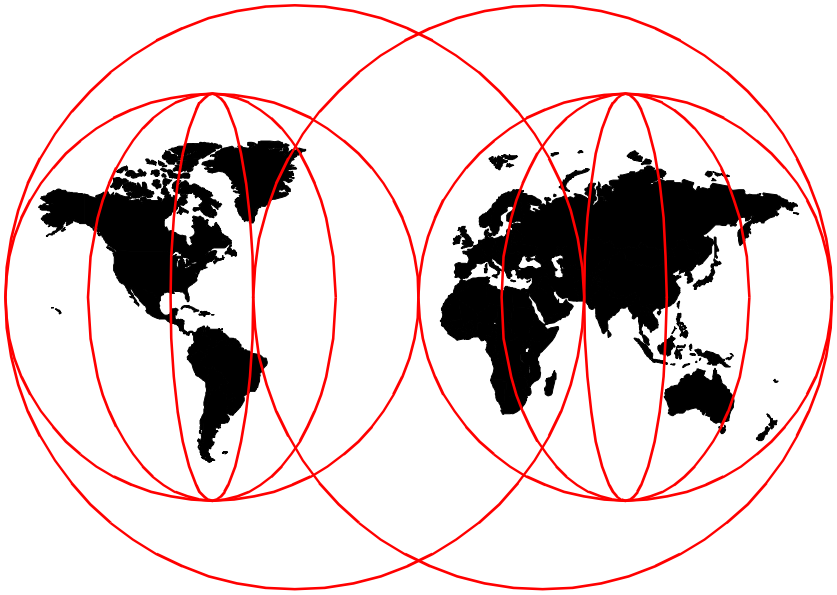
Scientific Applications in RS/6000 SP Environments

Marcelo Barrios, Stefan Andersson, Gyan Bhanot, John Hague, Frank Johnston, Swamy Kandadai, David Klepacki, John Levesque, Jarek Nieplocha, Frank O'Connell, Farid Parpia, Christoph Pospiech, Richard Treumann, Jim Tuccillo, Bob Walkup



International Technical Support Organization

www.redbooks.ibm.com

SG24-5611-00

SG24-5611-00

International Technical Support Organization

Scientific Applications in RS/6000 SP Environments

December 1999

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix A, “Special notices” on page 225.

First Edition (December 1999)

This edition applies to Version 3, Release 1, Modification 1 of Parallel System Support Programs for AIX (5765-D51) and to Version 2, Release 4 of IBM Parallel Environment for AIX (5765-543) for use with the AIX Operating System Version 4, Release 3.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JN9B Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

The announcement of POWER3-based Thin and Wide nodes in early 1999, along with the addition of High nodes this fall, positions the RS/6000 SP with a new and powerful offering for the scientific and technical community.

This redbook provides a description of the POWER3 SMP architecture exploited by the new POWER3-based nodes and performance information for standard benchmarks, such as LINPACK, NAS 2, and Spec95.

This redbook offers hints and tips as well as sample code for various aspects of parallel programming on POWER3 SMP architectures. Discussions of distributed memory, threads, MPI, OpenMP, LAPI, and several other facilities for developing parallel applications will help the reader understand and use these tools and take advantage of the parallel nature of the RS/6000 SP empowered by these new nodes.

Due to the technical nature of the book, it will be most valuable to readers with some background in parallel computing who are familiar with SMP and parallel architectures.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Marcelo Barrios is a project leader at the International Technical Support Organization, Poughkeepsie Center. He has been with IBM since 1993 working in different areas related to RS/6000. Currently, he is focusing on RS/6000 SP technology by writing redbooks and teaching IBM classes worldwide.

Stefan Andersson is an Advisory Engineer/Scientist at IBM Poughkeepsie. He has an MS in Mathematics from the University of Heidelberg. He began his work with IBM at the IBM Scientific Center, Heidelberg in 1990. He has been involved in parallel computing on the IBM RS/6000 SP since 1992. Currently, he is a member of the technical benchmark team at IBM Poughkeepsie. His areas of expertise include performance tuning for the POWER2 and POWER3 architectures and tuning and coding for distributed and shared memory on the IBM RS/6000 SP.

Gyan Bhanot earned his PhD in Theoretical Physics from Cornell University in 1979. He worked on simulations in Particle Physics and Statistical Mechanics at Brookhaven National Labs, The Institute for Advanced Study, Princeton, CERN, Geneva, and ITP Santa Barbara before joining the Supercomputer Institute at Florida State University and concentrating on parallel computers and computation. He joined IBM Research as a Research Staff Member in 1994 after working for five years at Thinking Machines Corporation. His current interests are in Parallel Algorithms/Languages, Computational Biology, and Computational Finance.

John Hague is an IBM IT Consultant in the UK. He obtained a PhD in High Energy Physics at University College, London, and worked in this field at the Rutherford Laboratory in the UK and the Lawrence Livermore Laboratory in Berkeley until he joined IBM in 1970. John was assigned to the IBM ITSO in Poughkeepsie in 1985 to provide worldwide technical support for the IBM Vector Facility. Since then, he has worked exclusively in the scientific and technical field and has considerable expertise in vectorizing, parallelizing, and tuning scientific programs, particularly in the areas of Petroleum and Weather Forecasting.

Frank Johnston works in the SP Performance Department at IBM, Poughkeepsie, and specializes in the performance of standard floating-point benchmarks on the SP.

Swamy Kandadai is a computational chemist by training and has been published extensively on protein and nucleic acid simulations using parallel processing. He has been involved with parallel computing in IBM since 1985, initially working on the Loosely Coupled Attached Processors (LCAP) and on RS/6000 SP. Currently, he is a member of the performance group in the RS/6000 division responsible for SP performance.

David Klepacki holds a PhD degree in Theoretical Nuclear Physics from Purdue University. He began working in IBM POWER Parallel Systems Group in 1991 as a computational physicist and scientific applications specialist with an emphasis on performance benchmarking. Today, he is the Assistant Director of the Advanced Computing Technology Center at the IBM T.J. Watson Research facility in Yorktown Heights, New York. His current interests include scalable algorithms and portable high-performance computing tools.

John Levesque is the Director of the Advanced Computing Technology Center centered at IBM Research. Mr. Levesque has over 25 years experience in High Performance Computing. At IBM, Mr. Levesque leads a group focusing on HPC computing within IBM and supply solutions for customers porting numerically-intensive applications to IBM SP architecture.

The ACTC offers programming tools, optimized libraries, and applications as well as training on optimizing scientific applications for shared/distributed parallel computer systems.

Jarek Nieplocha is a Staff Scientist at Pacific Northwest National Laboratory. His recent research has been in interprocessor communications with an emphasis on one-sided communication, shared memory programming, and high-performance I/O. He received two best paper awards at international conferences in the parallel computing area and an R&D-100 award. As a member of MPI Forum, he participated in the definition of the MPI-2 standard.

Frank O'Connell is a Senior Engineer at the RS/6000 Server Development group in Austin, TX. He has been a member of the RS/6000 high-performance processor development effort since 1992. He has focused on scientific and technical computing performance within IBM for the past 13 years including microprocessor and systems design as well as application performance and tuning. Frank earned a BSME from the University of Connecticut and an MSE in Engineering-Economic Systems from Stanford University.

Farid Parpia is Senior Scientist in the Scientific, Technical, and Engineering Benchmarking Group at IBM Poughkeepsie. He has a PhD in Atomic Physics from the University of Notre Dame and still enjoys writing software for the modeling of atomic and molecular systems using the Dirac theory of the electron.

Christoph Pospiech is an IT specialist in Germany. He has 11 years of experience in high-performance computing. He has done message passing parallel computing ever since this was conceived within IBM. He holds a doctor rerum naturae in Mathematics from Heidelberg University.

Richard Treumann is an Advisory Software Engineer and MPI development team leader at IBM Poughkeepsie. He has a MS in Computer Science and has worked in software development since 1984. He participated in developing the MPI standard as an IBM representative on the MPI-2 Forum. His primary areas of expertise include compiler parsers, optimizers, and code generators as well as message passing systems for parallel programming.

Jim Tuccillo is an atmospheric scientist by training. He has attended Cornell University, Old Dominion University, and Johns Hopkins University. Jim has been involved in the development of Numerical Weather Prediction (NWP) Models on high-performance vector, parallel vector, and distributed memory systems since 1980. Jim has worked in the NWP development labs of the U.S. Weather Service and NASA where he has been involved in the

development of research and operational NWP codes for weather forecasting in the U.S. Jim currently works for the IBM Global Government Industry organization where he is involved in issues associated with NWP and high-performance computing on the IBM SP system. Jim has research interests in the areas of parallel algorithms and parallel programming paradigms for high-performance numerically-intensive computing.

Bob Walkup is a physicist by training. He received a PhD in physics from M.I.T. and has been in the physical sciences department at the IBM Thomas J. Watson Research Center for the past 17 years. He has been actively involved with parallel programming since the initial days of IBM scalable parallel systems.

Thanks to the following people for their invaluable contributions to this project:

Yoshinori Shimoda
IBM Japan

Dr. William G. Tuel, Jr.
IBM Poughkeepsie

Henry Zongaro
IBM Canada

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Please send us your comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks evaluation” on page 243 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Chapter 1. Introduction

The IBM RS/6000 SP is a general-purpose scalable parallel system based on a shared-nothing architecture. Each processing unit, or node, runs its own instance of the standard AIX operating system, and it can connect to any other node through a high-bandwidth low-latency network (SP Switch) or through any other TCP/IP network. Generally available SP systems range from one to 128 nodes, although much larger systems (over 512 nodes) have been built and delivered for the past few years.

The SP is a low-cost high-performance massively-parallel-processing (MPP) system with the tools necessary to exploit the combined capacity of hundreds of nodes for solving large problems. This ability is not only beneficial for scientific and technical applications, but also for commercial environments, such as business intelligence, data mining, or online transaction processing (OLTP).

In addition to the MPP capabilities, the RS/6000 SP provides high scalability to applications for both computation and data, far beyond what is possible with conventional symmetric multiprocessing (SMP) systems.

In order to properly address these diverse applications and environments, the RS/6000 SP is designed around three key areas: Programming models, flexible architecture, and system availability. In this redbook, we concentrate on the first of these three key areas: Programming models, and, within programming models, we concentrate on technical computing.

Programming models for commercial environments are somewhat similar to scientific and technical applications, but their scope is broader. The two main areas where massive parallel processing is a key factor in commercial environments are OLTP and data mining, where the data shipping or shared nothing model for parallel transactions and query processing are a perfect fit for the RS/6000 SP. Even for the shared data model, the SP offers high performance SMP nodes, which allows applications to exploit both MPP and SMP environments.

In technical computing, there are essentially three programming models used in large scale systems: The message-passing programming model, the shared-memory programming model, and the data parallel programming model.

Each of these three models offers advantages and disadvantages for parallel applications as follows:

- For the message-passing programming model, processes have their own address spaces, and they share data through messages; the source process explicitly sends a message, and the target process explicitly receives this message. As discussed in Chapter 3, “Distributed memory” on page 25, although synchronization may seem implicit in this send and receive model, most current implementations of this model, such as MPI or LAPI, offer asynchronous messaging, one-sided communication protocols, or active messaging.
- In shared-memory programming models, processes in an application share a common address space; so, there is no explicit action to share data across multiple CPUs. Since all CPUs have access to the same memory address space, the synchronization between them must be explicit, that is, either done by the compiler, compiler directives, or explicit calls. Chapter 4, “Shared memory” on page 105, provides a detailed discussion of two aspects of shared memory parallelization: The shared-memory directives in OpenMP with special emphasis on the new POWER3 architecture available on the RS/6000 SP and multithread programming with a focus on POSIX threads (Pthreads).
- The third programming model is the data parallel model. This model is supported by data parallel languages, such as High Performance Fortran (HPF). Programs are written using sequential Fortran to specify the computations on the data (using either interactive constructs or the vector operations provided by Fortran 90 and Fortran 95) and data mapping directives to specify how large arrays should be distributed across processes.

Another interesting implementation of this data parallel model is MPI-IO, which is part of the MPI-2 standard discussed in Chapter 6, “Input/output” on page 169. Through MPI-IO, processes can access non-overlapping regions of a common set of data. This data parallel model has the advantage of freeing the user from the need to explicitly distribute global arrays onto local arrays and change names and indices accordingly allocating buffers for data that must be communicated from one node to another and inserting the required communication calls or the required synchronizations.

In addition, with the introduction of more powerful SMP nodes, as discussed in Section 1.1, “POWER3-based nodes” on page 3, the RS/6000 SP provides programmers with several alternatives for exploiting the multiple CPUs available on each node. By using multiple nodes and multithreading the MPI tasks in each node, programmers are better able to exploit RS/6000 SP environments by using hybrid programming models.

When applications are parallelized for speed improvements using any programming model, their I/O bandwidth requirements also increase proportionately. The proper tuning of local I/O subsystems is critical to good performance as is discussed in Chapter 6, “Input/output” on page 169. However, even with the proper tuning, local I/O cannot cope with the I/O requirements of parallelized applications.

Standard distributed file systems, such as NFS, AFS, or DFS, do not satisfy the extremely high bandwidth to file data required by some I/O-intensive applications. Fortunately, GPFS addresses these unique requirements by providing a highly-scalable and highly-available parallel file system, which, along with MPI-IO, may prove to be an excellent alternative to distributed I/O and data sharing. The MPI-IO implementation is discussed in Section 6.5, “MPI-IO” on page 214.

1.1 POWER3-based nodes

POWER3-based nodes are the latest addition to the RS/6000 SP. Combining the POWER3 in SMP configurations and the tools available for parallel programming, the RS/6000 SP provides the best platform for CPU- and I/O-intensive applications.

This section provides an overview of the POWER3 SMP High node and benchmark results from pre-GA environments, which position these new nodes in the high end of the RS/6000 family.

1.1.1 Hardware architecture

The POWER3 SMP High node is a high-performance symmetric multiprocessing (SMP) node for the RS/6000 SP. It is intended to perform as a high-performance technical and workgroup server. The POWER3 SMP High node drawer can accommodate up to four processor cards, each with two processors per card, for a total of eight processors (valid configurations are two, four, six, or eight processors). The drawer can also host up to four memory cards for a maximum of 16 GB. The minimum memory configuration is 256 MB.

The processors within POWER3 SMP High nodes are 222 MHz POWER3 processors with a 4 MB direct-mapped L2 cache also clocked at 222 MHz. Each processor is attached to the memory subsystem via a 16-byte wide 111 MHz 6XX bus.

In its base configuration, the POWER3 SMP High node has two 32-bit 33 MHz PCI slots, an integrated service processor, a 10BaseT Ethernet

interface, three media bays, and two 6-pack DASD SLEDS. It can support up to six Remote I/O expansion drawers (a seventh Remote I/O unit is supported via PRPQ).

The POWER3 SMP High node memory subsystem consists of an address and data-switched memory subsystem. Figure 1 shows the overall structure of the POWER3 SMP High node design.

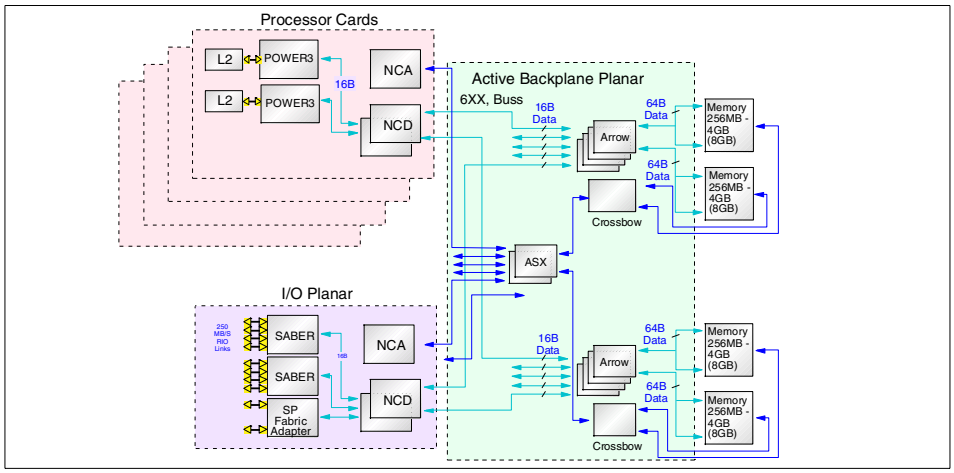


Figure 1. POWER3 SMP High node architecture

Each processor card has two 16-byte data bus connections that run at 111 MHz for a peak bandwidth of 3.5 GBps. The memory subsystem can provide two data transfers to each of the four processor cards concurrently for a peak aggregate rate of 14.2 GBps when fully populated with four memory cards. Memory is interleaved both within and across cache lines for maximum performance.

Table 1 explains the functionality of each of the chips appearing in Figure 1 on page 4.

Table 1. POWER3 internal chip functionality

Chip	Function
POWER3	Processor
NCA	Node Controller: Address. Handles address portion of all transactions issued by processors and I/O elements and coordinates data transfers among processors, memory, and I/O elements.
NCD	Node Controller: Data. Contains paths for interprocessor and processor-memory data transfers. Forms a portion of the data switch.
ASX	Address Switch.
Crossbow	Memory Controller. Controls data transfers across paths within the Arrow chips.
Arrow	Switch: Carries data to and from memory and between nodes. Forms a portion of the data switch.
Saber	Portal to I/O subsystems.
SP Fabric Adapter	Portal to inter-High node communication interconnect.

1.1.1.1 Memory subsystem

Figure 2 on page 6 shows a block diagram of the physical hierarchy for system memory. Memory is packaged on up to four cards connected to the four memory ports in the data cross point switch. Memory cards utilize 128 MB SDRAM modules. Each card can contain up to 4 GB of memory or up to 16 GB for the compute node. 1 GB of memory comes in the base system design. Memory can be added in 1 GB increments with the addition of cards and pluggable industry-standard DIMMs. This provides a highly-configurable and upgradable offering, which will grow with the application requirements.

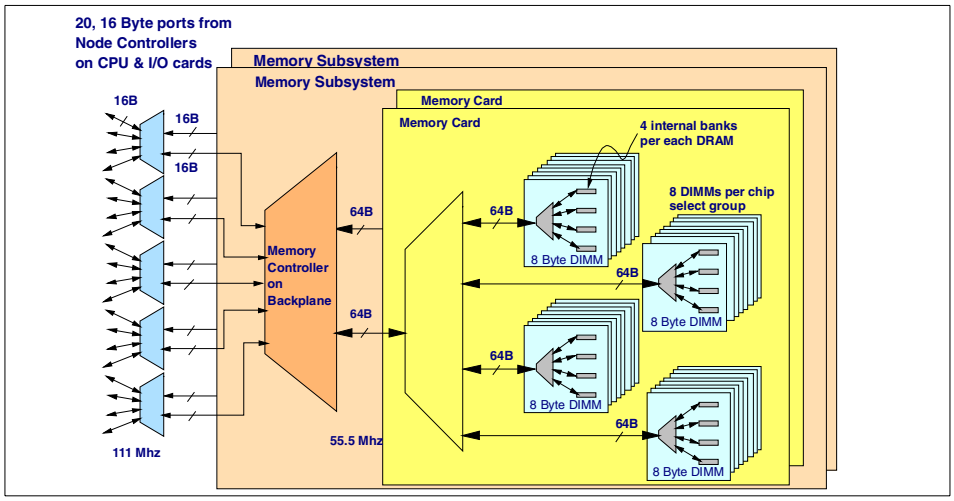


Figure 2. Memory physical hierarchy

Each memory subsystem is capable of accepting an address transaction every cycle at 111 MHz. Each memory data port is 64 bytes wide and operates at 55.5 MHz, which translates to a delivered bandwidth of 14.2 GBps.

1.1.1.2 I/O subsystem

Figure 3 on page 7 shows a block diagram of the I/O subsystem. The figure shows Sabers, which serve as portals to disk and communication subsystems, and an SP Switch adapter, which allows a compute node to connect to the SP Switch to form a multinode system.

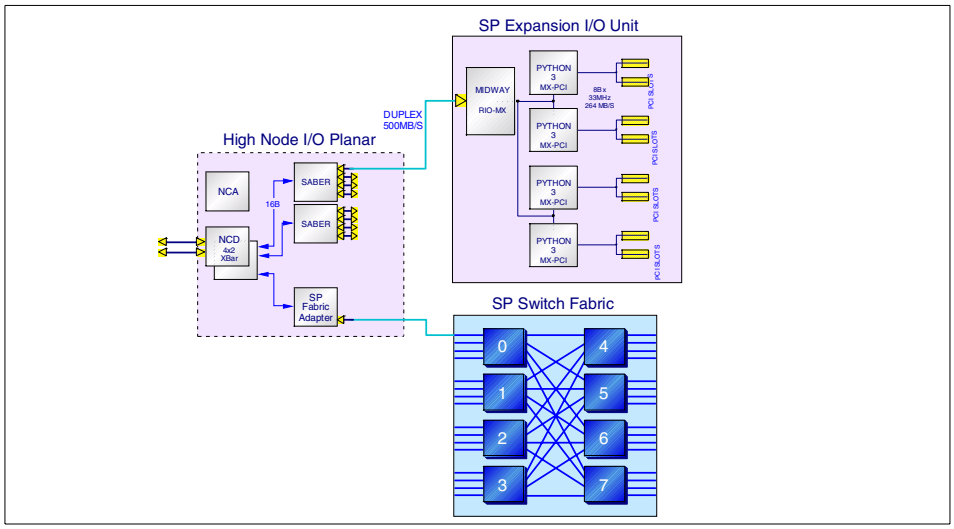


Figure 3. POWER3 SMP High node I/O topology

The High node I/O consists of four 64-bit PCI slots and one 32-bit PCI slot operating at 33 MHz. It provides integrated Ultra SCSI, 10/100 Ethernet, parallel port, and three serial ports. In addition, there are six remote I/O (RIO) connections. These connections allow SP Expansion I/O Expansion Units to be connected to the High node providing incremental growth for applications requiring more I/O connectivity.

The RIO ports operate full duplex at 250 MBps in each direction with cables up to 15 meters long. This provides outstanding I/O bandwidth to each RIO expansion node. With six SP Expansion I/O Units connected, a user can obtain a node complex with 53 PCI slots and 26 DASD bays. This combination of High node and SP Expansion I/O Units into a node complex provides the highest I/O configuration yet on the RS/6000 SP system.

Chapter 2. Performance

This chapter provides a comparison of the performance of POWER3 SMP High to that of POWER3 SMP Thin/Wide. POWER3 SMP High runs at 222 MHz while POWER3 SMP Thin/Wide runs at 200 MHz.

The last section of this chapter discusses how to access memory effectively in order to achieve better performance and to illustrate how multiple processors of the POWER 3 SMP High Node are able to achieve high efficiency when accessing memory concurrently.

Note

The POWER3 SMP High node runs were done on a system having pre-GA hardware and software; so, there is no guarantee that these results will be reproducible on GA systems.

The LINPACK and NAS 2 runs were done on POWER3 SMP Thin/Wide nodes, which did not have the firmware patch described in the IBM Service Bulletin dated July 8, 1999.

A pre-GA version of the ESSL library was used to run the tests presented in this section. Results may vary with the GA version.

2.1 LINPACK and NAS 2 benchmarks

The clock frequency suggests that POWER3 SMP High nodes should always outperform POWER3 SMP Thin/Wide nodes. However, the cost of an unprefetched cache miss on POWER3 SMP High nodes is more than double the cost on POWER3 SMP Thin/Wide nodes. The lower memory latency favors POWER3 SMP Thin/Wide nodes, but the higher clock frequency favors POWER3 SMP High node; so, in general, the single CPU performance of the two machines is similar.

Those applications that efficiently use all CPUs in an SMP should run faster on a POWER3 SMP High node than on a POWER3 SMP Thin/Wide node. The POWER3 SMP High node has eight CPUs compared to the two CPUs in a POWER3 SMP Thin/Wide node. A POWER3 SMP High node can have up to 16 GB of memory, which allows it to run problems that are too big to run on a POWER3 SMP Thin/Wide node.

This section illustrates these points using performance data for the standard LINPACK and NAS 2 benchmarks. These runs were done on POWER3 SMP

High nodes with eight CPUs, 8 GB of memory (128 MB DIMMs), and four memory cards per node. This configuration provides the maximum memory bandwidth.

2.1.1 LINPACK

The LINPACK benchmarks all solve systems of linear equations. Double precision (eight-byte) floating point data is used in all cases. A single precision version of LINPACK exists, but it is obsolete. The smallest LINPACK benchmark, LINPACK DP, solves a system of 100 equations. The source code provided by Jack Dongarra cannot be changed; so, all optimization must come from the Fortran compiler and preprocessor. Since this benchmark is so small, it is only run on a single processor.

The next largest LINPACK problem, LINPACK TPP (Toward Peak Performance), solves a system of 1000 equations. Tuned libraries, such as ESSL, can be used with this benchmark, and the benchmark spends most of its time in ESSL. This problem can be run on multiple CPUs in an SMP node, but it is not large enough to perform well on multiple nodes.

LINPACK HPC (Highly Parallel Computing) can be run on multiple nodes since the problem size is chosen to maximize performance. Larger problems lead to increased efficiency and higher FLOP rates - as long as the problem is not so large as to cause paging. Tuned libraries, such as PESSL and ESSL, are used, and most of the time is spent in these libraries. The benchmark is run with a single MPI task per node. Since PESSL/ESSL can run in parallel within an SMP node, the benchmark is run with only one or two MPI tasks per node. The POWER3 SMP Thin/Wide results were obtained with one MPI task per node, but the POWER3 SMP High results were obtained with two tasks per node.

Table 2 displays LINPACK results.

Table 2. LINPACK results

LINPACK		POWER3 SMP Thin/Wide			POWER3 SMP High		
Variant	**CPU**	**MFlops**	**Size**	**Nodes**	**MFlops**	**Size**	**Nodes**
DP	1	227	100	1	250	100	1
TPP	1	639	1000	1	684	1000	1
	2	1168	1000	1	1247	1000	1
	4				2153	1000	1
	8				3516	1000	1

LINPACK		POWER3 SMP Thin/Wide			POWER3 SMP High		
Variant	CPU	MFlops	Size	Nodes	MFlops	Size	Nodes
HPC	8	5130	22400	4	5540	13000	1
	16	10040	31600	8	11080	27000	2
	32	19920	44800	16	21000	38000	4

2.1.2 Discussion of LINPACK results

LINPACK DP is so small that it fits in L2 cache and scales like the clock frequency. The ratio of the clock frequencies is High_MHz/Thin_MHz = 222/200 = 1.11. The ratio of DP performance is very close: High_DP/Thin_DP = 250/227 = 1.10.

TPP runs seven percent faster on a POWER3 SMP High CPU than a POWER3 SMP Thin/Wide CPU. This is less than the improvement in the clock frequency, presumably because the higher memory latency on POWER3 SMP High can affect the performance obtained for this relatively small problem size. For two CPUs, the comparison is similar: POWER3 SMP High delivers seven percent more performance than POWER3 SMP Thin/Wide.

An eight-CPU POWER3 SMP High node delivers three times the LINPACK TPP performance of the two-CPU POWER3 SMP Thin/Wide node. Considering only the number of CPUs and the increase in clock speed, the ratio would be (8*222)/(2*200) = 4.4. The relatively small problem size of 1000 limits the SMP parallel efficiency of the eight-CPU POWER3 SMP High node.

Larger problems can be run for LINPACK HPC. In this case, an eight-CPU POWER3 SMP High node outperforms 4 POWER3 SMP Thin/Wide nodes, which have a total of eight CPUs. Since the calculation on POWER3 SMP High is done within a single node, it can be done more efficiently, because it can bypass PESSL and MPI. The larger problem size leads to excellent eight-way SMP parallel efficiency on POWER3 SMP High.

