CRAY T3E™ Fortran Optimization Guide

Document Number 004–2518–002

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

Copyright © 1996, 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 Unpublished rights reserved under the Copyright Laws of the United States.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/*Turbo*Kiva, 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, OLNET, 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. DynaWeb is a trademark of Electronic Book Technologies, Inc. EXABYTE is a trademark of EXABYTE Corporation. HP is a trademark of Hewlett-Packard Company. IBM, ESCON, and Magstar are trademarks of International Business Machines Corporation. Silicon Graphics is a trademark of Silicon Graphics, Inc. STK, TimberLine, RedWood, are trademarks of Storage Technology Corporation. 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.

New Features

CRAY T3E™ Fortran Optimization Guide 004–2518–002

This rewrite of the *CRAY T3E Fortran Optimization Guide*, publication SG–2518 supports the 3.0 release of the Programming Environment. It includes updates to existing chapters and a new chapter on optimizing I/O.

This rewrite of the *CRAY T3E Fortran Optimization Guide*, publication SG–2518 supports the 3.2 release of the Programming Environment.

Record of Revision

Version	*Description*
2.0.1	September, 1996 Original Printing.
3.0	June, 1997 Revised for the Programming Environment 3.0 release.
3.2	January, 1999 Revised for the Programming Environment 3.2 release.

Contents

Page

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 3

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

Page

Example 1: Transferring 32-bit data . . . 26
Sending and Receiving Stride-1 Data . . . 28
Example 2: PVMFPSEND and PVMFPRECV . . . 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: PVMFBCAST . . . 34
Example 4: PVMFMCAST . . . 35
Minimizing Synchronization Time When Receiving Data . . . 36
Using the Reduction Functions . . . 37
Example 5: PvmSum . . . 38
Gathering and Scattering Data . . . 39
Example 6: Gather operation . . . 41
Example 7: Scatter operation . . . 41

SHMEM [3] **43**
Using SHMEM_GET64 and SHMEM_PUT64 for data transfer . . . 44
Example 8: Example of a SHMEM_PUT64 transfer . . . 44
Optimizing Existing MPI and PVM Programs by Using SHMEM . . . 48
Example 9: PVM version of the ring program . . . 48
Optimizing by Using SHMEM_GET64 . . . 51
Example 10: SHMEM_GET64 version of the ring program . . . 52
Optimizing by Using SHMEM_PUT64 . . . 54
Example 11: SHMEM_PUT64 version of the ring program . . . 54
Passing 32-bit Data . . . 55
Example 12: 32-bit version of ring program . . . 56

Page

Copying Strided Data 58
Example 13: Passing strided data using SHMEM_REAL_IGET 58
Example 14: Passing strided data using SHMEM_REAL_IPUT 59
Gathering and Scattering Data 62
Example 15: SHMEM_IXPUT version of a reordered scatter 63
Broadcasting Data to Multiple PEs 66
Example 16: One-to-all broadcasting 66
Merging Arrays 68
Example 17: SHMEM_FCOLLECT 69
Reading and Updating in One Operation 70
Example 18: Remote fetch and increment 71
Using Reduction routines 71
Example 19: Minimum value reduction routine 72
Example 20: Summation using a reduction routine 74

Single-PE Optimization [4] **77**

Unrolling Loops 77
Software Pipelining 78
Optimizing a Program with Software Pipelining 78
Selecting the Level of Pipelining 79
Using the CONCURRENT and IVDEP Directives 79
Identifying Loops for Pipelining 80
How Pipelining Works 81
Optimizing for Cache 85
Rearranging Array Dimensions for Cache Reuse 85
Example 21: Unoptimized code 85
Padding Common Blocks and Arrays to Reduce Cache Conflict 88
Automatic Padding 93

Page

Example 22: Automatic padding 95
Example 23: Automatic padding for smaller arrays 95
Optimizing for Stream Buffers 95
Splitting Loops 96
Example 24: Original loop 97
Example 25: Splitting loops 98
Example 26: Stripmining 98
Example 27: Splitting loops across `IF` statements 98
Example 28: Splitting individual statements 99
Padding Common Blocks and Arrays for Loop Splitting 100
Changed Behavior from Loop Splitting 101
Maximizing Inner Loop Trip Count 101
Example 29: Rearranging array dimensions 102
Minimizing Stream Count 102
Example 30: Minimizing streams 103
Example 31: Reduced streams version 103
Grouping Statements That Use the Same Streams 103
Example 32: Original code 104
Example 33: Grouping statements within the loop 104
Example 34: Loop that will be split into four 104
Example 35: Loop that will be split into two 105
Enabling and Disabling Stream Buffers 105
Optimizing Division Operations 106
Example 36: Original code 107
Example 37: Modified code 108
Vectorization 109
Example 38: Transforming a loop for vectorization 111

Page

Using the IVDEP Directive . . . 112
Bypassing Cache . . . 113

Input/Output [5] **117**

Strategies for I/O . . . 117
Using a Single, Shared File . . . 118
Using Multiple Files and Multiple PEs . . . 120
Using a Single PE . . . 121
Unformatted I/O . . . 122
Sequential, Unformatted Requests . . . 123
Formatted I/O . . . 126
Reduce Formatted I/O . . . 126
Make Large I/O Requests . . . 126
Minimize Data Items . . . 127
Use Longer Records . . . 127
Format Manually . . . 127
Change Edit Descriptors for Character Data . . . 128
FFIO . . . 128
Memory-resident Data Files . . . 128
Distributed I/O . . . 130
Example 39: Distributed I/O . . . 132
Using the Cache Layer . . . 133
Using Library Buffers . . . 133
Random Access . . . 134
Striping . . . 134
Example 40: Disk striping from within a program . . . 135

Glossary [6] **141**

Page

Index 155

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 33
Figure 10. A PvmMax reduction 38
Figure 11. The gather/scatter process 40
Figure 12. SHMEM_PUT64 data transfer 45
Figure 13. Identification of neighbors in the ring program. 50
Figure 14. SHMEM_REAL_IGET and SHMEM_REAL_IPUT transfers 62
Figure 15. Reordering elements during a scatter operation 65
Figure 16. The broadcast operation 68
Figure 17. An example of SHMEM_FCOLLECT 70
Figure 18. The SHMEM_REAL8_MIN_TO_ALL example 74
Figure 19. Overlapped iterations 83
Figure 20. Pipelining a loop with multiplications 84
Figure 21. Before and after array A has been optimized 87
Figure 22. Arrays B and C in local memory 89
Figure 23. Cache conflict between arrays B and C 90
Figure 24. Arrays B and C in local memory after padding 91

Page

Figure 25. Data cache after padding 92
Figure 26. Multiple PEs using a single file 118
Figure 27. Multiple PEs and multiple files 121
Figure 28. I/O to and from a single PE 122
Figure 29. Data paths between disk and an array 124
Figure 30. Data layout for distributed I/O 131

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 85

Preface

This publication documents optimization options for the Cray CF90 Fortran compiler running on CRAY T3E systems.

Related Publications

The following documents contain additional information that may be helpful:

- *CF90 Commands and Directives Reference Manual*, publication SR–3901
- *Fortran Language Reference Manual, Volume 1*, publication SR–3902
- *Fortran Language Reference Manual, Volume 2*, publication SR–3903
- *Fortran Language Reference Manual, Volume 3*, publication SR–3905
- *CF90 Ready Reference*, publication SQ–3900
- *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*
- *Application Programmer's Library Reference Manual*
- *Application Programmer's I/O Guide*
- *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 `orderdsk@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:

Convention	Meaning
`command`	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.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: 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 Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.
variable	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, *number*)	References the glossary for a definition of the preceding term.

The following machine naming conventions are used throughout this document:

Term	Definition
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 43.
- Single-PE optimizations concern getting the best performance out of each processing element (PE). See Chapter 4, page 77.
- Input/output (I/O) optimizations help you move data between external devices (such as disk) and memory. See Chapter 5, page 117.

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 3), 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 147), 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 and subroutines that pass data in a variety of ways, provide synchronization, and perform *reductions* (glossary, page 149). SHMEM routines are implemented on Cray MPP 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 151), which is provided automatically with some of the other programming

styles. The following figure shows how the SHMEM routines enhance performance by dispensing with some of the processes followed by PVM.

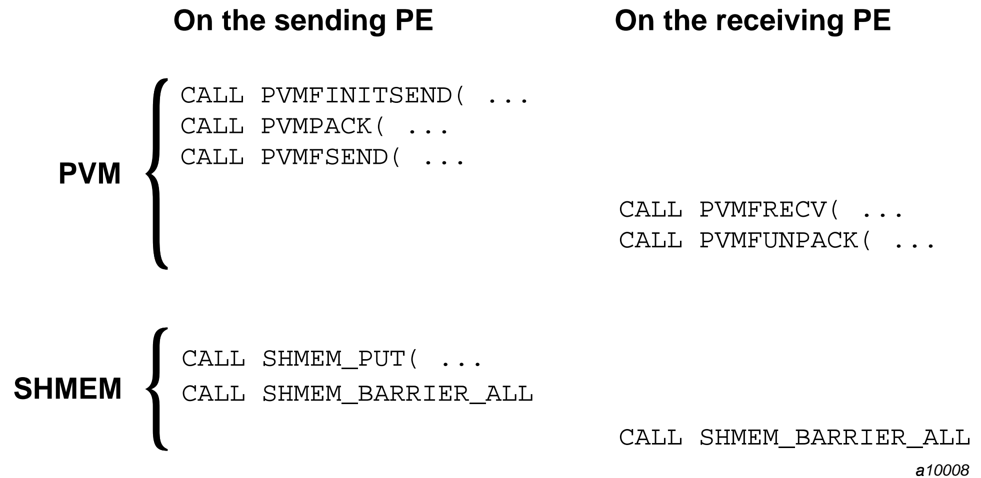

Figure 1. Data transfer comparison

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 routines to your code, or replacing the statements of another programming style with SHMEM routines, 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 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.

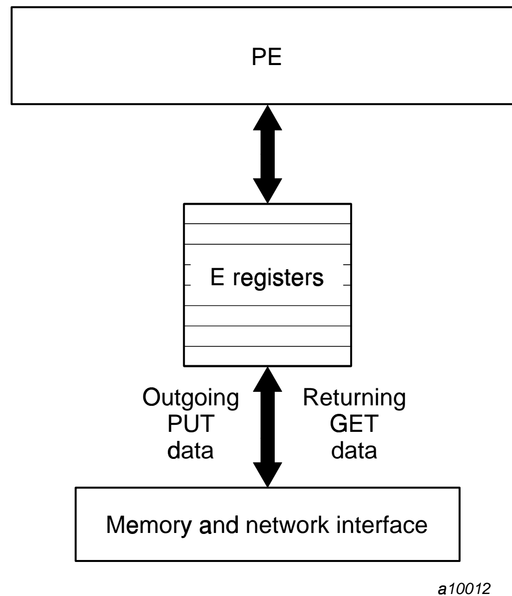

Figure 2. Position of E registers

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 153) 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 55). 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 144), 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 77.

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

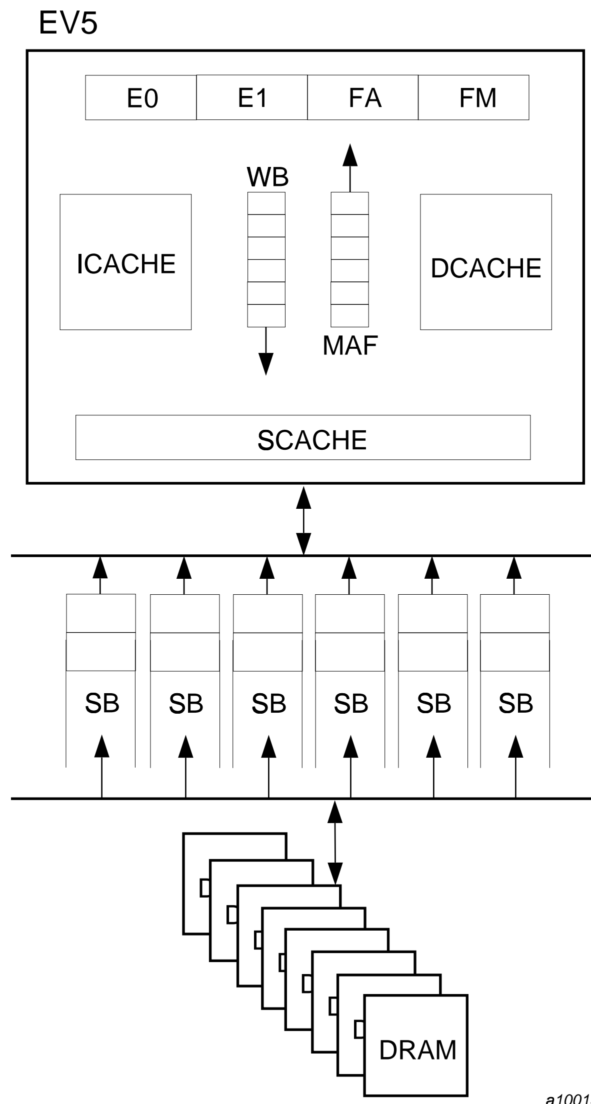

Figure 3. Flow of data on a CRAY T3E node

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.

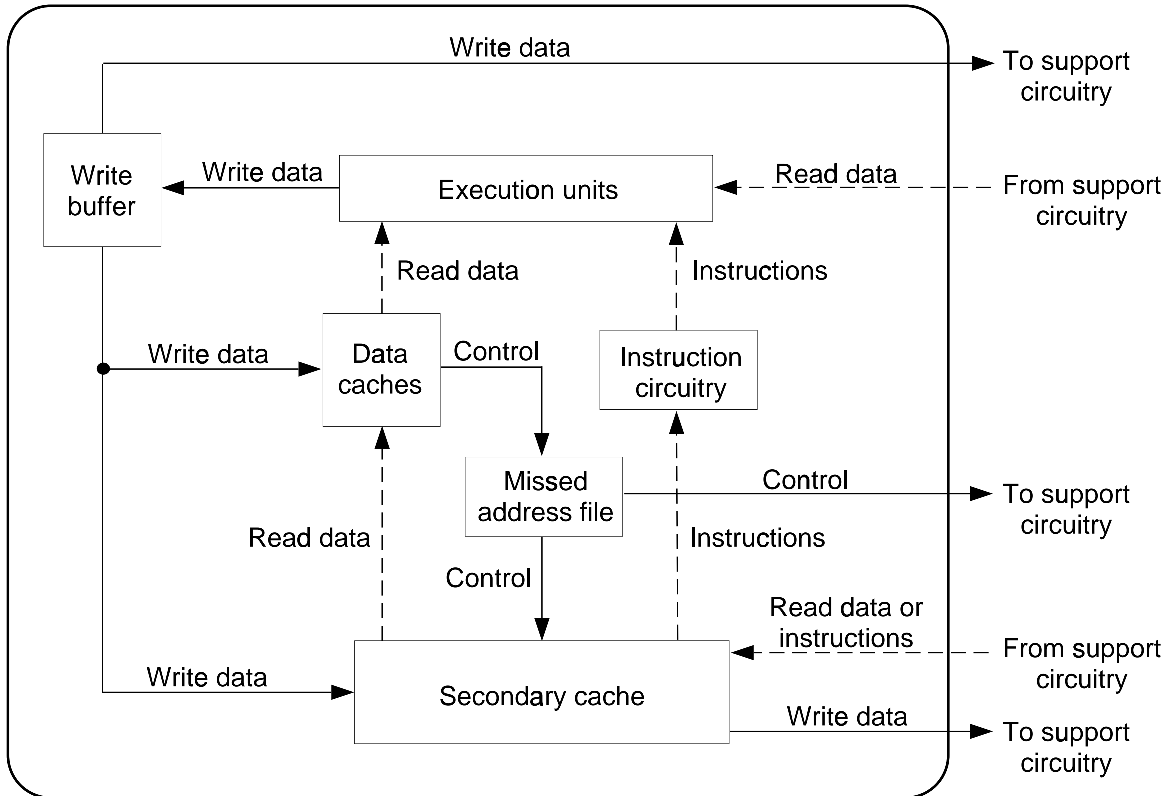

Figure 4. Data flow on the EV5 microprocessor

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)`):

```
DO I = 1, 1000
  A(I) = B(I) + C(I) + D(I) + E(I) + F(I)
END DO
```

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 4.4.7, page 105.

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:

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

Procedure 1: Moving data from memory

1. An EV5 register requests the value of `B(1)`from data cache.
2. Data cache does not have `B(1)`. It requests `B(1)` from secondary cache.
3. Secondary cache does not have `B(1)`. 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(1)` from data cache. The state of the data at this point is as illustrated in the following figure.

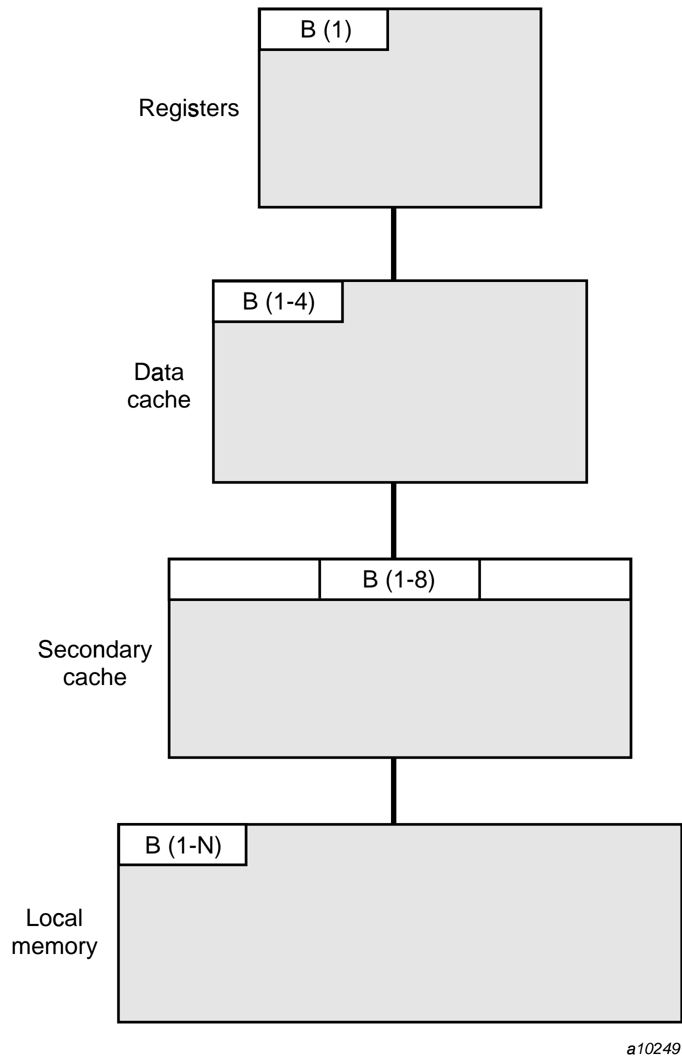

Figure 5. First value reaches the microprocessor

6. When other registers need `B(2)` through `B(4)`, they find them in data cache.

7. When a register needs `B(5)`, data cache does not have it.

8. Data cache requests `B(5)` through `B(8)` from secondary cache, which has them and passes them on.
9. Data cache passes `B(5)` through `B(8)` on to the appropriate registers as it gets requests for them. When the microprocessor finishes with them, it requests `B(9)` 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(9)` gets to the register, the situation is as illustrated in the following figure.

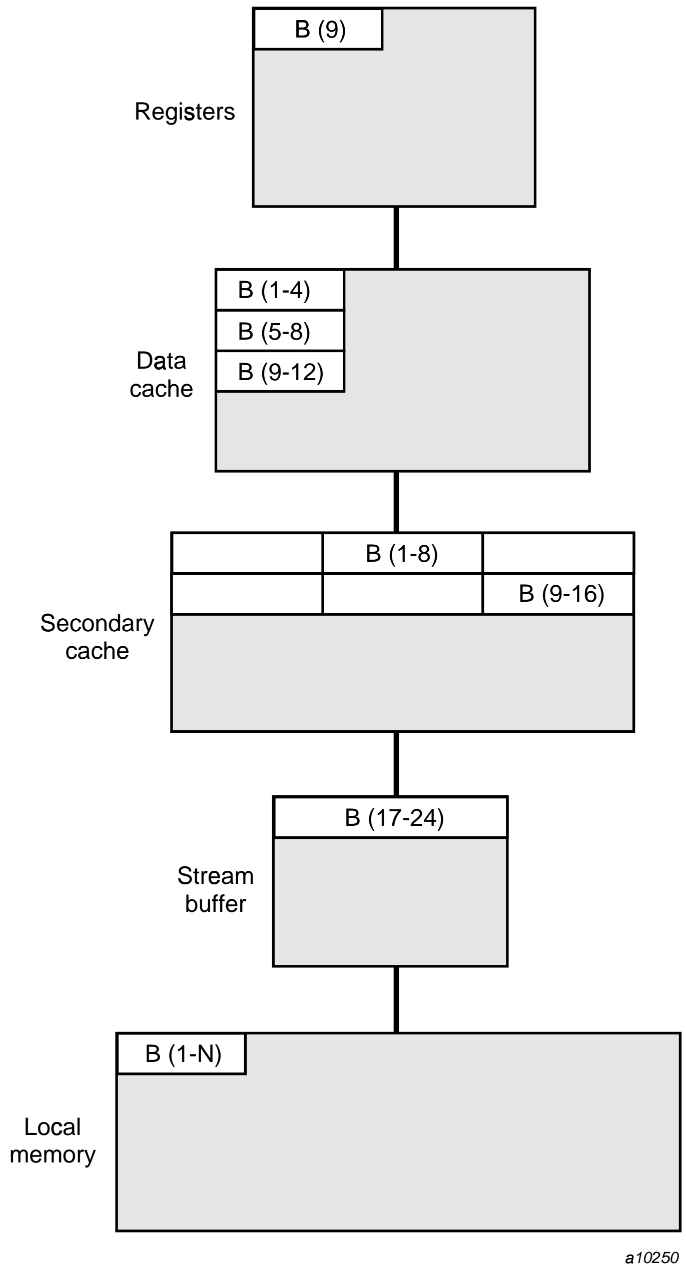

Figure 6. Ninth value reaches the microprocessor

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(17)` through `B(24)` 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.

