n; i+)
    a[i] = b[i] * 2.0 + val * c[j];
```

message-passing system

A software system that transfers explicit messages between PEs. The messages often contain data, such as array elements.

MFLOPS (or megaflops)

A timing measurement indicating how many millions of floating-point operations execute each second. A loop, for instance, that runs at 24 MFLOPS executes 24 million floating-point operations each second.

MPI, or Message Passing Interface

A message passing library that conforms to the Message Passing Interface Standard. It offers portability to other Cray

Research computer systems and to computer systems developed by other vendors.

multicast operation

In PVM, the multicast operation sends a message to each PE that has its task identifier number in an array.

multithreaded program

A program that is executed by multiple threads in parallel.

NaN

A value that is not a number but rather a symbolic entity encoded in floating-point format.

nanosecond

A measurement of time equal to one billionth of a second. A clock period on a CRAY T3E system running at 300 megahertz is equal to 3.3 nanoseconds.

nonblocking receive

A message receiving operation that will receive a message if one is present but will return immediately if one is not present.

overhead

In the context of this publication, time spent doing something other than the actual work of a program. For instance, the time required to move data, as opposed to the time spent processing data, is viewed as overhead.

overindexing

When a program attempts to access an array element that is outside the declared bounds of that array.

pad array

An unused array that aligns arrays containing data in cache in an optimal way.

PEs, or processing elements

The microprocessors that execute code on the CRAY T3E system. The CRAY T3E system can have up to 2,048 PEs configured.

process

In MPI, an independent, parallel code that runs on a PE. It is equivalent to a PVM task. See also *task*, page 150.

protocol

A standardized set of rules for transmitting data that allows communication between various entities.

PVM, or Parallel Virtual Machine

A message-passing library used when programming Cray MPP systems, Cray PVP systems, or certain other vendors' products. The principle virtues of PVM are portability and flexibility.

rank

In MPI, the number identifying a process. This is equivalent to a PE number.

read ahead

Reading data from local memory before it is needed. First, a block of memory is read from local memory and sent to the microprocessor. Then, the following block in memory is read, anticipating that it will be needed next. See also, *write behind*, page 152.

recurrence

A computation in a loop in which the values produced in the current iteration depend on values produced in previous iterations. A typical example is:

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

reduction

A loop or operation that reduces an array to a scalar value by doing a cumulative operation on many of the array elements.

For example, a summation is a reduction that adds all the elements of an array together to yield one number.

scatter operation

Distributing data from a single array on a single PE to multiple arrays on multiple PEs. Also, distributing consecutive data from one PE to nonconsecutive elements on another PE. See also *gather operation*, page 145.

secondary cache

A 96-Kbyte data cache, located between local memory and data cache, that optimizes memory access. See also *set-associative cache*, page 149, and *data cache*, page 143.

set-associative cache

A method of associating locations in local memory with locations in secondary cache. Each location in local memory is associated with a 3-line area of secondary cache. When a line is moved from local memory to secondary cache, it is moved into one of the three lines. CRAY T3E systems use a three-way associative cache, meaning each area in secondary cache can hold three lines at a time. See also *direct-mapped cache*, page 143.

SHMEM

A library of optimized subroutines that take advantage of shared memory. The most frequently used routines move data between the memory of a remote PE and the local PE, but a number of other facilities are available. The routines can either be used by themselves or in conjunction with another programming style, such as PVM. The principle virtue of SHMEM is its performance.

SM pool

In PVM, the shared memory (SM) pool lets a receiving PE continue to compute while receiving data.

software pipelining

A loop scheduling technique in which the execution of successive loop iterations are overlapped. Overlapping loop iterations exposes more instruction-level parallelism to the

processor, usually resulting in 100% utilization of one of its scheduled resources (such as functional units, cache bandwidth or memory bandwidth).

stream

On CRAY T3E systems, a stream is a series of data items between memory and the functional units of a PE. Similar to a pipeline on Cray PVP systems, a stream feeds data to the functional units in optimal fashion. For more information on streams, see Section 1.2.1, page 4. For information on how to make use of streams in your program, see Chapter 4, page 85.

stride

A term derived from the concept of walking through the data, from one location to the next. For instance, if every other element of an array were to be transferred, the stride through the array would be two.

stripmining

A single-PE optimization technique in which an array is broken down into chunks of convenient size (on the CRAY T3E system, the size of the cache) to allow optimal use of computational hardware.

symmetric

A data object is said to be symmetric if it has an address mapping across PEs that allows remote memory access. On CRAY T3E systems, a data object is symmetric if it has the same address on all PEs.

synchronization

Timing the actions of PEs to avoid problems. For instance, the `barrier` function might be used to prevent one PE from accessing a data location before another PE has updated that location.