For LINPACK HPC calculations across multiple nodes, two MPI tasks per node were run on POWER3 SMP High. Since the xlf SMP run time binds its threads to CPUs, it was necessary to use the bindprocessor subroutine to prevent multiple compute threads from being bound to the same CPU. Shared memory MPI (MP_SHARED_MEMORY=yes) was used for data transfers within each node.

Most of the MPI communication needed for LINPACK HPC is done in "interrupt" mode. Since the messages are long, increasing the hysteresis delay (MP_INTRDELAY=100) allows more data to be processed per interrupt, which improves performance. Since MPI currently supports only 32-bit addressing, and since MPI needs to reserve one of the eight 256 MB data segments for itself, the amount of user data per MPI task is limited to 1.75 GB. (When a 64 bit version of MPI is released, the memory available to a single MPI task will be the amount of physical memory installed on the node where it is running, less whatever is required by AIX, PSSP, etc.)

In order to run larger problems, two MPI tasks per node had to be used. Running larger problems with two MPI tasks per node improved performance, even though increasing the number of MPI tasks leads to additional MPI communication, and running with two MPI tasks per node may increase contention for the switch adapter.

Although multinode performance for equal numbers of CPUs is comparable, the eight-CPU POWER3 SMP High node delivers much more performance than the two-CPU POWER3 SMP Thin/Wide node. For four nodes, the ratio of POWER3 SMP High performance to POWER3 SMP Thin/Wide performance is NH_MFlop/WH_MFlop = 21000/5130 = 4.

The problem size was a larger on POWER3 SMP High, but that should not affect this comparison since the problem size on POWER3 SMP Thin/Wide was large enough to yield nearly maximum performance.

2.1.2.1 NAS 2 benchmarks

The NAS 2 benchmarks are a suite of eight codes developed by NASA Ames to compare performance on different computers. There are five kernels: CG, EP, FT, LU, and MG. These perform computation-intensive operations, such as FFT (FT) and sparse matrix vector multiply (CG). The three simulated applications, BT, LU, and SP, are intended to be more representative of full CFD programs. Each NAS 2 code has three different problem sizes denoted as Class A, B, and C, with A being smallest and C being largest. These are MPI codes, and the source provided by NASA Ames was not modified. Two codes, BT and SP, require that the number of MPI tasks be a perfect square (1,4,9,16, and so on) while CG, FT, IS, LU, and MG require that the number of tasks be a power of 2 (1,2,4,8, and so on).

2.1.3 Serial batch runs

Multiple copies of the single CPU class A problems were run. Since 1 is both a power of 2 and a perfect square, all eight codes could be run. The class A problem was chosen since it is the smallest, and eight simultaneous copies of

each code could easily be run on a POWER3 SMP High node. The POWER3 SMP Thin/Wide node 4 and 8 copy results are estimated, not measured. It is assumed that one would run four copies on POWER3 SMP Thin/Wide nodes by running two sets of two simultaneous copies back to back; so, the time for four copies is assumed to be double the time for two copies. Similarly, the time required to execute eight copies on POWER3 SMP Thin/Wide nodes is assumed to be four times the time to required to execute two copies.

Table 3 displays the run times in seconds as reported by the codes.

Table 3. Run times in seconds (as reported by the codes)

Code	POWER3 SMP High number of copies				POWER3 SMP Thin/Wide number of copies			
	1	2	4	8	1	2	4 (est)	8 (est)
btA	1606	1634	1693	1840	1333	1540	3080	6160
cgA	20.1	20.6	21.7	26.1	17.7	18.1	36.2	72.4
epA	206	206	206	206	226	227	454	908
ftA	68.5	69.8	74.1	79.0	64.7	70.4	140.8	281.6
isA	20.9	21.0	21.2	21.4	21.0	21.5	43.0	86.0
luA	845	865	875	913	755	789	1580	3160
mgA	33.0	33.3	34.5	37.2	27.6	30.8	61.6	123
spA	1104	1118	1158	1248	952	1070	2140	4280

2.1.4 Discussion of serial batch results

Immediately after a reboot, the time for a single copy of luA is about 700 seconds. The time increases to 845 sec after many jobs have been run. This behavior is reproducible, but it is currently not well understood. It may be that the allocation of pages in real memory immediately after a reboot allows this particular code to use L2 cache more efficiently.

For these codes, a single POWER3 SMP Thin/Wide CPU generally outperforms a single POWER3 SMP High CPU. The POWER3 SMP Thin/Wide node is faster for six of the eight codes: BT, CG, FT, LU, MG, and SP.

For BT, the POWER3 SMP Thin/Wide node is actually 20 percent faster than the POWER3 SMP High node. The lower latency on POWER3 SMP Thin/Wide nodes is responsible for the difference. The faster clock on

POWER3 SMP High nodes gives it better performance for the EP and IS codes.

Since the POWER3 SMP High node has eight CPUs, the POWER3 SMP High node can run eight copies of the serial NAS 2 codes much faster than a POWER3 SMP Thin/Wide node, which has only 2 CPUs. Note that the eight copy times on the POWER3 SMP Thin/Wide node are reasonable estimates, not actual measurements. A POWER3 SMP High node runs eight copies of most codes in less than 1/3 of the time estimated on a POWER3 SMP Thin/Wide node.

2.1.5 NAS 2 runs using shared memory MPI

A subset of the NAS 2 codes were run using shared memory MPI on the POWER3 SMP High node. Starting with PSSP 3.1.1, MPI supports a shared memory path for communication between two tasks on the same node. The POE environment variable MP_SHARED_MEMORY was set to yes. This environment variable is new with PSSP 3.1.1. The TB3MX2 adapter allows only four MPI/us tasks per node. These runs required eight MPI tasks on a single node, so the "ip" version of shared memory MPI was used instead of the "us" version. Since MP_EUILIB was set to "ip", MP_WAIT_MODE was set to "poll" (the default for "ip" is "yield").

For more information about MPI/POE environment variables, see *Parallel Environment for AIX: Operation and Use*, SC28-1979.

Those NAS 2 codes that can run with eight MPI tasks were run on single POWER3 SMP High node with eight CPUs using shared memory MPI. Results for class B and C problem sizes are shown. Those codes that would run on a POWER3 SMP Thin/Wide node with two CPUs and three GB of memory were run the same way. MPI/us results for four POWER3 SMP Thin/Wide nodes running PSSP 3.1.0 (no shared memory MPI) are also shown. Table 4 shows the times (in seconds) as reported by the codes.

Table 4. Times (in seconds) as reported by the codes

Code	Four POWER3 SMP Thin/Wide nodes eight CPUs MPI/us	One POWER3 SMP Thin/Wide node two CPUs MPI/shm	One POWER3 SMP High node eight CPUs MPI/shm
cgB	154	574	187
epB	117	477	104
ftB	175	571	162

Code	Four POWER3 SMP Thin/Wide nodes eight CPUs MPI/us	One POWER3 SMP Thin/Wide node two CPUs MPI/shm	One POWER3 SMP High node eight CPUs MPI/shm
isB	10.2	25	7.7
luB	425	1846	433
mgB	19.8	80.8	20.3
cgC	464	1874	553
epC	470	1904	414
ftC	829	N/A	775
isC	41.6	103	31.4
luC	1894	9089	2060
mgC	147	N/A	159

2.1.5.1 Discussion of NAS 2 results with shared memory MPI

Since POWER3 SMP High nodes can have up to 16 GB of memory, the largest class C problem size can be run for all six codes. The memory on POWER3 SMP Thin/Wide nodes is currently limited to 4 GB; so, the FT class C problem, which requires about 7 GB, will not fit on a single POWER3 SMP Thin/Wide node. MG class C requires about 4 GB of memory, but this could not be run since the POWER3 SMP Thin/Wide node had only 3 GB.

In general, the performance of a POWER3 SMP High node with eight CPUs is similar to that of four POWER3 SMP Thin/Wide nodes having a total of eight CPUs. Shared memory MPI helps the performance of FT and IS on POWER3 SMP High nodes. The higher frequency helps the performance of EP and IS on POWER3 SMP High nodes. The other codes run somewhat faster on POWER3 SMP Thin/Wide nodes. However, the performance of a single POWER3 SMP High node is three to four times better than that of a single POWER3 SMP Thin/Wide node.

2.2 Spec95 benchmark

The System Performance and Evaluation Cooperative (SPEC) benchmark (CPU95) consists of 10 FORTRAN 77 programs (SPECfp95) and eight C-language programs (SPECint95). The CFP95 group reflects "numeric-scientific applications", while the CINT95 group are "system or commercial". These benchmarks measure the performance of CPU, memory

system, and compiler code generation. The individual benchmarks are summarized in Table 5.

Table 5. Summary of individual benchmarks

Integer benchmarks: CINT95	
099.go	Artificial intelligence; plays the game "Go"
124.m88ksim	Motorola 88K chip simulator; runs test program
126.gcc	New version og GCC; builds SPARC code
129.compress	Compresses and decompresses file in memory
130.li	LISP interpreter
132.ijpeg	Graphic compression and decompression
134.perl	Manipulates strings (anagrams) and prime numbers in Perl
147.vortex	A database program
Floating point benchmarks: CFP95	
101.tomcatv	A mesh-generation program
102.swim	Shallow water model with 513 x 513 grid
103.su2cor	Quantum physics; Monte Carlo simulator
104.hydro2d	Astrophysics; Hydrodynamical Navier-Stokes equation
107.mgrid	Multi-grid solver in 3D potential field
110.applu	Parabolic/elliptic partial differential equations
125.turb3d	Simulates isotropic, homogeneous turbulence in a cube
141.apsi	Calculates statistics on temperature and pollutants in a grid
145.fpppp	Quantum chemistry
146.wave5	Plasma physics; solves Maxwell's equations on a cartesian mesh

The CPU95 benchmarks are used in speed measurement and throughput measurement. In the speed measurements, the results are expressed as the ratio of the wall clock time to execute one single copy of the benchmark compared to a fixed "SPEC reference time". For the CPU95 benchmarks, a SUN SPARCstation 10/40 was chosen as the reference. The SPECint_base95 and SPECfp_base95 provide the geometric mean when

compiled with conservative optimization for each benchmark while SPECint95 and SPECfp95 give the geometric mean when compiled with aggressive optimization for each benchmark.

In the throughput measurement, several copies of a benchmark are executed. This method is particularly suitable for multiprocessor systems and clusters. The results, called SPECrate, express how many jobs of a particular type (characterized by the individual benchmarks) can be executed in a given time. Therefore, the SPECrate characterizes the capacity of a system for computation-intensive jobs of similar characteristics. Table 6 displays SPEC95 performance.

Table 6. SPEC95 performance

Benchmark	**POWER3 SMP Thin/Wide (200 MHz)**	**POWER3 SMP High (222 MHz)**
SPECfp_base95	27.6	26.1
SPECfp95	30.1	28.2
SPECint_base95	12.5	13.1
SPECint95	13.2	14.2

Table 7 displays SPECrate results.

Table 7. SPECrate results

Number of CPUs	**POWER3 Thin/Wide (200 MHz)**				**POWER3 Thin/Wide (222 MHz)**			
	CFP95		CINT95		CFP95		CINT95	
	Base	Peak	Base	Peak	Base	Peak	Base	Peak
1	251	273	110	130	232	255	111	121
2	480	517	217	236	461	503	224	243
4	937	1008	375	402	900	977	441	472
8	1846	1983	738	794	1679	1849	872	931
16	3724	3984	1588	1716	3331	3611	1647	1801
32	7399	7894	2985	3260	6239	6830	3206	3325
64	14452	15291						
128	26960	28250						

2.3 Accessing memory effectively

In order for a single processor to access memory efficiently, it is important to use the L1 cache efficiently. This is discussed in Section 2.3.1, “Avoiding cache misses” on page 18. Section 2.3.2, “Multiprocessor throughput” on page 20, illustrates how the multiple processors of the POWER 3 SMP High Node are able to achieve high efficiency when accessing memory concurrently.

2.3.1 Avoiding cache misses

The key to avoiding L1 Cache misses is to understand how the Cache works.

In POWER3 SMP High nodes, as in POWER3 SMP Thin/Wide nodes, the L1 Cache is 64 KB in size and consists of 512 "lines". Each line is 128 bytes long and is loaded (and stored) from memory or L2 cache as a whole. The cache is 128-way associative, which means there are four blocks of 128 lines, and each word in memory can be loaded into just one of those blocks and has 128 lines into which it can be loaded. If the address is X, the block it can be loaded into can be determined from the algorithm (in Fortran notation): MOD(MOD(X,128),4).

It is now easy to see how to ruin the performance by creating a pathological situation with the Cache. If data is loaded with a stride of 1024 bytes, it will use all available Cache lines if the data size is 128 Double Precision words. For data sizes greater than this, the performance will be atrocious as shown in Figure 4.

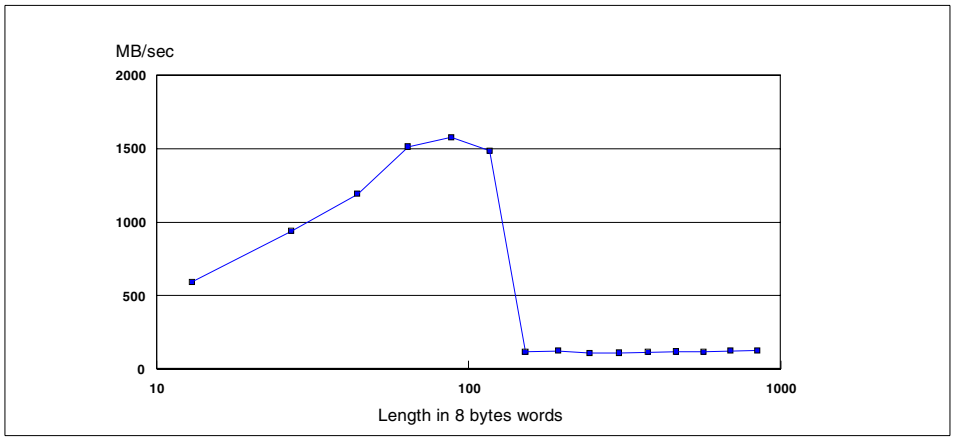


Figure 4. POWER3 SMP High node: Load with stride = 1024 bytes

Data is being loaded from level 2 cache. The access rate is approximately 120 MB/sec, although, since a whole Cache line is loaded for every access, the actual data transfer rate from L2 Cache is (128/8)*120 MB/sec = 1.9 MB/sec. The time taken to access the data is 220/(120/8) = approximately 15 cycles per word.

2.3.1.1 Array transpose

Obviously, to avoid cache misses, large strides should not be used if at all possible. For example, if the stride is caused by accessing a matrix in the "wrong" direction, the matrix can first be transposed using the ESSL routines. If ESSL is not available, the matrix could be transposed using Fortran code similar to the following:

```
do ii=1,N,NB
    do j=1,N
      do i=ii,min(ii+NB-1,N)
        y(i,j)=x(j,i)
      enddo
    enddo
  enddo
```

If NB is chosen correctly (somewhat less than 128), it will enable all values of x during one pass of the inner loop to be held in the Cache so that successive passes of the middle loop will find much of x data already in the Cache. However, since it is somewhat difficult to determine the best value of NB, it is preferable to use the ESSL routines.

2.3.1.2 Data prefetch

If data is accessed sequentially, the hardware will prefetch Cache lines. If data is accessed randomly, up to four Cache lines may be fetched concurrently. This can be illustrated by the following loops:

```
parameter(NS=16,N=32,M=2000)
   real*8 x(NS,N,M)
   . . .
   do j=1,M
     do i=1,N
       y = y + x(1,i,j)
     enddo
   enddo
   . . .
   do j=1,M
     do i=1,N
       y = y + x(1,ind(i),j)
     enddo
   enddo
```

In the first loop, the access is skip sequential, and one word is loaded from each Cache line sequentially. The word is loaded from memory because the total size of array x is 8 MB, which is too large for the 4 MB L2 Cache.

In the second loop, one word is loaded randomly from each line within a page. The reason for creating multiple random accesses within one page at a time (rather the whole array) is to avoid TLB (Translation Lookaside Buffer) misses.

The cycle times for these loops on POWER3 SMP High and POWER3 SMP Thin/Wide nodes are displayed in Table 8.

Table 8. Cycle times for POWER3 SMP nodes

Axis pattern	POWER3 SMP High	POWER3 SMP Thin/Wide
Skip Sequential	36	27
Random	30	25

If each line was fetched one at a time, the times would have a latency of approximately 85 cycles on the POWER3 SMP High node (and 35 on the POWER3 SMP Thin/Wide node) plus 16 cycles to transmit the line. These measurements clearly show that the hardware is looking ahead, and multiple lines are being fetched concurrently. However, remember that only four lines can be fetched concurrently.

It is also very important to try to avoid TLB misses as illustrated in the preceding code. There are 256 entries in the TLB buffer, each translating the address of a virtual page to a real page. This covers just 1 MB of address space. If an entry does not exist in the TLB buffer, an extra delay of about 50 cycles will occur.

2.3.2 Multiprocessor throughput

Some simple attempts were made to explore the throughput capabilities of the eight processor POWER3 SMP High node. Since it is memory access that is expected to cause conflict, three types of memory access were examined: Load, Store, and Copy.

In order to make the measurements, rather than running separate programs, Fortran's shared memory loop parallelization capability was used. The memory access operation was placed inside a loop that could be parallelized for multiple processes with Fortran directives (designated by CSMP$). The number of participating processes was determined at run time by setting the variable XLSMPOPTS to the required number.

2.3.2.1 Load

The Load operation was represented by the following Fortran loop:

```
parameter(M=4*1024*1024+7)
   real*8 y(100,8),x(M,8)
   . . . . . .
CSMP$ PARALLEL DO PRIVATE(it,irep,i)
   do it=1,NPROC
     do irep=1,NREP
       do i=1,N
         y(1,it)=y(1,it)+x(i,it)
       enddo
     enddo
   enddo
```

The variables `x` and `y` were double precision (64 bit). The variable `x` was summed, rather than simply loaded (for example, with y(1,it)=x(i,it)), to prevent Fortran from optimizing away the whole loop! The results for various values of N and NPROC are shown in Figure 5.

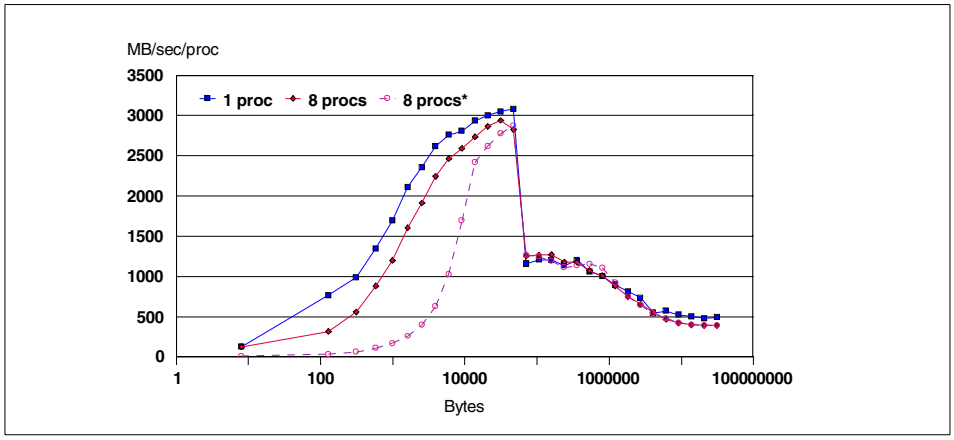


Figure 5. POWER3 SMP High node: Rate for y=y+x(i)

Note that the "1 proc" curve for a single processor reaches over 3000 MB/sec as long as the data is in the 64 KB Level 1 Data Cache. This is nearly equal to the maximum rate of 3.5 GBps. The lower rate for shorter lengths is due to the overhead of setting up the summation loop and storing y.

When the data is too large to fit in the L1 Cache, the rate drops to about 1.2 GBps, which is a little less that the maximum rate of about 1.75 GBps that can be expected from the 4 MB L2 Cache. As the data size approaches and

exceeds 4 MB, the data has to come directly from memory, and the rate falls off further.

However, what is really impressive is that the rate per processor for the "8 procs" curve is very similar to that of the "1 proc" curve. This is particularly so for large data sizes, which shows that there is very little memory interference. The "8 procs" curve reaches almost the same value as the "1 proc" curve while the data is in the L1 cache. This is to be expected since each processor has its own Level 1 cache.

The "8 procs" rate is lower than the "1 proc" rate for very short lengths due to the overhead of storing multiple y values and maintaining Cache coherence. Indeed, the "8 procs*" curve shows that very significant overhead will be incurred if the y values are stored in the same cache line (where each cache line consists of 128 consecutive bytes). This undesirable effect was inadvertently achieved by declaring y as "real*8 y(NPROC)". This caused a huge degradation for small data sizes; so, parallel programmers BEWARE.

2.3.2.2 Store

The Store operation was represented by the following Fortran loop:

```
   parameter(M=4*1024*1024+7)
   real*8 x(M,NPROC)
   . . . . . .
CSMP$ PARALLEL DO PRIVATE(it,irep,i)
   do it=1,NPROC
     do irep=1,NREP
       do i=1,N
         x(i,it)=1.d0
       enddo
     enddo
   enddo
```

The variable x was double precision (64 bit). The results, for various values of N and NPROC, are shown in Figure 6 on page 23.

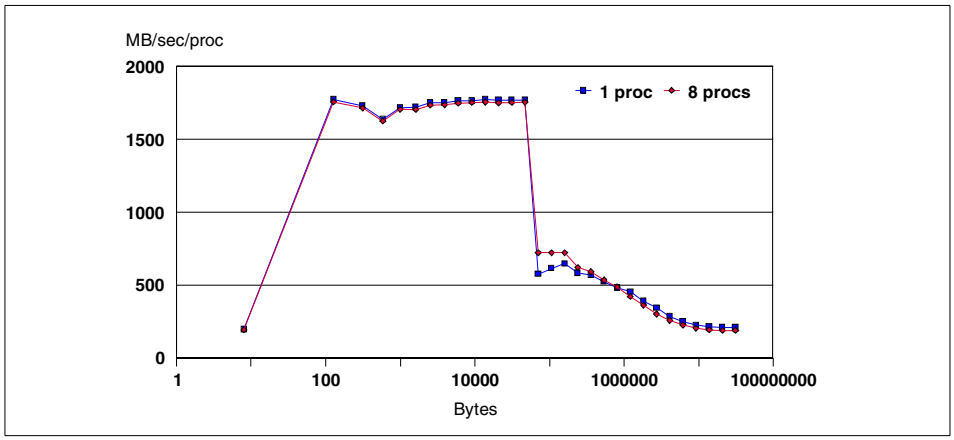


Figure 6. POWER3 SMP High node: Rate for x(i)=1.d0

Note that the rate per process for 8 processes is very similar to that for a single process.

2.3.2.3 Copy

The Copy operation was represented by the following Fortran loop:

```
parameter(M=4*1024*1024+7)
   real*8 y(M,8),x(M,8)
   . . . . . .
CSMP$ PARALLEL DO PRIVATE(it,irep,i)
   do it=1,NPROC
     do irep=1,NREP
       do i=1,N
         y(i,it)=x(i,it)
       enddo
     enddo
   enddo
```

The variables `x` and `y` were double precision (64 bit). The results for various values of N and NPROC are shown in Figure 7 on page 24.

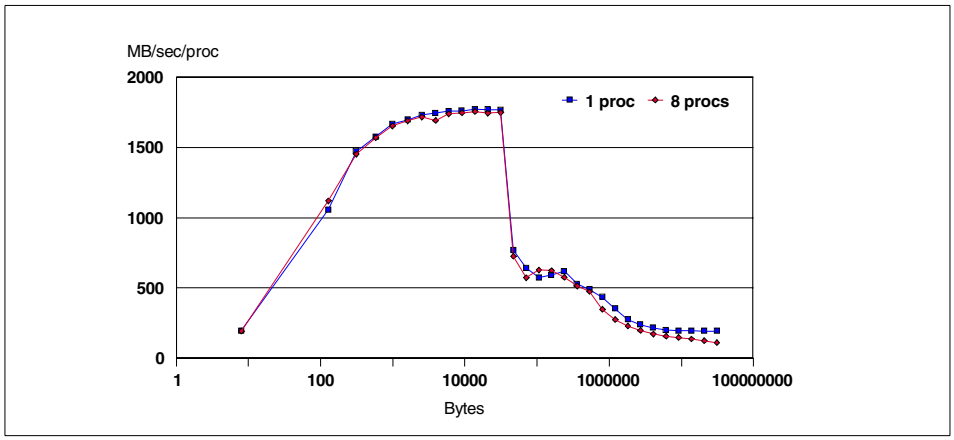


Figure 7. POWER3 SMP High node: Rate for y(i)=x(i)

Once again, note that the rate per process for eight processes is very similar to that for a single process.

Chapter 3. Distributed memory

This chapter serves as an introduction to programming paradigms for distributed memory parallelization. Mainly, the standardized Message Passing Interface (MPI) is being discussed, with particular emphasis on collective communications, MPI data types, and performance assessment. Finally, we will cover the Low-level Application Programming Interface (LAPI). After reading this chapter, you will understand advanced concepts of message passing and be able to use it more efficiently.

3.1 Introduction to MPI

This subsection introduces the message passing concepts of MPI starting with the basic functionality of send and receive subroutines. Next, we will discuss the way pitfalls and deadlock situations can easily occur when this basic functionality is used directly. This subsection is not intended to be a comprehensive introduction to MPI; so, the last part of this subsection offers suggestions for further reading.

3.1.1 Basic concepts

The message passing paradigm assumes that a computer program is split into several tasks, each having its own private data and addressing space. No global data is shared by several or all tasks of the program. All synchronization and communication between the tasks is accomplished by passing messages. This concept has been developed for parallel machines with distributed memory but is not confined to this architecture. By the very nature of this concept, any interaction between different tasks needs explicit coding, which makes it difficult to have message passing automatically done by the compiler. On the other hand, the programmer has total control over task interaction; so, by carefully designing synchronization and communication patterns, he or she can achieve better scalability than with any other parallelization technique.

In order to make parallel programs portable, the Message Passing Interface (MPI) was established. The goals of this standardization are best described with a quote from "MPI: A Message Passing Interface Standard (Version 1.1)", available at `http://www.mpi-forum.org/docs/docs.html`.

> “The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and

industry. The standardization process began with the Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia. At this workshop the basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process."

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher-level routines and/or abstractions are built upon lower-level message passing routines, the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard, such as that proposed here, provides vendors with a clearly-defined base set of routines that they can implement efficiently or, in some cases, for which they can provide hardware support, thereby enhancing scalability.

Simply stated, the goal of the Message Passing Interface is to develop a widely-used standard for writing message-passing programs. As such, the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows:

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying and allow overlap of computation and communication and off-load to communication coprocessor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface.
- Assume a reliable communication interface: The user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc., and provides extensions that allow greater flexibility.
- Define an interface that can be implemented on many vendors' platforms with no significant changes in the underlying communication and system software.

- The semantics of the interface should be language-independent.
- The interface should be designed to allow for thread-safety.