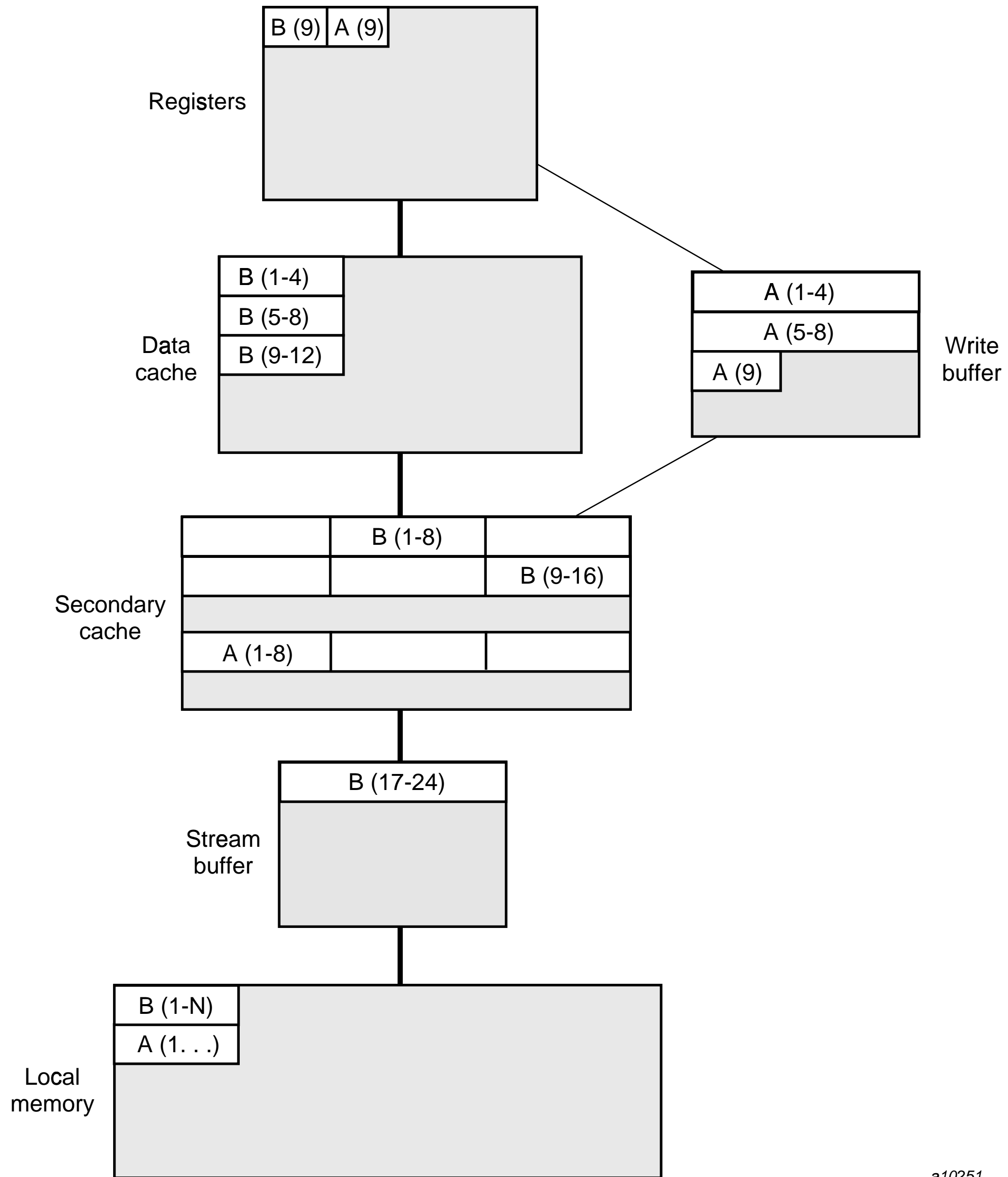

Figure 7. Output stream

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.)

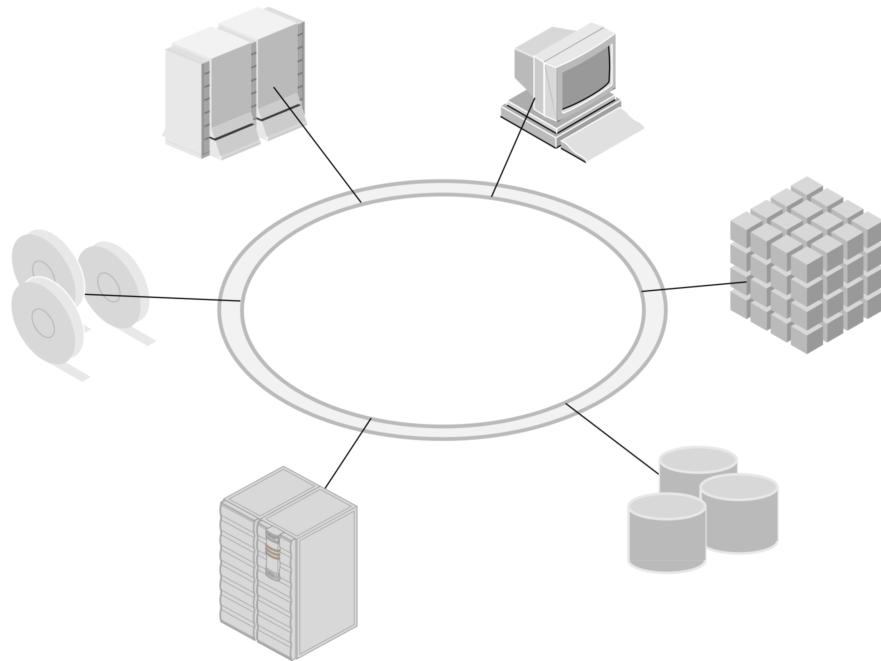

Figure 8. An external GigaRing network

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 (TimberlLine).
  - 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 `IRTC` intrinsic function returns values from the real-time clock in clock ticks. It is good for timing blocks of code that are part of a subprogram.The Fortran routine `PXFSYSCONF`, which returns the number of clock ticks in a second, helps convert an `RTC` value to seconds.
- 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 `IRTC` on the same code at the same time. MPP Apprentice introduces a significant amount of overhead that will be included in the `IRTC` 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 routines 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 routines (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 36).
- Using the reduction functions to execute an operation on multiple PEs (see Section 2.11, page 37).

- Distributing data from one PE to multiple PEs and gathering data from multiple PEs to a single PE (see Section 2.12, page 39).

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 PVMFGETOPT(3) routine, as follows. The variable MAXmax will hold the maximum message size value, in bytes.

```
CALL PVMFGETOPT(PVM_DATA_MAX, MAX)
```

You cannot, however, change the value within the program by using the PVMFSETOPT(3) routine. 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 `PVMFINITSEND`(3) subroutine 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:

```
CALL PVMFINITSEND(PvmDataInPlace, ISTAT)
```

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 4.4.7, page 105.
- 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 almost twice as fast as sending 64-bit data. 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 REAL(KIND=4) format:

Example 1: Transferring 32-bit data

```
1.        PROGRAM COMPARE2
2.  C
3.  C PVM version
4.  C
5.        INCLUDE 'fpvm3.h'
6.        INTEGER ME
7.        INTEGER ISTAT, SENDER, RECEIVER
8.        PARAMETER(N=1000)
9.        PARAMETER(SENDER = 0)
10.       PARAMETER(RECEIVER = 1)
11.       PARAMETER(MTAG = 2)
12.       REAL (KIND=4) D_SEND(N), D_RECV(N)
13. C
14. C Get PE info
15.       CALL PVMFMYTID(MYTID)
16.       CALL PVMFGETPE(MYTID, ME)
```

```
17. C
18. C Initialize data
19.       DO I=1,N
20.         D_SEND(I) = I * 1.0
21.       END DO
22.
23.       IF (ME .EQ. SENDER) THEN
24. C
25. C Send data
26.         CALL PVMFINITSEND(PvmDataRaw, ISTAT)
27.         CALL PVMFPACK(REAL4, D_SEND, N, 1, ISTAT)
28.         CALL PVMFSEND(RECEIVER, MTAG, ISTAT)
29.       ELSE IF (ME .EQ. RECEIVER) THEN
30. C
31. C Receive data
32.         CALL PVMFRECV(SENDER, MTAG, ISTAT)
33.         CALL PVMFUNPACK(REAL4, D_RECV, N, 1, ISTAT)
34. C
35. C Print results
36.         WRITE(*,*) 'Receiver=',ME,' D_RECV=', D_RECV(1), D_RECV(2)
37.     1              ,D_RECV(3),D_RECV(4),D_RECV(5),D_RECV(6),D_RECV(7)
38.       ENDIF
39.       END
```

The program in this example has the following output:

```
 Receiver=1 D_RECV=1.,  2.,  3.,  4.,  5.,  6.,  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 5 references the PVM `INCLUDE` file. See your system administrator for the actual location of the `INCLUDE` file on your system. You must also either load the Message Passing Toolkit (MPT) module with the `module`(1) command or specify the location of the `INCLUDE` file with the `-I` option on the `f90`(1) command line.

```
5.       INCLUDE 'fpvm3.h'
```

Lines 9 and 10 define the sending PE as PE 0 and the receiving PE as PE 1.

```
9.       PARAMETER(SENDER = 0)
10.      PARAMETER(RECEIVER = 1)
```

Line 12 defines the sizes of the sending and receiving arrays. The (KIND=4) specification defines 32-bit data.

```
12.      REAL (KIND=4) D_SEND(N), D_RECV(N)
```

The fastest of the data encoding arguments to PVMFINITSEND, in this case, is PvmDataRaw because data conversion is not needed. If the program were transferring 32-bit integer data, PvmDataDefault would be faster (see Section 2.2, page 25). 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 26 through 28 initialize the send buffer, pack the 32–bit data, and send the data:

```
26.        CALL PVMFINITSEND(PvmDataRaw, ISTAT)
27.        CALL PVMFPACK(REAL4, D_SEND, N, 1, ISTAT)
28.        CALL PVMFSEND(RECEIVER, MTAG, ISTAT)
```

Lines 32 and 33 receive and unpack the 32–bit data:

```
32.        CALL PVMFRECV(SENDER, MTAG, ISTAT)
33.        CALL PVMFUNPACK(REAL4, D_RECV, N, 1, ISTAT)
```

2.4 Sending and Receiving Stride-1 Data

PVMFPSEND(3) and PVMFPRECV(3) are send and receive routines that transfer either a single data item or stride-1 data to one PE. You do not have to initialize a send buffer or pack and unpack the data when you use PVMFPSEND and PVMFPRECV. For short messages, they run faster than the traditional send and receive routines, PVMFSEND and PVMFRECV.

The trade-off for the increase in speed is reduced flexibility. Using PVMFPSEND, 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 PVMFPRECV. But, after PVMFPRECV completes, it is done with the message. Using PVMFRECV, one or more unpack calls may follow the PVMFRECV call, and information about the message must be kept around in case the user calls PVMFBUFINO(3). In both the send and the receive, the PVMFPSEND and PVMFPRECV routines offer much simpler and faster code.

The speedups from using PVMFPSEND and PVMFPRECV 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 PVMFSEND and PVMFRECV are not significant.

The following example shows a program that passes data by using the PVMFPSEND and PVMFPRECV routines. The SRC PE passes data to the DEST PE, which in turn passes it back to the SRC PE.

Example 2: PVMFPSEND and PVMFPRECV

```
1.        PROGRAM PSEND_PRECV
2.        INCLUDE 'fpvm3.h'
3.        INTEGER SRC, DEST
4.        PARAMETER(SRC = 0)
5.        PARAMETER(DEST = 1)
6.        PARAMETER(LEN = 10)
7.        PARAMETER(BACK_AND_FORTH = 1000)
8.        REAL ARRAY(LEN)
9.        INTRINSIC MY_PE
10.
11.       ME = MY_PE()
12. C Initialize data
13.       IF (ME .EQ. SRC) THEN
14.          DO I = 1, LEN
15.            ARRAY(I) = I * 1.0
16.          ENDDO
17.       ENDIF
18.
19. C Send and receive data BACK_AND_FORTH times
20.       DO I = 1, BACK_AND_FORTH
21. C Send data to DEST PE
22.         IF(ME .EQ. SRC) THEN
23.            CALL PVMFPSEND(DEST, LEN, ARRAY, LEN, REAL8,
24.     $                   ISEND)
25. C Receive data from DEST PE
26.            CALL PVMFPRECV(DEST, LEN, ARRAY, LEN, REAL8,
27.     $                   IATID, IATAG, IALEN, IRECV)
28.         ELSE
29. C Receive data from SRC PE
30.            CALL PVMFPRECV(SRC, LEN,  ARRAY, LEN, REAL8,
31.     $                   IATID, IATAG, IALEN, IRECV)
32. C Send data to SRC PE
33.            CALL PVMFPSEND(SRC, LEN, ARRAY, LEN, REAL8,
34.     $                   ISEND)
```

```
35.          ENDIF
36.        ENDDO
37.        END
```

The MY_PE function, define in line 9 and referenced in line 11, returns the number of the PE on which is executes. It is available on the CRAY T3E system as both an intrinsic and an external library routine. While the intrinsic is slightly faster, the library version is more portable. Unless you specifically declare MY_PE as an intrinsic, as in line 9, you will get the external library version. The same is true for the constant N$PES and the library equivalent NUM_PES. N$PES is slightly faster, but NUM_PES is more portable.

2.5 Mixing Send and Receive Routines

The PVMFPSEND(3) and PVMFPRECV(3) routines work together with the PVMFSEND and PVMFRECV send and receive calls. You can incrementally 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 PVMFSEND(3), you can change it to use PVMFPSEND without worrying about finding and changing the matching PVMFRECV call. You can also continue to use PVMFBCAST(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 PVMFPRECV 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 information on choosing between the more portable NUM_PES library routine and the slightly faster N$PES constant, see Section 2.4, page 28, and Section 3.2.1, page 51.)

```
DO I = 1, NUM_PES
  CALL PVMFINITSEND(PvmDataRaw, ISTAT)
  CALL PVMFPACK(REAL8, ARRAY, N, 1, ISTAT)
  CALL PVMFSEND(I, MTAG, ISTAT)
END DO
```

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

```
CALL PVMFINITSEND(PvmDataRaw, ISTAT)
CALL PVMFPACK(REAL8, ARRAY, N, 1, ISTAT)
DO I = 1, NUM_PES
  CALL PVMFSEND(I, MTAG, ISTAT)
END DO
```

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 PVMFBCAST(3) and PVMFMCAST(3) (see Section 2.9, page 33), provide alternatives to what remains of the DO loop.

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

```
DO I = 1, NUM_PES-1
  CALL MYOWN_SEND(I, NSIZE, N, ISTAT)
END DO
```

In this example, MYOWN_SEND is a message-passing envelope that contains PVM code that might look something like the following. This subroutine makes the same mistake as the previous example by initializing and packing for each send.

```
SUBROUTINE MYOWN_SEND(I, NSIZE, N, ISTAT)
  CALL PVMFINITSEND(PvmDataRaw, ISTAT)
  CALL PVMFPACK(REAL8, NSIZE, N, 1, ISTAT)
  CALL PVMFSEND(I, MTAG, ISTAT)
RETURN
END
```

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 some work done while waiting for a message to arrive. In such cases, the PVMFNRECV(3) routine is a good substitute for PVMFRECV(3) for receiving a message. PVMFNRECV 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

PVMFNRECV 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 PVMFNRECV:

```
      CALL PVMFNRECV (-1, 4, ARRIVED)
        IF (ARRIVED .EQ. 0) THEN
C             Do something else
        ELSE
C             Process data in message
        ENDIF
```

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 PVMFRECV 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 PVMFNRECV routine (see Section 2.7, page 31). On the CRAY T3E system, PVMFNRECV provides comparable performance to the PVMFRECV routine. 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 section.

2.9 Using Broadcast or Multicast

The PVMFBCAST(3) and PVMFMCAST(3) routines offer two methods of sending messages to multiple PEs in a single call. The *broadcast* (glossary, page 142) routine, PVMFBCAST, 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) routine, PVMFMCAST, sends to all PEs with PE numbers that appear in an array that you define.

Although PVMFMCAST provides more flexibility concerning which PEs will receive the message, PVMFBCAST is usually faster. If the group name you give to PVMFBCAST is the global name, PVMALL, 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, page 33.) 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.

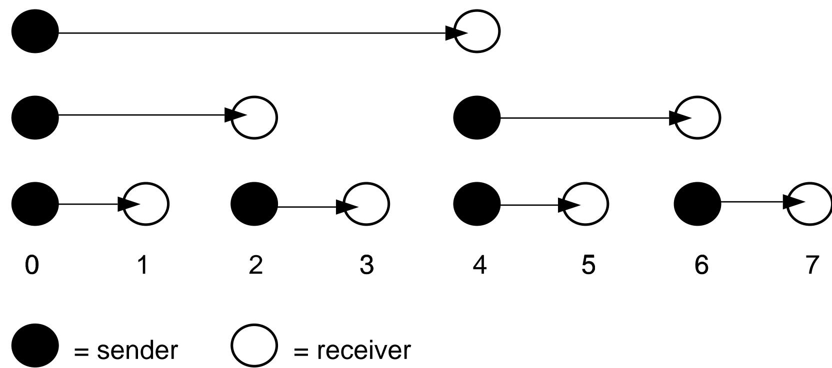

Figure 9. Fan-out method used by broadcasting routines

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

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

The following two examples use PVMFBCAST and PVMFMCAST, respectively, to transfer an array of 10 elements to all other PEs attached to the job:

Example 3: PVMFBCAST

```
      PROGRAM BCAST
      INCLUDE 'fpvm3.h'
      PARAMETER(LEN=10)
      INTEGER  MYTID, ME, NPES
      DIMENSION ARR(LEN)

C Use PVM method of obtaining task id, PE number, number of PEs
      CALL PVMFMYTID(MYTID)
      CALL PVMFGETPE(MYTID, ME)
      CALL PVMFGSIZE(PVMALL, NPES)

C PE 0 initializes, packs, and sends the array of 10 elements
      IF (ME .EQ. 0) THEN
         DO I = 1, LEN
           ARR(I) = I / 2.0
         ENDDO
         CALL PVMFINITSEND(PvmDataRaw, ISTAT)
         CALL PVMFPACK(REAL8, ARR, LEN, 1, ISTAT)
         CALL PVMFBCAST(PVMALL, LEN, ISTAT)
C All other PEs receive it
      ELSE
         CALL PVMFRECV(0, LEN, ISTAT)
         CALL PVMFUNPACK(REAL8, ARR, LEN, 1, ISTAT)
      ENDIF

C A representative PE prints the array
      IF (ME .EQ. 2) THEN
        WRITE(*,*) 'The array values are: ', ARR

      ENDIF
      END
```

Example 4: `PVMFMCAST`

```
      PROGRAM MCAST
      INCLUDE 'fpvm3.h'
      PARAMETER(LEN=10)
      INTEGER MYTID, ME, NPES
      DIMENSION ARR(LEN)
      INTEGER PE_ARR(NUM_PES)

C Use PVM method of obtaining task id, PE number, number of PEs
      CALL PVMFMYTID(MYTID)
      CALL PVMFGETPE(MYTID, ME)
      CALL PVMFGSIZE(PVMALL, NPES)

C Set up array of PE numbers
      DO I = 1, NPES-1
        PE_ARR(I) = I
      ENDDO

C PE 0 initializes, packs, and sends the array of 10 elements
      IF (ME .EQ. 0) THEN
         DO I = 1, LEN
           ARR(I) = I / 2.0
         ENDDO
         CALL PVMFINITSEND(PvmDataRaw, ISTAT)
         CALL PVMFPACK(REAL8, ARR, LEN, 1, IPACK)
         CALL PVMFMCAST(NPES, PE_ARR, LEN, ICAST)
C All other PEs receive it
      ELSE
         CALL PVMFRECV(0, LEN, IRECV)
         CALL PVMFUNPACK(REAL8, ARR, LEN, 1, IUPK)
      ENDIF

C A representative PE prints the array
      IF (ME .EQ. 1) THEN
        WRITE(*,*) 'The array values are: ', ARR
      ENDIF

      END
```

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 PVMFBCAST becomes more of a factor as the number of PEs increases, the advantage in using PVMFBCAST is most apparent when more PEs are involved in the job. But even when using as few as eight PEs, PVMFBCAST still has better performance than PVMFMCAST.

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 PVMFRECV(3) routine 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 PVMFRECV routine specifies the PE number from which the data is received.

```
OFFSET = 1
DO I = 1, NPES-1
  CALL PVMFRECV(I, MSGTAG, IRECV)
  CALL PVMFUNPACK(REAL8, ARRAY(OFFSET), LENGTH, 1, IUPK)
  OFFSET = OFFSET + LENGTH
ENDDO
```

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:

```
DO I = 1, NPES-1
  CALL PVMFRECV(-1, MSGTAG, IRECV)
  CALL PVMFUNPACK(REAL8, X((MSGTAG-1) * LENGTH + 1), LENGTH, 1, IUPK)
ENDDO
```

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:

```
DO I = 1, NPES-1
  CALL PVMFRECV(-1, MSGTAG, IRECV)
  CALL PVMFUNPACK(INT4, ISOURCE, 1, 1, IUPK1)
```

```
  CALL PVMFUNPACK(REAL8, X((ISOURCE-1) * LENGTH +1), LENGTH, 1, IUPK2)
ENDDO
```

This example assumes that each PE has sent its identifier in ISOURCE, 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 PVMFBUFINFO routine retrieves the value of the task identifier (the fourth argument), converts it to a PE number using PVMFGETPE(3), and places it in the variable NEXTPE. NEXTPE is then used in the PVMFUNPACK routine to provide the element number in the array X.