task

In PVM, an independent, parallel process that executes on a PE. There is a one-to-one relationship between a task and a PE. See also *process*, page 148, for the MPI equivalent.

thrashing

A phenomenon that occurs when you have a fixed quantity of a reusable resource, the resource is allocated on a least-recently-used basis, and the cycle length of the reuse is larger than the quantity of the resource. A common example of thrashing involves pages in a paging operating system. Assume that there is only room for two pages in main memory, allocated on a least-recently-used basis, but a loop is referencing three arrays, each on a different page. The following code fragment suggests the situation:

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

After the references to `a[0]` and `b[0]`, the page for `a` and the page for `b` are in memory. When `c[0]` is referenced, the operating system removes `a`'s page, because it has room for only two pages, and `a`'s page was least recently used. Next `a[1]` is referenced, but `a`'s page is now out of memory, so the operating system removes `b`'s page. Likewise, the reference to `b[1]` ends up removing `c`'s page. Because more pages are referenced than there are room for in memory, because they are referenced cyclically, and because they are allocated in a least-recently-used basis, reuse never occurs, and the paging mechanism gives no benefit.

thread

The activity entity of execution. A sequence of instructions together with machine context (CPU registers) and a stack.

tiling

An optimization technique that combines stripmining and loop interchange. It operates on inner loops to increase cache reuse. Tiling is not done automatically by the compiler. See also, *stripmining* page 150.

ulp

Unit of least precision. It is used to discuss the accuracy of floating-point data. It represents the minimum step between representable values near a desired number: the true result to infinite precision, assuming the argument is exact. For instance, the ulp of 1.977436582E+22 is 1.0E+13, since the least significant (leftmost) digit of the mantissa is in the 10¹³ place. Within 0.5 ulp is the best approximation representable.

unrolling

A single-PE optimization technique that lets the compiler exploit parallelism at the functional unit level and take maximum advantage of data in cache.

virtual address

A normal user address that starts at 0 or some other consistent value in every program. The hardware translates a virtual address to a physical address, which is the true location in a machine's memory.

word

A data item that is 64 bits, or 8 bytes, long on Cray Research systems.

write behind

An I/O operation in which a block of data is written to disk, and the next block of data is buffered, anticipating that it will be written sequentially to disk next. See also *read ahead*, page 148.

write buffer

A 6-entry buffer through which a write operation passes on its way to, first, secondary cache, and eventually to local memory.

write through

When a data item is being written from a microprocessor to local memory, it makes a brief stopover in data cache. No memory is allocated in data cache for a write-through operation.

32-bit data
SHMEM, 59

A

active set
 definition, 141
add operation
 and splitting loops, 99
advantages of unrolling, 85
aggressiveness levels, 103
Apprentice timing tool, 21
array
 how elements are stored, 93
 operations on, SHMEM, 78
 rearranging dimensions, 92, 100, 101
arrays
 merging with SHMEM, 73
asynchronous receive
 definition, 141
asynchronous send, SHMEM, 46
ATM networks, 19
atomic operation
 definition, 141
atomic swap
 definition, 141
atomic swap, SHMEM, 76
autoloaders, 18

B

background
 Parallel Virtual Machine (PVM), 2
 programming styles, 2
 SHMEM, 2
 topics, 1
background information, 1

bandwidth
 data and secondary cache, 19
 definition, 141
barrier
 definition, 141
 example, SHMEM, 50
 in SHMEM program, 55
barriers
 avoiding in PVM, 32
benefits of unrolling, 85
block multiplexer tape drives, 18
blocking receive
 definition, 142
blocking receive, PVM, 32
bottom loading
 definition, 142
broadcast operation
 definition, 142
broadcast vs. multicast, PVM, 33
broadcast, SHMEM, 71
buffer
 definition, 142
buffering, double, 121
buffers
 setting size for sequential I/O, 128
buffers, send
 allocating in PVM, 25
buffers, stream, optimizing, 95

C

cache
 coherence, 5
 data and secondary, 5
 how it works, 9
 load and store timings, 19
 miss, 9, 11