3.1.2 Pitfalls in point-to-point communication

The atomic units of every message passing library are send and receive operations, which have to come in pairs. The syntax of the blocking send operation is as follows:

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
[ IN buf] initial address of send buffer (choice)
[ IN count] number of elements in send buffer (nonnegative integer)
[ IN datatype] datatype of each send buffer element (handle)
[ IN dest] rank of destination (integer)
[ IN tag] message tag (integer)
[ IN comm] communicator (handle)
```

The send buffer specified by the MPI_SEND operation consists of count successive entries of the type indicated by data type starting with the entry at address buf. Note that we specify the message length in terms of the number of elements, not the number of bytes. The former is machine-independent and closer to the application level.

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the message envelope. These fields are:

- Source
- Destination
- Tag
- Communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the dest argument.

The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages.

A communicator specifies the communication context for a communication operation. Each communication context provides a separate “communication universe”. Messages are always received within the context they were sent,

and messages sent in different contexts do not interfere. A predefined communicator, MPI_COMM_WORLD, is provided by MPI. It allows communication with all processes that are accessible after MPI initialization, and processes are identified by their rank in the group of MPI_COMM_WORLD.

The syntax of the blocking receive operation is as follows:

```
MPI_RECV (buf, count, datatype, source, tag, comm, status)
[ OUT buf] initial address of receive buffer (choice)
[ IN count] number of elements in receive buffer (integer)
[ IN datatype] datatype of each receive buffer element (handle)
[ IN source] rank of source (integer)
[ IN tag] message tag (integer)
[ IN comm] communicator (handle)
[ OUT status] status object (Status)
```

The receive buffer consists of the storage containing count-consecutive elements of the type specified by data type starting at address buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit into the receive buffer without truncation.

If a message that is shorter than the receive buffer arrives, only those locations corresponding to the (shorter) message are modified.

Tag and source can be specified by the wild card constants MPI_ANY_TAG and MPI_ANY_SOURCE.

The send call described above is blocking; it does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. The receive is blocking as well; it returns when the message has been received.

Note that a finished MPI_SEND may not imply that the message has actually been sent. The MPI standard explicitly leaves it to the implementer to determine when the send actually occurs. Instead, MPI defines three more sending modes (each represented by a different MPI call).

A buffered mode send operation can be completed whether or not a matching receive has been posted. However, unlike the standard send, its completion never depends on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, MPI must buffer the outgoing message to allow the send call to complete. An error will occur if there is

insufficient buffer space. The amount of available buffer space is controlled by the user. This send mode is closest to the PVM send.

A send that uses the synchronous mode can be started whether or not a matching receive was posted. However, the send will only complete successfully if a matching receive is posted and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but also that the receiver has reached a certain point in its execution, namely, that it has started executing the matching receive.

A send that uses the ready communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous, and its outcome is undefined. The completion of the send operation does not depend on the status of a matching receive; it merely indicates that the send buffer can be reused. If a ready send is actually used without a corresponding receive being posted, IBM MPI issues the following error message indicating a fatal error:

```
ERROR: 0032-175 No receive posted for ready mode send  in MPI_RSEND or
MPI_IRSEND, task 1
ERROR: 0031-250  task 0: Terminated
ERROR: 0031-250  task 1: Terminated
v01n01:/u/pospiech/Redbook $
```

For implementation of the standard send, the MPI standard suggests that you use eager send for small messages and synchronous send for large messages. IBM follows this suggestion by adding a buffer at the receiving side for early arrival. This way, a standard send works very much like a buffered send for small messages, except that the buffering takes place at the receiver's side. The buffer size is controlled by the environment variable MP_BUFFER_MEM (poe option -buffer_mem). The limit between the two implementations is given by the environment variable MP_EAGER_LIMIT (the poe option -eager_limit). If neither MP_BUFFER_MEM nor MP_EAGER_LIMIT is set by the user, MP_EAGER_LIMIT defaults to Table 9.

Table 9. MP_EAGER_LIMIT per tasks

Number of Tasks	MP_EAGER_LIMIT (in bytes)
1—16	4096
17—32	2048
33—64	1024
65—128	512

The following example may be used to demonstrate the use of MP_EAGER_LIMIT and MP_BUFFER_MEM. If started with two MPI tasks, the program mutually exchanges buffer contents. Both MPI tasks first call a blocking send followed by a blocking receive. According to the standard, this is an unsafe practice and risks a deadlock.

```
/*********************************************************************
* xchange.c
*
* Program for testing MP_EAGER_LIMIT and MP_BUFFER_MEM
* Run on 2 MPI tasks.
*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
int i, tag;
int bufsize;
char *sendbuf, *recvbuf;
int me, comm_size, partner;
MPI_Status status[2];

/*  error check and allocation of sendbuf an recvbuf
     removed for better readability  */

/* meaning of command line argument */
bufsize = atoi(argv[1]);

/* initialize MPI */
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &me);
partner = (me ? 0 : 1);
tag = 1;

/* initialize buffers */
for (i=0; i<bufsize; i++) {
sendbuf[i] =  (char) me;
}

/* do communication */
printf("Start sending now ...\n");
MPI_Send(sendbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD);
printf("Posting receive now ...\n");
MPI_Recv(recvbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD, status);
printf("Received on %d: %d\n",me,recvbuf[0]);
```

```
MPI_Finalize();
printf("done\n");
return(0);
}
```

We get the following results:

```
v01n01:/u/pospiech/Redbook $ poe xchange 4096
0:Start sending now ...
1:Start sending now ...
1:Posting receive now ...
1:Received on 1: 0
0:Posting receive now ...
0:Received on 0: 1
0:done
1:done
v01n01:/u/pospiech/Redbook $ poe xchange 4097
0:Start sending now ...
1:Start sending now ... (deadlock; has to be canceled with ^C)
^CERROR: 0031-250 task 0: Interrupt
ERROR: 0031-250 task 1: Interrupt
v01n01:/u/pospiech/Redbook $ poe xchange -eager_limit 4097 4097
0:Start sending now ...
1:Start sending now ...
1:Posting receive now ...
1:Received on 1: 0
0:Posting receive now ...
0:Received on 0: 1
0:done
1:done
v01n01:/u/pospiech/Redbook $
```

In the first case, the program was started with a message size just at MP_EAGER_LIMIT. MPI chooses an eager send to handle the standard send, which was actually coded. The message is buffered for early arrival, and the send returns. Then, the receives are posted. Note that the return of the send does not even mean that the receives have been posted - let alone that the message has arrived.

In the second case, the message size surpasses the MP_EAGER_LIMIT; so, MPI does a synchronous send. This can only complete if a receive is posted. But this cannot happen since neither of the sends ever returns; so, neither of the MPI tasks arrives at the call of MPI_Recv.

In the last case, MP_EAGER_LIMIT is increased by one, and the example works again; so, if a program happens to show unstable behavior that disappears by increasing the MP_EAGER_LIMIT, it is likely to contain a code

section similar to the preceding. A safe MPI program will be able to run with MP_EAGER_LIMIT set to zero. It is recommended that you try this once if you are aiming for a portable MPI code.

Note that it is not possible to set MP_EAGER_LIMIT to a value larger than MP_BUFFER_MEM because MP_EAGER_LIMIT is limited to 65536 by implementation. MP_BUFFER_MEM cannot be lowered below 1 MB (the default for user space is 64 MB) because, otherwise, poe complains.

```
v01n01:/u/pospiech/Redbook $ poe xchange -buffer_mem 1000000 4096 1
   1:ERROR: 0031-309  Connect failed during message passing initialization,
task 1, reason: There is not enough memory available now.
   1:ERROR: 0031-007  Error initializing communication subsystem: return
code -1
   0:ERROR: 0031-309  Connect failed during message passing initialization,
task 0, reason: There is not enough memory available now.
   0:ERROR: 0031-007  Error initializing communication subsystem: return
code -1
v01n01:/u/pospiech/Redbook $
```

The standard way out of the aforementioned deadlock situation is to use non-blocking communication calls. A non-blocking send start call initiates the send operation but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, for example, to verify that the data has been copied out of the send buffer. Non-blocking send start calls can use the same four modes as blocking sends: Standard, buffered, synchronous, and ready. These carry the same meaning. Ready-excepted sends of all modes can be started whether a matching receive has been posted or not; a non-blocking ready send can be started only if a matching receive is posted. In all cases, it returns immediately, irrespective of the status of other processes.

A separate send complete call is needed to complete the communication, for example, to verify that the data has been copied out of the send buffer. The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning depending on the send mode.

If the send mode is synchronous, the send complete call only succeeds if a matching receive has started. That is, a receive has been posted and matched with the send.

If the send mode is buffered, the message must be buffered if there is no ending receive. In this case, the send-complete must succeed irrespective of the status of a matching receive.

If the send mode is standard, the send-complete call may return before a matching receive occurred if the message is buffered. On the other hand, the send-complete may not complete until a matching receive occurs and the message is copied into the receive buffer.

Nonblocking sends can be matched with blocking receives and vice-versa.

For example, the interface for standard send start is as follows:

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
[ IN buf] initial address of send buffer (choice)
[ IN count] number of elements in send buffer (integer)
[ IN datatype] datatype of each send buffer element (handle)
[ IN dest] rank of destination (integer)
[ IN tag] message tag (integer)
[ IN comm] communicator (handle)
[ OUT request] communication request (handle)
```

C interface:

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,int dest, int
tag, MPI_Comm comm, MPI_Request *request)
```

FORTRAN interface:

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

There are several send complete calls possible. For example, the interface for MPI_WAIT is given. MPI_WAIT takes the request handle from the send start command and returns when the corresponding send has completed.

```
MPI_WAIT(request, status)
[ INOUT request] request (handle)
[ OUT status] status object (handle)
```

C interface:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

FORTRAN interface:

```
MPI_WAIT(REQUEST, STATUS, IERROR)
                    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

But, even when nonblocking communication is used, an error, such as the one shown above, can happen. As an example, one might consider the following program, which shows a sneaky way to get along with only one

buffer for sending and receiving in an all-to-all communication. First, every task starts all send operations in nonblocking mode. Then, an MPI_Waitany is used to determine which send buffer can be reused. Whenever a buffer is free, the corresponding receive places its data into this buffer.

```
/*********************************************************************
 *                          all2all.c
 *
 *  Try to save memory for all to all Communication
 *  Run on n MPI tasks.
 *********************************************************************/

#include <stdio.h>
#include <stdlib.h>

#include "mpi.h"

int main(int argc, char *argv[])
{
int i, tag, sender;
int bufsize;
char *commbuf;
int me, comm_size;
MPI_Status status[2];
MPI_Request *requests;

/*  error check and allocation and initialization of
     commbuf removed for better readability  */

/* meaning of command line argument */
bufsize = atoi(argv[1]);

/* initialize MPI */
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
tag = 1;

/* do communication */

for (i=0; i<comm_size; i++)  {
    if ( i==me ) {continue;}
    printf("Start sending now to %d...\n", i);
    MPI_Isend(commbuf+i*bufsize, bufsize, MPI_CHAR, i, tag,
              MPI_COMM_WORLD, requests+i);
  }
for (i=0; i<comm_size; i++)  {
```

```
    printf("Waiting for a send to complete ...\n");
    MPI_Waitany(comm_size, requests, &sender, status);
    if (sender == MPI_UNDEFINED) {break;}
    printf("Posting receive now for %d...\n", sender);
    MPI_Recv(commbuf+sender*bufsize, bufsize, MPI_CHAR, sender, tag,
             MPI_COMM_WORLD, status);
    printf("Received on %d: %d\n",me,commbuf[sender*bufsize]);
  }

MPI_Finalize();
printf("done\n");
return(0);
}
```

When run on three CPUs for different message sizes, the outcome of this example is the following:

```
v01n01:/u/pospiech/Redbook $ poe all2all -procs 3 4096
   0:Start sending now to 1...
   0:Start sending now to 2...
   0:Waiting for a send to complete ...
   0:Posting receive now for 1...
   2:Start sending now to 0...
   1:Start sending now to 0...
   1:Start sending now to 2...
   1:Waiting for a send to complete ...
   1:Posting receive now for 0...
   1:Received on 1: 0
   1:Waiting for a send to complete ...
   1:Posting receive now for 2...
   1:Received on 1: 2
   1:Waiting for a send to complete ...
   0:Received on 0: 1
   0:Waiting for a send to complete ...
   0:Posting receive now for 2...
   0:Received on 0: 2
   0:Waiting for a send to complete ...
   2:Start sending now to 1...
   2:Waiting for a send to complete ...
   2:Posting receive now for 0...
   2:Received on 2: 0
   2:Waiting for a send to complete ...
   2:Posting receive now for 1...
   2:Received on 2: 1
   2:Waiting for a send to complete ...
   2:done
   0:done
   1:done
```

```
v01n01:/u/pospiech/Redbook $ poe all2all -procs 3 4097
   0:Start sending now to 1...
   0:Start sending now to 2...
   0:Waiting for a send to complete ...
   2:Start sending now to 0...
   2:Start sending now to 1...
   2:Waiting for a send to complete ...
   1:Start sending now to 0...
   1:Start sending now to 2...
   1:Waiting for a send to complete ...  (waiting forever; use ^C to stop)
^CERROR: 0031-250  task 0: Interrupt
ERROR: 0031-250  task 1: Interrupt
ERROR: 0031-250  task 2: Interrupt
v01n01:/u/pospiech/Redbook $
```

Again, this example shows significant dependence on the message size. If the message size exceeds MP_EAGER_LIMIT, the program is in deadlock. The explanation is easy and closely-linked to the question of where the data resides between send and receive. Small messages are buffered; so, all messages can take off, stay in the buffer, and land in their new destination. Large messages can only leave their old position if the new place is already free; so, all messages are waiting for someone to start the game. In fact, if started on two processors, it is the very first example in a new disguise; so, it is surprising that it behaves the same way. This is a good example of an unsafe code, which would have been caught by setting MP_EAGER_LIMIT to zero.

Note, however, that it is safe to call MPI_Waitany in a loop as shown in the example above. MPI_Waitany never waits on the same request twice because it is invalidating the request it has been waiting for.

Using nonblocking communication, there are indeed several ways to correct the erroneous MPI program for buffer exchange shown above. The erroneous statements were the following (for the sake of clarity, the printf statements are removed):

```
MPI_Send(sendbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD);
if (use_barrier)
  {
    MPI_Barrier(MPI_COMM_WORLD);
  }
MPI_Recv(recvbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD, status);
```

A standard send is implemented as synchronous send for message sizes larger than MP_EAGER_LIMIT; so, the call to MPI_Send can only return if it is matched by a posted receive. But, the receives are only posted after

completion of the send; so, the program deadlocks. The standard correction would be to replace these statements with the following:

```
MPI_Isend(sendbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD,
&request);
MPI_Recv(recvbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD, status);
MPI_Wait(&request, status+1);
```

Now, the send is started, and control is immediately given back to the program. Then, the receives are posted, and the sends can complete. An MPI_Wait is called to complete the send operation and guarantee that the send buffer can be reused safely. Thinking twice, one might note that the send still has to wait for the receive being posted; so, it might be advantageous to post the receive before starting the send. For obvious reasons, the receive has to be nonblocking now.

```
MPI_Irecv(recvbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD,
&request);
MPI_Send(sendbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD);
MPI_Wait(&request, status);
```

This works as smoothly as the previous solution, but the execution time tends to be slightly slower (see Figure 8 on page 38). Therefore, it may be tempting to replace the standard send with a ready send, since the receive is posted anyway. However, only the receive posted by the other MPI task counts; so, the following error message is almost inevitable:

```
ERROR: 0032-175 No receive posted for ready mode send  in MPI_RSEND or
MPI_IRSEND, task 0
ERROR: 0031-250  task 0: Terminated
ERROR: 0031-250  task 1: Terminated
```

Three errors occur because one task is ahead and gets to the send before the other task is able to post the receive. This can either be mended through the use of synchronous sends or by a barrier between receive and send. The latter guarantees that sends are only started when all receives are posted.

```
MPI_Irecv(recvbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD,
&request);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Rsend(sendbuf, bufsize, MPI_CHAR, partner, tag, MPI_COMM_WORLD);
MPI_Wait(&request, status);
```

A comparison of execution times is shown in Figure 8.

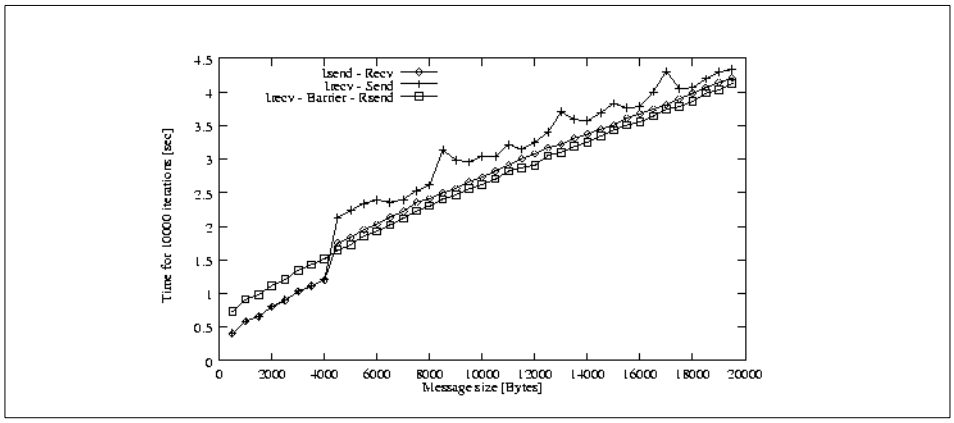


Figure 8. Different algorithms for message exchange

First, note the jump in execution time for the first two examples at a message size of 4096 bytes. This jump is due to the change in protocol for the standard send. The barrier is expected to take a similar amount of time for the rendezvous as takes place at the beginning of a synchronous send. Nevertheless, for large messages, this solution shows a slight advantage in execution time. Apparently, the barrier is too slow to make the last example efficient for small messages.

3.1.3 Suggestions for further reading

This redbook is not intended to be a complete introduction to MPI; rather, it concentrates on the effective use of possibly less-known parts of it. There are also two comprehensive introductions.

To start with, there is another redbook on this topic: *RS/6000 SP: Practical MPI Programming*, SG24-5380.

This redbook helps you write Message Passing Interface (MPI) programs that run on distributed memory machines, such as the RS/6000 SP. This publication concentrates on the real programs that RS/6000 SP solution providers want to parallelize.

A indispensable reference is the MPI standard itself, which is available via the Internet using the URL `http://www.mpi-forum.org/docs/docs.html`.

Quite a number of books have been published on this topic including the following:

- *Using MPI*, by William Gropp, Ewing Lusk, and Anthony Skjellum published by MIT Press ISBN 0-2625-7104-8. The example programs from this book are available at `ftp://ftp.mcs.anl.gov/pub/mpi/using`.
- *Designing and Building Parallel Programs*, Ian Foster's online book includes a chapter on MPI. It provides a succinct introduction to an MPI subset. (ISBN 0-2015-7594-9; Published by Addison-Wesley)
- *MPI: The Complete Reference*, by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra, The MIT Press .
- *MPI: The Complete Reference - 2nd Edition: Volume 2, The MPI-2 extensions*, by William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir, The MIT Press.
- *Parallel Programming With MPI*, by Peter S. Pacheco, published by Morgan Kaufmann.

3.2 MPI collective communication

As discussed in the previous section, the use of vanilla send and receive routines needs careful consideration; otherwise, a program may easily end up deadlocking. Moreover, some communication constructs may go well for some data but deadlock with others; so, at first sight, this just looks like the very opposite of ease of use.

Before blaming MPI for this complicated and error-prone setup, the atomic send and receive routines should be considered low-level layers. The high-level approach would be to use the collective communication routines discussed in this subsection.

3.2.1 Design concepts

Similar to the concept of Basic Linear Algebra Subroutines (BLAS), MPI also provides a concept of larger building blocks. These building blocks allow for a high-level approach to message-passing communication. This requires, as a first step, that you identify the communication pattern rather than thinking of single send and receive commands. The first question would be: Who has got what part of the information, and who is going to need it? This has to be compared with the given communication patterns provided by the collective communication routines. Experience shows that most communication can be mapped to collective communication routines, particularly in combination with MPI data types.

This approach produces less code, which is easier to read. Next, it is much less error-prone since all buffering, waiting, and blocking are done internally. Finally, the code is, in general, more efficient, at least on the RS/6000 SP; so, it is worth giving it a second thought. It is better to describe what communication is needed, and leave the question of how to set up the appropriate send and receive commands to MPI.

For better reading, a description of collective communication calls is quoted here from the MPI 1.1 standard. The functions of this type provided by MPI are the following:

- Barrier synchronization across all group members.
- Broadcast from one member to all members of a group. See Figure 9 on page 41.
- Gather data from all group members to one member. See Figure 9 on page 41.
- Scatter data from one member to all members of a group. See Figure 9 on page 41.
- A variation on Gather where all members of the group receive the result. This is shown as “allgather’’ in Figure 9 on page 41.
- Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all). This is shown as “alltoall’’ in Figure 9 on page 41.
- Global reduction operations, such as sum, max, min, user-defined functions where the result is returned to all group members, and a variation where the result is returned to only one member.
- A combined reduction and scatter operation.
- Scan across all members of a group (also called prefix).

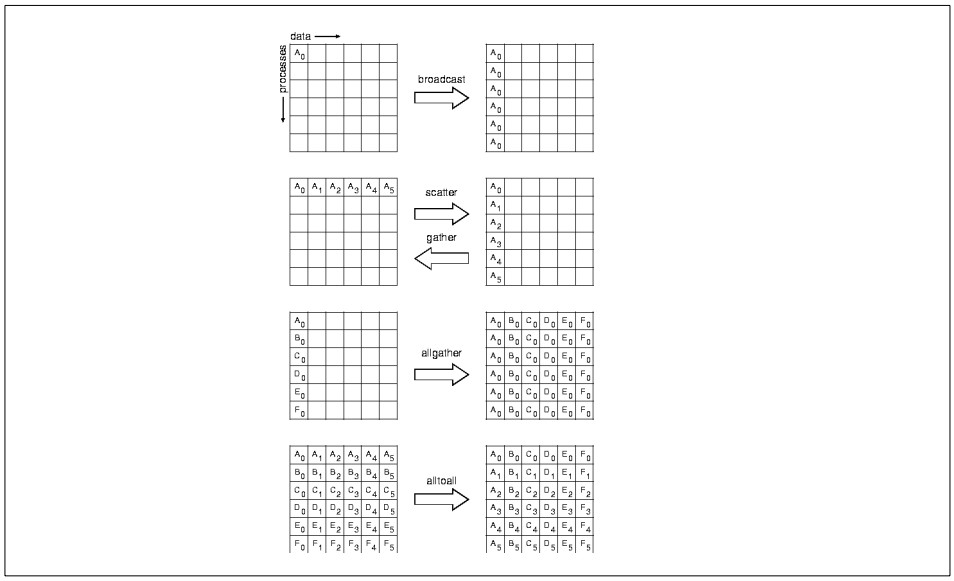


Figure 9. Collective move functions (group of six processes)

In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially, only the first process contains the data A_0, but, after the broadcast, all processes contain it.

A collective operation is executed by having all processes in the group call the communication routine with matching arguments. The syntax and semantics of the collective operations are defined to be consistent with the syntax and semantics of the point-to-point operations. The key concept of the collective functions is to have a “group” of participating processes. The routines do not have a group identifier as an explicit argument. Instead, there is a communicator argument. Several collective routines, such as broadcast and gather, have a single originating or receiving process. Such processes are called the root. Some arguments in the collective functions are specified as “significant only at root” and are ignored for all participants except the root.

Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise indicated in the description of the operation). Thus, a collective communication call may or may not have the effect of synchronizing all calling processes. Of course, this statement excludes the barrier function.

It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable. On the other hand, a correct portable program must allow for the fact that a collective call may be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it.

```
MPI_BARRIER( comm )
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Barrier(MPI_Comm comm)
```

Fortran interface:

```
MPI_BARRIER(COMM, IERROR) INTEGER COMM, IERROR
```

MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call. If most (if not all) communication is done via collective communication calls, an explicit barrier is rarely needed.

```
MPI_BCAST( buffer, count, datatype, root, comm )
[ INOUT buffer] starting address of buffer (choice)
[ IN count] number of entries in buffer (integer)
[ IN datatype] data type of buffer (handle)
[ IN root] rank of broadcast root (integer)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm )
```

Fortran interface:

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_BCAST broadcasts a message from the process with rank root to all processes of the group including itself. It is called by all members of the group using the same arguments for communication (root). Upon its return, the contents of root's communication buffer have been copied to all processes.

```
MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
root, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcount] number of elements in send buffer (integer)
[ IN sendtype] data type of send buffer elements (handle)
[ OUT recvbuf] address of receive buffer (choice, significant only at root)
[ IN recvcount] number of elements for any single receive (integer,
significant only at root)
[ IN recvtype] data type of recv buffer elements (significant only at root)
(handle)
[ IN root] rank of receiving process (integer)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Gather(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Fortran interface:

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE,
RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

Each process (including the root process) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. An alternative description is that the n messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI_RECV(recvbuf, recvcount n, recvtype, ...).

The receive buffer is ignored for all non-root processes. All arguments to the function are significant on process root while, on other processes, only the sendbuf, sendcount, sendtype, root, and comm arguments are significant. All processes must use the same communicator and the same value for root. The specification of counts and types should not cause any location on the root to be written more than once; such a call is erroneous. Note that the recvcount argument at the root indicates the number of items it receives from each process, not the total number of items it receives.

```
MPI_GATHERV( sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
recvtype, root, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcount] number of elements in send buffer (integer)
[ IN sendtype] data type of send buffer elements (handle)
```

```
[ OUT recvbuf] address of receive buffer (choice, significant only at root)
[ IN recvcounts] integer array (of length group size) containing the number
of elements that are received from each process (significant only at root)
[ IN displs] integer array (of length group size). Entry i specifies the
displacement relative to recvbuf at which to place the incoming data from
process i (significant only at root)
[ IN recvtype] data type of recv buffer elements (significant only at root)
(handle)
[ IN root] rank of receiving process (integer)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Gatherv(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

Fortran interface:

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE,
RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
COMM, IERROR
```

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since recvcounts is now an array. By providing the new argument, displs, it also allows more flexibility as to where the data is placed on the root. Messages are placed in the receive buffer of the root process in rank order, that is, the data sent from process j is placed in the jth portion of the receive buffer, recvbuf, on process root. The jth portion of recvbuf begins at the offset of displs[j] elements (in terms of recvtype) into recvbuf.

The receive buffer is ignored for all non-root processes. All arguments to the function are significant on process root while, on other processes, only the sendbuf, sendcount, sendtype, root, and comm arguments are significant. All processes must use the same communicator and the same value for root.

The type signature implied by sendcount, sendtype, on process i must be equal to the type signature implied by recvcounts[i], recvtype, at the root. This implies that the amount of data sent must be equal to the amount of data received pairwise between each process and the root.

```
MPI_SCATTER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
root, comm)
```

[IN sendbuf] address of send buffer (choice, significant only at root)
[IN sendcount] number of elements sent to each process (integer, significant only at root)
[IN sendtype] data type of send buffer elements (significant only at root) (handle)
[OUT recvbuf] address of receive buffer (choice)
[IN recvcount] number of elements in receive buffer (integer)
[IN recvtype] data type of receive buffer elements (handle)
[IN root] rank of sending process (integer)
[IN comm] communicator (handle)

C interface:

```
int MPI_Scatter(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Fortran interface:

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE,
RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

MPI_SCATTER is the inverse operation to MPI_GATHER.

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)
[IN sendbuf] address of send buffer (choice, significant only at root)
[IN sendcounts] integer array (of length group size) specifying the number of elements to send to each processor
[IN displs] integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i
[IN sendtype] data type of send buffer elements (handle)
[OUT recvbuf] address of receive buffer (choice)
[IN recvcount] number of elements in receive buffer (integer)
[IN recvtype] data type of receive buffer elements (handle)
[IN root] rank of sending process (integer)
[IN comm] communicator (handle)

C interface:

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Fortran interface:

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*),
SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM,
IERROR
```

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

```
MPI_ALLGATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
comm)
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcount] number of elements in send buffer (integer)
[ IN sendtype] data type of send buffer elements (handle)
[ OUT recvbuf] address of receive buffer (choice)
[ IN recvcount] number of elements received from any process (integer)
[ IN recvtype] data type of receive buffer elements (handle)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Allgather(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm)
```

Fortran interface:

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
RECVTYPE, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE,
RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLGATHER can be thought of as MPI_GATHER, except that all processes receive the result instead of just the root. The jth block of data sent from each process is received by every process and placed in the jth block of the buffer recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype, at any other process.

The outcome of a call to MPI_ALLGATHER() is as if all processes executed n calls to MPI_GATHER (sendbuf, sendcount, sendtype, recvbuff, recvcount, recvtype, root, comm) for root = 0 , ..., n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER.