```
DO I = 1, NPES-1
  CALL PVMFRECV(-1, MSGTAG, IRECV)
  CALL PVMFBUFINFO(IBUFID, IBYTES, ITAG, ISOURCE, ISTAT)
  CALL PVMFGETPE(ISOURCE, NEXTPE)
  CALL PVMFUNPACK(REAL8, X((NEXTPE-1) * LENGTH + 1), LENGTH, 1, IUPK2)
ENDDO
```

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

2.11 Using the Reduction Functions

The PVM reduction subroutine, PVMFREDUCE(3), performs common functions across multiple PEs, returning the results to a single PE. For instance, if each PE contains an array of integers, PVMFREDUCE 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, page 38:

```
CALL PVMFREDUCE(PvmMax, ARR, 10, INTEGER8, MTAG, PVMALL, 0, IMAX)
```

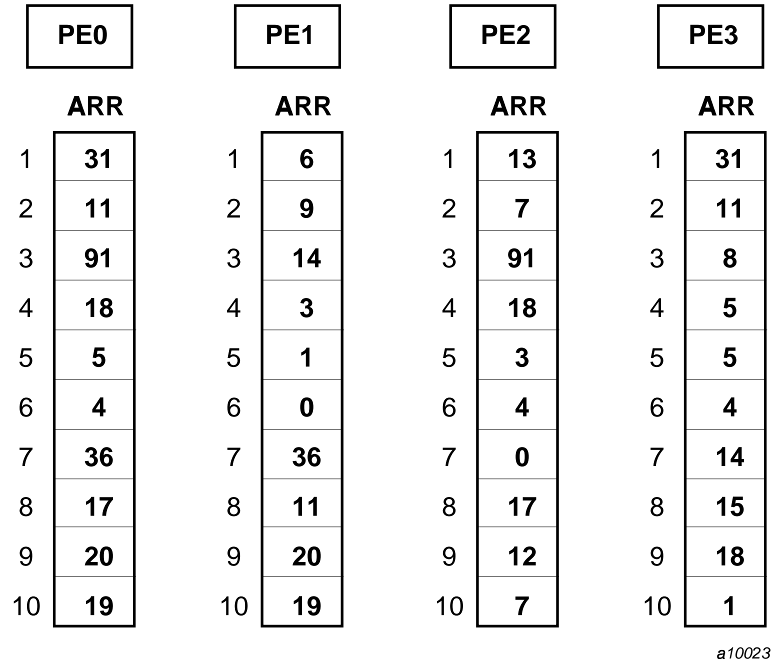

Figure 10. A `PvmMax` reduction

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: `PvmSum`

```
      PROGRAM REDUCE
      INCLUDE 'fpvm3.h'
      INTEGER MYTID, ME, NPES, LEN
      INTEGER ARR(10), RESULTS(10)
      EXTERNAL PVMSUM

      CALL PVMFMYTID(MYTID)
      CALL PVMFGETPE(MYTID, ME)
      CALL PVMFGSIZE(PVMALL, NPES)

C Initialize the array
      LEN = 10
      DO I = 1, LEN
```

```
          ARR(I) = ME * I
        ENDDO

C Make sure initialization is complete
        CALL BARRIER

C Find the sums at each location
        CALL PVMFREDUCE(PvmSum,ARR,LEN,INTEGER8,LEN,PVMALL,0,IRED)

C Write the answers on PE 0
        IF (ME .EQ. 0) THEN
          WRITE(*,*) 'The array sums are: ', ARR
        ENDIF
        END
```

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

```
The array sums are: 28, 56, 84, 112, 140, 168, 196, 224, 252, 280
```

2.12 Gathering and Scattering Data

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

The PVMFSCATTER routine distributes sections of an array among a group of PEs, and PVMFGATHER combines arrays from multiple PEs into a single array. The process is illustrated in Figure 11, page 40.

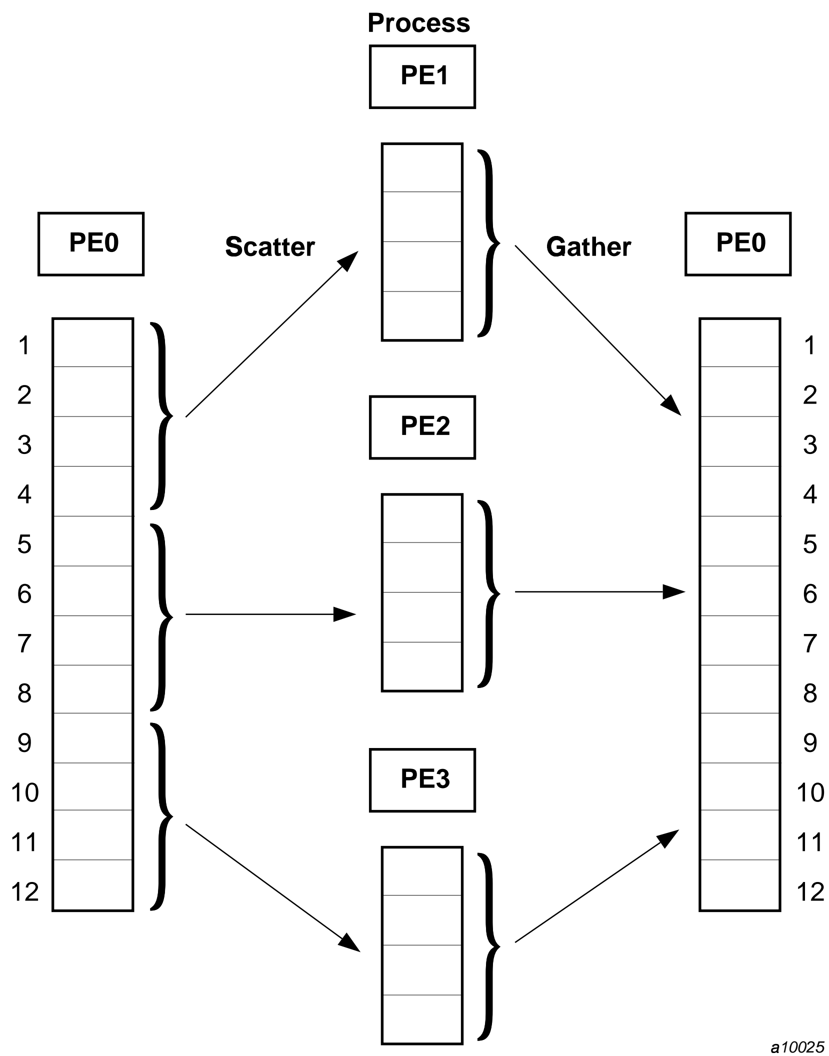

Figure 11. The gather/scatter process

The following example collects each PE's `SMALL_C` array into the `BIG_C` array on the root PE:

Example 6: Gather operation

```
      PROGRAM GATHER
      INCLUDE "fpvm3.h"

C Initialize PE0 as the root PE
      INTEGER, PARAMETER :: ROOT_PE = 0

      PARAMETER(IXDIM = 64, IYDIM = 128)
      DIMENSION A(IXDIM, IYDIM)
      DIMENSION B(IYDIM)
      DIMENSION SMALL_C(IXDIM)
      DIMENSION BIG_C(IXDIM*4) !Assume 4 PEs
C
      CALL PVMFMYTID(MYTID)
      CALL PVMFGETPE(MYTID, MYPE)
      MSGTAG = 9
      A = MYPE
      B(1:IYDIM-1:2) = -1.2
      B(2:IYDIM:2) = 1.0
      SMALL_C = 0.0
C Perform matrix-vector multiplication on each PE
      DO I = 1, IXDIM
         DO J = 1, IYDIM
            SMALL_C(I) = SMALL_C(I) + A(I,J) * B(J)
         ENDDO
      ENDDO
C Gather each PE's SMALL_C into ROOT_PE's BIG_C
      CALL PVMFGATHER(BIG_C,SMALL_C,IXDIM,REAL8,MSGTAG,PVMALL,ROOT_PE,IGATHER)
      IF (MYPE .EQ. ROOT_PE) THEN
         WRITE(6,1) (BIG_C(I), I = 1, IXDIM*4)
1        FORMAT(8(1X, F7.2))
      ENDIF
      END
```

The following example scatters the BIG_X array into the smaller SMALL_X arrays.

Example 7: Scatter operation

```
        PROGRAM SCATTER
      INCLUDE "fpvm3.h"
C Initialize PE0 as the root PE
      INTEGER, PARAMETER :: ROOT_PE = 0
```

```
      PARAMETER(IY = 128, IX = IY/4) !Assume 4 PEs
      DIMENSION SMALL_X(IX)
      DIMENSION BIG_X(IY)             !Assume 4 PEs
      PARAMETER(MTAGBCAST = 10, MTAGSCAT = 11)

      CALL PVMFMYTID(MYTID)
      CALL PVMFGETPE(MYTID, MYPE)

      IF (MYPE.EQ.IROOT_PE)THEN
C       Initialize BIG_X
        I = 1
        DO  J = 1,4             ! Assume 4 PEs
            DO INDX = 1,IX
                BIG_X(I) = 1. * J
                I = I+1
            ENDDO
        ENDDO
      ENDIF
      CALL PVMFSCATTER(SMALL_X, BIG_X, IX, REAL8, MTAGSCAT, PVMALL,
     x   IROOT_PE, ISCATTER)
      PRINT *,'FOR PE ',MYPE,' SMALL_X IS ',SMALL_X
      END
```

SHMEM [3]

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 147), 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 44). 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 48).
- Avoiding performance pitfalls when passing 32-bit data rather than 64-bit data (see Section 3.3, page 55).
- 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 58).
- Gathering and scattering data and reordering it in the process (see Section 3.5, page 62).
- *Broadcasting* (glossary, page 142) data from one PE to all PEs (see Section 3.6, page 66).
- Merging arrays from each PE into a single array on all PEs (see Section 3.7, page 68).
- Executing an *atomic* memory operation (glossary, page 141) to read and update a remote value in a single process (see Section 3.8, page 70).

- Using *reduction* (glossary, page 149) routines to execute a variety of operations across multiple PEs (see Section 3.9, page 71).

3.1 Using SHMEM_GET64 and SHMEM_PUT64 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_PUT64 and SHMEM_GET64 routines 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.

For both small and large transfers, the SHMEM_PUT64 routine, which moves data from the local PE to a remote PE, and the SHMEM_GET64 routine, which moves data from a remote PE to the local PE, are virtually the same in terms of performance. At times, SHMEM_PUT64 may be the better choice because it lets the calling PE perform other work while the data is in the network. Because SHMEM_PUT64 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. SHMEM_GET64 forces the calling PE to wait until the data is in *local memory* (glossary, page 146), meaning that no early work can be done.

Passing data in large chunks is always faster than passing it in small chunks because it saves subroutine overhead. Whenever possible, put all of your data (such as an array) into a single SHMEM_PUT64 or SHMEM_GET64 call rather than calling the routine iteratively.

In the following example, eight 64-bit words are transferred from PE 1 to PE 0 by using SHMEM_PUT64. PE numbering always begins with 0.

Example 8: Example of a SHMEM_PUT64 transfer

```
 1.        INCLUDE "mpp/shmem.fh"
 2.        INTEGER SOURCE(8), DEST(8)
 3.        INTRINSIC MY_PE
 4.        SAVE DEST
 5. C On the sending PE
 6.        IF (MY_PE() .EQ. 1) THEN
 7.          DO I = 1,8
 8.            SOURCE(I) = I
```

```
 9.          ENDDO
10. C PE 1 sends the data to PE 0.
11.          CALL SHMEM_PUT64(DEST, SOURCE, 8, 0)
12.        ENDIF
13.
14. C Make sure the transfer is complete.
15.        CALL SHMEM_BARRIER_ALL()
16.
17. C On the receiving PE
18.        IF (MY_PE() .EQ. 0) THEN
19.          PRINT *, 'DEST ON PE 0: ', DEST
20.        ENDIF
21.
22.        END
```

See the following figure for an illustration of the transfer.

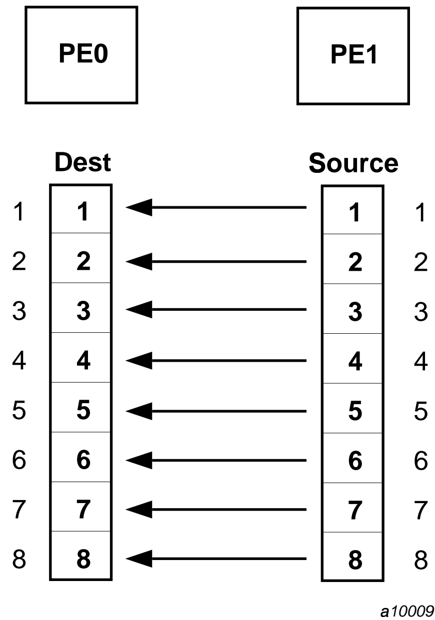

Figure 12. SHMEM_PUT64 data transfer

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 having the number of PEs defined as a power of 2.

For information on optimizing existing PVM and MPI programs using SHMEM_GET64 and SHMEM_PUT64, see Section 3.2, page 48. For a complete description of the MPP-specific statements in the preceding example, continue on with this section.

In the SHMEM_PUT64 example (see Example 8, page 44), line 1 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.

```
1.       INCLUDE "mpp/shmem.fh"
```

Line 3 declares the intrinsic function MY_PE, which returns the number of the PE on which it executes. Two versions of MY_PE function exist on the CRAY T3E system, one in the external library and one as an intrinsic. The intrinsic version is marginally faster than external library version, but the external library version is now, and will be in the future, on more Cray Research and Silicon Graphics supercomputer systems. Declaring MY_PE as an intrinsic is not necessary, but it will ensure you of getting the slightly faster version of the routine.

```
3.       INTRINSIC MY_PE
```

The defined constant N$PES, which returns the number of PEs in a program, is also slightly faster than the more portable external library routine NUM_PES. Like MY_PE, both versions return the same information.

The intrinsic function MY_PE and the constant N$PES are also faster than using equivalent message-passing routines, such as SHMEM_MY_PE and SHMEM_N_PES. Both methods return the same information and are available on Cray PVP systems as well as Cray MPP systems.

Line 4 ensures that the remote array (DEST) is *symmetric* (glossary, page 150), which means that it has the same address on remote PEs as on the local PE.

```
4.        SAVE DEST
```

You can make sure DEST is symmetric in any of the following ways. (None of these methods is significantly faster than the others.)

- Name it in a SAVE statement, as in the example.
- Include it in a common block.
- If it is an array, allocate it by using shpalloc(3).
- If it is a stack variable, declare it by using the !CDIR$ SYMMETRIC directive.

In line 6, the MY_PE function is called. The function returns the number of the calling PE, meaning only PE 1 will execute the THEN clause. As a result, the array SOURCE is initialized only on PE 1.

```
5. C On the sending PE
6.        IF (MY_PE() .EQ. 1) THEN
7.          DO I = 1,8
8.            SOURCE(I) = I
9.          ENDDO
```

In line 11, PE 1 executes the SHMEM_PUT64 routine call that sends the data. SHMEM_PUT64 is the variant of SHMEM_PUT64 that transfers 64–bit (KIND=8) data. It sends eight array elements from its SOURCE array to the DEST array on PE 0.

```
11.         SHMEM_PUT64(DEST, SOURCE, 8, 0)
12.       ENDIF
```

Line 15 is a *barrier* (glossary, page 141), which provides a *synchronization* point (glossary, page 151). 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 could print the DEST array before receiving the data. Calling SHMEM_BARRIER_ALL is as fast as calling the BARRIER routine directly.

```
15.      CALL SHMEM_BARRIER_ALL()
```

Line 18 selects PE 0, which is passively receiving the data. Because 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, PE 0 prints DEST, and the program exits.

```
18.      IF (MY_PE() .EQ. 0) THEN
19.        PRINT *, 'DEST ON PE 0: ', DEST
20.      ENDIF
```

3.2 Optimizing Existing MPI and PVM Programs by Using SHMEM

The following sections show two methods of optimizing a PVM 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 PVM 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 PVM 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 74.

Example 9: PVM version of the ring program

```
1.          PROGRAM RING_SUM_1
2.          INCLUDE 'fpvm3.h'
3.          INTEGER ME, NPES, NEXT, PREV, ISTAT
4.    C
5.    C Get PE info
6.          CALL PVMFMYTID(MYTID)
7.          CALL PVMFGSIZE(PVMALL, NPES)
8.          CALL PVMFGETPE(MYTID, ME)
9.    C
10.   C Define the ring
11.         NEXT = ME + 1
12.         IF (NEXT .GE. NPES) NEXT = NEXT - NPES
13.         PREV = ME - 1
14.         IF (PREV .LT. 0) PREV = PREV + NPES
15.   C
```

```
16.  C Initialize data
17.  C And begin loop
18.  C
19.         I_SEND  = ME
20.         I_TOTAL = ME
21.         DO I=2, NPES
22.  C
23.  C Send data to next PE
24.           CALL PVMFINITSEND(PvmDataRaw, ISTAT)
25.           CALL PVMFPACK(INTEGER4, I_SEND, 1, 1, ISTAT)
26.           CALL PVMFSEND(NEXT, I, ISTAT)
27.  C
28.  C Receive data from previous PE
29.           CALL PVMFRECV(PREV, I, ISTAT)
30.           CALL PVMFUNPACK(INTEGER4, I_RECV, 1, 1, ISTAT)
31.  C
32.  C Perform work
33.           I_TOTAL = I_TOTAL + I_RECV
34.           I_SEND  = I_RECV
35.         ENDDO
36.
37.         WRITE(*,*) ' PE = ', ME, ' Result = ', I_TOTAL,
38.       $            ' Expect = ', (NPES-1)*NPES*.5
39.         END
```

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 `I_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_GET64 version of the same program, see Section 3.2.1, page 51; otherwise, see the remainder of this section for a more detailed description of the PVM version.

Line 2 references the PVM INCLUDE file. Line 3 declares some necessary variables.

```
2.          INCLUDE 'fpvm3.h'
3.          INTEGER ME, NPES, NEXT, PREV, ISTAT
```

The next few statements define a PE's neighbors. The variables NEXT and PREV, defined in lines 11 through 14, specify which of its neighbors a PE will be passing to and which it will be receiving from, respectively. Lines 12 and 14 define neighbors for the PEs on the end, namely PE and PE 7 in an 8-PE configuration. Line 12 causes PE 7 to pass to PE 0, and line 14 causes PE to receive from PE 7.

```
10. C Define the ring
11.         NEXT = ME + 1
12.         IF (NEXT .GE. NPES) NEXT = NEXT - NPES
13.         PREV = ME - 1
14.         IF (PREV .LT. 0) PREV = PREV + NPES
```

The values for NEXT and PREV in each PE are as illustrated in the following figure.

PE0	PE1	PE2	PE3	PE4	PE5	PE6	PE7
NEXT=1	NEXT=2	NEXT=3	NEXT=4	NEXT=5	NEXT=6	NEXT=7	NEXT=0
PREV=7	PREV=0	PREV=1	PREV=2	PREV=3	PREV=4	PREV=5	PREV=6

a10003

Figure 13. Identification of neighbors in the ring program.

Line 19 initializes the variable that each PE will pass on to NEXT, and line 20 initializes the variable that will hold the running total in each PE. Both variables at first contain the number of the respective PE.

```
19.         I_SEND  = ME
20.         I_TOTAL = ME
```

Next comes the DO loop, within which the PVM 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 24, the PVMFINITSEND routine initializes a *buffer* (glossary, page 142) for each PE to be used to send the data. In line 25, the PVMFPACK routine puts the data (I_SEND) into the buffer, and in line 26, PVMFSEND sends the buffer on to the next PE. Each PVM routine is described in more detail on its man page.

```
21.         DO I=2, NPES
22. C
23. C Send data to next PE
24.           CALL PVMFINITSEND(PmvDataRaw, ISTAT)
25.           CALL PVMFPACK(REAL8, I_SEND, 1, 1, ISTAT)
26.           CALL PVMFSEND(NEXT, I, ISTAT)
```

Lines 28 and 29 receive the data from the PE that is defined in PREV. The PVMFRECV routine receives the buffer, and, in line 30, the PVMFUNPACK routine takes the data from the buffer and puts it into the variable I_RECV.

```
28.  C Receive data from previous PE
29.           CALL PVMFRECV(PREV, I, ISTAT)
30.           CALL PVMFUNPACK(REAL8, I_RECV, 1, 1, ISTAT)
```

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

```
32. C Perform work
33.           I_TOTAL = I_TOTAL + I_RECV
34.           I_SEND  = I_RECV
35.         ENDDO
```

3.2.1 Optimizing by Using SHMEM_GET64

To optimize the PVM version of the ring program shown in Example 9, page 48, you can replace the PVM 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 PVM send and receive routines. The optimization described in this section and shown in the following example uses a form of the SHMEM_GET64 routine; in Example 11, page 54, PVM is replaced by SHMEM_PUT64.

Example 10: `SHMEM_GET64` version of the ring program

```
1.          PROGRAM RING_SUM_2
2.          INCLUDE 'mpp/shmem.fh'
3.          INTEGER ME, NPES, NEXT, PREV, ISTAT
4.          COMMON /D_DATA/ D_SEND
5.          INTRINSIC MY_PE, N$PES
6.   C
7.          ME = MY_PE()
8.          NPES = N$PES()
9.   C
10. C Define the ring
11.         NEXT = ME + 1
12.         IF (NEXT .GE. NPES) NEXT = NEXT - NPES
13.         PREV = ME - 1
14.         IF (PREV .LT. 0) PREV = PREV + NPES
15. C
16. C Initialize data
17.         D_ME = ME
18. C
19.         D_SEND  = D_ME
20.         D_TOTAL = D_ME
21.         DO I=2, NPES
22. C
23. C Synchronize - Make sure data is ready on other PE
24. C
25.           CALL SHMEM_BARRIER_ALL()
26. C Get data from previous PE
27.           CALL SHMEM_GET64(D_RECV, D_SEND, 1, PREV)
28.           CALL SHMEM_BARRIER_ALL()
29. C Perform work
30.           D_TOTAL = D_TOTAL + D_RECV
31.           D_SEND  = D_RECV
32.         ENDDO
33.
34.         WRITE(*,*) ' PE = ', ME, ' RESULT = ', D_TOTAL,
35.       $            ' EXPECT = ', (NPES-1.)*NPES*.5
36.         END
```

Line 2 references the SHMEM `INCLUDE` file. Line 4 puts the value `D_SEND` into a common block to ensure that the remote and local arrays are *symmetric* (glossary, page 150). The addresses must be the same for all of the PEs involved

in the program, and using named common guarantees that. Ensuring that the arrays are symmetric is not a requirement for PVM.

```
2.          INCLUDE 'mpp/shmem.fh'

4.          COMMON /D_DATA/ D_SEND
```

Lines 7 and 8 use SHMEM functions that get the same information (the number of the calling PE and the number of PEs involved in the job) as the PVM calls in lines 6 through 8 of the PVM version of the program (see Example 9, page 48). As mentioned before, the intrinsic MY_PE and the constant N$PES are slightly faster than the SHMEM equivalents and the external library versions, MY_PE and NUM_PES. For more information, see page 46.