- optimization, 92
- reuse, 92
- cache coherence
 - definition, 142
- cache hit
 - definition, 142
- cache layer, FFIIO
 - with random access I/O, 127
- cache, system
 - bypassing, 128
- cache-aligned data
 - definition, 142
- cache_align directive, 5
- cache_align directive, SHMEM, 60
- channels feature not available, 23
- clock period
 - definition, 143
- clock, real time, 21
- collective routine
 - definition, 143
- commit
 - definition, 143
- communicator
 - definition, 143
- compile time
 - increases with loop splitting, 99
- concurrent directive, 87
- conditional OPEN statements, 116
- converting data, 119
- converting data, PVM, 26

D

- dangers of loop splitting, 99
- data
 - 32-bit in SHMEM, 59
 - 32-bit optimization, PVM, 26
 - 32-bit packing, PVM, 26
 - broadcast with SHMEM, 71
 - conversion, 119
 - dependencies, 94
 - gather/scatter, PVM, 40

- gather/scatter, SHMEM, 67
- initializing and packing, PVM, 30
- movement, 9
- reuse, 26
- streams, 8
- stride-1, PVM, 28
- strided, SHMEM, 62
- transfer, SHMEM, 46
- data cache, 5
 - definition, 143
- data conversion, PVM, 26
- data flow, 7
- data transfer
 - comparing PVM and SHMEM, 3
- differences, to PVM on CRAY T3D systems, 23
- dimensions, rearranging, 100, 101
- direct access I/O, 128
- direct-mapped cache
 - definition, 143
- directive
 - pipelining, 87
- disk mirroring
 - definition, 144
- disk sector boundaries
 - for I/O requests, 121
- disk striping
 - automatic, 128
 - definition, 144
- disk support, 16
- distributed I/O, 125
- division operations, 104
- division outside of loop, 105
- division, how to avoid it, 104
- division, IEEE, 105
- double buffering, 121
- DRAM, same as local memory, 7

E

- E registers, 4
- envelope

definition, 144
ESCON tapes, 18
EtherNet, 18
eureka
definition, 144

F

fan-out
definition, 144
fan-out distribution, PVM, 34
FDDI network, 18
FFIO
description, 123
Fiber Channel disks, 16
Flexible File I/O
description, 123
flow of data, 7
flushing cache
definition, 145
formatted I/O
optimizations, 122
reducing, 122
functional unit
and pipelining, 91
functional units, 8

G

gather data, SHMEM, 67
gather operation
definition, 145
gathering data, PVM, 40
get_d_stream routine, 104
GigaRing
definition, 145
GigaRing network, 15
global I/O, 125
glossary
description, 1
grmview example, 123

group
definition, 145
grouping statements, 102
GSEG
definition, 145

H

hardware
illustration, 7, 8
hardware overview, 4
HIPPI disks, 17
HIPPI network, 19

I

I/O, 113
I/O from a single PE, 118
I/O requests
using large, 122
I/O strategies, 113
IEEE division, 105
if statement, splitting loop with, 98
individual routine
definition, 145
initializing data, PVM, 30
inner loop trip count, maximizing, 100
instance number, 145
intrinsic routines
vectorized, 109
intrinsics.h include file, 49
invariant references, maximizing, 93
IPI-2 disks, 17
ivdep directive
and pipelining, 88

L

large transfers

- how handled by PVM, 24
- latencies
 - data and secondary cache, 19
- latency, memory
 - definition, 146
- line
 - definition, 146
- local memory
 - checking your system for, 123
 - definition, 146
- logical PE number, 21
- loop
 - overlapping
 - examples, 89
- loop invariant
 - definition, 146
- loop iterations
 - overlapping, 86
- loop splitting
 - can change program, 99
 - examples, 97
- loops
 - identifying for pipelining, 88
- loops, splitting, 96
- loops, unrolling, 85

M

- memory
 - checking your system for, 123
 - performance information, 19
- memory overview, 4
- memory-resident FFIO layer
 - with random access I/O, 128
- memory-resident I/O
 - description, 123
- merging arrays, SHMEM, 73
- Message Passing Interface
 - definition, 146
- message size
 - finding and changing with PVM, 24
- message size, PVM, 24

- message-passing system
 - definition, 146
 - PVM, 2
- Mflops, or megaflops
 - definition, 146
- microprocessor, description, 15
- mirroring, disk, 16
- mixing send and receive routines, PVM, 30
- MPI
 - definition, 146
- MPI, switching to SHMEM, 51
- multicast operation
 - definition, 147
- multicast vs. broadcast, PVM, 33
- multiple file, multiple PE I/O, 116
- multiply operation
 - and splitting loops, 99
- _my_pe intrinsic function, 50

N

- NaN
 - definition, 147
- nanosecond
 - definition, 147
- network illustration, 16
- network overview, 15
- network protocols, 18
- nonblocking receive
 - definition, 147
- nonblocking receive, PVM, 31
- nonstandard, SHMEM, 45
- nosplit directive, 96
- _num_pes intrinsic function, 50

O