```
MPI_ALLGATHERV( sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
recvtype, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcount] number of elements in send buffer (integer)
[ IN sendtype] data type of send buffer elements (handle)
[ OUT recvbuf] address of receive buffer (choice)
[ IN recvcounts] integer array (of length group size) containing the number
of elements that are received from each process
[ IN displs] integer array (of length group size). Entry i specifies the
displacement (relative to recvbuf) at which to place the incoming data from
process i
[ IN recvtype] data type of receive buffer elements (handle)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Allgatherv(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvtype, MPI_Comm comm)
```

Fortran interface:

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
DISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE,
RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
IERROR
```

MPI_ALLGATHERV can be thought of as MPI_GATHERV, except that all processes receive the result instead of just the root. The jth block of data sent from each process is received by every process and placed in the jth block of the buffer recvbuf. These blocks need not all be the same size. The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[j], recvtype, at any other process.

The outcome is as if all processes executed calls to MPI_GATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm) for root = 0 , ..., n-1. The rules for correct usage of MPI_ALLGATHERV are easily found from the corresponding rules for MPI_GATHERV.

```
MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
comm)
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcount] number of elements sent to each process (integer)
[ IN sendtype] data type of send buffer elements (handle)
[ OUT recvbuf] address of receive buffer (choice)
```

```
[ IN recvcount] number of elements received from any process (integer)
[ IN recvtype] data type of receive buffer elements (handle)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Alltoall(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm)
```

Fortran interface:

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE,
RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j and is placed in the ith block of recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype, at any other process. This implies that the amount of data sent must be equal to the amount of data received pairwise between every pair of processes. As usual, however, the type maps may be different.

All arguments on all processes are significant. All processes must use the same communicator.

```
MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
rdispls, recvtype, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcounts] integer array equal to the group size specifying the
number of elements to send to each processor
[ IN sdispls] integer array (of length group size). Entry j specifies the
displacement (relative to sendbuf from which to take the outgoing data
destined for process j
[ IN sendtype] data type of send buffer elements (handle)
[ OUT recvbuf] address of receive buffer (choice)
[ IN recvcounts] integer array equal to the group size specifying the
number of elements that can be received from each processor
[ IN rdispls] integer array (of length group size). Entry i specifies the
displacement (relative to recvbuf at which to place the incoming data from
process i
[ IN recvtype] data type of receive buffer elements (handle)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

Fortran interface:

```
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*),
SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
RECVTYPE, COMM, IERROR
```

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls, and the location of the placement of the data on the receive side is specified by rdispls.

The type signature associated with sendcount[j], sendtype, at process i must be equal to the type signature associated with recvcount[i], recvtype, at process j. This implies that the amount of data sent must be equal to the amount of data received pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed. All arguments on all processes are significant. All processes must use the same communicator.

```
MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm)
[ IN sendbuf] address of send buffer (choice)
[ OUT recvbuf] address of receive buffer (choice, significant only at root)
[ IN count] number of elements in send buffer (integer)
[ IN datatype] data type of elements of send buffer (handle)
[ IN op] reduce operation (handle) [ IN root] rank of root process
(integer)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm)
```

Fortran interface:

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT,
COMM, IERROR
```

MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count, and datatype; the output buffer is defined by the arguments recvbuf, count, and datatype; both have the same number of elements with the same type. The routine is called by all group members using the same arguments for count, data type, op, root, and identical communicators. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence.

The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative but not commutative. The “canonical’’ evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity, in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

The predefined operations displayed in Table 10 are supplied for MPI_REDUCE and the related functions: MPI_ALLREDUCE, MPI_REDUCE_SCATTER, and MPI_SCAN. These operations are invoked by placing the following in op:

Table 10. Predefined MPI operations

Name	Meaning
[MPI_MAX]	maximum
[MPI_MIN]	minimum
[MPI_SUM]	sum
[MPI_PROD]	product
[MPI_LAND]	logical and
[MPI_BAND]	bit-wise and
[MPI_LOR]	logical or
[MPI_BOR]	bit-wise or
[MPI_LXOR]	logical xor
[MPI_BXOR]	bit-wise xor

Name	Meaning
[MPI_MAXLOC]	max value and location
[MPI_MINLOC]	min value and location

MPI includes variants of each of the reduce operations where the result is returned to all processes in the group. MPI requires that all processes participating in these operations receive identical results.

```
MPI_ALLREDUCE( sendbuf, recvbuf, count, datatype, op, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ OUT recvbuf] starting address of receive buffer (choice)
[ IN count] number of elements in send buffer (integer)
[ IN datatype] data type of elements of send buffer (handle)
[ IN op] operation (handle)
[ IN comm] communicator (handle)
```

C interface:

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Fortran interface:

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM,
IERROR
```

This is the same as MPI_REDUCE except that the result appears in the receive buffer of all the group members. The all-reduce operation can be thought as a reduce followed by a broadcast.

MPI includes variants of each of the reduce operations where the result is scattered to all processes in the group upon return.

```
MPI_REDUCE_SCATTER( sendbuf, recvbuf, recvcounts, datatype, op, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ OUT recvbuf] starting address of receive buffer (choice)
[ IN recvcounts] integer array specifying the number of elements in result
distributed to each process. Array must be identical on all calling
processes.
[ IN datatype] data type of elements of input buffer (handle)
[ IN op] operation (handle) [ IN comm] communicator (handle)
```

C interface:

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf,
int *recvcounts, MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm)
```

Fortran interface:

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE,
OP, COMM, IERROR
```

MPI_REDUCE_SCATTER first does an element-wise reduction on the vector in the send buffer defined by sendbuf, count, and data type. Next, the resulting vector of results is split into n disjoint segments where n is the number of members in the group. Segment i contains recvcounts[i] elements. The ith segment is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i], and data type.

The MPI_REDUCE_SCATTER routine is functionally equivalent to an MPI_REDUCE operation function with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts.

```
MPI_SCAN( sendbuf, recvbuf, count, datatype, op, comm )
[ IN sendbuf] starting address of send buffer (choice)
[ OUT recvbuf] starting address of receive buffer (choice)
[ IN count] number of elements in input buffer (integer)
[ IN datatype] data type of elements of input buffer (handle)
[ IN op] operation (handle) [ IN comm] communicator (handle)
```

C interface:

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

Fortran interface:

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*),
RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM,
IERROR
```

MPI_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,...,i (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are the same as for MPI_REDUCE.

3.2.2 Performance considerations

As an example, all-to-all communication is considered more closely. All tasks send messages of equal length to every other task, and, consequently, they also have to receive messages from all tasks. It is assumed that all outgoing messages are stored in the same large buffer and aligned with the task ID of the receiver. Similarly, the incoming messages are to be stored in another larger buffer and aligned by the task ID of the sender. The send and receive buffers are assumed to be disjointed. From the previous subsection, recall that the example in which the last assumption was dropped turned out to be erroneous. This is not surprising because the communication pattern can also be described as each MPI task is mutually swapping message contents with every other task. But, swapping of data cannot be done in place. At least, a temporary buffer is needed, which, of course, may be hidden by the use of a buffered send.

In the context of each MPI task occupying one CPU and one communication adapter and message sizes being independent of the number of MPI tasks, the execution time for all-to-all communication is expected to grow, at least linearly, in the number of MPI tasks. This can easily be concluded from the fact that all outgoing messages have to be processed by the same CPU and have to pass through the same adapter, which serializes the communication. This linear behavior can also be observed from the timings that follow shortly.

Given this, every all-to-all communication would spoil parallel scalability. Fortunately, in most cases, the assumption of message sizes being independent of the number of MPI tasks is false. The most common examples for data to be communicated are vectors with elements distributed blockwise among the tasks or, even, matrices with rows and columns distributed blockwise. In the first case, the message size is inversely proportional to the number of MPI tasks and the execution time stays constant, in the latter case it is even inversely proportional to the square of the number of MPI tasks and the execution time decreases with the number of MPI tasks.

Following the lines of the previous subsection, a straightforward method of implementation would be the following:

```
for (i=0; i<num_tasks; i++)
  {
    MPI_Irecv(recvbuf+i*bufsize, bufsize, MPI_CHAR, i, tag,
              MPI_COMM_WORLD, requests+i);
  }
MPI_Barrier(MPI_COMM_WORLD);
for (i=0; i<num_tasks; i++)
  {
    MPI_Rsend(sendbuf+i*bufsize, bufsize, MPI_CHAR, i, tag,
```

```
      MPI_COMM_WORLD);
  }
MPI_Waitall(comm_size, requests, status);
```

Given its simplicity, this version is shown below to work surprisingly well. The barrier is needed to guarantee that all Ireceives are posted before the first Rsend begins. The only drawback is that each task also communicates with itself, which does not seem to be the most efficient way. Indeed, this can be remedied in the following way:

```
for (i=0; i<num_tasks; i++)
  {
    if ( i==me ) {requests[i] = MPI_REQUEST_NULL; continue;}
    MPI_Irecv(recvbuf+i*bufsize, bufsize, MPI_CHAR, i, tag,
              MPI_COMM_WORLD, requests+i);
  }
MPI_Barrier(MPI_COMM_WORLD);
for (i=0; i<num_tasks; i++)
  {
    if ( i==me )
      {
memcpy(recvbuf+me*bufsize, sendbuf+me*bufsize, bufsize);
continue;
      }
    MPI_Rsend(sendbuf+i*bufsize, bufsize, MPI_CHAR, i, tag,
      MPI_COMM_WORLD);
  }
MPI_Waitall(comm_size, requests, status);
```

Both versions just post all necessary sends and receives and leave it to the communication subsystem to sort out how all messages travel through the network without excessive congestion. This offers room for improvement by designing congestion-free communication plans.

Breaking the all-to-all communication into mutual data exchanges, the design of a congestion-free communication plan boils down to grouping the set of all possible pairs into a minimal number of groups with disjoint members. This problem is well known in the field of combinatorical mathematics. It goes by the name of the"chess tournament problem". This is best described by the task of planning a chess tournament with each participant playing against everyone else in a minimum number of rounds. If the number of participants is a power of two, there is an obvious solution by building the groups along the parallel edges, face diagonals, and space diagonals of a hypercube. If not, the next larger hypercube can be used to distribute the participants among the corners of the hypercube. But, the grouping just described leaves at least one participant without a partner in every round because not all

hypercube corners are occupied. This is congestion-free, but may not be effective.

Relaxing the idea of mutual data exchange leads to the situation that each MPI task sends and receives at the same time, except with different partners. This paves the way for a very simple and congestion-free way of implementing all-to-all communication, which is best described as follows: All participants are arranged in a circle. In the first round, everyone talks to their right neighbor but listens to their left neighbor. In the next round, everyone talks to the next one on the right and listens to the next one on the left. This is continued around the circle until everyone is talking to his left neighbor and listening to his right neighbor. A C program implementing this might look like the following:

```
size = num_tasks;
for (i=1; i<size; i++)
  {
    dest = (me+i<size ? me+i : me+i-size );
    source = (me-i>=0 ? me-i : me-i+size );
    MPI_Sendrecv(sendbuf+bufsize*dest, bufsize, MPI_CHAR, dest, tag,
 recvbuf+bufsize*source, bufsize, MPI_CHAR, source, tag,
 MPI_COMM_WORLD, &status);
  }
memcpy(recvbuf+me*bufsize, sendbuf+me*bufsize, bufsize);
```

Finally, all these program examples have to be compared to the collective communication version:

```
MPI_Alltoall(sendbuf, bufsize, MPI_CHAR,
             recvbuf, bufsize, MPI_CHAR,
             MPI_COMM_WORLD);
```

The following three figures show the timings of different all-to-all implementations for different message sizes with two, four, and eight MPI tasks.

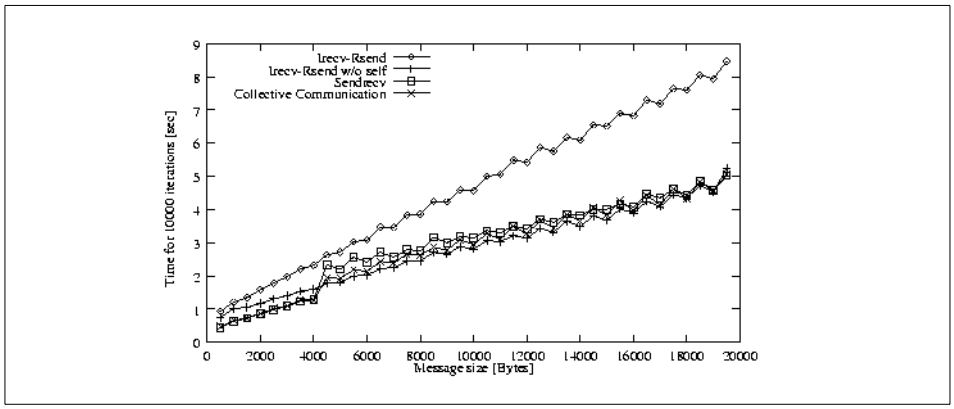


Figure 10. Algorithms for all-to-all collective communication (two MPI tasks)

There are 10000 repetitions of all-to-all communication among two MPI tasks for different message sizes (in bytes).

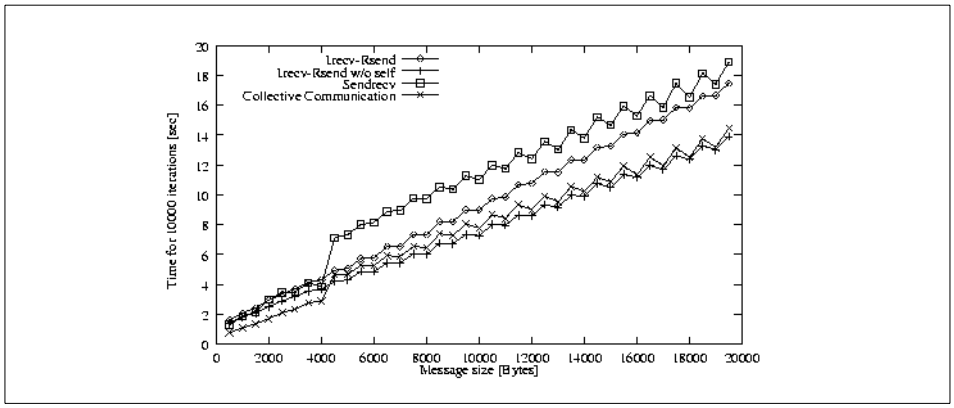


Figure 11. Algorithms for all-to-all collective communication (four MPI tasks)

There are 10000 repetitions of all-to-all communication among four MPI tasks for different message sizes (in bytes).

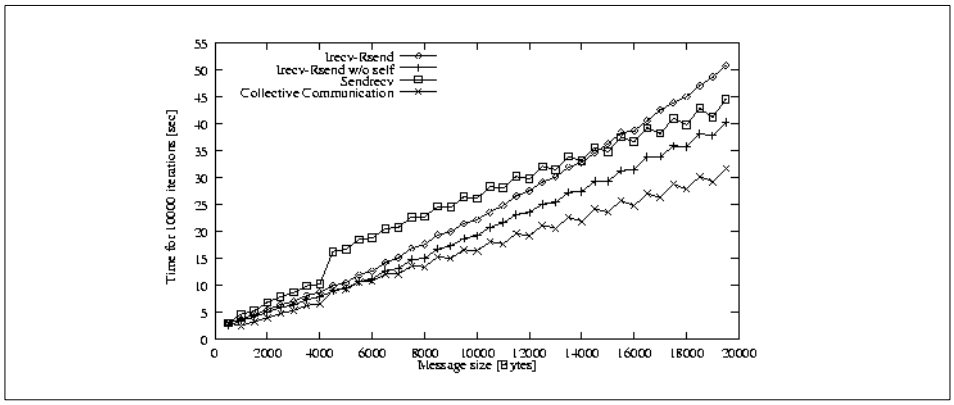


Figure 12. Algorithms for all-to-all collective communication (eight MPI tasks)

There are 10000 repetitions of all-to-all communication among eight MPI tasks for different message sizes (in bytes).

Not surprisingly, the difference between the first two methods decreases with the number of MPI tasks. In the second method, one send and receive pair is saved, but this loses importance as the total number of send and receive operations grows.

At first sight, it may be surprising that the second method (Irecv-Rsend without communication to itself) works so well for two MPI tasks. But, this is just the exchange example from the previous subsection in disguise. In particular, there is no congestion. If the number of MPI tasks and the message size increases, this changes dramatically.

Both the Sendrecv and the collective communication version show the characteristic jump at message size equal to MP_EAGER_LIMIT, which indicates an internal use of standard sends.

Lastly, all examples show that the pain of designing a congestion-free communication plan does not pay off. The sendrecv method is always worse than the collective communication version. A closer examination shows that the gap between the two versions matches the jump at MP_EAGER_LIMIT. Apart from the fact that the curves are roughly parallel. Apparently, the sendrecv version loses time during the hand shaking between sender and receiver for messages larger than MP_EAGER_LIMIT.

This picture gets even clearer if the above timings are transformed to bandwidth rates. This is done by dividing the message size by the time. This has to be multiplied by 10000 to accommodate the number of repetitions and

by the number of MPI tasks to get the amount of data that is actually sent. Figure 13 displays bandwidth for different all-to-all algorithms in the case of 8 MPI tasks.

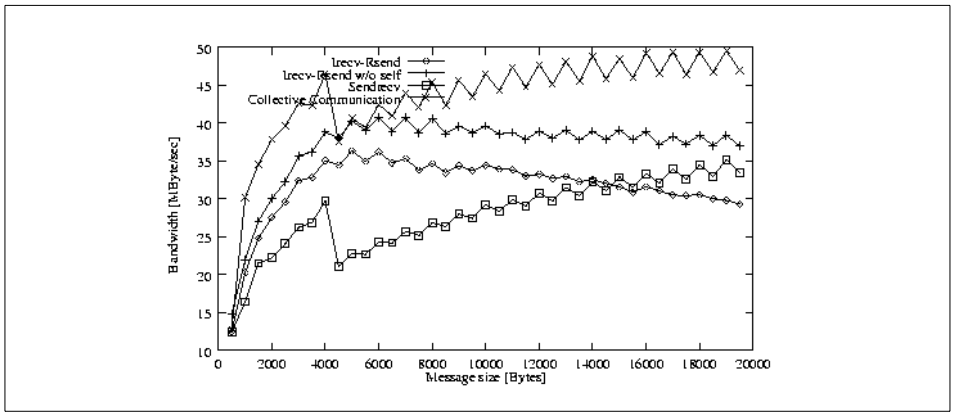


Figure 13. Bandwidth for different all-to-all algorithms (eight MPI tasks)

This figure shows collective communication to be the clear winner over the other alternatives that have been discussed in this subsection.

3.3 MPI data types

Most major message-passing libraries provide support for basic data types, such as integer or reals. In PVM 3.x, the data is “packed” into a buffer using a specific function for each data type. In MPI, each communication routine takes the data type and the number of elements as arguments. The disadvantage of the MPI approach compared to the PVM approach is that you cannot mix data types in one communication routine, and you cannot send noncontiguous data areas, such as a row in a matrix stored in column order, as the memory layout of a Fortran program.

MPI has two solutions to this problem: First, it provides a pack and unpack routine, but, more importantly, to solve this problem, MPI introduced the concept of derived data types. You can define your own communication data types using basic or derived data types.

There are several reasons to familiarize yourself with the new concept:

- Abstraction - By defining data types, you make an abstraction of your data and memory layout. For example, instead of sending 100 reals, which are on the border of your domain, you create a data type describing the

border; then, you can send the border directly. This will generally increase the readability of your cod,e an, since you can use the data type in several parts of your program, reusing the data structure will reduce the number errors in your code.

- Performance - Because the MPI standard does not tell the vendors how to implement derived data types, they do not have to go the PVM way and copy the data elements into a buffer before sending them. This could reduce memory movement and, therefore, make your program perform better.
- MPI-IO - One of the chapters of the MPI-2 standard is MPI-IO. To use MPI-IO efficiently, you have to be familiar with MPI derived data types. MPI-IO is discussed in Section 6.5, “MPI-IO” on page 214.

In the following sections, we discuss the use of MPI derived data types. Note that we point out differences between MPI-1.2 and MPI-2, even if some MPI-2 functions are not provided by the IBM implementation of MPI.

3.3.1 Basic concepts

Table 11 and Table 12 contain the MPI functions used to construct derived data types. Table 12 shows the functions added in MPI-2. They correct some problems with the MPI-1.x versions (mainly 64 bit addressing), and they replace the old functions. Note that they are not supported in the IBM MPI version at this time.

Table 11. MPI 1.2 Data types constructors

Syntax of constructor
MPI_Type_contiguous(count, oldtype, newtype)
MPI_Type_vector(count, blength, displ, oldtype, newtype)
MPI_Type_hvector(count, blength, displ, oldtype, newtype)
MPI_Type_indexed(count, blengths, displ’s, oldtype, newtype)
MPI_Type_hindexed(count, blengths, displ’s, oldtype, newtype)
MPI_Type_struct(count, blengths, displ’s, oldtypes, newtype)

Table 12. MPI-2 Data types constructors

Name of constructor
MPI_Type_create_hvector
MPI_Type_create_hindexed

Name of constructor
MPI_Type_create_struct
MPI_Type_create_subarrary
MPI_Type_create_darray

Table 13 displays MPI functions, which helps you build the derived data types. Again, the MPI-2 versions are not implemented in the IBM MPI.

Table 13. MPI utility functions for creating derived data types

Name
MPI_Type_commit(type)
MPI_Type_free(type)
MPI_Type_extend(type, extend)
MPI_Type_size(type, size)
MPI_Type_lb(type, displ)
MPI_Type_ub(type, displ)
MPI_Address(location, address)
MPI_Get_address (MPI-2)

Figure 14 on page 61 displays the general structure of the data type shown. Each data type contains:

- A sequence of data types
- A sequence of integer displacements

The displacements are not required to be positive, distinct, or in increasing order. A derived data type consists of n data types of type(i), each having the displacement displ(i), i=0,n-1.

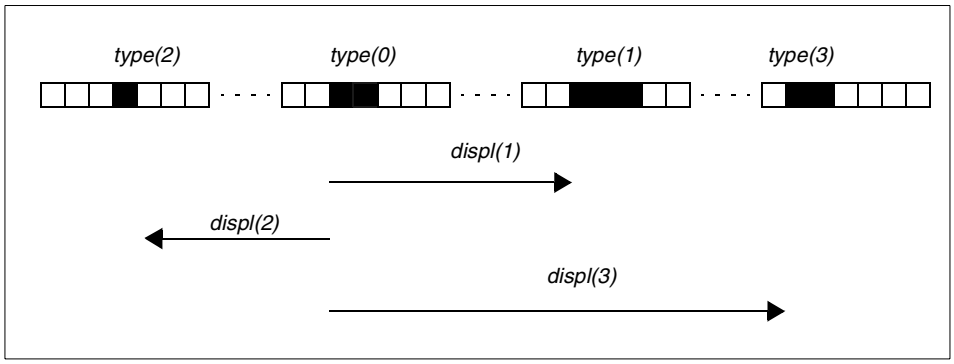


Figure 14. General derived data types

The MPI standard defines the type signature to be

```
typesig={type(0),..., type(n-1)}
```

and the type map to be

```
typemap = {(type(0),displ(0)),...,(type(n-1),displ(n-1)}.
```

The type map together with a base address, buf, specifies a communication buffer. The communication buffer consists of n entries where the i-th entry is at address buf+displ(i) and has the type type(i).

The basic data types listed in Table 14 on page 62 can also be expressed by a type map. For example, the basic data type MPI_INT has the type map, {(int,0)}, which is an integer at a displacement of 0.

The extent of a data type is defined to be the span from the first byte to the last byte occupied by entries in this data type and rounded to satisfy alignment requirements.

```
lb(typemap) = min displ(i)
ub(typemap) = max (dipsl(i)+sizeof(type(i)))+eps
extent(typemap) = ub(typemap)-lb(typemap)
```

This definition is only true if none of the types are of MPI_UB or MPI_LB. This case will be explained in more detail in Section 3.3.3, “Two dimensional parallel FFT” on page 63.

The simplest data type constructor is MPI_Type_contiguous(). It replicates one data type into contiguous locations.

To create a vector containing 100 single precision variables, you would type:

```
call MPI_Type_contiguous(100, MPI_REAL, newtype, mpierr)
```

A more general constructor is MPI_Type_vector(). It allows replication of a data type into locations that consist of equally-spaced blocks. Each block contains the same number of copies of the old data type. The spacing between each block is a multiple of the extent of the old data type. By using the second version, MPI_Type_hvector(), you can specify the blocking in bytes.

The next level of generalization is MPI_Type_indexed(). It allows each block to have a different length and displacement. Again, there is a version of this function, MPI_Type_hindex(), which takes the displacement in bytes rather than oldtypes.

The most general type function in MPI is MPI_type_struct(). In addition to MPI_Type_index(), it also allows all oldtypes to be distinct. This function can be used to model a FORTRAN 90 type or a C structure to an MPI data type.

3.3.2 Use of derived data types in collective communications

The derived data types can be used as the basic data types in all communication routines. The only problem is with routines, such as MPI_Reduce or MPI_Allreduce, that perform operations on the data. Because the reduction operations provided by MPI, such as MPI_SUM or MPI_MAX, are not defined for derived data types; you have to create them.

Table 14. Predefined MPI data types

Type	Permitted operations
MPI_INTEGER{1,2,4,8}	MPI_BAND, MPI_BOR, MPI_BXOR, MPI_MAX, MPI_MIN, MPI_PROD, MPI_SUM
MPI_REAL{4,8,16}	MPI_MAX, MPI_MIN, MPI_PROD, MPI_SUM
MPI_COMPLEX{8,16,32}	MPI_PROD, MPI_SUM
MPI_LOGICAL{1,2,4,8}	MPI_LAND, MPI_LOR, MPI_LXOR
MPI_CHARACTER	NONE
MPI_BYTE	MPI_BAND, MPI_BOR, MPI_BXOR
MPI_UB	NONE
MPI_LB	NONE

3.3.3 Two dimensional parallel FFT

In this section, we solve the two-dimensional FFT problem in parallel. Our test problem is the complex quadratic *n* x *n* matrix A:

$$A = \begin{bmatrix} (1,0) & (0,0) & \dots & (0,0) \\ (0,0) & (0,0) & \dots & (0,0) \\ \dots & & & \dots \\ (0,0) & \dots & \dots & (0,0) \end{bmatrix}$$

The algorithm used to solve the two-dimensional FFT takes advantage of the fact that you can express it with a series of one-dimensional FFT's executed first on the columns and then on the rows of the matrix. In order to keep the examples small, we only allow the number of used processes *np* to be a power of two and divide n.This leads to the following parallel algorithm:

1. Split the matrix A among *np* processes. Each process get a submatrix A_s with the dimension *n* x *(n/np).*
2. Each process solves the one-dimensional FFT for each column of the Matrix A_s. This involves no communication and is done by the ESSL routine dcft().
3. Transpose matrix A globally.
4. Solve the one-dimensional FFT for each new column of matrix A using the same routine as in step 2.