```
7.          ME = MY_PE()
8.          NPES = N$PES()
```

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 25 and 28, which set barriers, are necessary when using the SHMEM_GET64 routine. Synchronization is implicit in the PVM version because of the PVM mode of operation: each send is matched by a receive. You must provide your own synchronization when using SHMEM. The SHMEM_BARRIER_ALL routine takes advantage of the fast hardware barrier mechanism, making these calls relatively inexpensive. The implicit synchronization in PVM can be replaced by this faster synchronization method when you are converting between PVM and SHMEM.

```
25.           CALL SHMEM_BARRIER_ALL
26. C Get data from previous pe
27.           CALL SHMEM_GET64(D_RECV, D_SEND, 1, PREV)
28.        CALL SHMEM_BARRIER_ALL
```

Other performance improvements that you will see when converting to SHMEM data passing are as follows. They apply whether you are using SHMEM_PUT64 or SHMEM_GET64 routines.

- 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_PUT64`

The `SHMEM_PUT64` routines can deliver the answers from the ring program as fast as the `SHMEM_GET64` routines. The following example shows the `SHMEM_PUT64` version of the ring program.

Example 11: `SHMEM_PUT64` version of the ring program

```
1.          PROGRAM RING_SUM_3
2.          INCLUDE 'mpp/shmem.fh'
3.          INTEGER ME, NPES, NEXT, PREV, ISTAT
4.          COMMON /D_DATA/ D_RECV
5.   C
6.   C Get PE info
7.          ME = MY_PE()
8.          NPES = N$PES()
9.   C
10. C Define the ring
11.         NEXT = ME + 1
12.         IF (NEXT .GE. NPES) NEXT = NEXT - NPES
13.         PREV = ME - 1
14.         IF (PREV .LT. 0) PREV = PREV + NPES
15. C
16. C Initialize data
17.         D_ME = ME
18. C
19.         D_SEND  = D_ME
20.         D_TOTAL = D_ME
21.         DO I=2, NPES
22. C
23. C Send data to next PE
24.           CALL SHMEM_PUT64(D_RECV, D_SEND, 1, NEXT)
25. C
26. C Synchronize - Ensure all have valid data in D_RECV
```

```
27.           CALL SHMEM_BARRIER_ALL
28. C
29. C Perform work
30.           D_TOTAL = D_TOTAL + D_RECV
31.           D_SEND  = D_RECV
32. C
33. C Synchronize - Ensure all have accessed D_RECV
34.           CALL SHMEM_BARRIER_ALL
35.         ENDDO
36.
37.         WRITE(*,*) ' PE = ', ME, ' RESULT = ', D_TOTAL,
38.       $            ' EXPECT = ', (NPES-1.)*NPES*.5
39.         END
```

The first half of the program is the same as the SHMEM_GET64 version (see Example 10, page 52), except that the remote variable declared in the common block is now D_RECV, which is the target of the data transfer.

In line 24, each PE passes the data to the next PE. Lines 27 and 34 both contain synchronization routines, and both are needed to guarantee that the transfers are complete.

```
24.           CALL SHMEM_PUT64(D_RECV, D_SEND, 1, NEXT)
25. C
26. C Synchronize - Make sure everyone has data
27.           CALL SHMEM_BARRIER_ALL
28. C
29. C Perform work
30.           D_TOTAL = D_TOTAL + D_RECV
31.           D_SEND  = D_RECV
32. C
33. C Synchronize - Ensure everyone is ready to continue
34.           CALL SHMEM_BARRIER_ALL
```

3.3 Passing 32-bit Data

Two variants of SHMEM_GET64 and SHMEM_PUT64, 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, 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:

```
CALL SHMEM_PUT32(DEST(1), SOURCE(6), NLONG, PE)
```

Instead, cache-align the two arrays and begin the transfer either on two even-numbered or two odd-numbered array elements. The CF90 compiler directive CACHE_ALIGN serves the purpose of aligning cache.

```
!DIR$ CACHE_ALIGN DEST, SOURCE
      SHMEM_PUT32(DEST, SOURCE, NLONG, PE)
```

The following 32-bit version of the ring program uses the SHMEM_PUT32 routine:

Example 12: 32-bit version of ring program

```
1.       PROGRAM RING_NSUM_3_4
2. C
3. C Summing around a ring
4. C SHMEM_PUT64 version
5. C
6.         INCLUDE 'mpp/shmem.fh'
7.         INTEGER ME, NPES, NEXT, PREV
8.         INTEGER ISTAT
9.  C Pass arrays of size N
10.        PARAMETER(N=100000)
11.        REAL(KIND=4) D_SEND(N), D_RECV(N), D_TOTAL(N)
12.        COMMON /D_DATA/ D_RECV
13. C
14. C Get PE info
15.        ME = MY_PE()
16.        NPES = N$PES()
17. C
18. C Define the ring
19. C
20.        NEXT = ME + 1
21.        IF (NEXT .GE. NPES) NEXT = NEXT - NPES
22.        PREV = ME - 1
23.        IF (PREV .LT. 0) PREV = PREV + NPES
24. C
25. C Initialize data
26. C
```

```
27.        D_ME = ME
28. C
29.        D_SEND(1:N)  = D_ME
30.        D_TOTAL(1:N) = D_ME
31.        DO I=2, NPES
32. C
33. C Send data to next PE
34.          CALL SHMEM_PUT32(D_RECV, D_SEND, N, NEXT)
35. C
36. C Synchronize -
37. C   Wait for data to arrive at other PE (implicit SHMEM_QUIET)
38. C   and make sure everyone has data
39.          CALL SHMEM_BARRIER_ALL
40. C
41. C Flush cache if on a PVP system
42.          CALL SHMEM_UDCFLUSH()
43. C
44. C Perform work
45.          CALL WORK(N, D_TOTAL, D_SEND, D_RECV)
46. C
47. C Synchronize - Make sure everyone is ready to continue
48.          CALL SHMEM_BARRIER_ALL
49.        ENDDO
50.
51.        WRITE(*,*) ' PE = ', ME, ' Result = ', D_TOTAL(1),
52.      $            ' Expect = ', (NPES-1.)*NPES*.5
53.        END
54.
55.        SUBROUTINE WORK(N, D_TOTAL, D_SEND, D_RECV)
56.        REAL(KIND=4) D_SEND(N), D_RECV(N), D_TOTAL(N)
57.        D_TOTAL(1:N) = D_TOTAL(1:N) + D_RECV(1:N)
58.        D_SEND(1:N)  = D_RECV(1:N)
59.        RETURN
60.        END
```

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_REAL_IPUT and SHMEM_REAL_IGET routines, 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 routine calls. Speed and the ability to reorder the elements separate the SHMEM versions of the strided copy operation from their PVM equivalent. (For information on reordering data, see Section 3.5, page 62.) The following examples take elements from an array on PE and move them to an array on PE 1. Both examples, one using SHMEM_REAL_IGET and the other SHMEM_REAL_IPUT, use strides other than 1.

Example 13: Passing strided data using SHMEM_REAL_IGET

```
1.       PROGRAM STRIDED
2. C
3. C SHMEM_REAL_IGET version
4. C  The sending array is accessed with stride 2
5. C  The receiving array is accessed with stride 3
6. C
7.         INCLUDE 'mpp/shmem.fh'
8.         INTEGER ME
9.         INTEGER ISTAT, SENDER, RECEIVER
10.        PARAMETER(N=100)
11.        PARAMETER(SENDER = 0)
12.        PARAMETER(RECEIVER = 1)
13.        REAL D_SEND(2*N), D_RECV(3*N)
14.        INTRINSIC MY_PE
15.        COMMON /D_DATA/ D_SEND
16. C
17. C Get PE info
18.        ME = MY_PE()
19. C
20. C Initialize data
21.        DO I=1,2*N
22.        D_SEND(I) = I + ME
23.        ENDDO
24.        D_RECV = 0.0
25. C
```

```
26.        IF (ME .EQ. SENDER) THEN
27. C
28. C Synchronize - Make sure data is ready
29.          CALL SHMEM_BARRIER_ALL
30. C
31. C Note: Sender does nothing but synchronize,
32.        ELSE IF (ME .EQ. RECEIVER) THEN
33. C
34. C Synchronize - Make sure data is ready on other PE
35.          CALL SHMEM_BARRIER_ALL
36. C
37. C Get data
38.          CALL SHMEM_REAL_IGET(D_RECV, D_SEND, 3, 2, N, SENDER)
39. C
40. C Print results
41.          WRITE(*,*) 'Receiver=',ME,' d_recv=', D_RECV(1),
42.      1   D_RECV(2),D_RECV(3),D_RECV(4),D_RECV(5),D_RECV(6),D_RECV(7)
43.        ENDIF
44.        END
```

Example 14: Passing strided data using `SHMEM_REAL_IPUT`

```
1.        PROGRAM STRIDED_2
2. C
3. C SHMEM_REAL_IPUT version
4. C  The sending array is accessed with stride 2
5. C  The receiving array is accessed with stride 3
6. C
7.          INCLUDE 'mpp/shmem.fh'
8.          INTEGER ME
9.          INTEGER ISTAT, SENDER, RECEIVER
10.         PARAMETER(N=100)
11.         PARAMETER(SENDER = 0)
12.         PARAMETER(RECEIVER = 1)
13.         REAL D_SEND(2*N), D_RECV(3*N)
14.         INTRINSIC MY_PE
15.         COMMON /D_DATA/ D_RECV
16. C
17. C Get PE info
18.         ME = MY_PE()
19. C
20. C Initialize data
21.         DO I=1,2*N
```

```
22.        D_SEND(I) = I + ME
23.        ENDDO
24.        D_RECV = 0.0
25. C
26. C Synchronize - Make sure all arrays are initialized
27.        CALL SHMEM_BARRIER_ALL
28.        IF (ME .EQ. SENDER) THEN
29. C
30. C Send data
31.          CALL SHMEM_REAL_IPUT(D_RECV, D_SEND, 3, 2, N, RECEIVER)
32. C
33. C Synchronize - Make sure data has arrived
34.          CALL SHMEM_BARRIER_ALL
35.        ELSE IF (ME .EQ. RECEIVER) THEN
36. C
37. C Synchronize - Make sure data has arrived
38.          CALL SHMEM_BARRIER_ALL
39. C
40. C Cache update not required on
41. C CRAY T3E system (no-op)
42.          CALL SHMEM_UDCFLUSH()
43. C
44. C Print results
45.          WRITE(*,*) 'Receiver=',ME,'D_RECV=',D_RECV(1),D_RECV(2)
46.      1   ,D_RECV(3),D_RECV(4),D_RECV(5),D_RECV(6),D_RECV(7)
47.        ENDIF
48.        END
```

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

The SHMEM_REAL_IGET and SHMEM_REAL_IPUT routines are faster than the SHMEM_IXGET and SHMEM_IXPUT routines, 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:

```
C Send data
         CALL PVMFINITSEND(PvmDataRaw, ISTAT)
         CALL PVMFPACK(REAL8, D_SEND, N, 2, ISTAT)
         CALL PVMFSEND(RECEIVER, MTAG, ISTAT)
C
```

```
C Receive data
         CALL PVMFRECV(SENDER, MTAG, ISTAT)
         CALL PVMFUNPACK(REAL8, D_RECV, N, 3, ISTAT)
```

For a description of how to efficiently collect data from all PEs and distribute it to all PEs, see Section 3.5, page 62. Or continue on with the remainder of this section for a brief description of the two SHMEM strided data programs.

In line 15 of the programs in Example 13, page 58, and Example 14, page 59, both routines name the remote arrays in a `COMMON` statement, which ensures that the remote address will be the same as the local address. This is a requirement for these routines.

`SHMEM_REAL_IGET` version:

```
15.       COMMON /D_DATA/ D_SEND
```

`SHMEM_REAL_IPUT` version:

```
15.       COMMON /D_DATA/ D_RECV
```

The data is transferred from within `IF` statements, beginning on line 26 in the `SHMEM_REAL_IGET` version and on line 28 in the `SHMEM_REAL_IPUT` version.

The structures of the `IF` statements are identical in that the sender (PE 0) executes the `IF` clause and the receiver (PE 1) executes the `ELSE IF` clause, but the placement of the data transfer routines differs. The `SHMEM_REAL_IGET` routine, executing on PE 1, retrieves the data from PE 0. The `SHMEM_REAL_IPUT` routine, executing on PE 0, copies the data to PE 1.

`SHMEM_REAL_IGET` version:

```
38.          CALL SHMEM_REAL_IGET(D_RECV, D_SEND, 3, 2, N, SENDER)
```

`SHMEM_REAL_IPUT` version:

```
31.          CALL SHMEM_REAL_IPUT(D_RECV, D_SEND, 3, 2, N, RECEIVER)
```

As the third and fourth arguments of the routine calls specify, both routines 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 routines are the same. See the following figure for an illustration of the transfers:

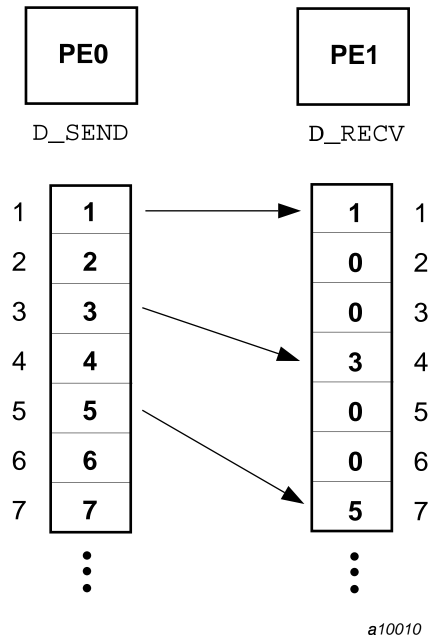

Figure 14. SHMEM_REAL_IGET and SHMEM_REAL_IPUT transfers

As mentioned earlier (see page 54), the following SHMEM_UDCFLUSH call is ignored on CRAY T3E systems. It is left in the program for compatibility reasons.

```
41. C CRAY T3E system (no-op)
42.          CALL 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 routines 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 routines 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.

Example 15: SHMEM_IXPUT version of a reordered scatter

```
1.       PROGRAM SCATTER_2A
2. C
3. C  SHMEM_IXPUT version
4. C  Use the index array for the receiving array (scatter)
5. C
6.       INCLUDE 'mpp/shmem.fh'
7.       INTEGER ME
8.       INTEGER ISTAT, SENDER, RECEIVER
9.       PARAMETER(NN=100)
10.      PARAMETER(N=10)
11.      PARAMETER(SENDER = 0)
12.      PARAMETER(RECEIVER = 1)
13.      REAL D_SEND(N), D_RECV(NN)
14.      INTRINSIC MY_PE
15.      COMMON /D_DATA/ D_RECV
16.      INTEGER INDEX(N)
17.      DATA INDEX / 99, 19, 28, 91, 82, 37, 73, 46, 64, 55 /
18. C
19. C Get PE info
20.      ME = MY_PE()
21. C
22. C Initialize data
23.      DO I=1,N
24.        D_SEND(I) = I + ME
25.      ENDDO
26.      D_RECV(1:NN) = 0.0
27. C
28. C Synchronize - Make sure data arrays are initialized
29.      CALL SHMEM_BARRIER_ALL
30.      IF (ME .EQ. SENDER) THEN
31. C
32. C Send data
33.        CALL SHMEM_IXPUT(D_RECV, D_SEND, INDEX, N, RECEIVER)
34. C
35. C Synchronize - Make sure data has arrived
36.        CALL SHMEM_BARRIER_ALL
```

```
37.      ELSE IF (ME .EQ. RECEIVER) THEN
38. C
39. C Synchronize - Make sure data has arrived
40.        CALL SHMEM_BARRIER_ALL
41. C
42. C Make sure cache is up to date
43.        CALL SHMEM_UDCFLUSH()
44. C
45. C Print results
46.        WRITE(*,*) 'Receiver=',ME,' D_RECV=', D_RECV(100), D_RECV(20)
47.   1               ,D_RECV(29),D_RECV(92),D_RECV(83),D_RECV(38)
48.      ENDIF
49.      END
```

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

```
35.          DO I=1,N
36.            CALL PVMFUNPACK(REAL8, D_RECV(1+INDEX(I)), 1, 1, ISTAT)
37.          ENDDO
```

In the following lines of the SHMEM program, the INDEX array (defined in line 17) is referenced in the call to the SHMEM_IXPUT routine, which reorders the array itself.

```
17.      DATA INDEX / 99, 19, 28, 91, 82, 37, 73, 46, 64, 55 /

33.          CALL SHMEM_IXPUT(D_RECV, D_SEND, INDEX, N, RECEIVER)
```

Because the INDEX array is zero-based, the respective data element will wind up in the position specified by the INDEX array plus one. For instance, the first value of the D_SEND array will transfer to D_RECV(100), not D_RECV(99). The output from the SHMEM version on a 2-PE configuration is as follows:

```
 Receiver=1 D_RECV=1.,  2.,  3.,  4.,  5.,  6.
```

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

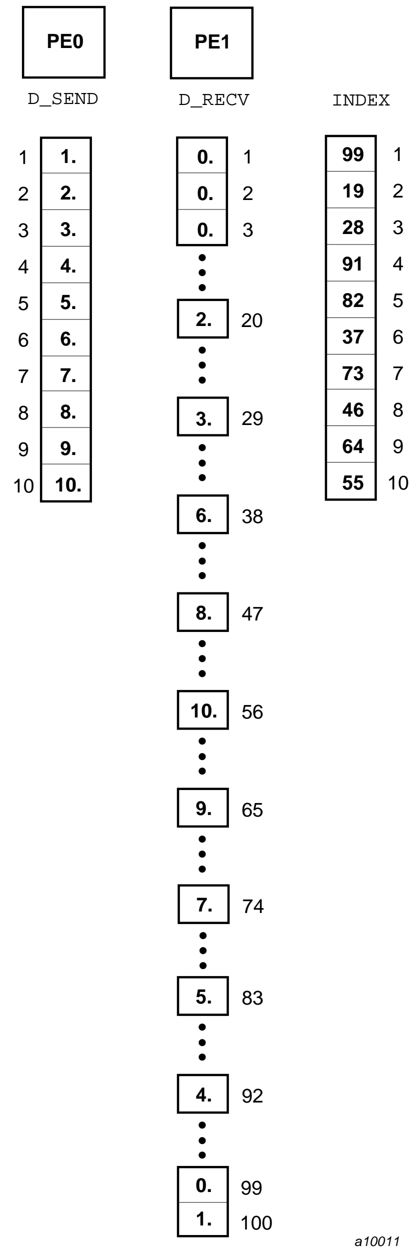

Figure 15. Reordering elements during a scatter operation

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:

```
CALL SHMEM_IXGET(D_RECV, D_SEND, INDEX, N, SENDER)
```

3.6 Broadcasting Data to Multiple PEs

The SHMEM_BROADCAST(3) routine copies an array on a single PE to a target array on each of the other PEs. It uses the *fan-out* (glossary, page 145) method of copying the data, which makes it the most efficient routine 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 33. SHMEM_BROADCAST gives you significant performance advantages when compared to the PVMFBCAST(3) routine.

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. Remember that the sixth argument to SHMEM_BROADCAST, the stride, is specified in base 2. If you specify 0, the stride will be 1. Also, the fourth argument, the sending PE, is a number relative to the active set; if you have defined PE 4 through PE 7 as an active set, a 0 will select PE 4.

Example 16: One-to-all broadcasting

```
1.        INCLUDE 'mpp/shmem.fh'
2.        INTEGER DEST(8), SOURCE(8)
3. C Declare the PSYNC array
4.        INTEGER PSYNC(SHMEM_BCAST_SYNC_SIZE)
5.        INTRINSIC MY_PE
6.
7. C Use the DATA statement to initialize PSYNC
8.        DATA PSYNC /SHMEM_BCAST_SYNC_SIZE*SHMEM_SYNC_VALUE/

9.        SAVE DEST, SOURCE
10.
11. C Initialize the SOURCE array
12.       IF (MY_PE().EQ.0) THEN
13.         DO I = 1, 8
14,           SOURCE(I) = I * I
15,         ENDDO
```

```
16.         PRINT *, 'THE ORIGINAL ARRAY VALUES ARE ', SOURCE
17.       ENDIF
18.
19. C Broadcast an 8-element array from PE 0
20.       CALL SHMEM_BROADCAST(DEST, SOURCE, 8, 0, 0, 0, 4, PSYNC)
21. C Don't forget the PSYNC argument
22.
23. C Print the DEST array on all PEs
24.       PRINT *, 'PE ', MY_PE(), ' HAS ', DEST
25.
26.       END
```

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.