- organization of manual, 1
- output stream, 13
- overhead

- by grouping statements, 102
- definition, 147
- SHMEM, 46
- overindex command-line option, 99
- overindexing
 - dangerous with loop splitting, 99
 - definition, 147
- overlapping loop iterations, 86

P

- packing data, PVM, 30
- pad array
 - definition, 147
- Parallel Virtual Machine (PVM), 23
- PE, 15
 - logical number, 21
 - physical number, 21
- performance
 - SHMEM, 2
- peripherals overview, 15
- PEs
 - definition, 148
- physical PE number, 21
- pipelined, division is not, 104
- pipelining, 86
 - command-line options, 87
 - definition, 149
 - directive, 87
 - how it works, 89
- portability
 - PVM, 2
 - SHMEM, 2
- portability, SHMEM, 45
- powers of 2, 49
- prescheduling division, 105
- process
 - definition, 148
- processing element (PE), 15
- processing elements
 - definition, 148
- programming styles

- background, 2
- protocol
 - definition, 148
- PVM, 23
 - background, 2
 - PVM, or Parallel Virtual Machine
 - definition, 148
 - pvm_bcast function, 33
 - pvm_bufinfo routine, 38
 - pvm_bufinfo(3), 28
 - PVM_DATA_MAX environment variable, 24
 - pvm_gather function, 40
 - pvm_getopt, use of, 24
 - pvm_initsend routine, 25
 - pvm_mcast function, 33, 34
 - pvm_nrecv routine, 31
 - pvm_prerecv routine, 28
 - pvm_psend routine, 28
 - pvm_reduce, 38
 - pvm_scatter function, 40
 - PvmDataDefault, 25
 - PvmDataInPlace, 25
 - PvmDataRaw, 25, 28

R

- random access I/O, 128
- rank
 - definition, 148
- read ahead
 - definition, 148
- real-time clock, 21
- rearranging array dimensions, 92
- receiving stride-1 data, PVM, 28
- reciprocal multiplication, 104
- reduction
 - definition, 148
- reduction routines, SHMEM, 78
- reductions, PVM, 38
- remote memory, 4
- reorder an array, SHMEM, 67

reusing data, 26
 _rtc timing tool, 21

S

scatter data, SHMEM, 67
 scatter operation
 definition, 149
 scattering data, PVM, 40
 SCSI disks, 16
 SCSI tape drives, 17
 search, eureka, 144
 secondary cache, 5
 definition, 149
 send buffers
 allocating in PVM, 25
 sending stride-1 data, PVM, 28
 sequential I/O, 119
 set-associative cache
 definition, 149
 set_d_stream function, 104
 shared memory (SHMEM), 45
 SHMEM, 45
 background, 2
 definition, 149
 shmem.h include file, 49
 shmem_broadcast function, 71
 shmem_double_min_to_all function, 79
 shmem_fcollect function, 73
 shmem_get function, 46
 shmem_get32 function, 59
 shmem_iget function, 62
 shmem_int_sum_to_all function, 81
 shmem_iput function, 62
 shmem_ixget function, 67
 shmem_ixput function, 67
 shmem_my_pe function, 50
 shmem_n_pes function, 50
 shmem_put function, 46
 shmem_put32 function, 59
 shmem_short_fadd function, 76
 shmem_short_finc function, 76

shmem_swap function, 76
 single file, multiple PE I/O, 114
 single-PE optimizations, 85
 size of memory, 4
 size of message
 finding and changing with PVM, 24
 size of message, PVM, 24
 SM pool
 definition, 149
 software pipelining, 86
 definition, 149
 split command-line option, 96
 split directive, 96
 splitting loop with if statement, 98
 splitting loops, 96
 statistics
 memory performance, 19
 strategies for I/O, 113
 stream
 definition, 150
 reducing startup costs, 100
 stream buffers
 setting aggressiveness level, 103
 stream buffers, optimizing, 95
 stream references, maximizing, 100
 streams
 example of, 8
 reducing number, 101
 stride
 definition, 150
 stride-1
 streams, maximizing, 93
 stride-1 data, PVM, 28
 strided data, SHMEM, 62
 striping, disk, 16
 automatic, 128
 stripmining
 definition, 150
 stripmining example, 97
 subtract operation
 and splitting loops, 99
 swap, atomic, SHMEM, 76

symmetric
 definition, 150
synchronization
 avoiding in PVM, 32
 definition, 150
 minimizing with receive, PVM, 37
 PSYNC array, SHMEM, 71
 PVM, 2, 32
 SHMEM, 3
synchronization, SHMEM, 55
system cache
 bypassing, 128
system call I/O, 120

T

tapes supported, 17
task
 definition, 150
task identifier, PVM, 37
TCP/IP network, 19
thrashing
 definition, 151
tiling
 definition, 151
time to compile
 increases with loop splitting, 99
timing your code, 21
timings
 memory operations, 19
transfer
 32-bit data, PVM, 26

 of data, SHMEM, 46
transfer of data
 comparing PVM and SHMEM, 3
transfer, large
 how handled by PVM, 24

U

ulp
 definition, 152
unformatted I/O, 119
unroll directive, 86
unrolling
 by the compiler, 86
 command-line option, 86
 definition, 152
 outer loops, 86
 unrolling loops, 85

V

vectorization, 107

W

work while waiting,
 PVM, 31
write behind
 definition, 152