Figure 15 on page 64 shows the global transpose for np=4. Each MPI process will store only a part of the matrix. Each of these matrix parts is further divided into *np* quadratic submatrices; so, for example, process 1 stores the four submatrices A1 through A4 in its local memory. In order to transpose the matrix, we must do an Alltoall operation as described in Section 3.2.2, "Performance considerations" on page 53. After this operation, process 1 has the submatrices A1, B1, C1, and D1. Each of these submatrices must also be transposed locally.

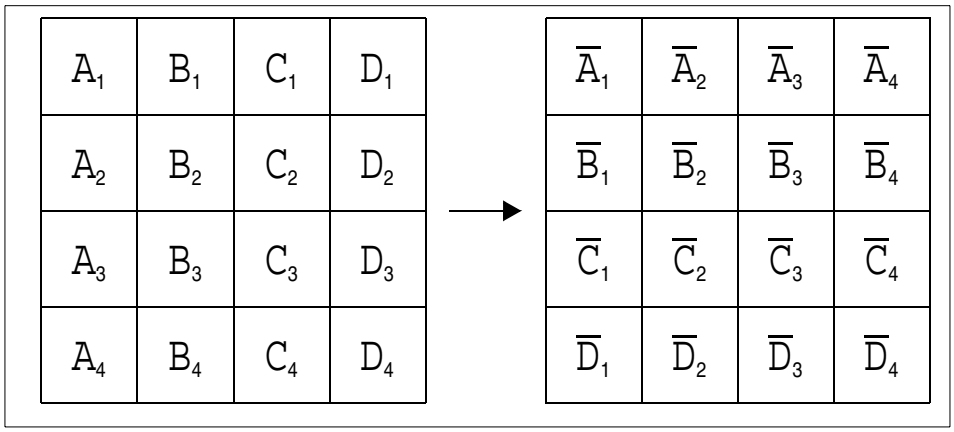


Figure 15. Matrix A for np=4

Knowing MPI, the obvious way to perform this operation is to use the MPI_Alltoall() call followed by the local transpositions, but there are other strategies possible. We implemented some of them. The timing results are listed in Table 15 on page 69.

The communication scheme using a buffered send and a blocking receive looks like:

```
call MPI_Buffer_attach(vectorbuf, size, mpierr)
do i=1,np-1
   dest=xor(myid,i)
   row=dest*(n1/np)+1
   do j=1, n2/np
      call MPI_Bsend(a(row,j), n1/np, MPI_COMPLEX16, dest, j,...)
      call MPI_Recv (a(row,j), n1/np, MPI_COMPLEX16, dest, j,...)
   end do! j
end do! i
call MPI_Buffer_detach(vectorbuf, size, mpierr)
```

The i-loop covers each submatrix that has to be sent. The j-loop sends all columns belonging to a specific submatrix. Since we restricted the number of processes to be a power of two, we use a hypercube processor topology to implement the communication. In a hypercube, we calculate the ID of the neighbors with the exclusive *or* function xor(). Since we have the same numbers of submatrices as we have processors, and since our numbering of the submatrices is consistent with the numbering of the processors, we use a destination variable to calculate the index of the submatrix to send and receive.

The buffered send needs a buffer in which the data is stored. Unlike PVM, you have to provide the buffer in MPI. In our case, this is done by allocating a buffer using the FORTRAN 90 capability of declaring a variable *allocatable.* In our case, the size of the buffer has to be at least:

```
(n2/np)*((n1/np)*sizeof(MPI_COMPLEX16)+MPI_BUFFER_OVERHEAD)
```

bytes. MPI_BUFFER_OVERHEAD is an MPI constant, which describes how many extra bytes MPI needs for the buffering. If you are sending many messages and are not sure when they will be received and you cannot check the status of your buffer, you should also add an extra amount of “safety bytes”. The buffer has to be made known to MPI by a call to MPI_BUFFER_ATTACH(). You can remove the buffer with MPI_BUFFER_DETACH(). The detach function will return as soon as all messages in the buffer have been transmitted. This is why you should not deallocate the buffer until it has been detached.

As can be seen in the Table 11 on page 59, this communication scheme is not the fastest (bsend no data types). In order to improve it, we define an MPI derived data type describing a submatrix. First, we define a subcolumn containing of n1/np MPI_COMPLEX 16 data types:

```
call MPI_Type_contiguous(n1/np, MPI_COMPLEX16, Type1, mpierr)
```

Because we do not want to MPI_Type1 directly in a communication operation, we do not have to commit it. The next step is to replicate this data type n2/np times:

```
call MPI_Address(a(1,1), address1, mpierr)
call MPI_Address(a(1,2), address2, mpierr)
displ=address2-address1
call MPI_Type_hvector(n2/np,1,displ, Type1, Type2, mpierr)
call MPI_Type_commit(Type2, mpierr)
```

By using the `Type2` data type in the communication, we get the following code structure:

```
call MPI_Buffer_attach(vectorbuf, size, mpierr)
do i=1,np-1
   dest=xor(myid,i)
   row=dest*(n1/np)+1
   call MPI_Bsend(a(row,j), 1, Type2, dest, j,...)
   call MPI_Recv (a(row,j), 1, Type2, dest, j,...)
end do! i
call MPI_Buffer_detach(vectorbuf, size, mpierr)
```

Besides being faster than the first approach (bsend column type), the communication pattern is written down more easily and, thus, is less error

prone. We still need the same extra buffer for the buffering. In order to reduce the buffer, we introduce an extra call to MPI_Barrier() in the i-loop, which is called every *memfact* iteration. In our example runs, we set memfact equal to 2. This allows us to make sure that all sends are completed and that the buffer is available for reuse:

```
memfact=2
size=(n1/np)*(n2/np)*16+MPI_BUFFER_OVERHEAD! 16=sizeof(complex)
allocate(vectorbuf(memfact*size))
call MPI_Buffer_attach(vectorbuf, size, mpierr)
do i=1,np-1
   dest=xor(myid,i)
   row=dest*(n1/np)+1
   call MPI_Bsend(a(row,j), 1, Type2, dest, j,...)
   call MPI_Recv (a(row,j), 1, Type2, dest, j,...)
   if (mod(i,memfact).eq.0) call MPI_Barrier(MPI_COMM_WORLD, mpierr)
end do! i
call MPI_Buffer_detach(vectorbuf, size, mpierr)
```

Surprisingly, this version is faster (bsend column barrier) than the version without a call to MPI_Barrier(). This is consistent with the finding in Section 3.2, “MPI collective communication” on page 39.

If we use the MPI_Alltoall() call to do the communication, the code will fail with the above definition of `Type2`. The reason for this is the extent of the data type. Figure 16 shows the memory layout of this derived MPI Type with *lb* and *ub* being the default lower and upper bounds of the data type.

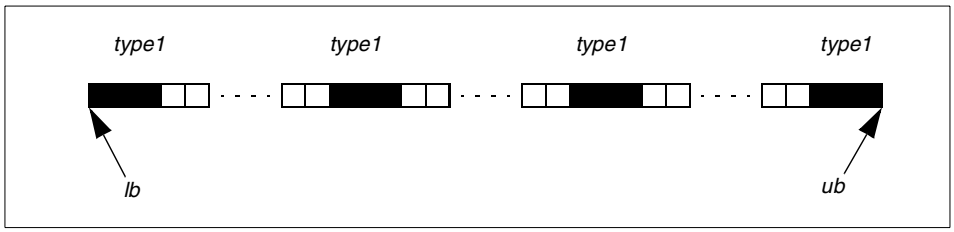


Figure 16. Memory layout of Type2

The problem is that the default, *ub*, is pointing to the end of `Type2`. The MPI standard says that you can view the receives in a collective communication, in our case, MPI_Alltoall(), as if each message from a task, i, was received as:

```
mpi_recv(recvbuf+i*recvcount*extent(recvtype),....)
```

So, instead of interleaving the data in memory as we want, we attach them together using memory that does not belong to our matrix. The solution to the

problem is to move the MPI_UB pointer after the first type1. This can be seen in Figure 17.

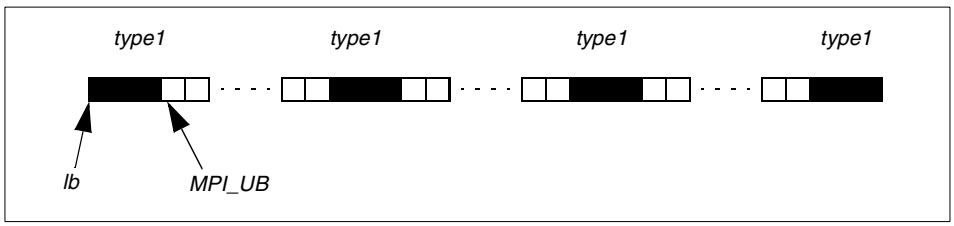


Figure 17. Memory layout of Type2 with MPI_UB moved

The easiest way to do this would be to use our Type2:

```
types(1)=TYPE2
types(2)=MPI_UB
len(1)=1
len(2)=1
displ(1)=0
call MPI_Address(a(1,1), address1, mpierr)
call MPI_Address(a(n1/np+1,1), address2, mpierr)
displ(2)=address2-address1
call MPI_Type_struct(2, len, displ, types, Type3, mpierr)
```

Another approach using Type1 and FORTRAN 90 vector notion would be:

```
types(1:n2/np)=Type1
types(n2/np+1)=MPI_UB
len(1:n2/np+1)=1
call MPI_Address(a(1,1), address1, mpierr)
call MPI_Address(a(1,2), address2, mpierr)
displ(1)=0
displ(2:n2/np)=displ(1:n2/np-1)+address2-address1
call MPI_Address(a(1,1), address1, mpierr)
call MPI_Address(a(n1/np+1,1), address2, mpierr)
displ(n2/np+12)=address2-address1
call MPI_Type_struct(n2/np+1, len, displ, types, Type3, mpierr)
```

Which version to use is a matter of taste and programming style. There is, at least for the IBM RS/6000 SP, no performance difference.

There are some problems with using the above approach to move the upper or lower bound in a data type. First, MPI_UB and MPI_LB are “sticky”. This means that you cannot remove them once you put them into a data type. This is important if you are building new data types using data types that already

have MPI_UB or MPI_LB set directly. For example, consider a case where we set MPI_UB explicitly for Type2 to point to the standard position as in Figure 16. We then use this new data type to build Type3 as in the first example above. Type3 would contain two MPI_UB's as shown in Figure 18. MPI_UB_1, the MPI_UB with the highest memory address, would be used to determine the extent of Type3, which, again, would be wrong for collective communications.

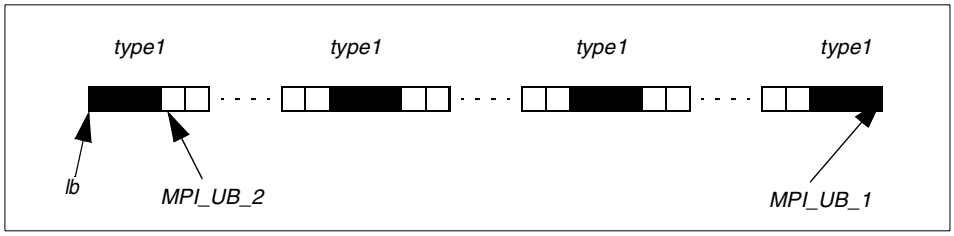


Figure 18. Memory layout of Type2 with a "sticky" upper bound

The second problem with MPI_UB and MPI_LB is that the code gets more error-prone because we have to add more lines of code to build the data type.

In MPI-2, there is a function to solve both problems:

```
MPI_Create_type_resized(oldtype, lb, extend, newtype, mpierr)
```

This function generates a new data type that is identical to the old data type - except that the lower bound is set to `lb`, and the upper bound is set to `lb+extend`. It also removes extra MPI_LB and MPI_UB in the new data type. To create Typ3, the previous example would look like the following:

```
call MPI_Address(a(1,1), address1, mpierr)
call MPI_Address(a(n1/np+1,1), address2, mpierr)
extend=address2-address1
call MPI_Type_create_resized(Type2, 0, extend, Type3, mpierr)
```

Note that this function was not part of the IBM MPI version when this publication was written.

Looking closer at the algorithm, we see that it has three parts: The one-dimensional FFT calculation, the Alltoall communication, and the local transposition of each submatrix. Since we are using ESSL to solve the one-dimensional fft's, there is no optimization to be done here. We already discussed the communication, leaving the transposition open. So far, we have done this transposition using a subroutine, which, by unrolling, uses all four prefetch streams of the POWER3 chip.

Another approach to the transposition is to let MPI do the work because MPI does not require that the send and receive type be identical; it only requires that the type signature on the receiver side match the type of signature of the sender; so, by defining row-oriented submatrices as the opposite of the above definition of column-oriented submatrices, we can let MPI do both the local and global transposition.

```
call MPI_Type_vector(n2/np, 1, lda, MPI_COMPLEX16, MPI_Type1, mpierr)
call MPI_Type_hvector(n1/np, 1, 16, MPI_Type1, MPI_Type2, mpierr)
```

If we are using a collective communication, we also have to move the MPI_UB as shown in the preceding code.

We have also tested the MPI_Pack and MPI_Unpack routine. As can be seen, they perform about as well as our first version using MPI_Bsend().

The results of our runs are documented in Table 15. The numbers reported are seconds as measured with the FORTRAN function rtc(). The Total Time is the time to do the two-dimensional FFT on a double complex Matrix with the dimension of 4096 x 4096. The first row is the timing for the two-dimensional FFT solver provided by PESSL. The second row is our first implementation followed by the version using the MPI submatrix column-oriented data type. The fastest version is our implementation with calls to the MPI_Barrier(). The next five rows show timings for different data types. The ssend example tries to reduce the memory needed to synchronize the sends and receives. The remaining three versions use the standard Alltoall implementation and the MPI_Pack routine.

Table 15. Timings for different parallel FFT's (n1=n2==4096, np=4)

Testcase	Total Time	Comm. Time	Local Trans. Time
Parallel ESSL pdcft2	3.338	---	---
bsend no data types	3.546	1.456	.336
bsend column type	3.275	1.182	.335
bsend column barrier	3.246	1.152	.335
bsend column/row	4.580	2.739	.084
bsend row/column	5.084	3.242	.084
bsend column thread	3.288	1.195	.335
bsend 1d column/row	4.636	2.795	.084
bsend 1d row/column	5.722	3.879	.084

Testcase	Total Time	Comm. Time	Local Trans. Time
ssend column (gen.)	3.551	1.464	.331
All2All column	3.654	1.279	.621
Pack standard	3.599	1.508	.335
Pack column	3.364	1.272	.335

Conclusions:

The idea of letting MPI take care of the local transposition was not a good one. Even if the local transposition time went down, the increased communication time was far higher. Knowing that we are transposing a matrix and not simply moving data allows us to write a more efficient routine than MPI can provide.

In this testcase, the version using MPI_Barrier performed a little better than the PESSL version. There were several other test cases in which the PESSL version were slightly faster and, considering that our versions are not yet as general as the ESSL version, it is safe to say that the PESSL version is a very good choice from a performance point of view. The main advantage our code has is that it uses much less memory space than the PESSL version. The MPI call Alltoall() requires that the input and output buffer of the communication be disjointed. This increases the memory needed to solve the problem by n1*n2/np data elements. Using our version, the memory increase is only memfact*(n1/np)*(n2/np). This memory advantage would allow us to solve the same problem as PESSL using a smaller number of MPI tasks or to solve a bigger problem using the same number of tasks.

3.3.4 Domain splitting

Another good example for MPI is the Domain Splitting method. It assumes you have a domain that you split onto different MPI processes. The problem with this approach is that you will have to exchange border values between the different processes as shown in Figure 19. In general, you have to send information between processes that share a common edge. In our example, this involves communication between domains A and B and B and C. Depending on your problem, you might have communication between domains sharing a common edge, C and A in our example. If you have to do this, you might need a total of eight communications. There is an algorithm that, in a two-dimensional domain splitting, reduces the number of communications needed from eight unordered to four ordered ones. In a

three-dimensional problem, you would only need six communication steps instead of 26. Figure 19 illustrates domain splitting with nine domains.

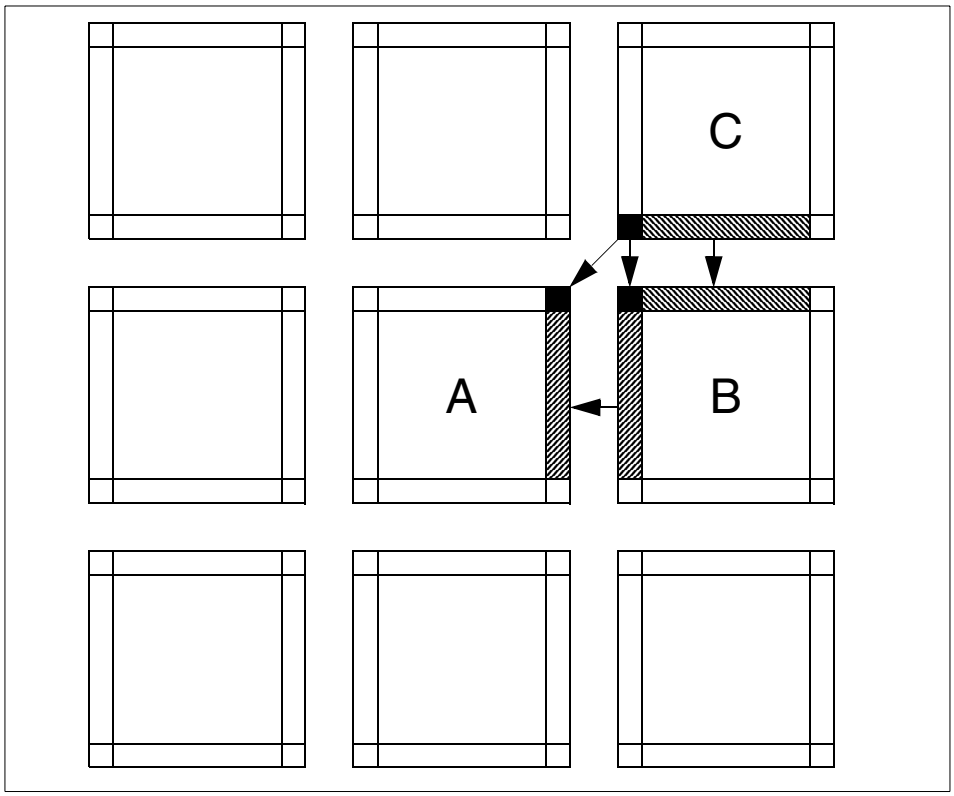


Figure 19. Domain splitting with nine domains

Here is a code fragment that transfers MPI_COMM_WORLD into a two dimensional Cartesian topology, defines two MPI data types on the border, and, finally, uses MPI_Sendrecv() to do the update. The trick is to define the data types to include the corners and also transfer them. In the code fragment, the update of the corner between each communication step is missing; this can be a problem depending on how the transposition is done.

```
call MPI_Cart_create(MPI_COMM_WORLD, dim, dims, period, reorder,
comm_cart, mpierr)
call MPI_Cart_coords(comm_cart, myid, dim, coords, mpierr)

! Get my neighbors
call MPI_Cart_shift(comm_cart, 0, 1, west, east, mpierr)
call MPI_Cart_shift(comm_cart, 1, 1, north, south, mpierr)
```

```
! Build border
call MPI_Type_contiguous(n1/ny, MPI_REAL8, border_ns, mpierr)
call MPI_Type_vector(n2/nx, 1, n1, MPI_REAL8, border_we, mpierr)

! Exchange borders
  call MPI_Sendrecv (domain(1,n2-1), 1, border_ns, east, 1,&
&                    domain(1,1)   , 1, border_ns, west, 1,&
&                    comm_cart, status, mpierr)
  call MPI_Sendrecv (domain(1,2)   , 1, border_ns, west, 2,&
&                    domain(1,n2)  , 1, border_ns, east, 2,&
&                    comm_cart, status, mpierr)
  call MPI_Sendrecv (domain(n1-1,2), 1, border_we, south, 3,&
&                    domain(1,1)   , 1, border_we, north, 3,&
&                    comm_cart, status, mpierr)
  call MPI_Sendrecv (domain(2,2)   , 1, border_we, north, 4,&
&                    domain(n1,1)  , 1, border_we, south, 4,&
&                    comm_cart, status, mpierr)
```

The whole domain has the size of n1 $\times$ n2 grid points, and it is mapped onto an ny $\times$ nx processor grid. In Figure 19 on page 71, a 3 $\times$ 3 processor grid is shown.

This example is written using a structured grid called `domain`. The change to a nonstructured grid or even a domain without a grid is rather easy. MPI provides the function MPI_Graph_create() to describe a more general processor topology than the above Cartesian one. If your border is irregular but still constant in time, you can still define a data type describing it using MPI_Type_indexed() or MPI_Type_struct(). In the more general case, where your border is not constant in time, you might want to use `MPI_Pack()` to set up the buffers for the communication.

3.4 MPI Performance assessment

After discussing various methods of efficient implementation in MPI in the previous subsections, this subsection covers some easy ways of assessing the performance of a parallel program. First, different ways of using the time/timex command are discussed, followed by a discussion of different timing calls in the program. MPI offers a portable API for profiling, which is introduced next. The profiling support provided by the parallel operating environment (POE) and, in particular, the visualization of profiling results by xprofiler is highly recommended. The `poe` command also offers a trace facility that allows detailed insight into various aspects of the parallel program.

3.4.1 Timing considerations

The quickest way to get some timing information is by using the /bin/time command. But, there are some caveats to it. On the SP, the poe command is used to start a parallel program (unless mpich is used). When /bin/time precedes poe, the result looks like the following:

```
v07n01:/u/pospiech/Redbook $ /bin/time poe iall2all 8000 1 10000 -procs 6

[ ... ] (Program output)

Real   70.68
User   0.06
System 0.09
v07n01:/u/pospiech/Redbook $
```

This way, the overall wall clock time for requesting six processors and loading and executing the parallel program is measured. The user time refers to the time used by poe, which, obviously, does not include the CPU time of the parallel program. The situation does not change if time is replaced by timex.

```
v07n01:/u/pospiech/Redbook $ /bin/timex poe iall2all 8000 1 10000 -procs 6

[ ... ] (Program output)

real 69.12
user 0.09
sys  0.12
v07n01:/u/pospiech/Redbook $
```

Unlike time, timex also includes CPU and Real time of all child processes of the process under investigation. But, poe does not spawn the MPI tasks; rather, it starts them by a process similar to rsh; so, these processes are not accounted for by timex, and the difference between using time or timex this way is negligible.

In order to get CPU time on the parallel program, the order of time and poe should be exchanged.

```
v07n01:/u/pospiech/Redbook $ poe /bin/time iall2all 8000 1 10000 -procs 6

[ ... ] (Program output)

   3:
   3:Real   60.71
   3:User   36.64
   3:System 1.95
   2:
```

```
   2:Real   60.71
   2:User   35.70
   2:System 2.93
   0:
   0:Real   60.66
   0:User   51.75
   0:System 2.23
   1:
   1:Real   60.66
   1:User   52.21
   1:System 2.28
   4:
   4:Real   60.72
   4:User   55.04
   4:System 2.13
   5:
   5:Real   60.72
   5:User   54.93
   5:System 2.24
v07n01:/u/pospiech/Redbook $
```

In this case, /bin/time gets the role of the parallel program. Thus, the timing information is printed for each MPI task. poe actually copies the complete environment to the parallel tasks including the value of $PATH; so, the specification of the path for /bin/time is not necessary. The User time now has a meaningful value. The Real time is somewhat less now because the time for requesting the processors is no longer included. Note that the same trick does not work for timex as can be seen from the following.

```
v07n01:/u/pospiech/Redbook $ export EUILIB=ip
v07n01:/u/pospiech/Redbook $ poe /bin/timex iall2all 8000 1 10000 -procs 6
   1:ERROR: 0031-303  mp_euilib specifies us, css library loaded is not us.
   0:ERROR: 0031-303  mp_euilib specifies us, css library loaded is not us.
   0:
   0:real 0.31
   0:user 0.00
   0:sys  0.01
   0:
   1:
   1:real 0.18
   1:user 0.00
   1:sys  0.01
   1:
ERROR: 0031-250  task 2: Terminated
ERROR: 0031-250  task 3: Terminated
ERROR: 0031-250  task 4: Terminated
```

```
ERROR: 0031-250  task 5: Terminated
v07n01:/u/pospiech/Redbook $
```

The reason that timex does not work as expected is because timex is a "set uid" program. Such programs reset LIBPATH to NULL for security reasons, but poe relies on being able to set LIBPATH to select between the User Space and IP libraries, with the IP library as the default. In the example, had you set MP_EUILIB=ip (not EUILIB=ip), it would have worked. If you need User Space, you must specifically set -L/usr/lpp/ppe.poe/lib/us in your mpcc (or mpxlf) compilation.

There is actually a point on how to interpret CPU-time for a parallel program. Each MPI task is doing computation and communication, which may even overlap. Even if they do not, part of the communication time accounts for CPU time. Some other part is spent by the CPU on the adapter and may not be caught at all; so, CPU time is very difficult to interpret.

Experience shows that, in many cases, wall clock time can be fitted quite well to some model derived from Amdahl's law; so, some reasonable interpretation can be easily derived from these timings. On the other hand, reproducible results (if there are any) can only be obtained on a dedicated system.

For timings inside the program, the Fortran function second() may not be the right choice, given the preceding discussion. A better choice would be the rtc() function, available only with xlf, which returns wall clock time in seconds as a double precision value. Calling this function before and after the code segment to be timed, the difference of both results is the wall clock time spent in this code segment.

There is also a portable way, because MPI also defines a timer. Even though it is not “message-passing”, a timer is specified because existing timers (in both POSIX 1003.1-1988 and 1003.4D 14.1 and Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers.

```
MPI_WTIME()

double MPI_Wtime(void)
DOUBLE PRECISION MPI_WTIME()
```

MPI_WTIME returns a floating-point number of seconds representing elapsed wall-clock time since some time in the past.

The “time in the past” is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not “ticks”); it allows higher resolutions and carries no unnecessary baggage. One would use it in the following way:

```
{
   double starttime, endtime;
   starttime = MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = MPI_Wtime();
   printf("That took %f seconds\n",endtime-starttime);
}
```

All timing in this chapter was done this way.

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns the number of seconds between successive clock ticks as a double precision value:

```
MPI_WTICK()

double MPI_Wtick(void)
DOUBLE PRECISION MPI_WTICK()
```

The resolution of MPI_WTIME on the SP depends on the actual hardware clock that is being used for this function. AIX provides a clock on each node, and there is also a synchronized clock on the switch. The MP_CLOCK_SOURCE environment variable provides additional control on this. Of course, the actual clock source depends on the availability of hardware and is determined by Table 16.

Table 16. Clock source (MP_CLOCK_SOURCE variable)

MP_CLOCK_SOURCE	Library Version	All Nodes SP?	Source used
not set	ip	yes	switch
not set	ip	no	AIX
not set	us	yes	switch
not set	us	no	Error
SWITCH	ip	yes	switch
SWITCH	ip	no	AIX

MP_CLOCK_SOURCE	Library Version	All Nodes SP?	Source used
SWITCH	us	yes	switch
SWITCH	us	no	Error
AIX	ip	yes	AIX
AIX	ip	no	AIX
AIX	us	yes	AIX
AIX	us	no	AIX

The time provided by MPI_WTIME may not be global (synchronized on all nodes for example). The standard does not require this. It is only global if the clock source is the switch. The parallel environment guarantees that the MPI_COMM_WORLD attribute, MPI_WTIME_IS_GLOBAL, is set accordingly to true or false on all nodes.

3.4.2 MPI intrinsic routines

MPI comes with its own profiling interface, which is just that - an interface. It says nothing about the way in which it is used. What information is collected through the interface or how the collected information is saved, filtered, or displayed is entirely up to the user.