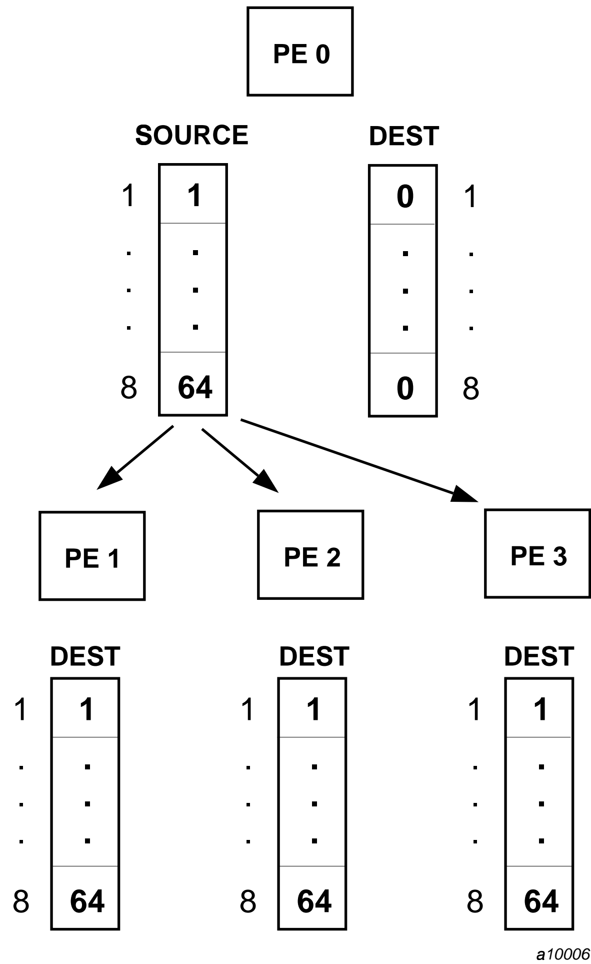

Figure 16. The broadcast operation

3.7 Merging Arrays

The SHMEM_FCOLLECT(3) routine 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: SHMEM_FCOLLECT

```
1.         PROGRAM SHARE_ARRAY
2.         INCLUDE 'mpp/shmem.fh'
3.         INTEGER MYVALS(N$PES), ALLVALS(16)
4.         INTRINSIC MY_PE
5.         INTEGER PSYNC(SHMEM_COLLECT_SYNC_SIZE)
6.         DATA PSYNC /SHMEM_COLLECT_SYNC_SIZE*SHMEM_SYNC_VALUE/
7.
8. C Assume 4 PEs
9.         NPES = 4
10.
11. C The values to be passed will be based on PEs numbers
12.        N = NPES * MY_PE()
13.        DO I = 1, NPES
14.           N = N + 1
15.           MYVALS(I) = N
16.        END DO
17.
18. C Wait until all PEs are initialized
19.        CALL SHMEM_BARRIER_ALL
20.
21.        CALL SHMEM_FCOLLECT(ALLVALS, MYVALS, 4, 0, 0, NPES, PSYNC)
22.
23.        PRINT *, 'PE ', MY_PE(), ' HAS ', ALLVALS
24.        END
```

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 PVMFGATHER followed by a call to PVMFBCAST would produce the same result as one call to SHMEM_FCOLLECT, but the call to SHMEM_FCOLLECT is faster.

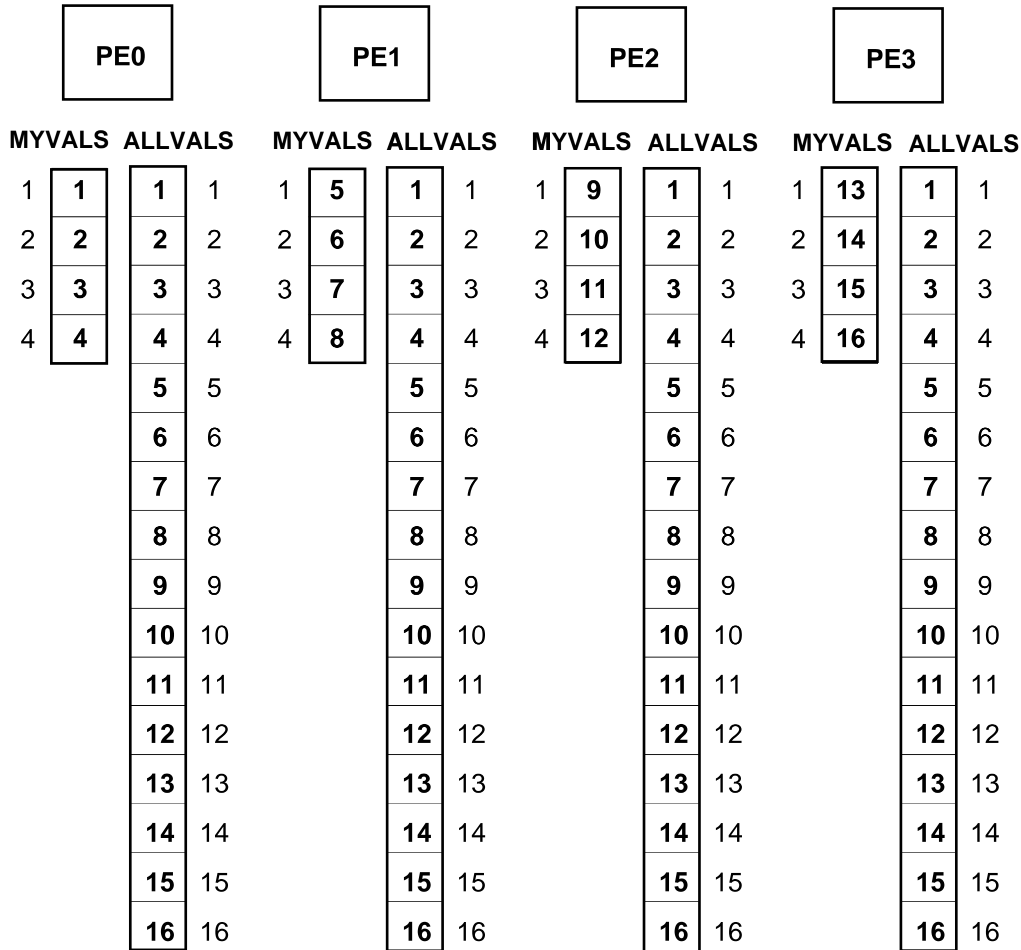

Figure 17. An example of SHMEM_FCOLLECT

3.8 Reading and Updating in One Operation

The SHMEM_SWAP(3), SHMEM_INT4_FINC(3), and SHMEM_INT4_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_INT4_FINC function reads and toggles a value. One possible use of the

SHMEM_INT4_FINC function is to implement your own locks, as shown in the following example.

Example 18: Remote fetch and increment

```
1.  C
2.  C       These LOCK/UNLOCK routines implement a take-a-number lock
3.  C       using an array of 2, assumed to be initialized to 0.
4.  C
5.            SUBROUTINE LOCK(LCK)
6.            INTEGER(KIND=4) LCK(2), NOW_SERVING
7.            INTEGER(KIND=4) SHMEM_INT4_FINC
8.
9.  C       Take the next number.
10.
11.         MYNO = SHMEM_INT4_FINC(LCK(1), 0)
12.
13. C       Wait until my number comes up.
14.
15.         CALL SHMEM_GET4(NOW_SERVING, LCK(2), 1, 0)
16.         DO WHILE (NOW_SERVING .NE. MYNO)
17.                 CALL SHMEM_GET4(NOW_SERVING, LCK(2), 1, 0)
18.         ENDDO
19.         END
```

Using the SHMEM_BARRIER_ALL routine is faster than writing your own synchronization routine 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 routines

The *reduction* (glossary, page 149) routines combine array elements from each active PE to yield an array of results, which are distributed to all PEs. For example, one routine 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 routine completes. In its simplest form, a reduction routine is a *collective* (glossary, page 143) operation that involves all PEs.

The SHMEM reduction routines are usually faster than those of PVM and MPI because of the difference in overhead, but they are only marginally faster. They

are faster than other means of executing operations across PEs (such as SHMEM_PUT64, SHMEM_GET64, 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 72)
- 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 74)
- 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

```
1.         PROGRAM MINVAL
2.         INCLUDE 'mpp/shmem.fh'
3.         INTRINSIC MY_PE, RANF, RANSET
4.
5.         INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE), NR, SEED
6.         DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
7.
8. C Make the number of results a constant
9.         PARAMETER(NR=4)
10.        REAL FOO(NR), FOOMIN(NR), PWRK(SHMEM_REDUCE_MIN_WRKDATA_SIZE)
11.        COMMON /COM/ FOO, FOOMIN, PWRK
12.
13. C Put some values into the FOO arrays
14.        DO I = 1, 4
15.         SEED = RANSET((MY_PE()+I) * (I*I))
16.         FOO(I) = RANF()
17.        END DO
18.
19. C Print the preliminary numbers on each PE
20.        PRINT 100, MY_PE(), FOO
21. 100  FORMAT('At first, PE ', I2, ' has ', 4F8.5)
```

```
22.
23.       CALL SHMEM_BARRIER_ALL
24.       PRINT *
25.
26.       CALL SHMEM_REAL8_MIN_TO_ALL(FOOMIN,FOO,NR,0,0,4,PWRK,PSYNC)
27.
28. C All the values should be the same
29.       PRINT 200, MY_PE(), FOOMIN
30. 200  FORMAT('Result on PE ', I2, ' :', 4F8.5)
31.
32.       END
```

The output from the program is as follows:

```
At first, PE  0 has  0.15804 0.42240 0.26721 0.27292
At first, PE  3 has  0.79022 0.31894 0.69247 0.85908
At first, PE  1 has  0.47413 0.05458 0.84766 0.80164
At first, PE  2 has  0.47413 0.68676 0.11202 0.33036

Result on PE  0 : 0.15804 0.05458 0.11202 0.27292
Result on PE  1 : 0.15804 0.05458 0.11202 0.27292
Result on PE  2 : 0.15804 0.05458 0.11202 0.27292
Result on PE  3 : 0.15804 0.05458 0.11202 0.27292
```

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

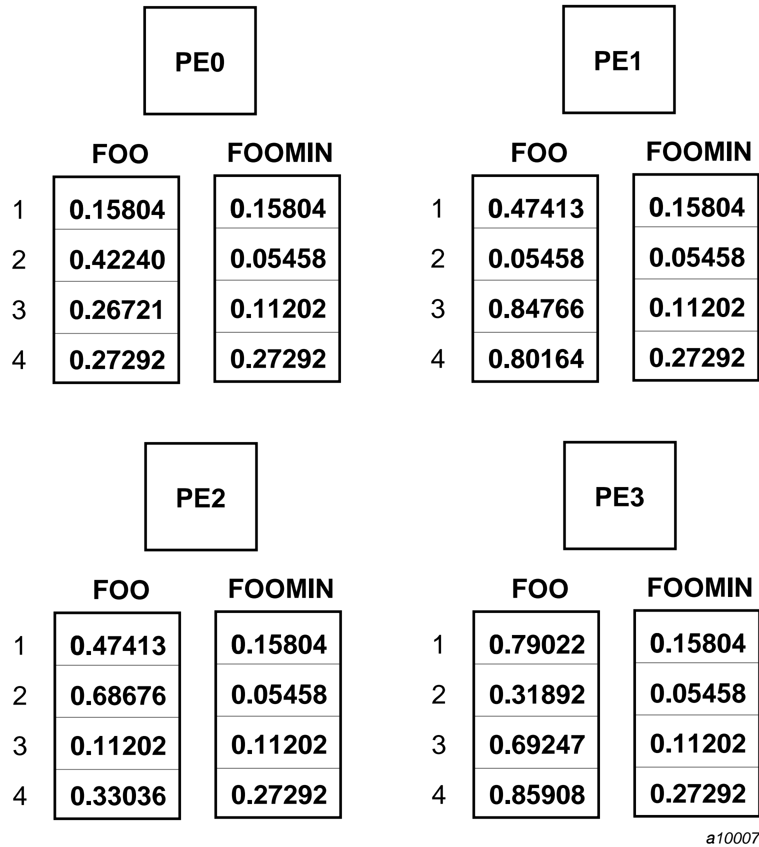

Figure 18. The `SHMEM_REAL8_MIN_TO_ALL` example

In the next example, the summation performed by the ring program is implemented by using the `SHMEM_INT4_SUM_TO_ALL`(3) reduction routine. The reduction is faster than using either `SHMEM_GET64` or `SHMEM_PUT64`, 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

```
1.       PROGRAM RING_NSUM_4
2. C
3. C SHMEM_INT4_SUM_TO_ALL version of ring nsum
4. C
5.         INCLUDE 'mpp/shmem.fh'
```

```
6.          INTEGER ME, NPES, NEXT, PREV
7.          INTEGER ISTAT
8.          INTRINSIC MY_PE, N$PES
9.          PARAMETER(N=100000)
10.         INTEGER(KIND=4) SEND(N), TOTAL(N)
11.         COMMON /IDATA/SEND, TOTAL
12.         REAL PWRK(MAX(N/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
13.         INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
14.         DATA PSYNC/SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
15. C
16. C Get PE info
17.         ME = MY_PE()
18.         NPES = N$PES
19. C
20. C Initialize data
21. C
22.         SEND = ME
23. C
24. C Synchronize - make sure data is ready on all PEs
25.         CALL SHMEM_BARRIER_ALL
26. C
27. C Perform work
28.         CALL SHMEM_INT4_SUM_TO_ALL(TOTAL, SEND,
29.       *  N, 0, 0, NPES, PWRK, PSYNC)
30.
31.         WRITE(*,*) ' PE = ', ME, ' Result = ', TOTAL(1),
32.      $            ' Expect = ', (NPES-1.)*NPES*.5
33.         END
```

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_GET64, and SHMEM_PUT64 versions of the ring program, SHMEM_INT4_SUM_TO_ALL delivers the best performance of the four.

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 77).
- Using pipelining for loop optimization (see Section 4.2, page 78).
- Making the best use of cache (see Section 4.3, page 85).
- 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 106).
- Vectorizing some math operations within a loop (see Section 4.6, page 109).
- Bypassing cache (see Section 4.7, page 113).

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 unrolling on important loops by including the `-O unroll1` option on the `f90`(1) command line and placing the `UNROLL` directive immediately in front of a loop, as follows:

```
!DIR$ UNROLL
      DO I = 1, N
...
```

Specifying the `-O unroll2` command-line option instructs the compiler to make all loops in a program candidates for unrolling. You can instruct the compiler not to unroll a 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 loops that are not the innermost loop, a technique called unroll and jam is performed. A loop must meet special criteria, however, to ensure that correct behavior is maintained. In particular, the loop must have no data dependencies across its iterations. Also, the compiler will perform 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 `DO` loops, `DO WHILE` loops, and `WHILE` loops, provided that the loops contain no conditional code or subroutine calls. Software pipelining is often effective when used in conjunction with the `-O3` or `-Ovector3` option to vectorize intrinsic functions. For information on vectorization, see Section 4.6, page 109.

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 `-O pipeline`*n* option specifies the level of pipelining in effect for the program. The levels are as follows:

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

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 gives the compiler information about dependencies between different array references. Sometimes the compiler cannot understand ambiguous array references, forcing it to assume a dependency exists (for safety reasons) where there are none. Using the `CONCURRENT` directive allows the compiler to assume no dependencies exist. This information is used by the software pipeliner 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 no dependencies exist between the current iteration of the loop and *n* preceding or subsequent iterations.

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

```
!DIR$ CONCURRENT SAFE_DISTANCE=3
      DO I = K+1, N
        X(I) = A(I) + X(I-K)
      ENDDO
```

The CONCURRENT directive in this example informs the software pipeliner that the relationship K>=3 is true. This allows the pipeliner, for example, to safely load any of the array references X(I-K), X(I-K+1), X(I-K+2), and X(I-K+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 provides information on vector dependencies, which are data dependencies between a memory reference and those memory references that lexically precede it in the loop.

The SAFE_VL clause of the IVDEP directive is currently not available on the CRAY T3E system. The IVDEP directive on the CRAY T3E system assumes an infinite vector length. For more information on the IVDEP directive, see Section 4.6.1, page 112.

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 149) 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.

  ```
  DO I = P+1, N
      X(I) = A(I) + X(I-P)
  END DO
  ```

  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:

  ```
  DO I = 2, N
      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
      ...                          ! More work
  END DO
  ```

  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:

```
DO 1 I = 1, N
   Y(I) = X(I)
END DO
```

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 = 1
DO
   T = X(I)

   Y(I) = T
   I = I+1
   IF (I.GT.N) EXIT
END DO
```

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÷2) has been achieved.

The new loop (see the following figure) has parts of multiple iterations executing at the same time, 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 100% busy within the loop.

```
I=1
T1=X(I)
I=I+1
                  T2=X(I)
DO
Y(I-1)=T1         I=I+1
IF(I.GT.N) EXIT                     T1=X(I)
                  Y(I-1)=T2         I=I+1
                  IF(I.GT.N) EXIT                     T2=X(I)
END DO
```

Figure 19. Overlapped iterations

Figure 20, page 84, illustrates the overlap of iterations for the following loop:

```
DO I=1,N
  ... = A(I) * B(I)
  ... = C(I) * D(I)
END DO
```

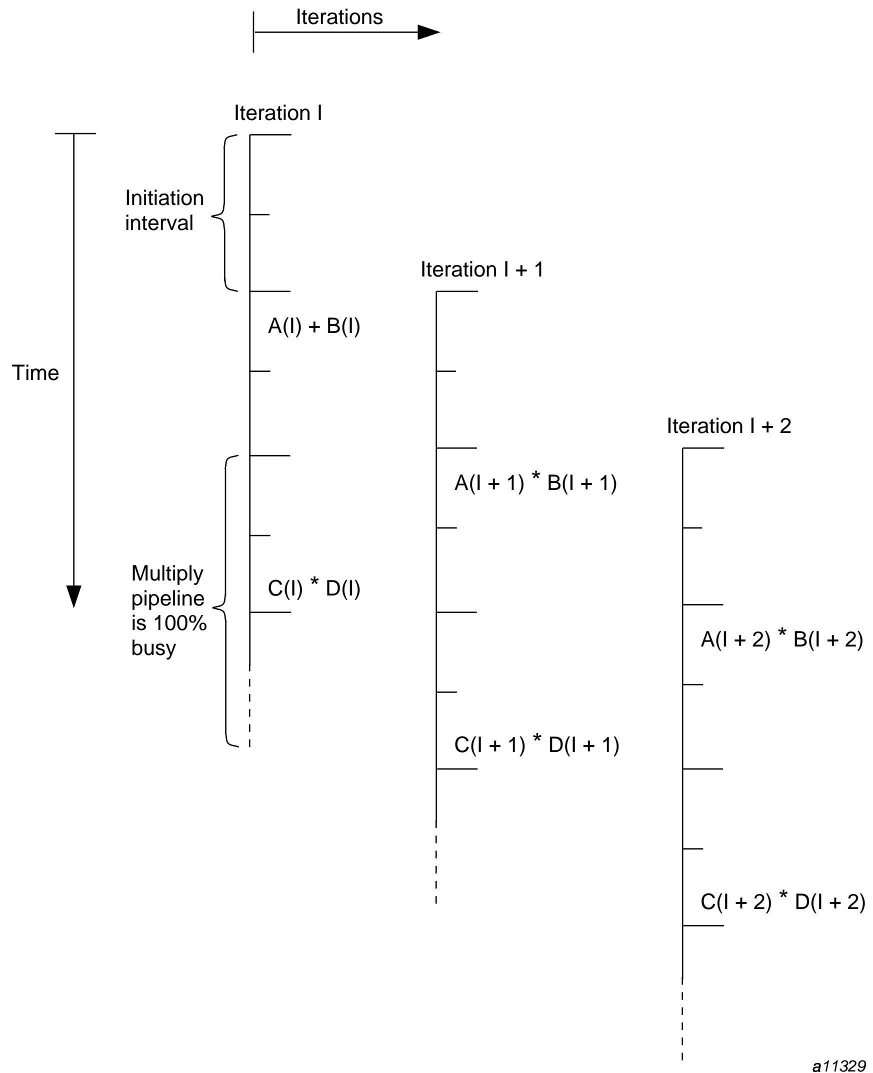

Figure 20. Pipelining a loop with multiplications

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 144). 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 sections describe how to rearrange array dimensions (see the next section) and add pad arrays (see Section 4.3.2, page 88). 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 leftmost, dimensions of an array.

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

Example 21: Unoptimized code

```
COMMON A(N,3,3), B(N,3,3), C(N,3,3)
    DO I=1,3
      DO K=1,3
        DO L=1,N
```

```
        C(L,I,K) = A(L,I,1) * B(L,1,K)
&           +A(L,I,2) * B(L,2,K)
&           +A(L,I,3) * B(L,3,K)
      ENDDO
    ENDDO
  ENDDO
```

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 second and third 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, leftmost dimensions. The dimension declarations can be changed as follows:

```
COMMON A(3,3,N), B(3,3,N), C(3,3,N)
```

Rearranging a dimension should be accompanied by changing the order of the nested `DO` 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. The `K` and `I` loops have more invariant references than the `L` loop. The `I` loop gives more stride-1 references, thus the loop nest should be changed in the following way:

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

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

a10179

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 152). 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 26, page 98.

4.3.2 Padding Common Blocks and Arrays to Reduce Cache Conflict

Reducing cache conflict is another method of increasing cache reuse. Reducing data cache conflict between arrays accessed in the same loop can reduce the run time of loops by up to 66%. Reducing secondary cache conflict can reduce the run time of loops by up to 85%.

Due to the addition of a 96-Kbyte secondary cache on CRAY T3E systems, data cache misses are not as costly as they were for CRAY T3D systems. When a value needed by the microprocessor is not in data cache, it is often in secondary cache. Accessing either data cache or secondary cache is faster than accessing local memory. Maximizing the number of data references to cache can effectively decrease the execution time of your program, and adding pad arrays is one way to do that.

You can either pad arrays yourself, or you can specify the command-line option `-a pad` and have the compiler do the padding. Because the compiler does not do extensive analysis of the data or any analysis of data reference patterns in your code, you may be able to get better results adding your own pad arrays. This section concentrates on how to add your own padding arrays. Information on instructing the compiler to add them automatically is included in Section 4.3.3, page 93.

Cache conflict in common blocks is a function of the declaration of the common block, the code that references it, and the size of the cache. Such conflict often happens when a common block's arrays have sizes that are powers of two, as in the following code fragment:

```
COMMON /AAA/ A(1024), B(1024), C(1024)

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

Data cache is 8 Kbytes, or 1,024 words, in size. Each of the arrays in this example is the same size as data cache. Every word in the B array is mapped to a line in data cache, element B(1) to the first cache line and B(1024) to the last cache line. Then the next data item in memory, in this case C(1), is mapped to the first line of data cache. (See the following figure.)

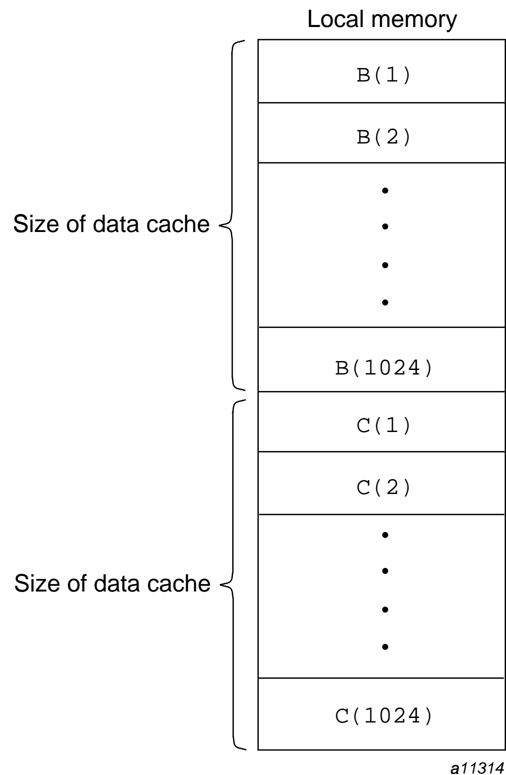

Figure 22. Arrays B and C in local memory

Because of the array sizes, all references to B(I) and C(I) for the same value of I will map to the same line in data cache. Data cache is direct mapped, meaning it can contain only one of the two, either B(I) or C(I). Usually B(1-4) will be loaded and, after B(1) is moved to a register, overwritten by C(1-4). (See Figure 23, page 90.) When B(2) is needed, it will have to be loaded into data cache a second time. Subsequent references to B and C may mean reloading the same cache line in the same manner for each array element.