According to the MPI-1.1 standard, the objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

Since MPI is a machine-independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It is, therefore, necessary to provide a mechanism by which the implementers of such tools can collect whatever performance information they wish without access to the underlying implementation.

To accomplish this:

- All of the MPI defined functions may be accessed with a name shift. Thus, all of the MPI functions, which normally start with the prefix “MPI_”', are also accessible with the prefix “PMPI_”.
- Those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.

Suppose that the user wishes to accumulate the total amount of data sent by the MPI_SEND function along with the total elapsed time spent in the function. This could trivially be achieved in the following manner:

```
static int totalBytes;
static double totalTime;

int MPI_Send(void * buffer, const int count, MPI_Datatype datatype,
             int dest, int tag, MPI_comm comm)
{
   double tstart = MPI_Wtime();       /* Pass on all the arguments */
   int extent;
   int result    = PMPI_Send(buffer,count,datatype,dest,tag,comm);

MPI_Type_size(datatype, &extent);  /* Compute size */
   totalBytes += count*extent;

totalTime  += MPI_Wtime() - tstart;         /* and time          */

return result;
}
```

To run this profiling, the following must be done

- Compile the above function with mpcc, and generate the file MPI_Send.o.
- Include MPI_Send.o in the list of objects for linking.
- Run the resulting executable normally.

The MPI-1.1 standard also provides an interface for dynamically controlling the profiling at run time. This is normally used for (at least) the purposes of:

- Enabling and disabling profiling depending on the state of the calculation
- Flushing trace buffers at non-critical points in the calculation
- Adding user events to a trace file

These requirements are met by using MPI_PCONTROL:

```
MPI_PCONTROL(level, ...)
[ IN level] Profiling level
```

C interface:

```
   int MPI_Pcontrol(const int level, ...)
```

Fortran interface:

```
   MPI_PCONTROL(level)
                      INTEGER LEVEL
```

MPI libraries themselves make no use of this routine and simply return immediately to the user code. However, the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, the standard does not specify the number and types of arguments. To provide some level of portability of user codes to different profiling libraries, the following meanings for the first argument to MPI_PCONTROL are suggested in the standard.

- level==0 - Profiling is disabled.
- level==1 - Profiling is enabled at a normal default level of detail.
- level==2 - Profile buffers are flushed. (This may be a no-op in some profilers).

All other values of level have profile library-defined effects and additional arguments.

The provision of MPI_PCONTROL as a no-op in the standard MPI library allows them to modify their source code to obtain more detailed profiling information but still be able to link exactly the same code against the standard MPI library.

3.4.3 gprof profiling

The Application Development Toolbox on AIX (ADT) provides another way of getting a code profiled. This applies to both serial and parallel applications.

To get the profiling started, the code has to be recompiled and linked with the option -pg. Failing to link with -pg is, by far, the most frequent user error of an inexperienced user of this tool. If properly recompiled and linked, the program only has to be rerun in the same way as without profiling enabled. The -pg option causes the compiler to insert a call to the mcount subroutine into the object code generated for each recompiled function of your program. During program execution, each time a parent calls a child function, the child calls the mcount subroutine to increment a distinct counter for that parent-child pair. Programs not recompiled with the -pg option do not have the mcount subroutine inserted and, therefore, keep no record of who called them.

However, some profiling features are only available if the code has been additionally compiled with -g and possibly with -qfullpath. The -g option adds debugging information into the code. With this information, each assembler opcode can be traced back to the original line of code from which it was generated. This information includes the file name and the location of the line

of code given by relative path names. If either the object decks or the executable are relocated prior to execution, the pathnames should be absolute rather than relative. This feature is invoked by specifying the -qfullpath option . The -g option is compatible with any optimization level. It is recomended to use it with the same optimization level that has ben used for the non-profiling run. This gives more reliable information than turning off optimization despite the fact that the optimizer may “damage” the debugging information. As a matter of experience, the amount of “damage” is, in general, quite moderate, and little harm is done by it, provided the results are interpreted with some care.

A serial run produces one extra file with the fixed name gmon.out. poe is adapted to use this tool. A parallel run produces files gmon.out.<task-Id>, where task_Id is a number ranging from 0 to one minus the number of MPI tasks used for this run. The gmon.out.<task-Id> files are created in the user’s current directory, initially named gmon.out, and then renamed gmon.out.<task_id>.

The gmon.out files are binary, and two tools are available to make this information readable. The syntax of the first is as follows:

```
/usr/ucb/gprof [ .-b ] [ -e Name ] [ -E Name ] [ -f Name ] [ -F Name ] [ -L
PathName ] [ -s ] [ -z ] [ a.out [ gmon.out ... ] ]
```

The `gprof` command produces three items:

- A flat profile is produced similar to that provided by the prof command. This listing gives total execution times and call counts for each of the functions in the program, sorted by decreasing time. The times are then propagated along the edges of the call graph. Cycles are discovered and calls into a cycle are made to share the time of the cycle.
- The functions are sorted according to the time they represent including the time of their call-graph descendents. Below each function entry are its (direct) call-graph children with an indication of how their times are propagated to this function. A similar display above the function shows how the time of the function and the time of its descendents are propagated to its (direct) call-graph parents.
- Cycles are also shown with an entry for the cycle as a whole and a listing of the members of the cycle and their contributions to the time and call counts of the cycle.

While `gprof` produces ASCII output, Parallel Environment also provides a user-friendly GUI for examination of gmon.out files, which is called Xprofiler.

Though provided by the Parallel Environment, Xprofiler can also be used for serial programs.To run Xprofiler, type:

```
xprofiler a.out gmon.out [gmon2.out gmon3.out ...]
```

where `a.out` is the application's binary executable file, and one or more gmon.out files can be specified. This tool is almost self-explanatory and comes with an extensive help function. In addition to gprof, it also provides profiling at the source code level, provided the executable has been compiled with -g.

3.4.4 IBM trace interface

The Parallel Operating Environment (POE) comes with a built-in trace file generator that is unleashed by setting the value of MP_TRACELEVEL to a nonzero value prior to parallel execution. No recompilation or relink of the code is needed.

However, some features are only available if the code has been compiled with -g and possibly with -qfullpath.

If the code is rerun with a non-zero value of MP_TRACELEVEL set, a trace file is generated. This trace file may contain the following:

- Separate entries for the start and end of each call of an MPI communication routine - Each entry contains all relevant information about this MPI call including message sizes and the status of completion and is equipped with a time stamp generated by the synchronized switch hardware clock, if available.
- Entries for AIX kernel statistics - Among much other information, each entry contains average CPU usage and disk I/O from this MPI task.
- Entries for application markers - Application markers can be defined by the user to mark different code portions.

Table 17 explains which of these features are switched on and off by different values of MP_TRACELEVEL.

Table 17. MP_TRACELEVEL values

Flag Value	Communication	AIX Kernel Statistics	Application Markers
0	NO	NO	NO
1	NO	NO	YES
2	NO	YES	YES

Flag Value	Communication	AIX Kernel Statistics	Application Markers
3	YES	NO	YES
9	YES	YES	YES

Due to this detailed information, the generated trace files tend to be large. A couple of features are provided to control the size of the trace file. None of them are found to work without some complications.

To view the trace file, Parallel Environment provides the graphical interface vt.

Synopsis

```
vt [-tracefile trace_file] [-tfile trace_file] [-configfile
configuration_file] [-cfile configuration_file] [-spath directory_list]
[-norm] [-cmap] [-go]
```

The `vt` command starts the Visualization Tool for visualizing performance characteristics of a program or the system. This X-Windows tool consists of a group of displays that present specific, often complex information in easily-interpretable forms, such as bar charts and strip graphs. VT can be used to play back traces generated during a program's execution (trace visualization) or as an online monitor to study the operational status and activity of processor nodes (performance monitoring).

Flags

- **-tracefile** or **-tfile** loads a specified trace file for playback. (A trace file can also be loaded after VT is started.)
- **-configfile** or **-cfile** loads a specified configuration file. A configuration file contains previously-saved arrangements of VT windows as well as input field specifications. (A configuration file can also be loaded after VT is started.)
- **-spath** indicates a search path to a program's source code. Like the AIX PATH environment variable, this is a series of colon-delimited directory names to search. Unless the program's source is in the current directory, the search path is needed to display it in the VT's Source Code view. (A search path to the program's source code can also be indicated after VT is started.)
- **-norm** indicates that LoadLeveler is unavailable. In performance monitoring mode, VT normally uses LoadLeveler to learn which nodes are available for monitoring. If this flag is specified, VT instead gets this information from the host list file indicated by the MP_HOSTFILE

environment variable and from the LAN. If you are going to use VT for online performance monitoring of a cluster or mixed environment, you must use this flag.

- **-cmap** requests a private color map. If this flag is not used, VT attempts to use the default color map shared by all active X-Windows applications. Depending on the number of active X-Windows applications, there might not be enough available colors for VT. When this happens, VT displays a message indicating the spectrums it cannot allocate and uses black in place of the unallocated colors. VT will still run, but, in extreme cases, some display spectrums may be unusable because of the missing colors. When you use this flag, VT makes a private copy of the default X-Windows color map.
- **-go** starts playing back the trace file immediately upon starting VT. When you use this flag, you must also specify a trace file and a configuration file using the -tracefile (or -tfile) flag and the -configfile (or -cfile) flags.
- **-log_file** specifies the file name where the results of the trace file post-processing will be written. The default name is $HOME/tracefilename.pplog.
- **-h**, **-?**, or **-helpVT_trc.h** gets help information.
- **-mp_source** specifies which task's source code is displayed in the Source Code view.

The structure of the trace file is open. The header file for the trace records can be found in /usr/lpp/ppe.vt/include. These files enable a C programmer to write his or her own program for working on the trace file. Each record in the trace file starts with a common header, which contains the time stamp, the process ID, the thread ID and the type of the following body of this record. Any of these types are coded by a unique pair of numbers, which are specified in the file VT_trc.h. The structure (and, thus, the length) of every trace file record is described in either of the files VT_mpi.h or VT_trc.h .

3.5 Low-level Application Programming Interface (LAPI)

The Low-level Application Programming Interface (LAPI) is a low-level programming interface designed to provide optimal communication performance on the IBM SP equipped with a high-performance switch. It offers remote memory copy, active messages, and synchronization operations. Remote memory copy operations enable any process in a parallel program to access data residing in the address space of another process. This capability facilitates a system-wide shared memory-like programming model. In addition, LAPI allows you to send a message to another process

that, upon arrival, will invoke a handler function (active message communication). A variety of mechanisms are provided for interprocess and communication synchronization. LAPI has been designed for use by libraries and power programmers for whom performance is more important than code portability.

3.5.1 Concepts

This section explains basic concepts in LAPI programming to a reader familiar with other communication models, such as MPI.

LAPI offers a more flexible and lower-level interface than MPI. In fact, the MPI interface can be implemented on top of LAPI. Unlike the point-to-point message-passing model (MPI-1), which requires cooperation between sender and receiver processes to move data between a pair of processes, LAPI operations are unilateral. This means that any process can initiate a data transfer between its local memory and the memory of another (remote) process and that the data transfer will complete without any explicit participation of the remote process. To expose application concurrency in communication over the SP Switch, the LAPI remote memory copy (put and get) were designed to operate in an asynchronous (non-blocking) fashion. The user calls initiate a data transfer operation and return to the caller before the data transfer is complete. This approach allows applications to overlap data transfer with computations.

Remote memory copy operations in LAPI resemble the put/get operations in the MPI-2 (a set of extensions to MPI standard approved by MPI Forum in 1997) one-sided communication model. However, the two models differ in terms of progress rules (LAPI is less restrictive) and functionality (the MPI model is a higher level than that of LAPI). In addition to remote memory copy, LAPI, unlike MPI-2, offers active message operations. LAPI lacks the accumulate operation of the MPI-2 specification. However, the accumulate operation can be fairly easily implemented with the LAPI active messages.

Active message (AM) functionality in LAPI allows any process to send a special type of a message (active message) to another process. The message contains an address of a handler function that will be invoked in the address space of the remote process upon the message arrival. In addition to the function address, the message can deliver data to be available in the active message handler when it is executing at the remote process. An important difference between the active message and the traditional message passing models is that the active messages are unilateral, that is, the AM send operation does not have or need a matching receive operation.

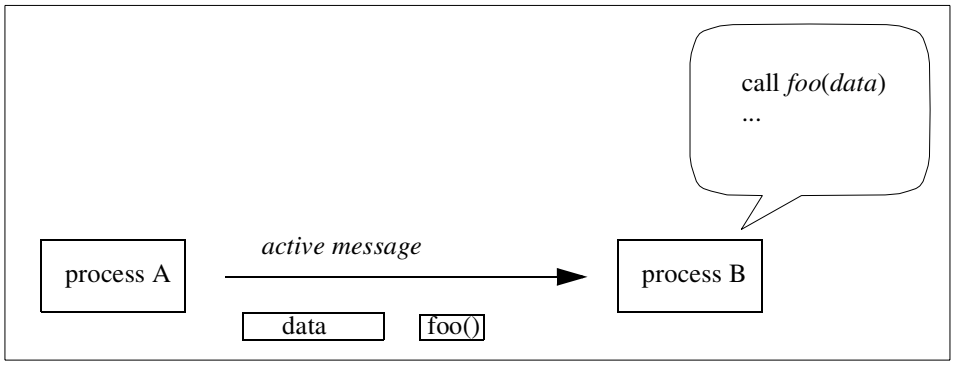


Figure 20. Active message concept

LAPI provides a set of synchronization mechanisms for:

- Synchronization of all processes (*global fence*), similar to MPI_Barrier
- Synchronization of communication operations that allow tasks to wait for completion of outstanding asynchronous communication operations

To support mutual exclusion in a shared memory style of programming, interprocess synchronization can be implemented by using the atomic read-modify-write operation available in LAPI. This operation updates an integer variable in the remote process memory in an uninterruptible atomic fashion and returns the original value of this variable to the calling process. With this operation, shared memory synchronization operations, such as locks or semaphores, can be implemented.

3.5.1.1 Properties of the SP Switch network

In order to use LAPI effectively, some knowledge of the IBM SP interconnect properties is useful. The IBM SP network is a low-latency high-bandwidth network with an omega-type interconnect topology. It is integrated with the processing nodes through the SP Switch adapter. All the user-level communication libraries available on the machine transfer data across the network in packets. The packet size is fixed and equals 1024 bytes in the most recent implementations of the network (as of 1999). Since the network uses multiple-path adaptive routing, packets sent between any pair of nodes can arrive out of order. In addition, the network hardware does not gurantee reliable delivery of 100 percent of the packets.

3.5.1.2 Terminology

The following are important terms:

- **target/origin** - LAPI uses the term of *origin* task/process in the context of active messages and remote memory copy operations to identify the task/process that initiates a communication operation. The term *target* is used to identify the remote task/process whose address space is targeted by such an operation.
- **nonblocking/blocking** - These terms are used to distinguish between the operations that return to the caller before (nonblocking) or after (blocking) the data transfer completes.
- **LAPI messages** - LAPI messages are communication operations that move data between origin and target and/or trigger the execution of the user-specified handler function at the target process. Remote memory copies and active messages are different user interfaces to the for LAPI messages.

3.5.1.3 Infrastructure of LAPI

LAPI messages and Pthreads form an underlying infrastructure of LAPI. LAPI makes extensive use of threads to achieve good performance.

Remote memory copy operations in LAPI are implemented with LAPI messages. LAPI provides a generalized yet efficient mechanism for customizing the communication interface to the specific requirements by allowing applications to use their own handlers for the active messages. In particular, applications can implement their own versions of remote memory copies on top of active messages instead of using the ones provided.

LAPI supports error-free (reliable) interprocess communication. It uses the Hardware Abstraction Layer (HAL) library that encapsulates the network adapter-specific programming interfaces. HAL is responsible for sending and receiving packets through the SP network, but it does not assure reliability. LAPI assembles and disassembles packets into and from messages and assures that data is delivered reliably. LAPI supports messages that can be larger than the size supported by the underlying HAL. Data sent with an active message will arrive in multiple packets; furthermore, these packets can arrive out-of-order.

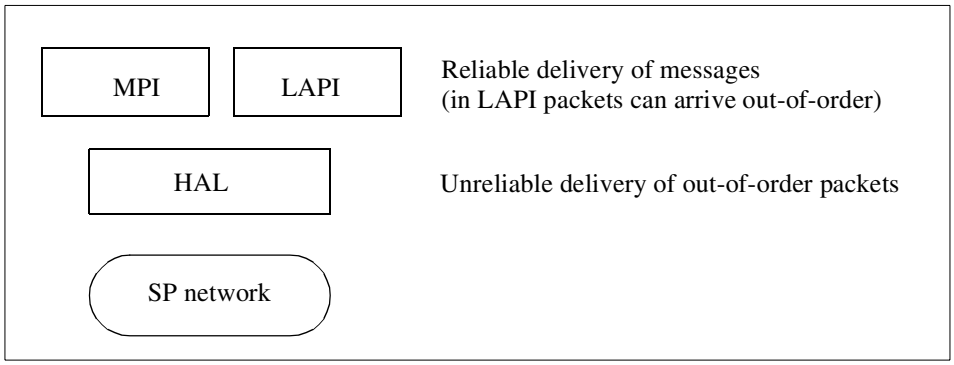


Figure 21. Hierarchy of communication libraries

LAPI is a thread-safe library and can be used by multithreaded applications. LAPI itself is implemented with multiple threads. At initialization time, the library creates two additional threads in the context of each user task. These threads are blocked and do not consume CPU cycles in absence of the LAPI communication traffic. They can be activated when they are needed to process incoming LAPI messages. A centerpiece of the LAPI implementation is the *LAPI dispatcher*. This software layer deals with the arrival of messages and the invocation of handlers. LAPI dispatcher can be executed by the application thread when it makes a LAPI function call, or it can be executed by one of the threads that LAPI creates at initialization time.

When the active message brings with it data from the originating process, the library requires that the “handler” be written as two separate routines:

- A *header* handler function: This is the function that is specified in the active message call. It will be called when the message first arrives at the target process, and it provides the LAPI dispatcher with:
  - An address where the arriving data must be copied
  - The address of the optional *completion* handler
- A *completion* handler will be called after the whole message has been received. If the user supplies NULL as the completion handler, after the data has been copied into the user-specified buffer, the active message completes.

The Pthreads library provides locking functionality, which is needed to protect user’s data structures that may be shared by user and LAPI threads in the same task.

3.5.1.4 Interrupt and polling modes of communication processing

Data transfers implemented by LAPI, MPI, and other communication libraries require interactions with the hardware and the operating system. There are two alternative modes in which communication libraries interact with the network interfaces: Polling and interrupts.

Polling mode requires the user process to check the network interface for occurrences of communication-related events, such as the arrival of a message, and then initiate appropriate processing of such events, for example, assembling packets into an MPI message. The network interface is usually accessed implicitly from a communication library whenever a call to the communication library is made by the application.

In the interrupt mode, a user process does not need to make any calls to the communication library to assure communication progress. The execution of the user task is interrupted, and the necessary processing of communication events is accomplished through appropriate interrupt handler functions. They are invoked by the system when a hardware event is detected. The handlers can be provided by a communication library or defined by the user (for example, active message handlers in LAPI or *rcvncall* handlers in the pre-MPI IBM communication library called MPL on early implementations of the IBM SP architecture).

The polling mode can offer a lower overhead of message processing than the interrupt mode. However, this benefit can be realized when the arrival of messages is expected, and the application can pass control to a communication library in advance, for example, by issuing a blocking receive call. Selection of the appropriate mode (interrupt or polling) depends on the nature of the application. For regular and/or well-balanced applications, polling mode usually works better. Applications with irregular data and/or communication patterns or dynamic load balancing might work better when interrupt mode is used. Although both MPI and LAPI allow users to select a mode of communication processing, LAPI provides more flexibility in this area. In MPI, the user-selected mode (through the MP_INTERRUPT environment variable) applies to the entire program execution. The default mode in LAPI is interrupt processing. However, using the LAPI_Setenv() call applications can change this mode into polling for entire or only certain phases of the program on some or all tasks.

3.5.1.5 Completion of operations and LAPI counters

Put, Get, and AM are unilateral communication operations that are initiated by one task (the origin), but an indication of the completion of a communication operation is provided at both ends. The definition of when a put or get

operation is complete needs some discussion. Intuitively, the origin may consider a put as complete when the data has been moved from the origin to the target, (for example, the data is available at the target, and the origin data may be changed). However, another equally valid interpretation is one where the origin task considers the operation to be complete when the data has been copied out of its buffer and either the data is safely stored away or is on its way to the target. The target task would consider the put complete when the data has arrived into the target buffer. Similarly, for a get, the target task may consider the operation to be complete when the data has been copied out of the target buffer but has not yet been sent to the origin task.

In order to provide the ability to exploit these different intuitive notions, LAPI has a completion notification mechanism via the use of counters. The user is responsible for associating counters with events related to message progress. However, as the counter structure is an opaque object internally defined by the LAPI counter, it can be accessed using only the appropriate interfaces provided in LAPI. The same counter can be used across multiple messages. This allows you to group different communication calls with the same counter and check their completion as a group. LAPI updates the counters when a particular event (or one of the events) with which the counter was associated has occurred. The application can either periodically check the counter value (using the non-blocking polling LAPI function *Getcntr*) or can wait until the counter reaches a specified value (using the blocking LAPI *Waitcntr* function). Upon its return from the *Waitcntr* call, the counter value is automatically decremented by the value specified in the *Waitcntr* call.

There are three types of counters in LAPI operations: *Origin*, *target*, and *completion* counters. The origin counter is incremented when the buffer with data sent by the origin process can be reused by the application. The target counter is incremented at the target process when the data is copied into the destination location at target. Finally, the completion counter at origin is incremented when the entire data transfer operation completes. Applications can ignore the capability provided by any or all counters by specifying NULL pointer as the counter argument address.

Table 18. Properties of LAPI counters

Counter	Location	When incremented
origin	sender (origin)	data can be modified
target	receiver (target)	data reached destination
completion	sender (origin)	operation completed

3.5.1.6 Message ordering

Two LAPI operations that have the same origin task are considered to be ordered with respect to the origin if one of the operations starts after the other and has completed at the origin task. Similarly, two LAPI operations that have the same target task are considered to be ordered with respect to that target if one of the operations starts after the other and has completed at the target task. If two operations are not ordered, they are *concurrent*.

LAPI provides no guarantees of ordering for concurrent communication operations. For example, consider the case where a node issues two non-blocking puts to the same target node where the target buffers overlap. These two operations may complete in any order including the possibility of the first put partially overlapping the second, in time. Therefore, the contents of the overlapping region will be undefined, even after both *puts* complete. Waiting for the first to complete (for instance, using the completion counter) before starting the second will ensure that the overlapping region contains the result of the second after both *puts* have completed. Alternatively, a fence call (LAPI_Fence) can be used to enforce order.

3.5.1.7 Synchronization

In the message-passing (MPI) model, a message transfers data as well as synchronizes the sender with receiver. Even the nonblocking (asynchronous) message-passing communication that diffuses the synchronization of processes does not eliminate it completely since a coordination between sender and receiver is still necessary. On the other hand, truly one-sided communication systems, such as LAPI, decouple synchronization from data movement and make the bilateral coordination of data transfers between the origin and target unnecessary.

For maximum performance, concurrent operations may complete out of order. As a result, data dependencies between the source and the destination must be enforced using explicit synchronization as is the case in the shared memory programming style. However, in many cases, the program structure makes it unnecessary to synchronize on each data transfer. LAPI provides target counters and atomic operations to be used for synchronization if necessary.

3.5.2 Using LAPI

The set of LAPI functions is shown in Table 19 and discussed briefly in the following section.

Table 19. LAPI functionality

Operations	Functions
Setup	LAPI_Init, LAPI_Term
Active Message	LAPI_Amsend
Data Transfer	LAPI_Put, LAPI_Get
Mutual Exclusion	LAPI_Rmw
Signaling Communication Progress	LAPI_Setcntr,LAPI_Waitcntr, LAPI_Getcntr
Ordering	LAPI_Fence, LAPI_Gfence
Address Exchange	LAPI_Address_init
Environment Query/Setup	LAPI_Qenv, LAPI_Senv

3.5.2.1 Operations on LAPI counters

The LAPI remote memory copy and active message operations allow an application to control and monitor the progress of communication processing by offering counter variables. These opaque data objects store integer values. By using a set of LAPI operations, counters can be initialized, tested, or waited on until they reach a certain value. A counter is incremented when a corresponding phase in a communication process completes. Since LAPI counters are opaque objects, LAPI provides three operations to access and modify their values:

- LAPI_Setcntr - This initializes a specified counter to a desired value.
- LAPI_Getcntr - This reads the value of a counter.
- LAPI_Waitcntr - This waits until a counter reaches a specified value.

3.5.2.2 Remote memory copy operations

LAPI_Get and LAPI_Put are two versions of the remote memory copy operation in LAPI. LAPI_Get transfers *len* bytes of data from the memory of the emote process *target* starting at address *tgt_addr* to the memory of the calling process starting at *org_addr*. LAPI_Put is similar except the direction of data transfer is reversed: Data is moved from the memory of the calling process to the memory of the remote (target) process. A variable *handle* in LAPI operations identifies a communication context in which the operation is

used. Both operations are nonblocking, that is, they initiate the data transfer and return to the calling process as soon as possible. The progress of the data transfer can be monitored using LAPI counters:

```
LAPI_Get(handle, target, len, tgt_addr, org_addr, tgt_cntr, org_cntr)
LAPI_Put(handle, target, len, tgt_addr, org_addr, tgt_cntr, org_cntr, cmpl_cntr)
```

For example, Figure 22 illustrates communication phases in the LAPI_Put operation. The phases are numbered to identify a sequence of events associated with the data transfer. LAPI_Put initiates data transfer and returns control to the application as soon as possible. The buffer containing data at the origin process can be modified by the program when the *org_cntr* origin counter has been incremented. LAPI can increment this counter only when it is certain that the user data cannot be lost in a transmission process over the unreliable network.

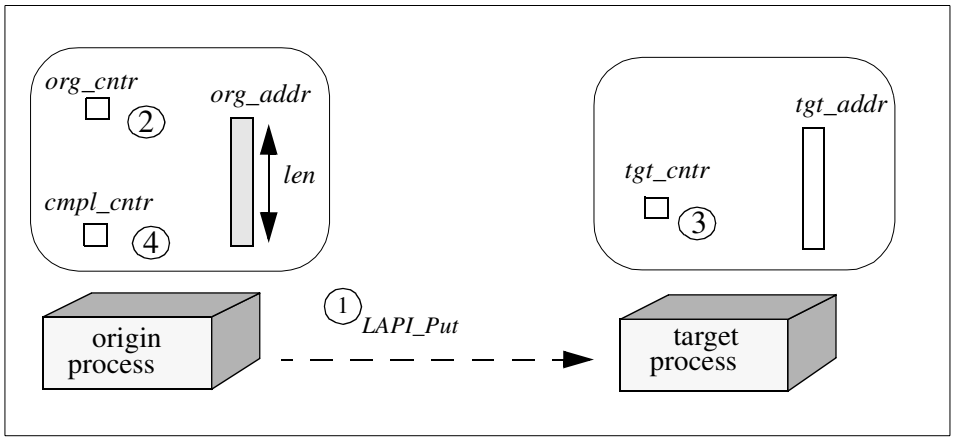


Figure 22. Remote memory copy interfaces and progress of communication

For small requests, it is not necessary to wait for data to be delivered to the target process; LAPI can copy user data to one of its internal (retransmission) buffers and then release the user buffer by incrementing the origin counter. This copy of the user data would be needed to repeat the transfer in the rare case that the data does not reach the destination the first time it is sent. For larger requests, LAPI does not copy the data to its retransmission buffers; therefore, the counter cannot be incremented before the data safely reaches the target process.

On the target side, LAPI, after copying data to the destination location, increments the target counter *tgt_cntr.* For example, this counter can be used to notify the target process that data has been put into its address space. In the next step, LAPI sends a control packet to the origin to indicate that the

data reached the destination safely. When that packet is received, the origin process is notified by incrementing the *cmpl_cntr* completion counter that the LAPI_Put operation completed at the origin and target sides. The counters provide a convenient and efficient mechanism to monitor the progress of communication events. The application can use any, all, or none of these counters depending on its needs. In the case of the LAPI_Put operation, all three counters can be ignored by simply specifying their addresses as NULL.

Example 1: Nonblocking put:

```
double a[100], *rem_b
lapi_cntr_t counter;
LAPI_Setcntr(&counter,0);
rc = LAPI_Put(handle, 0, 100*sizeof(double), rem_b, a, NULL, NULL,
&counter);
/* do some work overlapping comunication with computations*/
rc = LAPI_Waitcntr(handle, &counter, 0, NULL);
/* now we can be sure that data reached process 0 */
```

The interface to LAPI_Get is similar. However, since the origin counter is incremented when data is fully copied from target to origin, a completion counter is redundant and, therefore, not available. The incremented origin counter indicates that the operation completed at the origin as well as the target.

Example 2: Nonblocking get

```
double a[100], *rem_b
lapi_cntr_t counter;
LAPI_Setcntr(&counter,0);
rc = LAPI_Get(handle, 0, 100*sizeof(double), rem_b, a, NULL, &counter);
/* do some work overlapping comunication with computations*/
rc = LAPI_Waitcntr(handle, &counter, 1, NULL);
/* now we can use data in array a */
```

3.5.2.3 Active messages

The LAPI_Amsend operation is used to send an active message to a remote (target) process. Upon arrival, LAPI invokes a handler function passed as an argument to LAPI_Amsend. LAPI active message handlers are divided into a header handler and a completion handler. The handler address specified by the sender refers to header handler. A completion handler function is specified by the header handler executing at target. In addition to invoking a handler function, the LAPI_Amsend operation can transfer data to the target process, similar to LAPI_Put. The data can be moved in two portions:

- User header data - This portion of data is available when the header handler is executed. The amount of data that can be sent in this way is fairly limited. Essentially, after LAPI fills its control information into a single network packet, it makes the remaining space in the packet available to the user. In the current SP network, the maximum size of user header data is approximately over 900 bytes. The exact value of the limit (MAX_UHDR_SZ) can be obtained by calling LAPI_Qenv.
- User data - This portion of data can be of arbitrary size. If it is larger than zero, the header handler must return an address of a memory location to which the data will be copied. LAPI copies the data before the completion handler is called.

Because of the way LAPI dispatches are implemented, no LAPI operations are allowed from within the header handler. Furthermore, if any blocking system calls are made within the header handler, it is possible to deadlock because no communication progress can be made until the header handler returns. If an active message is short and carries no user data from the originating process but requires communication or uses blocking calls in the handler, it is still required to write the handler in two parts because of these restrictions.

LAPI calls can be made from within the completion handler. However, caution is required when writing completion handlers that do long computations and then issue LAPI calls. While long computations take place, the arriving new messages may cause the internal LAPI completion handler queues to fill up if many active messages are received by the same process before the completion handler completes. In such cases, forking a separate thread from within the completion handler and having the forked thread make LAPI calls eliminates the possibility of a deadlock occurring.

Figure 23 on page 95 illustrates the flow of data and control in an LAPI active message. A process on the origin makes the LAPI_Amsend call. The call initiates a transfer of the header *uhdr* and data *udata* at the origin process to the target process specified in the LAPI active message call. As soon as the application is allowed to reuse *uhdr* and *udata*, an indication is provided via *org_cntr* at the origin process. At some point (Step 1), the header and data arrive at the target. Upon arrival at the target, an interrupt that results in the invocation of the LAPI dispatcher is generated. The LAPI dispatcher identifies the incoming message as a new message and calls the *hdr_hndlr* specified by the user (Step 2) in the LAPI active message call. The handler returns a buffer pointer where the incoming data is to be copied (Step 3). The header handler also provides LAPI with an indication of the completion handler that must be executed when the entire message is copied into the target buffer

specified by the header handler. The LAPI library moves the data, which may be transferred as multiple network packets, into the specified buffer. Upon completion of the data transfer, the user-specified completion routine is invoked (Step 4). After the completion routine finishes execution, the *tgt_cntr* at the target process and *cmpl_cntr* at the origin process are updated indicating that the LAPI active message call is now complete:

```
LAPI_Amsend(handle, target, hdr_hdl, uhdr, uhdr_len, udata, udata_len, tgt_cntr,org_cntr,
cmpl_cntr)
```

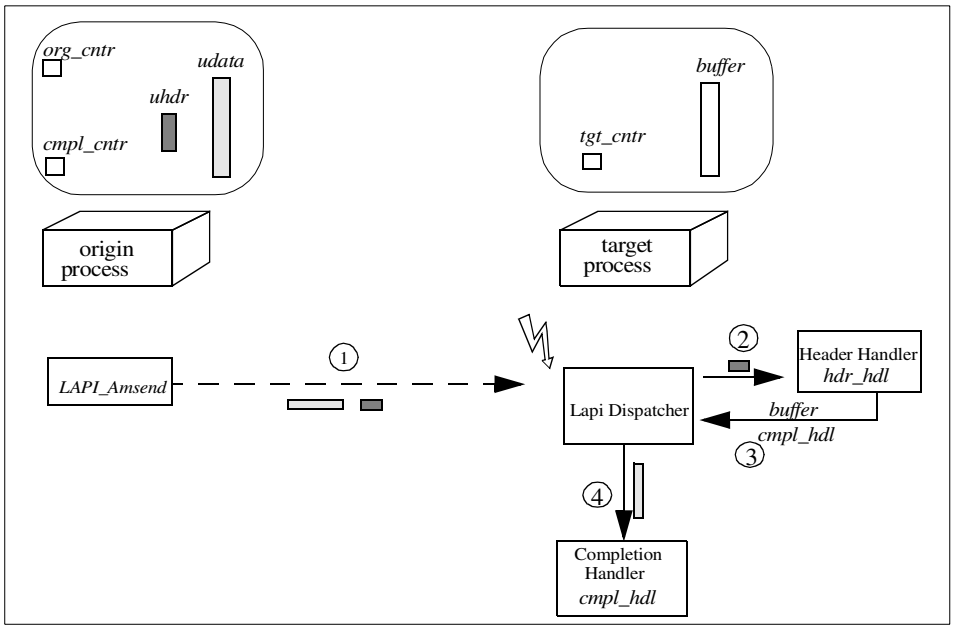


Figure 23. Active message communication in LAPI

When transferring large amounts of data with LAPI_Amsend, it is important to consider availability of memory for *udata* at the target, especially if multiple messages from different sources could be targeting the same task. Since the application must provide buffer space for *udata* in every active message arriving from the network, some form of flow control might be needed at the application level to avoid depleting local memory in case of contention.

3.5.2.4 Thread scheduling

To provide high performance to the applications, the LAPI implementation relies on the default thread scheduling policy in AIX. The application should not change the scheduling policy; otherwise, the LAPI performance can deteriorate.

3.5.2.5 LAPI environment variables and settings

LAPI provides two functions for querying its execution environment parameters and changing some of its parameters.

Some of the parameters that can be retrieved with LAPI_Qenv are:

- TASK_ID - This is the current task/process ID.
- NUM_TASKS - This is the number of tasks.
- INTERRUPT_SET - This is the mode of operation polling (0) or interrupt (1).
- ERROR_CHK - This indicates whether more extensive error checking mode in LAPI is enabled (1) or not (0). The less extensive error-checking mode offers higher performance.

The last two parameters can be modified with LAPI_Senv to change from interrupt to polling mode and vice versa and set the desired error-checking mode.

3.5.2.6 Address exchange

LAPI provides a collective operation, LAPI_Address_init, to exchange addresses of memory areas that are to be used in the context of remote memory copy operations. On IBM SP, even for statically-allocated data structures on other tasks, the addresses might differ.

3.5.2.7 Interoperability of LAPI and MPI

LAPI is interoperable with MPI. This feature allows you to take advantage of the LAPI functionality in the existing MPI applications, or develop new applications that use hybrid MPI and LAPI programming models. The MPI library on the IBM SP is available in four versions depending on the implementation approach and communication protocols used. Table 20 shows compatibility data for the MPI and LAPI libraries.

Table 20. Compatibility of LAPI with MPI library versions

Library version	User space protocol	IP protocol
Thread safe version	+	+
Signal version	-	-

Unlike MPI, there is no implementation of LAPI based on the IP protocol.

3.5.3 Programming examples

The following section offers programming examples.

3.5.3.1 Accumulate operation

Accumulate operation combines vector (S) on the calling process with another vector (D) on the target node and puts the results in the vector at the target:

```
D[0..N-1] = D[0..N-1] + S[0..N-1]
```

where

S[N] is a vector of length N in the address space of the origin process

D[N] is a vector of length N in the address space of the target process

We can implement the accumulate operation using LAPI active messages. Before making the active message call, we obtained the address of the target counter (*targetcntraddr*) and the address of the header handler to be executed on the target process *accumulateaddr* (possibly using the LAPI_Address_init function). The user header udhr is initialized based on the header expected by accumulate. For our example, the structure of uhdr is as follows:

```
typedef struct {
    void *target_addr;
    uint length;
} put_add_hdr_t;
put_add_hdr_t uhdr;
uhdr.target_addr = D;
uhdr.length = N;
```

In this case, the specific call to LAPI_Amsend is:

```
LAPI_Amsend(handle,target_process,accumulate_addr,&uhdr,sizeof(put_add_hdr
_t),&S[0], N*sizeof(S[0]), target_cntr_addr, &origin_cntr, &compl_cntr)
```

Upon receipt of this message at the target (assuming that all data is contained within a packet without a loss of generality), the handler is invoked by the LAPI dispatcher. The structure of the header handler is as follows:

```
void *header_handler (void *uhdr, uint uhdr_len, completion_handler_t
*completion_handler, void *saved_info)
void completion_handler(void *saved_info)
```

If any state information about the message is required by the completion handler, this information should be saved by the header handler in a user buffer. The header handler passes the address of this buffer to the dispatcher through the parameter *savedinfo*. The dispatcher uses this pointer as a parameter for the completion handler.

The specific calls for our example are as follows:

Within the dispatcher:

```
     read lapi header
     extract uhdr and uhdr_len from the lapi header
     buf =(*accumulate_addr)(uhdr, uhdr_len, &completion_handler,
&saved_info);
     copy udata into buf;
     (*completion_handler)(saved_info);
    /* Note that if the message was not contained within a packet, the LAPI
     ** layer will save the necessary information and will invoke the
     ** completion handler after all the udata has arrived and copied into
     ** buf
     */
```

User defined functions:

```
accumulate(uhdr, uhdr_len, completion_handler, saved_info)
{
        buf = addr where incoming data should be buffered
        saved_info = address where parameters to completion handler are
saved
        save (target_addr=D, length=N, buf) in saved_info
        completion_handler = complete_accumulate
        return buf
}
complete_accumulate (saved_info)
{
        retrieve required data (namely D, N and buf) from saved_info;
        for (i=0; i<N; i++) D[i] = D[i] + buf[i];
        return
}
```

The accumulate handler is the header handler and is called by the LAPI layer when the message first arrives at the target process. The header handler saves the information required by *complete_accumulate* (namely, target_addr, length, and buf) in saved_info and passes back a pointer to the *complete_accumulate* handler. Additionally, it returns the address of a buffer buf.

The LAPI layer stores the incoming data as it arrives in buf. When all the data has been received, it calls the *complete_accumulate* function, which uses *saved_info* to access the two vectors, adds them, and stores them at the desired location. After the return from the *complete_accumulate* routine, the lapi layer bumps up *target_cntr*. The origin_cntr is incremented as soon as the data has been copied out of the origin buffer and the origin buffer is available for reuse (there are no guarantees about the data being available at

the target when *origin_cntr* is updated). *compl_cntr* is updated after the return from the *complete_accumulate* routine.

3.5.3.2 Gather operation

Here, we implement a non-collective gather operation (unrelated to MPI_Gather) that can transfer an arbitrary number of double precision elements from a remote (target) process memory (see Figure 24). Unlike the LAPI_Get operation, gather fetches elements that are stored in non-adjacent locations in memory. We make the following assumptions:

- N is the number of elements.
- A[0..N-1] is the local array to which the data should be copied.
- ptrB[N..1] is the array of pointers to doubles (ptrB[i] stores an address of the i-th element in memory of the target process).
- P is the target process.

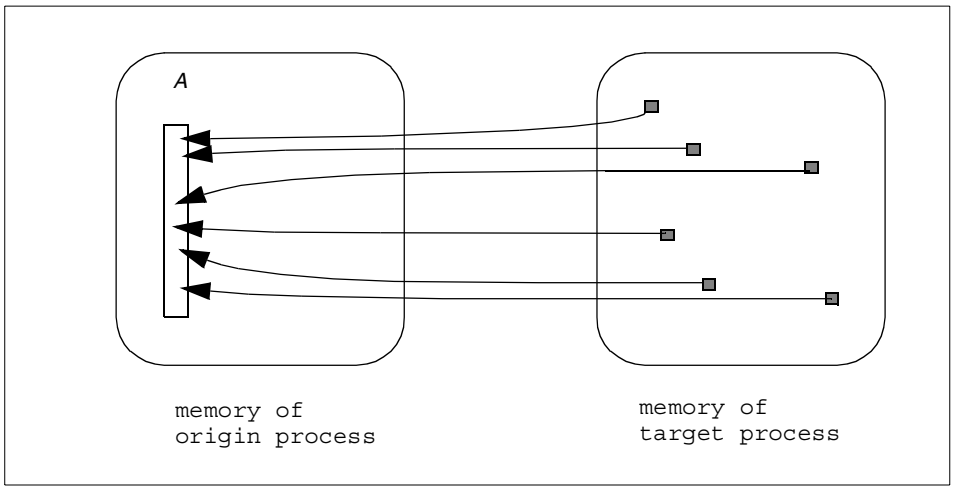


Figure 24. Gather operation

The most straightforward implementation of gather uses a series of N nonblocking LAPI_Get operations:

```
lapi_cntr_t counter;
LAPI_Setcntr(&counter,0);
for(i=0;i<N;i++)
     rc = LAPI_Get(handle, P, sizeof(double), ptrB+i, a+i, NULL, &counter);
rc = LAPI_Waitcntr(handle, &counter, N, NULL);
```

With this approach, since each LAPI_Get operation is used to transfer only one element at a time, the network utilization and expected performance cannot be very good. A better solution is based on active messages.

We send an active message containing pointers to the elements we want to gather to target process P. In addition, we include an address of a counter at the origin that will be incremented when the response arrives.

```
API_Setcntr(&counter,0);
LAPI_Amsend(handle,P,header_handler,&uhdr,sizeof(uhdr),&ptrB,
N*sizeof(double*), NULL, NULL, NULL)
rc = LAPI_Waitcntr(handle, &counter, 1, NULL);
```

where

```
struct gather_request{
   uint from; /* origin process */
   double *A; /* array at origin where elements should be copied to */
   int N;     /* number of elements */
   lapi_cntr_t *pcounter; /* =&counter address of Lapi counter at origin */
} uhdr;
```

The header handler is used to allocate memory for the addresses arriving as the user data in the active messages. We cannot make any LAPI calls here.

```
void *header_handler(uhdr, uhdr_len, completion_handler, saved_info)
{
   ptr_arr = malloc(((struct gather_request)uhdr).N * sizeof(double*));
   memcpy(saved_info, uhdr, uhdr_len)
   completion_handler = gather_handler;
   return ptr_arr;
}
```

In the completion counter, we copy the requested data into a temporary array, and then call LAPI_Put to transfer it back to the origin process. When completed, this operation will increment two LAPI_Put counters:

- One in the memory of a process that issued LAPI_Amsend call (*target counter*)
- One allocated in the completion handler counter (*origin counter*)

When incremented, *Target counter* will signal to the process expecting data arrival that the operation has completed. When incremented, the *origin counter* will signal that the data has been moved out and, therefore, the temporary array can be freed.

```
gather_handler(struct gather_request *uhdr)
```

```
{
lapi_cntr_t cntr;
    LAPI_Setcntr(&cntr,0);
    double *tmp = malloc(uhdr->N * sizeof(double));
    for(i=0; i< uhdr->N; i++) temp[i]=*ptr_arr[i];
    LAPI_Put( handle, uhdr->from, tmp, uhdr->N, uhdr->A, uhdr->pcounter,
NULL, &cntr);
    LAPI_Waitcntr(handle, &cntr, 1, NULL);
    free(tmp);
    free(ptr_arr);
}
```

On the IBM SP with P2SC processors, this implementation of gather achieves 17 times greater asymptotic bandwidth than our first (LAPI_Get based) implementation.

3.5.3.3 Distant disk I/O operation

By combining I/O operations with LAPI active messages, we can perform non-collective read/write operations on disk connected to a different node than the one on which the given task is running. In this way, any task can access data on local disks at any node (local or remote) that the application is using.

For example, a read operation from a file created by a remote task could have the following interface:

```
dio_read(fd, offset, bytes, buffer, task)
```

where *fd* is a file descriptor at a remote task, *buffer* represents the memory area to which the data should be read, and *task* represents the remote task ID.

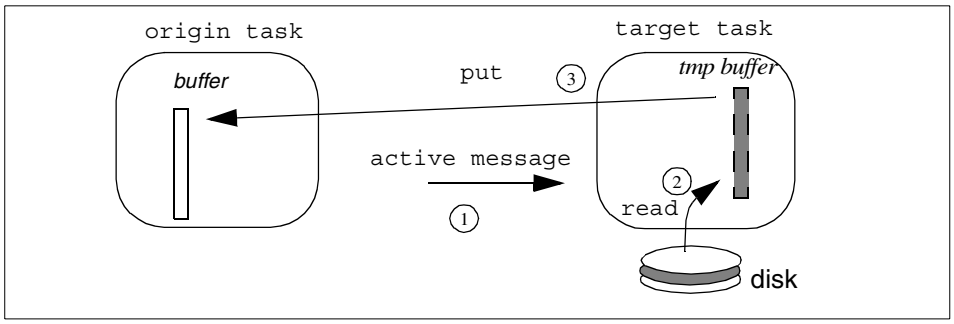


Figure 25. Reading from disk on remote node

We can implement this operation by sending *fd*, *offset*, and *buffer* address arguments in *uhdr* of the LAPI_Amsend operation. In addition, a LAPI counter, *io_cntr*, is used to notify the origin task when the requested data arrives into *buffer*. *dio_read* is nonblocking; so, in order to access data in *buffer*, we need to wait until the *io_cntr* is incremented.

```
struct read_request{
   uint from;             /* origin process */
   int fd;                /* remote file descriptor */
   char *buffer;          /* array at origin where elements should be copied to */
   int bytes;             /* number of bytes to read*/
   off_t offset;          /* offset in file to read from */
   lapi_cntr_t *pcounter; /* address of io_cntr counter at origin */
} uhdr;
```

We assume that the file has already been opened by the target task in a mode allowing writing to that file. The actual I/O operation is performed in a LAPI completion counter at target:

```
read_cmpl_handler(struct read_request *uhdr)
{
lapi_cntr_t cntr;
char *tmp = malloc(uhdr->bytes); /* temporary buffer for reading */
   LAPI_Setcntr(&cntr,0);
   lseek(fd,uhdr->offset,SEEK_SET);
   read(uhdr->fd, tmp, uhdr->bytes);
   LAPI_Put(handle, uhdr->from, tmp, uhdr->bytes, uhdr->buffer,
uhdr->pcounter, NULL, &cntr);
   LAPI_Waitcntr(handle, &cntr, 1, NULL); /*wait until all data is sent */
   free(tmp);
}
```

Since the bandwidth of LAPI_Put operation on the IBM SP is much higher than disk I/O bandwidth to the local SCSI disk at a node, the performance of distant read operation can be very close to that of local I/O for all but very small (<10KB) requests.

3.5.3.4 Ordering of remote memory copy operations

Certain applications require ordering of remote memory copy operations, for example, to maintain a consistent view of shared data structures in the presence of put operations targeting overlapping memory areas. If overlapping put operations complete out-of-order, the result of the data transfer is unpredictable. Ordering of operations is the simplest technique for preserving consistency of the data structures.

Since LAPI messages arrive out-of-order, ordering should be addressed at the application level. The LAPI fence and counter operations allow you to order operations by enforcing completion of the outstanding operations:

```
LAPI_Put(handle, proc, len, rem_addr1, loc_addr1, NULL, NULL, NULL);
LAPI_Fence(proc)
LAPI_Put(handle, proc, len, rem_addr2, loc_addr2, NULL, NULL, NULL);
```

or using a completion counter

```
LAPI_Setcntr(&counter,0);
LAPI_Put(handle, proc, len, rem_addr1, loc_addr1, NULL, NULL, &counter);
LAPI_Waitcntr(&counter, 1, NULL)
LAPI_Put(handle, proc, len, rem_addr2, loc_addr2, NULL, NULL, &counter);
```

The ordering of operations by enforcing completion works, but it is not the most efficient technique for maintaining consistency of remote data structures. It reduces the potential concurrency in communication over the network since it allows only one outstanding put operation to a given target process. This constraint might not be necessary if all we need is to assure that the overlapping puts complete in order. For example, we can use the sliding window technique to monitor memory areas addressed by a certain number of outstanding put operations and selectively block only the operations that target overlapping memory areas.

Chapter 4. Shared memory

This chapter covers two aspects of shared-memory parallelization. The first section discusses OpenMP shared memory directives on POWER3 SMP nodes, and the second section discusses thread programming.

4.1 Shared memory parallelization with OpenMP

This section discusses OpenMP Shared Memory directives and how they can be used to effectively parallelize a Fortran or C program on the POWER3 SMP nodes. These directives are designed to allow the user to specify variable scoping and work sharing necessary for the compiler to generate a threaded parallel application. With the release of Version 7.1 of XL Fortran, the xlf compiler will be fully OpenMP-compliant.

This section does not discuss all of OpenMP - only those OpenMP features pertaining to DO loop parallelism. A full discussion of the language can be found at the following URL:

`http://www.openmp.org`

4.1.1 Introduction to shared memory parallelization

A majority of the High Performance Computing architectures today are clusters of nodes in which each node is a high-performance SMP machine. This section discusses the optimization of applications for Shared Memory. This optimization comes from parallelizing the work across the processors on the node. The two principal ways of accomplishing the parallelization is by using hand-coded pthreads (see Section 4.2, “Programming with threads” on page 130) and OpenMP directives. Either way, the user must be concerned with issues that can degrade the performance.

First, we should be perfectly clear that parallelization does not reduce CPU time; if anything, it will increase CPU time. Parallelization will, however, decrease wallclock time. When you are timing your parallel code, be sure that you use a library call that returns wallclock time and not CPU time. The rtc() function measures wallclock and is recommended for measuring parallel code. The major issues that the user should watch are:

- The overhead of parallelizing a piece of code including thread generation and synchronization
- The bandwidth available on the SMP node and the amount of bandwidth required by the application being parallelized

- Load balancing when parallelizing iterations of a loop with differing compute times

The overhead for parallelizing on any RISC-based SMP is significant and should not be ignored. Figure 26 shows the crossover point for the Stream benchmark. Notice that one must have a significant loop iteration count before a performance gain is realized by parallelization. This crossover point will be dependent on the DO loop. It depends on the granularity of the computation.

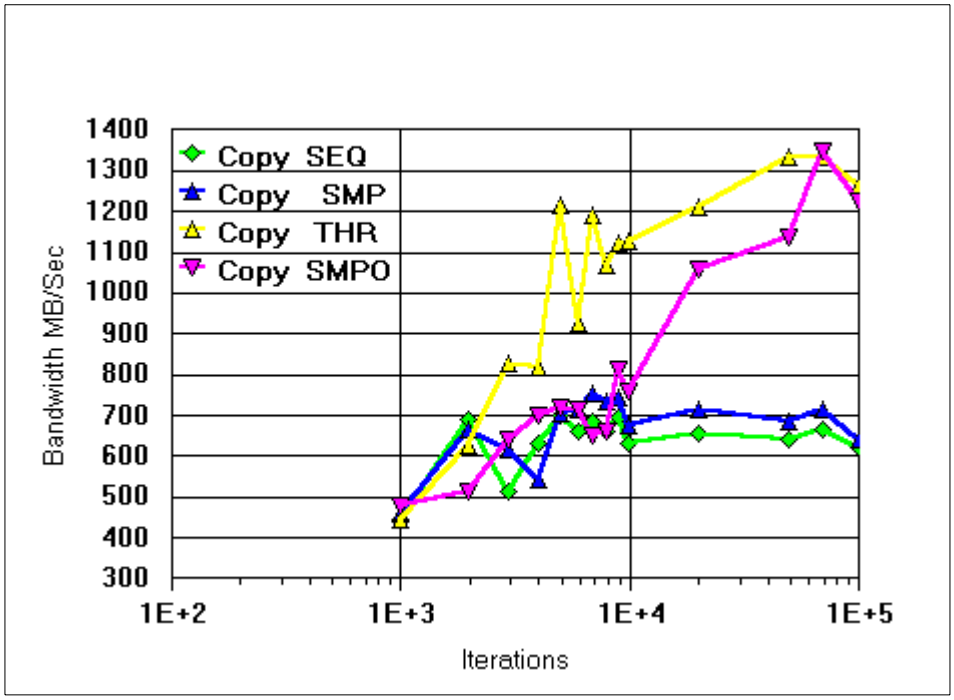


Figure 26. Stream benchmark on POWER3 SMP Thin/Wide and High nodes

Figure 26 shows Memory Bandwidth for the COPY kernel for the Stream Benchmark. This simple benchmark measures the memory bandwidth achieved for copying one array into another. There are four versions of the kernel: The first is the sequential version measured for the generic Stream Benchmark; the second is for the default automatic parallelization of the compiler; the third is for an OpenMP version (in Section 4.1.9, “Automatic parallelization” on page 123, we discuss the runtime parallelization analysis that is used and how it affects performance), and, the fourth is a manually-coded Pthread example. This timing was obtained on a two-processor, SMP POWER3, Thin/Wide (200 MHZ) system.

Notice that, for iteration counts less than 1000000, the OpenMP example and the Pthread implementation for this iteration count perform about 1.8 times better on two processors. For iteration counts around 100000, the forced automatic parallelization example and the Pthread example are twice as fast as the sequential. The default, which uses a runtime decision that checks the granularity of the loop, never uses the parallel version at these iteration counts. Also, notice that, for iteration counts below 10000, the parallel version runs slower than the sequential version. Remember that this loop is only doing a copy, not a computation. If a more significant loop is used, say, the vector = vector + scalar * vector, the granularity of the DO loop will be larger, and the benefit from parallelization is better.

Figure 27 displays the stream rates for Triad.