Such data cache *thrashing* (glossary, page 151) can be avoided by adding pad arrays of at least 4 words between the arrays in conflict.

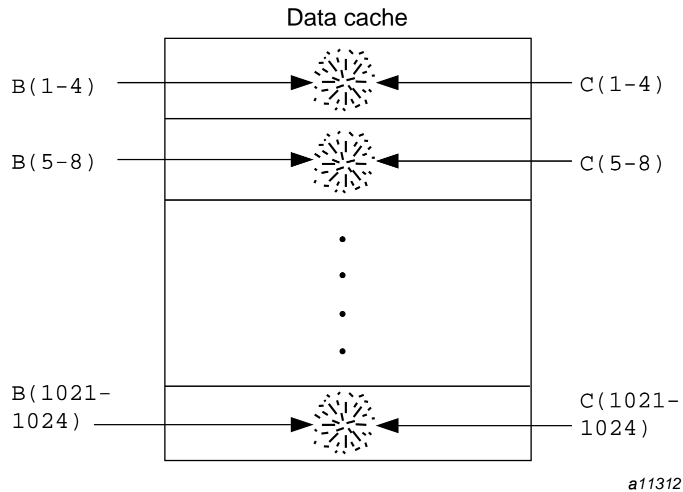

Figure 23. Cache conflict between arrays B and C

Adding a CACHE_ALIGN directive as well guarantees that the common block starts at the beginning of an 8-word line, implying that 4 words of padding are sufficient to avoid conflict in this case.

```
      COMMON /AAA/ A(1024), B(1024), P(4), C(1024)
!DIR$ CACHE_ALIGN /AAA/
```

Now the layout in memory looks different. The P array, placed as it is between the B and the C arrays, causes the mappings of the two arrays to change. (See Figure 24, page 91.)

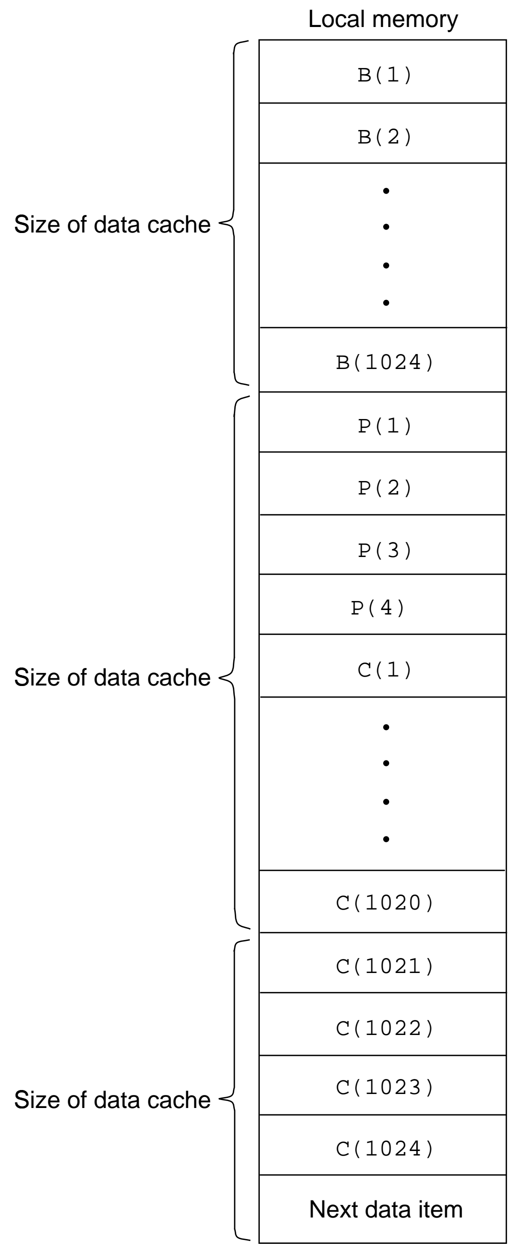

Figure 24. Arrays B and C in local memory after padding

The first elements of C now map to the second line of data cache, and the conflicts have disappeared. C(1-4) will not be replaced by B(5-8) until they are no longer needed. (See the following figure.)

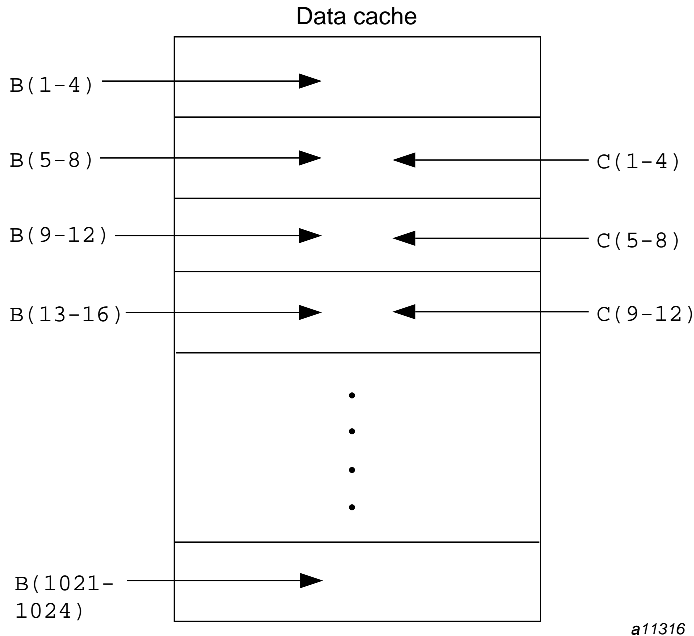

Figure 25. Data cache after padding

The preceding example contains no conflict in secondary cache either. Although secondary cache is larger than data cache, keep in mind the need to maximize its performance, especially when you are dealing with large arrays. The following example is a similar code fragment but with larger arrays. This time the size of the arrays match the size of secondary cache. Any multiple of that size causes conflicts.

```
      COMMON /AAA/ A(4096), B(4096), C(4096)
!DIR$ CACHE_ALIGN /AAA/

      DO I=1,4096
        A(I) = B(I) + C(I)
      ENDDO
```

Output values and input values are both written to secondary cache, meaning you must be concerned about conflict with A as well as B and C. (For an example, see Procedure 1, page 9.) The values A(I), B(I), and C(I) will all map to the same line in secondary cache for the same values of I.

Although secondary cache is three-way, set-associative (three 8-word values can be held in each line), its random replacement strategy creates the potential for cache thrashing. Because its lines are 8 words, a pad array of 8 words (as opposed to 4 words for data cache) is necessary to eliminate conflicts in secondary cache. The P1 and P2 arrays are pad arrays in the following example:

```
      COMMON /AAA/ A(4096), P1(8), B(4096), P2(8), C(4096)
```

Padding individual dimensions within the same array may also reduce cache conflict, as in the following code fragment:

```
      COMMON /AAA/ A(1024,64), B(1024,64)
!DIR$ CACHE_ALIGN /AAA/

      DO J=1,64
        DO I=1,1024
          A(I,J) = B(I,J) - B(I,J+1)
        ENDDO
      ENDDO
```

Not only do the A(I,J) and B(I,J) references conflict in both data cache and secondary cache, but the two B references conflict with each other in data cache. Extending the first dimensions of both A and B by 8 words avoids any conflict in either cache. You need not use the extra words defined by the pad array. The loop bounds will remain the same using the following arrays:

```
      COMMON /AAA/ A(1032,64), B(1032,64)
```

This technique will work for all arrays, not just for those in common blocks.

Note: Use caution when applying these padding techniques yourself. Because they change the sequence association and storage association of the data that is padded, they can change the behavior of a program.

4.3.3 Automatic Padding

The compiler adds padding automatically after every array in a common block and after local static data when you specify the -a pad option on the f90 command line. The compiler computes a padding value for each array. It pads

static arrays and common blocks if the arrays do not appear in an EQUIVALENCE statement.

You can specify your own pad size by adding a value to the -a pad option. The compiler adds padding after every array if you execute one of the following command line options:

```
% f90 -a pad ...
% f90 -a pad 64 ...
```

Using the first option, the compiler adds padding that depends on the size of the array, as follows:

- For large arrays of 1,024 words or more, it adds up to 264 words of padding for every 4,096 words in the array to reduce secondary cache conflict. A pad of 264 words reflects the following:
  - Loop splitting is stripmined by 256.
  - Secondary cache lines are 8 words long.
- For smaller arrays, between 128 and 1,023 words, it adds 8 words to reduce data cache and secondary cache conflict. Arrays with fewer than 128 words are not padded.

Automatic padding also adds words so that the next array starts on a secondary cache line (8–word) boundary. It assumes that the preceding array also starts on an 8–word boundary. You can ensure that it does by using either the CACHE_ALIGN directive or the following compiler/loader option:

```
-Wl"-D allocate(alignsz)=64b"
```

For the pad 64 argument, the compiler adds a pad of 64 words after all arrays. Using a large pad value on small arrays is not recommended.

Note: Automatic padding by the compiler could potentially slow some programs down or cause results to change, since the option breaks standard Fortran sequence association and storage association specifications for padded data. For restrictions that apply to automatic padding, see the *CF90 Commands and Directives Reference Manual*, publication SR–3901.

In the application that defined the following common block, every array is aligned with two other arrays in cache:

Example 22: Automatic padding

```
COMMON A(1024), B(1024), C(1024), D(1024), E(1024), F(1024),
% G(1024), H(1024), R(1024), S(1024), T(1024), U(1024)
```

The compiler automatically pads the arrays as follows if the `-a pad` option is specified:

```
 COMMON A(1024), PAD1(72),  B(1024), PAD2(72),  C(1024), PAD3(72),
&       D(1024), PAD4(72),  E(1024), PAD5(72),  F(1024), PAD6(72),
&       G(1024), PAD7(72),  H(1024), PAD8(72),  R(1024), PAD9(72),
&       S(1024), PAD10(72), T(1024), PAD11(72), U(1024), PAD12(72)
```

Now none of the arrays start at the same location in cache. The pad at the end of the common block is added to potentially improve effects across common blocks in the same way that it improves things within common blocks.

A common block of smaller arrays such as the following can be saved from data cache conflicts:

Example 23: Automatic padding for smaller arrays

```
 COMMON A(256), B(256), C(256), D(256), E(256), F(256),
& G(256), H(256), R(256), S(256), T(256), U(256)
```

The compiler automatically pads these arrays as follows:

```
 COMMON A(256), PAD1(8),  B(256), PAD2(8),  C(256), PAD3(8),
&       D(256), PAD4(8),  E(256), PAD5(8),  F(256), PAD6(8),
&       G(256), PAD7(8),  H(256), PAD8(8),  R(256), PAD9(8),
&       S(256), PAD10(8), T(256), PAD11(8), U(256), PAD12(8)
```

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).
- Padding common blocks and arrays for loop splitting (see Section 4.4.2, page 100).
- Avoiding undesirable side effects from loop splitting (see Section 4.4.3, page 101).
- Getting the most out of a stream by maximizing the inner loop trip count (see Section 4.4.4, page 101).
- Rearranging the dimensions of an array in order to cut down on the number of streams (see Section 4.4.5, page 102).
- Reducing overhead by grouping statements that use the same stream (see Section 4.4.6, page 103).
- Enabling or disabling stream buffers to get the most benefit from the hardware (see Section 4.4.7, page 105).

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 CF90 compiler provides a command-line option (`-O split`*n*) and source directives (`SPLIT` and `NOSPLIT`) to split loops. The compiler does not split loops without direction from you, 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 4.4.3, page 101.

Place the `SPLIT` compiler directive immediately before the `DO` 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.
- There are few, if any, IF statements. Loops with IF statements can be split profitably but not 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; it splits the loop only if it can be done without changing the results of the computation. It will not cause incorrect behavior in codes that conform to the Fortran standard. 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.

Loop splitting should also be used in conjunction with loop unrolling (see Section 4.1, page 77). The -Ounroll2 command-line option has been shown to improve performance when used with the SPLIT directive.

The following examples show optimizations based on splitting loops:

Example 24: Original loop

```
!DIR$ SPLIT
      DO I=1,4000
         A(I) = B(I) * C(I)
         T = D(I) + A(I)
         E(I) = F(I) + T * G(I)
         H(I) = H(I) + E(I)
      ENDDO
```

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 25: Splitting loops

```
      DO I=1,4000
         A(I) = B(I) * C(I)
         TA(I) = D(I) + A(I)
      ENDDO
      DO I=1,1000
         E(I) = F(I) * TA(I) * G(I)
         H(I) = H(I) + E(I)
      ENDDO
```

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 26: Stripmining

```
      DO I1=1,4000,256
         I2 = MIN(I1+255, 1000)
         DO I=I1,I2
            A(I) = B(I) * C(I)
            TA(I-I1+1) = D(I) + A(I)
         ENDDO
         DO I=I1,I2
            E(I) = F(I) * TA(I-I1+1) * G(I)
            H(I) = H(I) + E(I)
         ENDDO
      ENDDO
```

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 27: Splitting loops across `IF` statements

```
!DIR$ SPLIT
      DO I=1,4000
         IF (A(I) .LT. 0.0)
             B(I) = C(I) * D(I)
         ELSE
             E(I) = F(I) * G(I)
         ENDIF
      ENDDO
```

The preceding loop is split as follows:

```
DO I=1,4000
   L(I) = A(I) .LT. 0.0
   IF (L(I))
       B(I) = C(I) * D(I)
   ENDIF
ENDDO
DO I=1,1000
   IF (.NOT. L(I))
       E(I) = F(I) * G(I)
   ENDIF
ENDDO
```

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 28: Splitting individual statements

```
!DIR$ SPLIT
      DO I=1,4000
         A(I) = B(I) * C(I) + D(I) * E(I) + F(I) * G(I)
      ENDDO
```

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

```
DO I=1,4000
   T(I) = B(I) * C(I) + D(I) * E(I)
ENDDO
DO I=1,1000
   A(I) = T(I) + F(I) * G(I)
ENDDO
```

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

The `-O split2` command-line option to the `f90`(1) command can be used to apply the `SPLIT` directive to all loops in a file. The `-O split1` 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 NOSPLIT compiler directive inhibits loop splitting and overrides a -O split2 command-line specification. The -O split0 directive disables all loop splitting.

For more information on loop splitting options and directives, see the *CF90 Commands and Directives Reference Manual*.

4.4.2 Padding Common Blocks and Arrays for Loop Splitting

In addition to decreasing cache conflict, padding common blocks and arrays can improve the performance of loop splitting, especially in the case of array reuse within a loop.

The following code fragment contains a loop that has already been split and stripmined:

```
      COMMON /AAA/ A(4096), B(4096), C(4096), D(4096),
     &                      X(4096), Y(4096), Z(4096)

      DO I1=1,4096,256
        I2 = MIN(I1+255, 4096)
        DO I=I1,I2
          A(I) = B(I) * C(I) + D(I)
        ENDDO
        DO I=I1,I2
          X(I) = Y(I) * Z(I) - D(I)
        ENDDO
      ENDDO
```

Notice the reuse of D between the two inner loops. All of the array references conflict in both data cache and secondary cache. Adding pad arrays of 8 words between each array will eliminate data cache conflict within a single loop, however much of the strip of D will likely have been thrown out of both data cache and secondary cache before the second loop begins. Adding padding arrays of 264 words before and after D will ensure that the strip of D is still in secondary cache when the second loop begins. Arrays P3 and P4 serve that purpose in the following example:

```
      COMMON /AAA/ A(4096), P1(8), B(4096), P2(8), C(4096), P3(264),
     &       D(4096), P4(264), X(4096), P5(8), Y(4096), P6(8), Z(4096)
```

The compiler has a command-line option that adds such pad arrays automatically. See Section 4.3.3, page 93.

4.4.3 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 `-O overindex` option to the compiler command line. It disables some assumptions made by the compiler that *overindexing* (glossary, page 148) 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.

If correct behavior cannot be restored, you may have to disable loop splitting for the offending loop by using the `NOSPLIT` directive.

4.4.4 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 29: Rearranging array dimensions

```
DIMENSION X(32, 1000), Y(32, 1000), Z(32, 1000)
DIMENSION S(32, 1000), T(32, 1000), W(32, 1000), U(32, 1000)

DO J=1,N1000
  DO I=1,N32
    X(I,J) = Y(I,J) + Z(I,J)
    W(I,J) = X(I,J) * T(I,J) - S(I,J) * U(I,J)
  ENDDO
ENDDO
```

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%:

```
DIMENSION X(1000, 32), Y(1000, 32), Z(1000, 32)
DIMENSION S(1000, 32), T(1000, 32), W(1000, 32), U(1000,32)

DO J=1,N32
  DO I=1,N1000
    X(I,J) = Y(I,J) + Z(I,J)
    W(I,J) = X(I,J) * T(I,J) - S(I,J) * U(I,J)
  ENDDO
ENDDO
```

4.4.5 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 leftmost) dimension. This is the opposite of the technique described in Section 4.4.4, page 101, 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. Such array dimensions and references are more common in older Fortran codes in which the programmer is simulating functionality now provided by derived types in Fortran 90.

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 30: Minimizing streams

```
DIMENSION X(1000, 6)
DIMENSION Y(1000, 6)
DO I=1,1000
  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)
  X(I,6) = Y(I,6)
ENDDO
```

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

Example 31: Reduced streams version

```
DIMENSION X(6, 1000)
DIMENSION Y(6, 1000)
DO I=1,1000
  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)
  X(6,I) = Y(6,I)
ENDDO
```

4.4.6 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 32: Original code

```
DIMENSION X(2,1000), Y(2,1000), Z(2,1000)
DIMENSION S(2,1000), T(2,1000), W(2,1000), U(2,1000)

DO I=1,1000
  X(1,I) = Y(1,I) + Z(1,I)
  W(1,I) = X(1,I) * T(1,I) - S(1,I) * U(1,I)
  X(2,I) = Y(2,I) + Z(2,I)
  W(2,I) = X(2,I) * T(2,I) - S(2,I) * U(2,I)
ENDDO
```

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 33: Grouping statements within the loop

```
DO I=1,1000
  X(1,I) = Y(1,I) + Z(1,I)
  X(2,I) = Y(2,I) + Z(2,I)
  W(1,I) = X(1,I) * T(1,I) - S(1,I) * U(1,I)
  W(2,I) = X(2,I) * T(2,I) - S(2,I) * U(2,I)
ENDDO
```

The situation in the preceding example is most likely to occur in older Fortran codes that contain arrays with small dimensions (between 2 and 20 elements) that simulate Fortran 90 derived types. But it can also 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 34: Loop that will be split into four

```
DIMENSION X(1000), Y(1000), Z(1000)
DIMENSION S(1000), T(1000), W(1000), U(1000)

DO I=1,1000,2
  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)
ENDDO
```

As in Example 32, page 104, and Example 33, page 104, 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 35: Loop that will be split into two

```
DO I=1,1000,2
  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)
ENDDO
```

4.4.7 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 call. The stream buffer level is changed to in the following example:

```
CALL SET_D_STREAM(0)
```

The `GET_D_STREAM` routine saves the current stream buffer level in order to restore 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 high performance Fortran (HPF) or PGHPF/CRAFT language features.
- A program that uses the CACHE_BYPASS directive (see Section 4.7, page 113).

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 for these programs using the SET_D_STREAM library routine 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 CF90 compiler does by default, can improve performance dramatically. In the following example, A, B, C, and D are all cache-resident arrays:

```
!DIR$ UNROLL
        DO I = 1, 256
           A(I) = (B(I) + 2.0 * C(I) + D(I)) / X
        ENDDO
```

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
!DIR$ UNROLL
      DO I = 1, 256
         A(I) = (B(I) + 2.0 * C(I) + D(I)) * XINV
      ENDDO
```

By default, the CF90 compiler changes a divide into a reciprocal multiply for you. You do not have to change the code at all. Unless you enable IEEE division by specifying the `-e u` option on the `f90`(1) 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 36: Original code

```
1.        DO 1300 JN = 1,LPR
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 = 1.0E0/(XW1**2+XW2**2+RWTMP**2)
8.  C The problem is here.  The result of divide is used
9.  C  in next calculation.
10.         DF2 = CGI*CG(J)*R2INV
11.         EELT = EELT+DF2
12.         R6 = R2INV**3
13.         F1 = CN12(1,IC)*(R6*R6)
14.         F2 = CN12(2,IC)*R6
15.         ENBT = ENBT + (F2-F1)
16.         DF = (DF2+6.0E0*((F2-F1)-F1))*R2INV
17.         FW1 = XW1*DF
18.         FW2 = XW2*DF
19.         FW3 = RWTMP*DF
20.         F(1,J) = F(1,J)  +FW1
21.         F(2,J) = F(2,J)  +FW2
22.         F(3,J) = F(3,J)  +FW3
23.
24.         TMP4 = TMP4 -FW1
25.         TMP5 = TMP5 -FW2
26.         TMP6 = TMP6 -FW3
27. 1300  CONTINUE
```

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 37: Modified code

```
1.  C 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 = 1.0E0/(XW1**2+XW2**2+RWTMP**2)
7.
8.       DO 1300 JN = 1,LPR
9.  C Compute the divide needed 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.        R2INV_NEXT = 1.0E0/(XW1_NEXT**2+XW2_NEXT**2+RWTMP_NEXT**2)
15.        IC = ICO(IACI+IAC(J))
16.        DF2 = CGI*CG(J)*R2INV
17.        EELT = EELT+DF2
18.        R6 = R2INV**3
19.        F1 = CN12(1,IC)*(R6*R6)
20.        F2 = CN12(2,IC)*R6
21.        ENBT = ENBT + (F2-F1)
22.        DF = (DF2+6.0E0*((F2-F1)-F1))*R2INV
23.        FW1 = XW1*DF
24.        FW2 = XW2*DF
25.        FW3 = RWTMP*DF
26.        F(1,J) = F(1,J)  +FW1
27.        F(2,J) = F(2,J)  +FW2
28.        F(3,J) = F(3,J)  +FW3
29.
30.        TMP4 = TMP4 -FW1
31.        TMP5 = TMP5 -FW2
32.        TMP6 = TMP6 -FW3
33. C Juggle the values for the next pass.
34.        J     = J_NEXT
35.        XW1   = XW1_NEXT
36.        XW2   = XW2_NEXT
```

```
37.         RWTMP = RWTMP_NEXT
38. C The result of divide not needed until here.  All the work
39. C above this can proceed concurrently with the divide.
40.         R2INV = R2INV_NEXT
41. 1300 CONTINUE
```

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 26, page 98.)
2. Split vectorizable operations into separate loops, if necessary. (For more information on loop splitting, see Example 26, page 98.)
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 78.)

The programming environment also offers the `libmfastv` library of faster, but less accurate, vector versions of the `libm` routines. These routines deliver results that are usually one-to-two bits less accurate than the results given by the `libm` routines. Less accurate scalar versions of the library routines 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 50 to 70%.

Because the vector routines may not provide a performance improvement in all cases (due to necessary loop splitting), vectorization is not turned on by default. It is enabled through the compiler command-line option -O vector3. The default is vector2, which currently does not do 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 -O vector3, you can further control vectorization by using the following:

- Vector directives NEXTSCALAR and [NO]VECTOR. They allow you to turn vectorization on and off for selected parts of your program.
- In the case of ambiguous data dependences within the loop, you can express a loop's vectorization potential by including the IVDEP directive (see Section 4.6.1, page 112). IVDEP tells the compiler to proceed with vectorization and ignore vector dependencies in the loop that follows.
- Access to the libmfastv routines can be controlled with the -l compiler option. For example, the following command line links in the faster, but less accurate math routines rather than the slower, default routines in libm:

```
% f90 -Ovector3 -lmfastv test.f
```

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 libm results generally within 2 *ulps* (glossary, page 152), although some results could differ by larger amounts. Exceptions may also differ from the libm versions, where some calls to libmfastv may generate only a *NaN* (glossary page 147) for a particular operand rather than an exception, causing exceptions later in the program.

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(3)

1/SQRT (replaced by a call to SQRTINV(3)

LOG(3)

EXP(3)

SIN(3)

COS(3)

COSS(3) (replaced by a combined call to SIN(3) and COS(3)

The following support 64-bit floating-point only:

RANF(3)

X**Y

POPCNT(3)

Note: Early versions of the Programming Environment 3.0 release may not vectorize loops with multiple RANF(3) calls. The IVDEP directive enables RANF vectorization, but the values returned may be different than those returned without vectorization.

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:

Example 38: Transforming a loop for vectorization

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

The loop in this example will be transformed into the following loop. The vector version of the square root function is called after the first ENDDO statement.

```
DO II = 1, N, 256
    NN = MIN( II+255, N )
    DO I = II, NN
        T(I-II+1) = C(I) + D(I)
    ENDDO
    SQRT_V( NN - II + 1, T, T2, 1, 1 )
    DO I = II, NN
        A(I) = B(I) * T2(I-II+1)
    ENDDO
ENDDO
```

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 enables the compiler to perform the following optimizations:

- Loop splitting
- Using the vectorizing intrinsic routines and operators listed in Section 4.6, page 109
- Speeding up memory access with the CACHE_BYPASS directive (see Section 4.7, page 113)
- Improving pipelining

Vector dependencies prevent the compiler from vectorizing a loop because of the unknown value of array indices at the time the program is executed. In particular, the first statement in the loop may be trying to read a value that was written by the second statement in an earlier iteration:

```
DO I=1,N
  ...
  X = A(I-K)
  A(I) = B(I) * C(I) - W
  ...
ENDDO
```

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.

In the following example, there may be a dependence, depending on the values of IX.

```
DO I=1,N
  ...
  B(I) = A(IX(I))
  ...
  A(IX(I)) = W(I) * X(I) * Z(I)
  ...
ENDDO
```

If there are duplicate values in IX, there is a dependence. If there are no duplicates in IX, there is no dependence. Again, the compiler will not vectorize the loop unless it is preceded by an IVDEP directive.

4.7 Bypassing Cache

The CACHE_BYPASS directive offers a semi-automatic method for speeding up certain memory references. The execution time for some loops can be reduced by up to 45%; others can be even faster.

Local memory references specified 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, they give you higher bandwidth for sparse index array accesses, such as gather/scatter and large stride accesses that do not take advantage of multiple accesses from the same cache load.

The following example illustrates a large stride access; it strides through the B array by incrementing the rightmost dimension. (For an illustration of how an array is stored in Fortran, see Figure 21, page 87.)

```
!DIR$ CACHE_BYPASS B
      DO I=1,N
        A(I) = B(1,I)
      ENDDO
```

The next example shows a gather/scatter loop. The references to the A and B arrays are drawn from a third array, IX. Both the following loop and the preceding loop will benefit from bypassing cache.

```
!DIR$ CACHE_BYPASS A, B
      DO I=1,N
        A(IX(I)) = B(IX(I)) + C
      ENDDO
```

Note: The CACHE_BYPASS directive makes no guarantees about the state of the cache before or after the specified loop. In particular, it does not do the following:

- Guarantee that the specified variables are not in cache before the loop.
- Guarantee that the specified variables are not in cache after the loop.
- Invalidate cache.
- Affect program results in any way.

The directive is strictly a performance hint to the compiler.

CACHE_BYPASS 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 DO loop and affects all of the named arrays within the scope of that loop.[1] In the following loop, array X is initialized through E registers rather than through cache:

```
!DIR$ CACHE_BYPASS X
      DO I = 1,N
        X(I) = 1.0
      ENDDO
```

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:

1 Only types whose base types are 64 bits (INTEGER(KIND=8), REAL(KIND=8), LOGICAL(KIND=8), and pointers to these types) can be named in the directive. Cray pointer pointee arrays, as well as allocatable and deferred-shape arrays, can be named. Arrays or pointers that are components of objects of derived types cannot be named in the directive. Specification of unsupported arrays is ignored and will cause a warning to be issued. Support for 32-bit base types is deferred.

- The loop must be an inner loop, if nesting is involved.
- 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 112. For more information on vectorization, see Section 4.6, page 109.)

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 77.

This technique can reduce the execution time of certain gather/scatter codes by up to 45%. 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%.

The following example loads only array A through E registers:

```
!DIR$ CACHE_BYPASS A,IB
        DO I=1,N
          C(I) = A(IB(IX(I)))
        ENDDO
```

The following example does not bypass cache for stores to the CA array:

```
        POINTER (CP,CA)
        ...
!DIR$ CACHE_BYPASS CA
        DO I=1,N
          CP = LOC(A(IX(I)))
          CA(I) = B(I)
        ENDDO
```

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 4.4.7, page 105.

Input/Output [5]

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 117).
- Using unformatted I/O whenever possible (see Section 5.2, page 122).
- Coping with formatted I/O when necessary (see Section 5.3, page 126).
- Making use of the performance-enhancing FFIO layers in your program (see Section 5.4, page 128).
- Optimizing random access I/O (see Section 5.5, page 134).
- Striping a file over disk partitions (see Section 5.6, page 134).

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 118).

- 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 120).
- 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 121).

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.

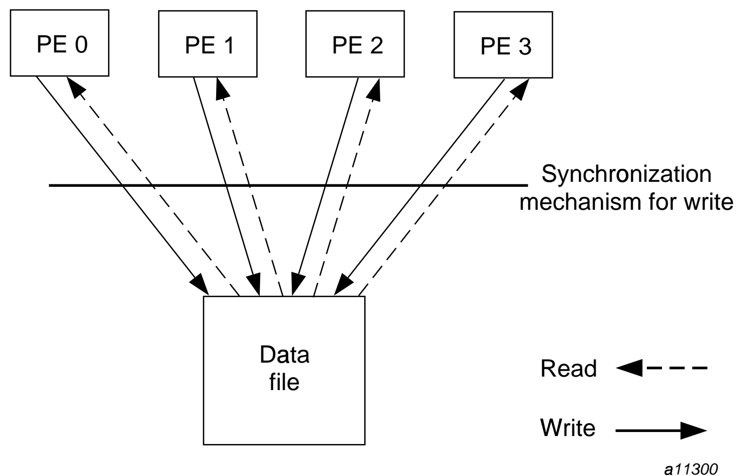

Figure 26. Multiple PEs using a single file

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 130.
- 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 133, and Section 5.4.4, page 133.
- Your file is striped over multiple disk drives. For information on disk striping, see Section 5.6, page 134.

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.

```
OPEN(UNIT=9,ACCESS=DIRECT,ACTION=READ,RECL=40960)
OPEN(UNIT=11,ACCESS=DIRECT,ACTION=WRITE,RECL=40960)
DO I = 1, NUMTODO
  NEXT = FINDNEXT(NEXT, MY_PE, N$PES)
  READ(9, REC=NEXT) Z
    ! Process the record
  WRITE(11,REC=NEXT) Z
ENDDO
```

All of the PEs execute all of the statements. The input file is opened for random access (`DIRECT`) and sets the record length to 40,960 bytes. Setting the `ACTION` argument to `READ` (rather than `READWRITE`, for instance) allows the I/O libraries to optimize the read operation.

A separate `OPEN` statement opens the output unit, which is also random access. Each PE writes its own record from `Z` each time through the loop.

Before the program that includes this code is run, execute the assign command twice, as follows:

```
% assign -F cachea:10 u:9 f:datadir/input
% assign -F cachea:10 -m on u:11 f:datadir/output
```

The assign command sets up a cache of 10 blocks, each containing 512 words. In the program, that size is matched in bytes by the RECL argument to the OPEN statements. Matching the record length with the cache size is crucial when writing in random access mode to avoid overwriting data. The -m on argument to the assign command prevents the PEs from truncating the end of the file, each in a different place, as they complete their output.

Caution: For a sequential, unformatted file, the I/O library normally adds a blocked file structure that contains control information to delimit records. This adds extra time to a program. However, if you want to do positioning in the file (such as backspacing), you must use the control information. Neither of the preceding assign statements specify control information.

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 OPEN statements, 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.

```
IF(MY_PE() .EQ. 0) THEN
  OPEN(9,ACTION=READWRITE,FILE='File0')
IF(MY_PE() .EQ. 1)
  OPEN(9,ACTION=READWRITE,FILE='File1')
IF(MY_PE() .EQ. 2)
  OPEN(9,ACTION=READWRITE,FILE='File2')
IF(MY_PE() .EQ. 3)
  OPEN(9,ACTION=READWRITE,FILE='File3')

READ(9,*) ARRAY
```

```
! Perform calculations

      WRITE(9,*) ARRAY
```

In this example, each PE opens its own file for reading and writing. The READ and WRITE statements are executed by all PEs. For an illustration of this process, see Figure 27.

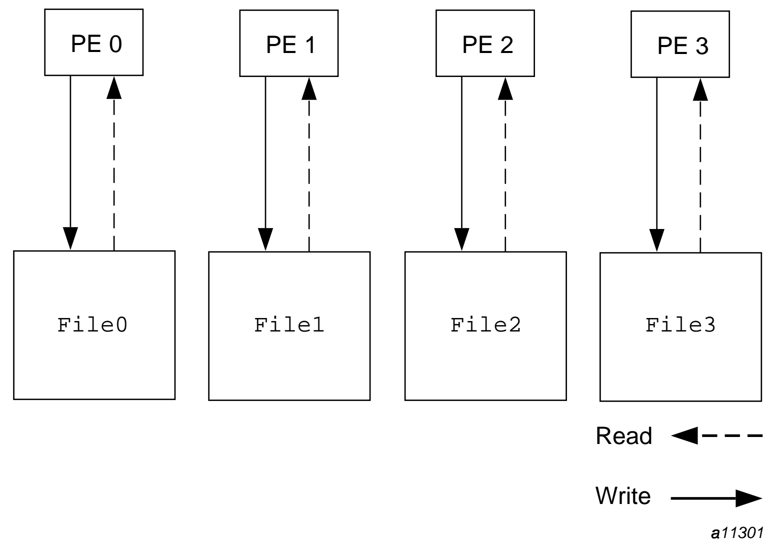

Figure 27. Multiple PEs and multiple files

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.

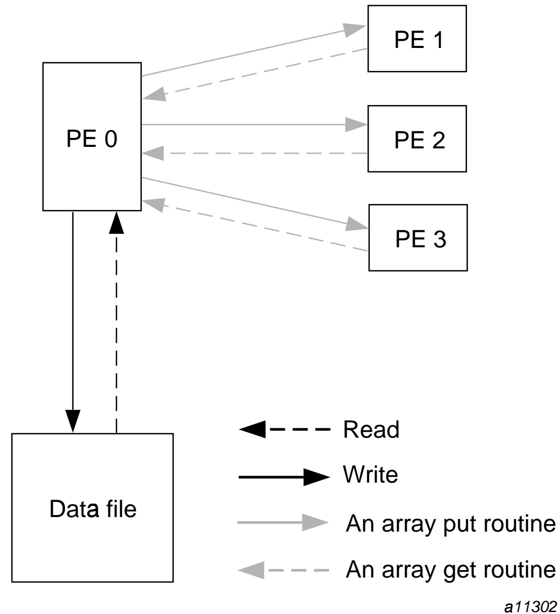

Figure 28. I/O to and from a single PE

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

- You can a message-passing system to pass on the data. PVM, MPI, and SHMEM all have broadcast routines that pass data to other PEs. If every PE needs all the data, the SHMEM_BROADCAST(3) 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 66. For information on PVMFBCAST(3), see Section 2.9, page 33.

- If each PE only needs part of the data, use array-handling get and put routines. SHMEM_IXPUT(3) and SHMEM_IXGET(3) are the fastest. For information on using SHMEM_IXPUT and SHMEM_IXGET, see Section 3.5, page 62. If portability is a concern, both PVM and MPI have put and get routines that can pass arrays, but they are slower.

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
  ```

- Remove FORMAT statements from your program and modify your OPEN statement as follows:

  ```
  OPEN (9, FORM='UNFORMATTED')
  ```

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.

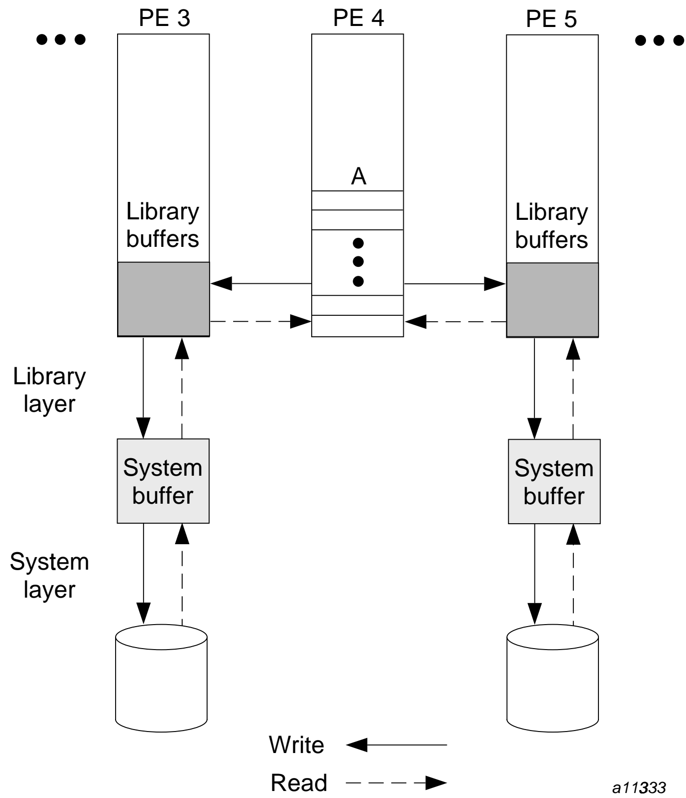

Figure 29. Data paths between disk and an array

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 128.) The following example selects `syscall` for unit 9:

```
% assign -F syscall u: 9
```

- 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
```

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 134.)

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:

- 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.

- 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 128.

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 code makes many small I/O requests by reading an array within a loop, element-by-element:

```
REAL A(I,J,K)
...
DO K = 1, N
  DO J = 1, M
    DO I = 1, L
      READ *, A(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

Instead, read the entire array with a single statement. Doing so will replace many library I/O requests with a single request, meaning fewer calls to the I/O library and ultimately fewer calls to the system. The following example reads the entire array in array-element order. For an illustration of the order in which array elements are stored, see Figure 21, page 87.

```
REAL A(I,J,K)
READ *, A
```

5.3.3 Minimize Data Items

The compiler reduces data items automatically when it detects an I/O statement with an excessive number. You, however, could speed up the compilation by minimizing data items. Without optimization, the following example could make up to 22 calls to the WRITE statement: up to 20 for the X array and one each for the Z array and M:

```
      DIMENSION X(20), Y(10), Z(5,30)
      WRITE (6,101) M, (X(I), I=1,20), Z(M,J)
```

The following example calls WRITE just three times:

```
      WRITE (6,101) M, X, Z(M,J)
```

5.3.4 Use Longer Records

Because reading and writing a record requires some processing, handling a few large records is more efficient than many smaller records. The following example writes one record at a time:

```
      WRITE (42, 100) X
  100 FORMAT (E25.15)
```

The following example processes five records per write, meaning it will handle 80% fewer records than the previous example:

```
      WRITE (42, 101) X
  101 FORMAT (5E25.15)
```

5.3.5 Format Manually

You can save some overhead by reducing the number of edit descriptors. For instance, if you are writing integers that fit into four digits (between –9999 and 9999), either of the following statements work, but the second is more efficient:

```
  200 FORMAT (16(X,I4))
  201 FORMAT (16I4)
```

The following lines of output generated from the two formats are identical. (Some data items are removed from the following because of line width restrictions):

```
 9911 9912 9913 9914 9915 9916 9917 9918 9919 9920 9921 9922 ...
```

```
9911 9912 9913 9914 9915 9916 9917 9918 9919 9920 9921 9922 ...
```

5.3.6 Change Edit Descriptors for Character Data

Use edit descriptors that specify the same width as the character variable when transferring character data. For CHARACTER**n* data, use the A or A*n* edit descriptor. The following example assumes that MYNAME is 12 characters long:

```
READ (*, '(A12)') MYNAME
```

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
            Ap. Size  Number Aps.
 Type   PE  min  max running limit    x  y  z Clock UsrMem FreMem
+ 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) routine.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.

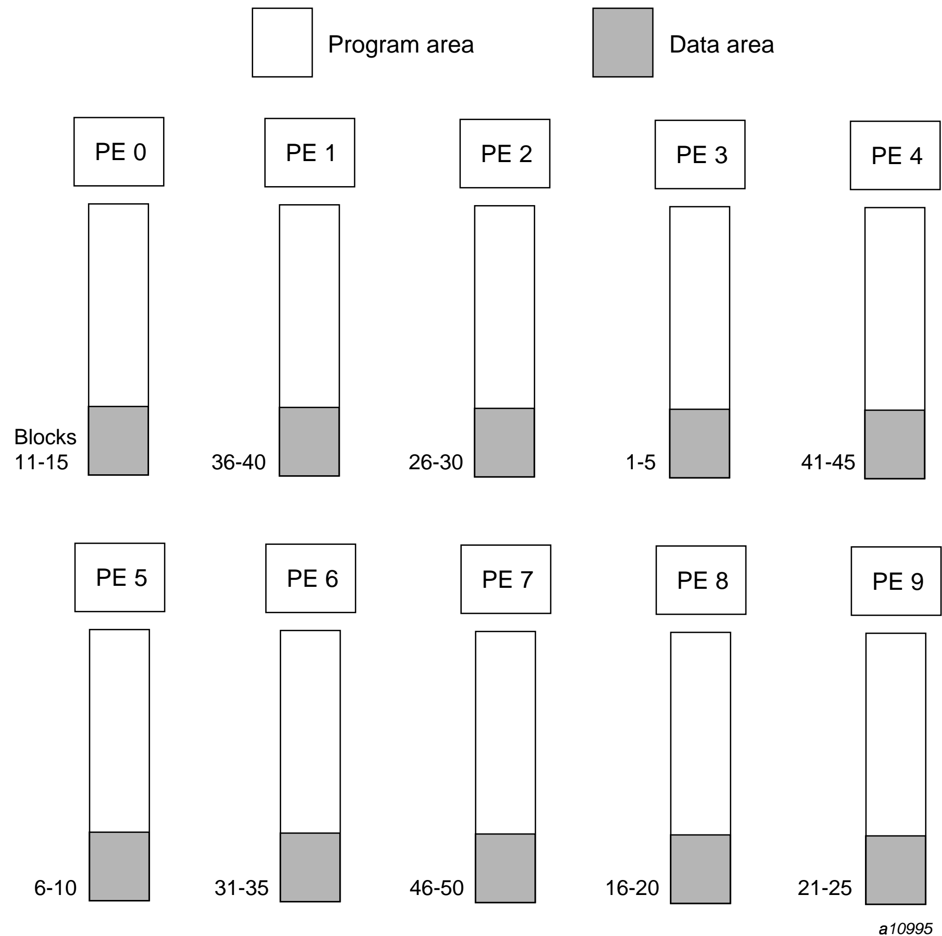

Figure 30. Data layout for distributed I/O

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.

The following example uses distributed I/O. Each processor simultaneously writes out to a different record of a direct access file spread across all of the PEs. The file is closed and PE 0 reads it to ensure that is was written correctly.

Example 39: Distributed I/O

```
      PROGRAM GLOBAL
C Compile like this:
C       f90 -VVv -o global global.f
C Run on any number of PEs >1
C This is private data different on each PE
      REAL GLOBOUT(512),GLOBIN(512)
C Clean up the assign environment
      CALL assign("assign -R")
C Define unit 20 as a global I/O file with each PE having its
C own private position in the file.
      CALL assign("assign -F global.privpos:1:1 u:20")
C OPENs and CLOSEs are "collective":  they must be done by all PEs
      OPEN (20,FILE="GLOBDAT",ACCESS="DIRECT",RECL=4096)
      CALL BARRIER()
C Put unique data in the array on each PE
      GLOBOUT=REAL(MY_PE())
      IREC=MY_PE()+1
      PRINT *,"PE#",MY_PE()," WRITING TO RECORD ",IREC
C Each PE will write out to a different record simultaneously
      WRITE(20,REC=IREC) GLOBOUT
      CLOSE(20)
      CALL BARRIER()
C Verify that this worked by having PE 0 read all the data back.
      IF (MY_PE().EQ.0) THEN
        PRINT *,"READING THE GLOBAL FILE ON PE#0"
        OPEN (30,FILE="GLOBDAT",ACCESS="DIRECT",RECL=4096)
        DO I=0,N$PES-1
          IREC=I+1
          READ(30,REC=IREC) GLOBIN
```

```
    PRINT *,"RECORD ",IREC,"=",INT(GLOBIN(1))
  ENDDO
  CLOSE(30)
ENDIF
CALL BARRIER()
STOP
END
```

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 128.
- 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.

You can also call the assign(3F) routine to stripe a disk file from within a program. The following program writes out a 64-Mbyte array to /usr/tmp, striping it one more level with each write up to the maximum number of partitions on the disk. It starts with the partition with the most free space and works towards the partition with the least space. It times each write and prints out the best rate at the end of the program. The program is long, but it should be general enough to run on any CRAY T3E system on which the /usr/tmp directory is striped.

Example 40: Disk striping from within a program

```
      PROGRAM STRIPEX
C
      IMPLICIT NONE
C Compile with "f90 -O stripex -VVv stripex.f", AND RUN ON ANY
C Number of PEs
      INTEGER NBLOCK,I,IPART,IERR,ILEN,ISHELL,NPART,PART1,MY_PE
      INTEGER IPARTMIN,LENDAT
C Secondary partitions sorted by free space
      INTEGER IFREE(100)
C Size of the array to write out
      PARAMETER(NBLOCK=16*1024)
      PARAMETER(LENDAT=NBLOCK*512)
      REAL A(LENDAT),T1,T2,DUR,SECONDR,RATE,DURMIN,RATEMAX
C This will be the string that does shell commands
      CHARACTER*80 SHELARG
C This will be the string that does the various assigns
      CHARACTER*360 ASGARG
C These strings will hold file names.
      CHARACTER*50 STRIPEFILE
C This will be the string that removes the stripe file
      CHARACTER*80 RMSTRIPE
C This will hold your user name
      CHARACTER*16 MYNAME
C This will hold the partition portion of the assign argment
      CHARACTER*3 PARTARG(100)
C Do your work on 1 PE.  You might want to run it as a 2 PE
C application, so that you don't get interrupted.  All the
C PEs except for 0 spend all their time at a barrier.
      IF (MY_PE().EQ.0) THEN
C Find the user name associated with this process
      CALL PXFGETLOGIN(MYNAME, ILEN, IERR)
      IF (IERR.NE.0) THEN
```

```
         PRINT *,"PXFGETLOGIN ERROR=",IERR," EXITING"
         GOTO 999
      ENDIF
C Get reasonably unique names for the striped data file
      WRITE(STRIPEFILE,500) MYNAME
500   FORMAT("/USR/TMP/STRIPE.",A16)
C Clean up any old files lying around.
      WRITE(RMSTRIPE,520) STRIPEFILE
520   FORMAT("RM -F ",A50)
      PRINT *,RMSTRIPE
      IERR=ISHELL(RMSTRIPE)
      IF (IERR.NE.0) THEN
         PRINT *,"ERROR DELETING OLD STRIPED FILE=",IERR
         GOTO 999
      ENDIF
C Fill up array with consecutive REAL (integers)
      DO I=1,LENDAT
        A(I)=REAL(I)
      ENDDO
      WRITE(SHELARG,540) STRIPEFILE
540   FORMAT("WRITING OUT DATA TO ",A50)
      PRINT *,SHELARG
      CALL DFREAD(MYNAME,NPART,PART1,IFREE,IERR)
      IF (IERR.NE.0) THEN
         GOTO 999
      ENDIF
      IF (NPART.EQ.0) THEN
         PRINT *,"/USR/TMP IS NOT STRIPED ON THIS SYSTEM"
         GOTO 999
      ENDIF
C STRIPE THE FILE ON ALL THE SECONDARY PARTITIONS
      DO I=PART1,PART1+NPART-1
        IPART=I-PART1+1
C We will use assign/FFIO to do the striping.  Call assign
C from the program so that you can change the environment.
C Clean up the old assign.
        CALL assign("assign -R")
C This routine will format the partition argument for the
C assign command.
        CALL PARTFMT(PARTARG,IPART,IFREE)
C Create the FORMAT statement that creates the assign
C ARGUMENT
        WRITE(SHELARG,501) IPART
```

```
501   FORMAT('("assign -p ", ',I2,'A3,',
     1  '" -N ",I6, " -Q 128 -F BUFA:128:",I2.2," U:20")')
        WRITE(ASGARG,SHELARG) PARTARG(1:IPART),NBLOCK,IPART
        PRINT *,ASGARG
        CALL assign(asgarg)
        OPEN (20,FILE=STRIPEFILE,
     1    FORM="UNFORMATTED")
C Time the striped file write.  I've included a close in the timing
C to make sure that the file was actually written out to disk.
        T1=SECONDR()
        WRITE (20) A
        CLOSE(20)
        T2=SECONDR()
        DUR=T2-T1
        RATE=(8*(NBLOCK*512))/(DUR*(10**6))
        PRINT 555,IPART,DUR,RATE
555     FORMAT("WRITING FILE TO ",I2," PARTITIONS TOOK ",F6.3,
     1  " SECONDS RATE=",F7.3, " MBYTES/SEC")
C This command will actually show you how the file is laid out if
C you uncomment it.  It produces a lot of output to the screen!
C        CALL ISHELL("/ETC/FCK -B /USR/TMP/STRIPE.OUT")
C Record the maximum I/O rate
        IF(IPART.EQ.1) THEN
          DURMIN=DUR
          RATEMAX=RATE
          IPARTMIN=IPART
        ELSE
          IF (RATE.GT.RATEMAX) THEN
            DURMIN=DUR
            RATEMAX=RATE
            IPARTMIN=IPART
          ENDIF
        ENDIF
        IERR=ISHELL(RMSTRIPE)
        IF (IERR.NE.0) THEN
          PRINT *,"ERROR DELETING STRIPED FILE=",IERR
        ENDIF
      END DO
      PRINT 565,RATEMAX,IPARTMIN
565   FORMAT("*** MAXIMUM RATE WAS ",F7.3," MBYTES/SEC ",
     1  "ACHIEVED ON ",I2," PARTITIONS")
999   CONTINUE
      ENDIF
```

```
      CALL BARRIER()
      STOP
      END

      SUBROUTINE DFREAD(MYNAME,NPART,PART1,IFREE,IRET)
C This routine finds out how many secondary partitions there are in
C /usr/tmp on this system and sorts them by the amount of free space
      IMPLICIT NONE
      INTEGER IERR,IRET,NPART,ISHELL,PART1,I,ITEMP,FTEMP,J
C IFREE will contain the partition numbers ordered by amount of
C free space, with the partition with the greatest free space
C first
      INTEGER IFREE(100)
      REAL FREE(100)
C This will be the string that removes the DF file
      CHARACTER*80 RMDF
      CHARACTER*80 SHELARG
      CHARACTER*80 DFREC
      CHARACTER*50 DFFILE
      CHARACTER*16 MYNAME
      CHARACTER*1 SECOND
      IRET=0
      NPART=0
      PART1=0
C Get reasonably unique names for the striped data and DF files
      WRITE(DFFILE,510) MYNAME
510   FORMAT("/USR/TMP/DFLOG.",A16)
      WRITE(RMDF,530) DFFILE
530   FORMAT("RM -F ",A50)
      PRINT *,RMDF
      IERR=ISHELL(RMDF)
      IF (IERR.NE.0) THEN
         PRINT *,"ERROR DELETING OLD DF FILE=",IERR
         IRET=IERR
         GOTO 999
      ENDIF
      WRITE(SHELARG,550) DFFILE
550   FORMAT("DF -P /USR/TMP > ",A50)
      PRINT *,SHELARG
      IERR=ISHELL(SHELARG)
      IF (IERR.NE.0) THEN
         PRINT *,"ERROR CREATING DF LOG=",IERR," EXITING"
         IRET=IERR
```

```
            GOTO 999
         ENDIF
         OPEN(30,FILE=DFFILE)
100      CONTINUE
         READ(30,600,END=999,ERR=900) DFREC
600      FORMAT(A80)
         READ(DFREC,610) SECOND
610      FORMAT(12X,A1)
         IF (SECOND.EQ."S") THEN
           NPART=NPART+1
           READ(DFREC,620) IFREE(NPART),FREE(NPART)
620        FORMAT(10X,I2,32X,F5.1)
C           PRINT*,"PART#",IFREE(NPART)," FREE=",FREE(NPART)
            IF(PART1.EQ.0) THEN
              PART1=IFREE(NPART)
            ELSE
              DO I=1,NPART-1
                IF(FREE(NPART).GT.FREE(I)) THEN
                  FTEMP=FREE(I)
                  ITEMP=IFREE(I)
                  FREE(I)=FREE(NPART)
                  IFREE(I)=IFREE(NPART)
                  DO J=NPART,I+2,-1
                    FREE(J)=FREE(J-1)
                    IFREE(J)=IFREE(J-1)
                  ENDDO
                  FREE(I+1)=FTEMP
                  IFREE(I+1)=ITEMP
                ENDIF
              ENDDO
            ENDIF
         ENDIF
         GOTO 100
900      CONTINUE
         PRINT *,"ERROR READING ",DFFILE
         IRET=-1
999      CONTINUE
         PRINT *,NPART," TOTAL SECONDARY PARTITIONS"
         DO I=1,NPART
C          PRINT*,"ORDERED PART#",IFREE(I)," FREE=",FREE(I)
         ENDDO
         IERR=ISHELL(RMDF)
         IF (IERR.NE.0) THEN
```

```
          PRINT *,"ERROR DELETING DF FILE=",IERR
        ENDIF
        RETURN
        END

        SUBROUTINE PARTFMT(PARTARG,IPART,IFREE)
C This routine will write a string containing the IPART secondary
C partitions with the most space suitable for an assign command.
        IMPLICIT NONE
        CHARACTER*3 PARTARG(IPART)
        INTEGER IPART,I
        INTEGER IFREE(IPART)
        DO I=1,IPART-1
          WRITE(PARTARG(I),500) IFREE(I)
500     FORMAT(I2.2,":")
        ENDDO
        WRITE(PARTARG(IPART),510) IFREE(IPART)
510     FORMAT(I2.2," ")
        RETURN
        END
```

Glossary [6]

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 CF90 `!DIR$ 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 146.

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.

derived type

In Fortran, a user-defined type, not an intrinsic type. It requires a type definition to name the type and specify its components. The components may be of intrinsic or user-defined types.

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 33, 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. The predefined variable `PVMALL` represents all of 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 PVMFGETINST(3) routine or the return value from the PVMFJOINGROUP(3) routine.

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:

```
DO I = 1,N
  A(I) = B(I) * 2.0 + VAL * C(J)
ENDDO
```

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 151.

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:

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

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 144.

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 77.

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:

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

After the references to A(1) and B(1), the page for A and the page for B are in memory. When C(1) 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(2) is referenced, but A's page is now out of memory, so the operating system removes B's page. Likewise, the reference to B(2) 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 10^{13} 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.

Index

32–bit data
SHMEM, 55

A

active set
definition, 141
add operation
and splitting loops, 99
advantages of unrolling, 77
aggressiveness levels, 105
Apprentice timing tool, 21
array
how elements are stored, 86
operations on, SHMEM, 71
padding, 88
rearranging dimensions, 85, 101, 102
array passing routines, 122
arrays
merging with SHMEM, 68
padding for loop splitting, 100
asynchronous receive
definition, 141
asynchronous send, SHMEM, 44
ATM networks, 19
atomic operation
definition, 141
atomic swap
definition, 141
atomic swap, SHMEM, 70
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, 47
in SHMEM program, 51
barriers
avoiding in PVM, 32
benefits of unrolling, 77
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, 66
buffer
definition, 142
buffering, double, 125
buffers
setting size for sequential I/O, 133
buffers, send
allocating in PVM, 25
buffers, stream, optimizing, 95
bypassing cache, 113

C

cache
bypassing, 113

coherence, 5
conflict, reducing, 88
data and secondary, 5
data, pad example, 88
how it works, 9
load and store timings, 19
miss, 9, 11
optimization, 85
reuse, 85
secondary, pad example, 92
cache coherence
definition, 142
cache conflict
illustration, 89
cache hit
definition, 142
cache layer, FFIO
with random access I/O, 133
cache, system
bypassing, 134
cache-aligned data
definition, 142
CACHE_ALIGN directive, 5, 90
CACHE_ALIGN directive, SHMEM, 56
CACHE_BYPASS directive, 113
channels feature not available, 23
clock period
definition, 143
clock, real time, 21
collective routine
definition, 143
commit
definition, 143
common block padding, 88
common blocks
padding for loop splitting, 100
communicator
definition, 143
compile time
increases with loop splitting, 100
CONCURRENT directive, 79
conditional OPEN statements, 120
converting data, 123
converting data, PVM, 26

D

dangers of loop splitting, 101
data
32–bit in SHMEM, 55
32–bit optimization, PVM, 26
32–bit packing, PVM, 26
broadcast with SHMEM, 66
conversion, 123
dependencies, 87
gather/scatter, PVM, 39
gather/scatter, SHMEM, 62
initializing and packing, PVM, 30
movement, 9
reuse, 26
streams, 8
stride-1, PVM, 28
strided, SHMEM, 58
transfer, SHMEM, 44
data cache, 5
definition, 143
data conversion, PVM, 26
data flow, 7
data items
minimizing, 127
data transfer
comparing PVM and SHMEM, 3
derived type
definition, 143
differences, to PVM on CRAY T3D systems, 23
dimensions, padding, 93
dimensions, rearranging, 101, 102
direct access I/O, 134
direct-mapped cache
definition, 144
directive
pipelining, 79
disk mirroring
definition, 144

disk sector boundaries
  for I/O requests, 124
disk striping
  automatic, 134
  definition, 144
disk support, 16
distributed I/O, 130
division operations, 106
division outside of loop, 106
division, how to avoid it, 106
division, IEEE, 107
double buffering, 125
DRAM, same as local memory, 7

E

E registers, 4
edit descriptors
  for character data, 128
envelope
  definition, 144
ESCON tapes, 18
EtherNet, 18
eureka
  definition, 144

F

fan-out
  definition, 145
fan-out distribution, PVM, 33
FDDI network, 18
FFIO
  description, 128
Fiber Channel disks, 16
Flexible File I/O
  description, 128
flow of data, 7
flushing cache
  definition, 145
formatted I/O
  optimizations, 126
  reducing, 126
formatting manually, 127
functional unit
  and pipelining, 84
functional units, 8

G

gather data, SHMEM, 62
gather operation
  definition, 145
gathering data, PVM, 39
GET_D_STREAM routine, 105
GigaRing
  definition, 145
GigaRing network, 15
global I/O, 130
glossary
  description, 1
grmview example, 128
group
  definition, 145
grouping statements, 103
GSEG
  definition, 145

H

hardware
  illustration, 7, 8
hardware overview, 3
HIPPI disks, 17
HIPPI network, 19

I

I/O, 117
I/O from a single PE, 121

I/O requests
  using large, 126
I/O strategies, 117
IEEE division, 107
IF statement, splitting loop with, 98
individual routine
  definition, 146
initializing data, PVM, 30
inner loop trip count, maximizing, 101
instance number, 146
intrinsic routines
  vectorized, 111
invariant references, maximizing, 86
IPI-2 disks, 17
IRTC timing tool, 21
IVDEP directive, 112
  and pipelining, 80
  and vectorization, 110

L

large transfers
  how handled by PVM, 24
latencies
  data and secondary cache, 19
latency, memory
  definition, 146
libfastmvvectorized math routines, 109
line
  definition, 146
local memory
  checking your system for, 128
  definition, 146
local memory optimization, 113
logical PE number, 22
loop
  overlapping
    examples, 81
loop invariant
  definition, 146
loop iterations
  overlapping, 78
loop splitting
  can change program, 101
  examples, 97
loops
  identifying for pipelining, 80
loops, splitting, 96
loops, unrolling, 77

M

memory
  checking your system for, 128
  performance information, 19
memory overview, 4
memory-resident FFIO layer
  with random access I/O, 134
memory-resident I/O
  description, 128
merging arrays, SHMEM, 68
Message Passing Interface
  definition, 147
message size
  finding and changing with PVM, 24
message size, PVM, 24
message-passing system
  definition, 147
  PVM, 2
Mflops, or megaflops
  definition, 147
microprocessor, description, 15
mirroring, disk, 16
mixing send and receive routines, PVM, 30
MPI
  definition, 147
multicast operation
  definition, 147
multicast vs. broadcast, PVM, 33
multiple file, multiple PE I/O, 120
multiply operation
  and splitting loops, 99
MY_PE

two versions of, 46

N

N$PES
versus NUM_PES and SHMEM_N_PES, 47
NaN
definition, 147
nanosecond
definition, 147
network illustration, 16
network overview, 15
network protocols, 18
NEXTSCALAR directive, 110
nonblocking receive
definition, 147
nonblocking receive, PVM, 31
nonstandard, SHMEM, 43
NOSPLIT directive, 96

O

organization of manual, 1
output stream, 13
overhead
by grouping statements, 103
definition, 147
SHMEM, 44
overindex command-line option, 101
overindexing
dangerous with loop splitting, 101
definition, 148
overlapping loop iterations, 78

P

packing data, PVM, 30
pad array
definition, 148
padding, 88
done by compiler, 93
illustration, 90
padding for loop splitting, 100
Parallel Virtual Machine (PVM), 23
PE, 15
logical number, 22
physical number, 22
performance
SHMEM, 2
peripherals overview, 15
PEs
definition, 148
physical PE number, 22
pipelined, division is not, 106
pipelining, 78
command-line options, 79
definition, 150
directive, 79
how it works, 81
portability
PVM, 2
SHMEM, 2
portability, SHMEM, 43
powers of 2, 46
prescheduling division, 107
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, switching to SHMEM, 48
PVM_DATA_MAX environment variable, 24
PvmDataDefault, 25
PvmDataInPlace, 25

PvmDataRaw, 25, 28
PVMFGATHER routine, 39
PVMFGETOPT, use of, 24
PVMFINITSEND routine, 25
PVMFMCAST routine, 33, 34
PVMFNRECV routine, 31
PVMFPRECV routine, 28
PVMFPSEND routine, 28
PVMFREDUCE routine, 37
PVMFSCATTER routine, 39

R

random access I/O, 134
rank
  definition, 148
read ahead
  definition, 148
real-time clock, 21
rearranging array dimensions, 85
receiving stride-1 data, PVM, 28
reciprocal multiplication, 106
reduction
  definition, 149
reduction routines, SHMEM, 71
reductions, PVM, 37
remote memory, 4
reorder an array, SHMEM, 62
reusing data, 26

S

scatter data, SHMEM, 62
scatter operation
  definition, 149
scattering data, PVM, 39
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, 123
set-associative cache
  definition, 149
SET_D_STREAM routine, 105
shared memory (SHMEM), 43
SHMEM, 43
  background, 2
  definition, 149
SHMEM_BROADCAST routine, 66
SHMEM_FCOLLECT routine, 68
SHMEM_GET32 routine, 55
SHMEM_GET64 routine, 44
SHMEM_INT4_FADD function, 70
SHMEM_INT4_FINC function, 70
SHMEM_INT4_SUM_TO_ALL routine, 74
SHMEM_IXGET routine, 62
SHMEM_IXPUT and SHMEM_IXGET
  for passing arrays, 122
SHMEM_IXPUT routine, 62
SHMEM_MY_PE routine, 47
SHMEM_N_PES routine, 47
SHMEM_PUT32 routine, 55
SHMEM_PUT64 routine, 44
SHMEM_REAL8_MIN_TO_ALL routine, 72
SHMEM_REAL_IGET routine, 58
SHMEM_REAL_IPUT routine, 58
SHMEM_SWAP function, 70
shpalloc routine, 47
single file, multiple PE I/O, 118
single-PE optimizations, 77
size of memory, 4
size of message
  finding and changing with PVM, 24
size of message, PVM, 24
SM pool
  definition, 150
software pipelining, 78
  definition, 150
split command-line option, 96

SPLIT directive, 96
splitting loop with IF statement, 98
splitting loops, 96
stack variable, 47
statistics
  memory performance, 19
strategies for I/O, 117
stream
  definition, 150
  reducing startup costs, 102
stream buffers
  setting aggressiveness level, 105
stream buffers, optimizing, 95
stream references, maximizing, 101
streams
  example of, 8
  reducing number, 102
stride
  definition, 150
stride-1
  streams, maximizing, 86
stride-1 data, PVM, 28
strided data, SHMEM, 58
striping, disk, 16
  automatic, 134
stripmining
  definition, 150
stripmining example, 98
subtract operation
  and splitting loops, 99
swap, atomic, SHMEM, 70
symmetric
  definition, 150
symmetric array, SHMEM, 47
SYMMETRIC directive, SHMEM, 47
synchronization
  avoiding in PVM, 32
  definition, 151
  minimizing with receive, PVM, 36
  PSYNC array, SHMEM, 66
  PVM, 2, 32
  SHMEM, 2
synchronization, SHMEM, 51
system cache
  bypassing, 134
system call I/O, 124

T

tapes supported, 17
task
  definition, 151
task identifier, PVM, 36
TCP/IP network, 19
thrashing
  definition, 151
tiling
  definition, 152
time to compile
  increases with loop splitting, 100
timing your code, 21
timings
  memory operations, 19
transfer
  32–bit data, PVM, 26
  of data, SHMEM, 44
transfer of data
  comparing PVM and SHMEM, 3
transfer, large
  how handled by PVM, 24

U

ulp
  definition, 152
unformatted I/O, 122
UNROLL directive, 78
unrolling
  by the compiler, 78
  command-line option, 78
  definition, 152
  outer loops, 78
unrolling loops, 77

V

variable
  stack, 47
VECTOR directive, 110
vectorization, 109
  command-line options, 110

W

work while waiting
  PVM, 31
write behind
  definition, 152