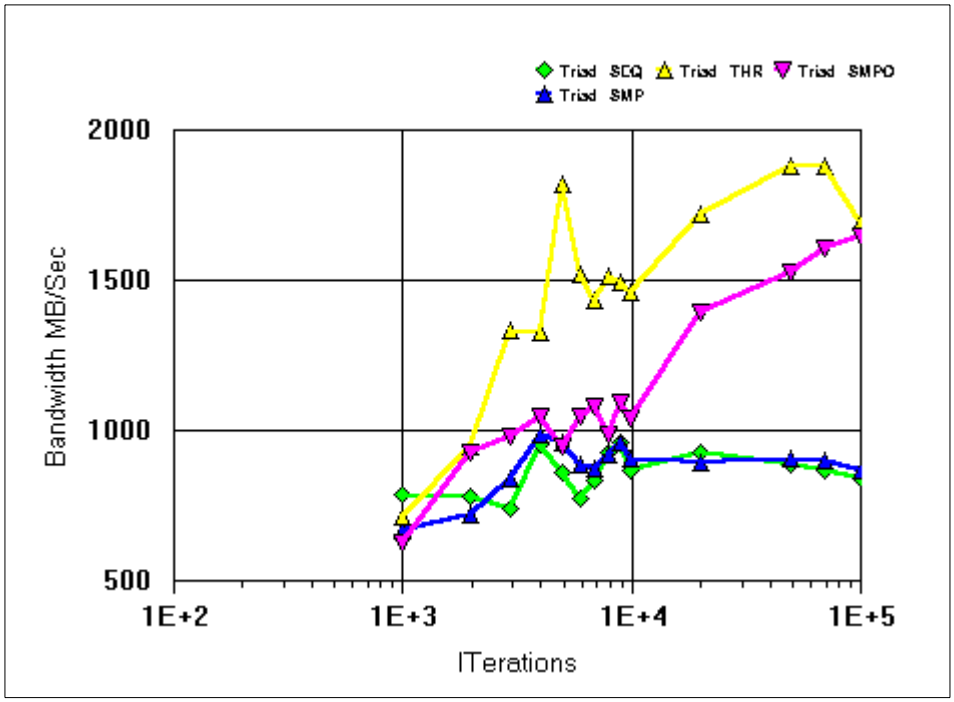


Figure 27. Stream rates for Triad

The OpenMP example always runs as well as the sequential example; however, we do not really get a good performance gain until we approach 100000 iterations. The cause of the poor scaling is simply a matter of the overhead with managing the threads; the loop is entirely load balanced. As the amount of computation increases in the DO loop or the iteration count

increases, the crossover point, where the parallel version of the DO loop runs significantly faster than the sequential, decreases.

Users are encouraged to test their major computational kernels with different lengths to determine individual crossover points.

IBM hardware has a particularly high memory bandwidth, and the node architecture is designed to supply enough memory bandwidth for the number of processors supplied on the node. The POWER3 SMP Thin/Wide node has a memory bandwidth of 1.6 Gbytes/sec and a maximum of four processors on the node. The POWER3 SMP High node has a memory bandwidth of 14.2 Gbytes/sec if all memory slots are populated and there is a maximum of eight processors. While the high node has significantly more memory bandwidth than the thin node, it may vary with the amount of memory installed on the high node.

The memory bandwidth on SMP nodes contributes to how well you will scale as you add more and more processors. Once again, the scaling is a function of the application, and the user should understand how much memory bandwidth is needed within their application.

4.1.2 OpenMP - Portable shared memory parallelization

Shared memory parallel systems have been around for 30 years. Each vendor had their own idea of how the user should program for their hardware. Ten years ago, an attempt was made to standardized shared memory parallelization; however, this failed because several vendors felt that they would stay with their implementation. With the recent appearance of SMP across all the RISC based systems, a new proposal has been put forth, which has been accepted by all the major hardware vendors; so, now, users have an approach to build portable shared memory parallel systems. A few of the most widely-used directives follow.

Note

The following directives are implemented as part of the IBM effort to support OpenMP. Some clauses, such as THREADPRIVATE, COPYIN, ORDERED, and others, are expected to be available in future versions of IBM compilers.

!$OMP PARALLEL / !$OMP END PARALLEL - Indicate a parallel region for each thread to execute - must scope all variables within the region.

- DefaultChange Default Scoping

- Private Assure each thread has own copy
- Shared Variable to be addressable by each thread
- First Private Master thread to copy its private variable to all threads
- Reduction Following specified operation to be applied across variable
- If Perform following loop in parallel if test true

!$OMP PARALLEL DO / !$OMP END PARALLEL DO - Indicate a parallel do for all threads to share in the work. This is a special case of PARALLEL where the region and the loop are identical. Parallel DO is simply a shortcut for combining the PARALLEL and the DO into one construct.

- Default - Change Default Scoping
- Private - Assure each thread has own copy
- Shared - Variable to be addressable by each thread
- First Private - Master thread to copy its private variable to all threads
- Last Private - Last value for private variable copied to Master thread
- Reduction - Following specified operation to be applied across variable
- If - Perform following loop in parallel if test true
- SCHEDULE - Specify work sharing option

!$OMP DO / !$OMP END DO - Indicates a parallel do for all threads to share in the work.

- Private - Assures that each thread has its own copy
- First Private - Master thread copies its private variable to all threads
- Last Private - Gets the last value for the private variable copied to the Master thread
- Reduction - Specifies the operation to be applied across variables
- Schedule - Specifies the work sharing option

4.1.3 Rationale for using OpenMP directives

The OpenMP directives address three important issues of parallelizing an application: First, clauses and directives are available for scoping variables. Frequently, variables should not be shared; that is, each processor should have its own copy of the variable. Second, work sharing directives specify how the work contained in a parallel region of code should be distributed across the SMP processors. Finally, there are directives for synchronization between the processors.

OpenMP directives do not parallelize the program; they are a set of commands to the compiler on how a particular DO loop should be parallelized. The existence of the directives in the source removes the need for the compiler to perform any parallel analysis on the parallel code. The use of OpenMP directives dictates that the compiler should parallelize the particular section of code, and wrong answers could result if the directives are used incorrectly.

This variable must be replicated for each processor that executes the DO loop; otherwise, one processor would set i, and then, prior to using the value of i, processor 2 may set i. To solve this problem, we make *n* (the number of processors) copies of *i* using the PRIVATE clause. Since the DO loop index is made private automatically, this is not necessary; however, for completeness, we include it.

Prior to investigating OpenMP directives in detail, a small example that illustrates the use of the directives is given. Consider the following DO loop:

```
dimension a(1000000),b(1000000),c(1000000)
      read *,n
      sum = 0.0
      call random (b)
      call random (c)
!$omp PARALLEL DO
!$omp+PRIVATE (i)
!$omp+SHARED (a,b,n)
!$omp+REDUCTION (+:sum)
      do i=1,n
        a(i) = sqrt(b(i)**2+c(i)**2)
        sum = sum + a(i)
      enddo
      print *,'sum = ',sum
end
```

Variable scoping is the hardest task the user must complete to parallelize a DO loop. The question to be addressed is how a particular variable should be allocated prior to the execution of the parallel DO loop. The first variable encountered in the DO loop is the loop index itself (i).

The other variables in the first sample DO loop are either only read, or they are arrays whose access is dependent upon the loop index. Any variable satisfying either of these characteristics should be shared by all processors.

The variable sum is a special case. It actually introduces a loop-carried dependency and could be an inhibitor to parallelization of the DO loop. Since this type of construct occurs often, a special name is given, and a special

directive is supplied. sum is a reduction variable. The way to parallelize this construct would be for each processor to calculate its own local sum and then outside the loop to have each processor add its value into a global shared value. Consider the following example:

```
      sum = 0.0
!$omp PARALLEL
!$omp+PRIVATE (i,sumx)
!$omp+SHARED (a,b,n,sum)
      sumx = 0.0
!$omp DO
      do i=1,n
        a(i) = sqrt(b(i)**2+c(i)**2)
        sumx = sumx + a(i)
      enddo
!$omp END DO
!$omp CRITICAL
      sum = sum + sumx
!$omp END CRITICAL
!$omp END PARALLEL
      print *,'sum = ',sum
      end
```

Several new directives have been introduced. First, the PARALLEL—END PARALLEL construct allows for scoping to be disjointed from work sharing. This allows the user to have control over some private variables. In this example, the private variable, sumx, is initialized outside the DO loop, updated in a work-sharing directive, DO—END DO, and then summed into the shared variable, sum, under the control of a synchronization directive, CRITICAL—END CRITICAL. The shortcut for forcing the compiler to do this is the use of the REDUCTION directive, which results in exactly this code.

We have one last comment on this example: One may get different answers when parallelizing this example. The parallelization of this example sums the elements of a(i) in a different order than the original sequential DO loop. Such permutation of addition could get different answers due to the accuracy of the computer functional units.

We now look at the variable scoping in more detail.

4.1.4 Variable scoping

Variable scoping tends to be most difficult in cases that benefit the most from parallelization. To obtain good parallelization performance, the outermost DO loops should be parallelized. In many cases, the outermost DO loop may

contain subroutine and function calls. When a DO loop contains subroutine and function calls, variable scoping is a challenge.

In earlier versions of Fortran, the user could be sure that, when a subroutine was called, all local variables would retain their values when leaving the subroutine or function. This implicit saving of the procedure's variables was assured because memory for the subroutine and/or function was "statically" stored in memory that lived throughout the execution of the program. With the advent of parallel systems, particularly shared memory parallel systems, a different way of handling a procedure's variables was formulated to assure that two different processors would use different memory locations when the procedure was called. The new method stored the variables on a "stack", when the procedure was called and released them with the return from the procedure.

Unfortunately, this method of allocating a procedure's local variables does not save the value of the variable from one invocation of the procedure to the next. With the advent of Fortran90, the default for saving a procedure's local variables changed to "stack". If a user wants to retain a value for a variable from one invocation to the next, he or she must use a SAVE statement. Actually, the SAVE statement is like placing the local variable in a COMMON block; it is still local, and it exists from one invocation to the next.

When using shared memory parallelization in a program, you must ensure that all local variables in the routines called in parallel must be allocated on the "stack". When using xlf on an IBM SMP, you must compile the routines with -qnosave. This is not necessary when using xlf90 or xlf95. Care must be taken to ensure that no SAVE statements are contained within the called routine and that no variables have the SAVE attribute, namely, those that are explicitly initialized.

In Fortran, there are two types of variables: GLOBAL variables and LOCAL variables. GLOBAL variables are frequently allocated at compile time and LOCAL variables without the SAVE attribute are always allocated on stack at runtime. In OpenMP, there are two types of variables: SHARED and PRIVATE. Variable scoping is much easier if you try to think of all GLOBAL variables as being SHARED and all LOCAL variables as being PRIVATE.

Perhaps, understanding what a compiler does to generate a parallel version of a DO loop after encountering an OpenMP directive will help. When an OpenMP directive is encountered on a DO loop, the DO loop is encapsulated in a subroutine, and, then, that subroutine is invoked by each active thread. All variables that are scoped PRIVATE on the OpenMP directive are allocated on "stack" when the routine is called, and all variables scoped SHARED are

passed to the subroutine in a COMMON block. In the transformed parallel DO loop, the variable scoping strictly adheres to the Fortran notions of GLOBAL and LOCAL.

PRIVATE variables are those variables that must be disjointed; that is, each processor must have a separate copy during the execution of the parallel code. SHARED variables are those variables that must be shared by each processor, that is, all processors use the same memory location for SHARED variables.

All variables that are scoped PRIVATE on the OpenMP directive are allocated on "stack" when the routine is called, and all variables scoped SHARED are passed to the subroutine as GLOBAL variables. In the transformed parallel DO loop, the variable scoping strictly adheres to the Fortran notion of GLOBAL and LOCAL.

Given this introduction to variable scoping, we now present some rules of thumb to assist in scoping variables in a DO loop.

A PRIVATE variable is a variable that satisfies one of the following rules:

- A scalar variable that is set and then used within the DO is PRIVATE.
- An array whose subscript is constant with respect to the DO and is set and then used within the DO is PRIVATE.

All other variables are SHARED including:

- A scalar or array whose subscript is constant with respect to the PARALLEL DO that is only used.
- A scalar or array whose subscript is constant with respect to the DO that is used and then set in the DO. This variable leads to loop-carried dependencies, and a loop containing this type of variable can only be parallelized when an ORDERED or CRITICAL region or a REDUCTION clause is used. The use of one of these OpenMP constructs will be dependent upon the use of the variable. An example of each will be given later in this section.
- An array whose subscript is dependent upon the DO.

Given this set of rules, we will now talk about exceptions. First, strictly speaking, a Fortran local variable can be an OpenMP SHARED variable. This will occur when the DO is contained within the routine that allocates the local variable. For example:

```
subroutine example1(n,b,c)
        real*8 a(1000000),B(1000000),c(1000000)
```

```
        integer n,i
        real*8 sum
c       read *,n
        sum = 0.0
c       call random_number (B)
c       call random_number (C)
!$omp PARALLEL DO
!$omp+PRIVATE (i)
!$omp+SHARED (a,b,n)
!$omp+REDUCTION (+:sum)
             do i=1,n
              a(i) = sqrt(b(i)**2+c(i)**2)
              sum = sum + a(i)
             enddo
        print *,'sum = ',sum
        end
```

The variable sum is local with respect to example1; however, it is SHARED with respect to OpenMP and is one of those nasty used-before-being-set variables. This occurrence of sum is handled quite nicely with the REDUCTION clause.

Additionally, a global variable may necessarily be scoped as a PRIVATE variable. When this occurs, there is almost always a problem. Consider the following example:

```
subroutine example4(n,m,a,b,c)
        real*8 a(100,100),B(100,100),c(100)
        integer n,i
        real*8 sum
!$omp PARALLEL DO
!$omp+PRIVATE (j,i,c)
!$omp+SHARED (a,b,m,n)
        do j=1,m
           do i=2,n-1
            c(i) = sqrt(1.0+b(i,j)**2)
           enddo
           do i=1,n
            a(i,j) = sqrt(b(i,j)**2+c(i)**2)
           enddo
        enddo
end
```

The variable *c* is passed into the routine; while it may be local within some other routine, it is definitely global with respect to example4. In the PARALLEL DO — do j=1,m , c(i) must be a PRIVATE variable. If this isn't clear, go back to the rules. c(i) is constant with respect to j, and it is set and

then used in the PARALLEL DO. Something is not quite right about this DO loop. Why does the user even have c(i)? Why not substitute sqrt(1.0+b(i,j)**2), and why are all values of *c* not being set? Well, maybe the user does not want all of *c* to be set. Maybe the values c(1) and c(n) are special and are passed into the subroutine. If the user really wants the c(1) and c(n) that are passed into the routine, making it a PRIVATE variable is not giving all of the processors a copy of c(1) and c(n). Basically, we have a problem. Consider the following modification to the OpenMP directives:

```
!$omp PARALLEL DO
!$omp+PRIVATE (j,i)
!$omp+FIRSTPRIVATE (c)
!$omp+SHARED (a,b,m,n)
```

The FIRSTPRIVATE directive copies the values of c owned by the master thread to each of the PRIVATE arrays c. This results in quite a large amount of data motion that is not really required. Alternatively, the user can employ the following code to produce a more efficient version of the parallel DO loop:

```
   subroutine example5(n,m,a,b,c)
           real*8 a(100,100),B(100,100),c(100)
           real*8 cc(100)
           integer m,n,i
           real*8 sum
   !$omp PARALLEL
   !$omp+PRIVATE (j,i,cc)
   !$omp+SHARED (a,b,m,n)
           cc(1) = c(1)
           cc(n) = c(n)
   !$omp DO
           do j=1,m
            do i=2,n-1
              cc(i) = sqrt(1.0+b(i,j)**2)
            enddo
            do i=1,n
              a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
            enddo
           enddo
   !$omp END DO
   !$omp END PARALLEL
   end
```

By using the PARALLEL/END PARALLEL construct, we have reduced the memory copies and achieved the same parallelization.

Whenever a global variable is made private, another problem arises. What if that variable is required when returning to the calling procedure? A global

variable is either passed in as an argument or in a COMMON block. What if it is needed after execution of the DO loop? In the OpenMP documentation, it is not clear if you can have both:

```
!$omp PARALLEL DO
!$omp+PRIVATE (j,i)
!$omp+FIRSTPRIVATE (c)
!$omp+LASTPRIVATE (c)
!$omp+SHARED (a,b,m,n)
```

There certainly could be a case where both were required, or one could use the PARALLEL/END PARALLEL construct:

```
!$omp PARALLEL
!$omp+PRIVATE (j,i,cc)
!$omp+SHARED (a,b,m,n)
        cc(1) = c(1)
        cc(n) = c(n)
!$omp DO
        do j=1,m
          do i=2,n-1
            cc(i) = sqrt(1.0+b(i,j)**2)
          enddo
          do i=1,n
            a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
          enddo
        enddo
!$omp END DO
        do i=1,n
          c(i) = cc(i)
        enddo
!$omp END PARALLEL
```

Is this right? LASTPRIVATE says the processor that has the last iteration of the DO loop must store all the values of c. Since we cannot be sure in which order the iterations are performed, this is incorrect. Perhaps the following is correct:

```
!$omp PARALLEL
      cc(1) = c(1)
      cc(n) = c(n)
!$omp DO
      do j=1,m
       do i=2,n-1
        cc(i) = sqrt(1.0+b(i,j)**2)
       enddo
       do i=1,n
        a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
```

```
       enddo
      enddo
!$omp END DO
      if(j.eq.m+1)then
       do i=1,n
        c(i) = cc(i)
       enddo
      endif
!$omp END PARALLEL
```

Variable scoping becomes more difficult when the parallel DO calls an external routine:

```
subroutine example5(n,m,a,b,c)
         real*8 a(100,100),B(100,100),c(100)
         integer m,n
!$omp PARALLEL DO
!$omp+PRIVATE (j)
!$omp+SHARED (a,b,m,n)
         do j=1,m
           call doit(j,n,a,b)
         enddo
end
CEND  EXAMPLE5
subroutine doit(j,n,a,b)
        real*8 a(100,100),B(100,100)
        COMMON cc(100)
        do i=2,n-1
         IF(a(i,j).gt.SIN(b(i,j)))THEN
            cc(i) = sqrt(1.0+b(i,j)**2)
          ENDIF
        enddo
        do i=1,n
          a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
        enddo
end
```

We have a global variable *cc*, which is contained in a COMMON block; however, each processor needs a different copy in the routine doit(). There are many problems with this one. We are not sure if the original values of *cc* are needed; it looks like they probably are. The real problem is that all processors will be using the same memory for *cc* and overwriting values, which is nothing but trouble. Unfortunately, this type of construct occurs quite often, especially when one parallelizes code at a very high level. OpenMP thought of this and introduced the following directive:

```
!$omp THREADPRIVATE (/BCOM/)
```

This construct specifies that the COMMON block BCOM will be allocated for each participating thread. Unfortunately, blank COMMON cannot be specified on a THREADPRIVATE directive, and we would have to change all occurrences of COMMON *cc* to COMMON/BCOM/ *cc*. How do we deal with getting a copy of *cc* for all the threads? This is handled by the COPYIN clause as follows:

```
!$omp PARALLEL DO COPYIN(/BCOM/)
```

When the THREADPRIVATE directive is used, none of its variables should be used in any PRIVATE or SHARED clause. The COMMON block variables can only be used in the COPYIN clause. The previous DO loop would, therefore, be changed as follows:

```
!$omp PARALLEL DO
!$omp+COPYIN(/BCOM/)
!$omp+PRIVATE (j)
!$omp+SHARED (a,b,m,n)
      do j=1,m
       call doit(j,n,a,b)
      enddo
end
CEND  EXAMPLE5
subroutine doit(j,n,a,b)
      real*8 a(100,100),B(100,100)
      COMMON/BCOM/ cc(100)
!$OMP THREADPRIVATE (/BCOM/)
      do i=2,n-1
       if(a(i,j).gt.SIN(b(i,j)))THEN
        cc(i) = sqrt(1.0+b(i,j)**2)
       endif
      enddo
      do i=1,n
       a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
      enddo
end
```

If one variable in a COMMON block must be made private, all variables must be made private. COPYIN can specify a list of variables that need to obtain a copy of the original value.

There are directives to specify default scoping for the compiler telling the compiler to treat everything not mentioned on a scoping directive as PRIVATE or SHARED depending on the directive (compilers typically have defaults for these directives); however, these directives and the compiler defaults can cause problems. The most important task of parallelizing a DO loop is

scoping all the variables; if this is not done correctly, wrong answers will be obtained. It is important for the user to scope all variables used in a DO loop.

Also, do not assume that the compiler is scoping the variables for you. You may get lucky, but, nine times out of ten, you will get incorrect answers if you just put a !$omp PARALLEL DO on the DO loop.

4.1.5 Work sharing concepts

The most widely-used method for work sharing is chunk distribution. For example, if there are four processors (np=4) being used, a simple work allocation is:

```
!$omp Schedule(Static)
   Processor      Iteration Count
   1               1,n/4
   2               n/4+1, n/2
   3              n/2+1, 3n/4
   4              3n/4+1, n
```

Each processor will get an equivalent chunk of work, and the processor will be given the work with one dispatch. A second option would be:

```
!$omp Schedule(Dynamic,1)
       Processor  Iteration Count
       1          1,7,9,...
       2          2,5,6,...
       3          3,8,10,...
       4          4,11,...
```

Each processor may not get an equivalent amount of work; however, it will receive dispatches of one iteration of work as long as they are available. Giving a thread a chunk of work has associated overhead. Why would this second approach be of any value? The answer has to do with load balancing of the work. If our DO loop had an IF test prior to the calculation dependent upon some loop dependent value, there would be a potential that the calculation would not be executed for a number of loop iterations. The first work-sharing mechanism assumes equal work for each iteration. The second is the best for load balancing if the work is variable. OpenMP has other mechanisms to allow the user to trade-off between minimizing overhead and good load balancing.

Consider the following DO loop:

```
   subroutine example5(n,m,a,b,c)
         real*8 a(100,100),B(100,100),c(100,100)
         integer i,j,m,n
```

```
!$omp PARALLEL DO
!$omp+PRIVATE (i,j)
!$omp+SHARED (a,b,c,n,m)
      do i=1,m
       do j=i+1,n
        a(j,i) = sqrt(b(j,i)**2 + c(j,i)**2)
       enddo
      enddo
end
```

In this example, each subsequent processor has less work than the previous processor. The default method of giving processor one the first m/nprocs iterations, the second processor the next m/nprocs iterations, and so on, will give the last processor significantly less work than the first, and the last processor will finish the DO loop first and wait for the rest. This DO loop is not "Load Balanced". OpenMP has other options to cover this area. The following directive:

```
!$omp SCHEDULE(STATIC,1)
```

would give each processor only one iteration at a time. This would give you good "Load Balancing"; however, the overhead of issuing *m* parallel tasks would be excessive. Consider the following:

```
!$omp SCHEDULE(STATIC,4)
```

This would give larger chunks of work; however, this could be dependent upon the value of *m*. If *m* were a zillion, we should set the chunk size pretty large:

```
!$omp SCHEDULE(STATIC,1000000)
```

There is a way to do this correctly. First, we figure out the size of the chunk size that achieves a good parallel speedup. Let us say we determine that the DO loop gets a factor of 3.7 on four processors for a value of around 250 if *m* is typically 59000. A very good work sharing directive would be:

```
!$omp SCHEDULE(GUIDED,500)
```

This directive starts by handing the iterations out in chunks equal to n/nprocs and then exponentially decreasing the chunk size to an iteration count of 50. This will effectively hand out a balanced number of chunks.

Table 21 shows the difference in iteration distribution for several different options for a DO loop that is not shown:

Table 21. Iteration distribution

Iterations	1,000
static,10	1-10, 41-50, 81-90 ... 11-20, 51-60, 91-100 ... 21-30, 61-70, 101-110 ... 31-40, 71-80, 111-120 ...
Dynamic,10	1-10, 41-50,71-80, 81-90 ... 11-20,91-100 ... 21-30,51-60, 61-70, 101-110 ... 31-40, 71-80, 111-120 ...
Guided	1-250, 686-764,927-945,971-978, ... 251-438,824-868,902-926 439-579,765-823,960-970 580-685,869-901,946-959

4.1.6 Other directives

The next set of directives deals with synchronizing the processors. Several of these directives have already been discussed. The major synchronization directives are:

```
!$OMP CRITICAL
!$OMP END CRITICAL
```

The `CRITICAL` directive allows only one processor in the region encompassed by the directives. Any order is allowed.

```
!$OMP ORDERED
!$OMP END ORDERED
```

The `ORDERED` directive allows only one processor in the region encompassed by the directives in the order specified by the DO loop. The thread executing the first iteration is followed by the thread executing the second iteration, which is followed by the thread executing the third iteration, and so on. Additionally, the PARALLEL DO must have an ORDERED clause.

```
!$OMP ATOMIC
```

Consider the following DO loop:

```
subroutine example8(n,m,a,b,c,ij)
      real*8 a(1000000),b(1000000),c(1000000),ij(1000)
      integer i,j,m,n
!$omp PARALLEL DO
!$omp+PRIVATE (i)
```

```
!$omp+SHARED (a,b,c,ij,m)
      do i=1,m
!$omp ATOMIC
       a(ij(i)) = a(ij(i)) + sqrt(b(i)**2 + c(i)**2)
      enddo
end
```

In the PARALLEL DO, the update of a(ij(i)) is performed atomically. That means that the load, update, and store of a(ij(i)) are performed under a critical region; however, the expression sqrt(b(i)**2 + c(i)**2) can be performed in parallel.

```
!$omp BARRIER
```

This directive specifies that all processors will wait at this point until all processors have arrived at this point.

4.1.7 Function calls

The following calls are self-explanatory and are useful when performing parallelism with OpenMP directives:

```
call OMP_SET_NUM_THREADS(integer expression)
nthrds = OMP_GET_NUM_THREADS()
nprocs = OMP_GET_NUM_PROCS()
```

The number of threads can be greater than the number of processors

```
myid = OMP_GET_THREAD_NUM()
```

4.1.8 Compiler options

When using shared memory parallelization, the _r version of the compiler should be used. The three most popular versions are xlf_r, xlf90_r, or mpxlf_r. These are the Fortran 77, Fortran 90, and distributed memory parallel version. When these commands are used, the -qsmp option invokes the parallelizer. Some sample uses follow:

```
xlf_r -qnosave -qsmp=omp
```

This invokes the Fortran compiler using default option settings that are consistent with typical FORTRAN 77 compiler behavior, requesting that all locals be stored on stack and asking for the recognition of OpenMP directives.

```
xlf90_r -qsmp
```

This invokes the Fortran compiler using default option settings that are consistent with typical FORTRAN 90 compiler behavior and requests automatic parallelization.

```
mpxlf_r -qnosave -qsmp=noauto
```

Invokes the compiler with MPI libraries, requests no automatic parallelization, and requests the interpretation of OpenMP and IBM directives. We strongly recommend that users use the xlf90_r and xlf95_r with the -qfixed option rather that xlf_r.

4.1.9 Automatic parallelization

The IBM compiler has state-of-the-art capabilities for parallelizing Fortran programs. The major problem with automatic parallelization is the potential that a parallelized DO loop may run slower than a sequential DO loop. When applying automatic parallelization across an entire program, some loops will result in a speed-up, and some may result in slower execution. The combination is questionable. Selective use of automatic parallelization can be very useful.

The runtime library performs checking of the execution of the parallel DO loop to decide whether to run the loop sequentially the next time the loop is encountered. This checking of the performance can be controlled with environment variables. The following variables control this runtime check.

```
export XLSMPOPTS="parthreshold = .1"
```

`parthreshold` specifies the time in milliseconds below which the loop will run in serial. The default value is 0.2.

```
export XLSMPOPTS=" profilefreq=1"
```

`profilereq` specifies the frequency with which the loop should be analyzed to check for parallel execution (a value of zero specifies that profiling should be turned offf).

```
export XLSMPOPTS=" seqthreshold = .05"
```

`seqthreshold` specifies the time in milliseconds beyond which a previously sequential loop will be run in parallel when being rechecked. The default is 5 milliseconds.

Consider the following graph of the stream benchmark, the TRIAD loop. Notice that the automatically-parallelized plot jumps from the sequential to the OpenMP version at 200,000 iterations. This implies that the default value for parthreshold is too large. Since the OpenMP version of this example best

performs around 10,000 iterations, we should set this environment variable to .05 or .025. Once again, the user is encouraged to make several tests to see what works best for their application.

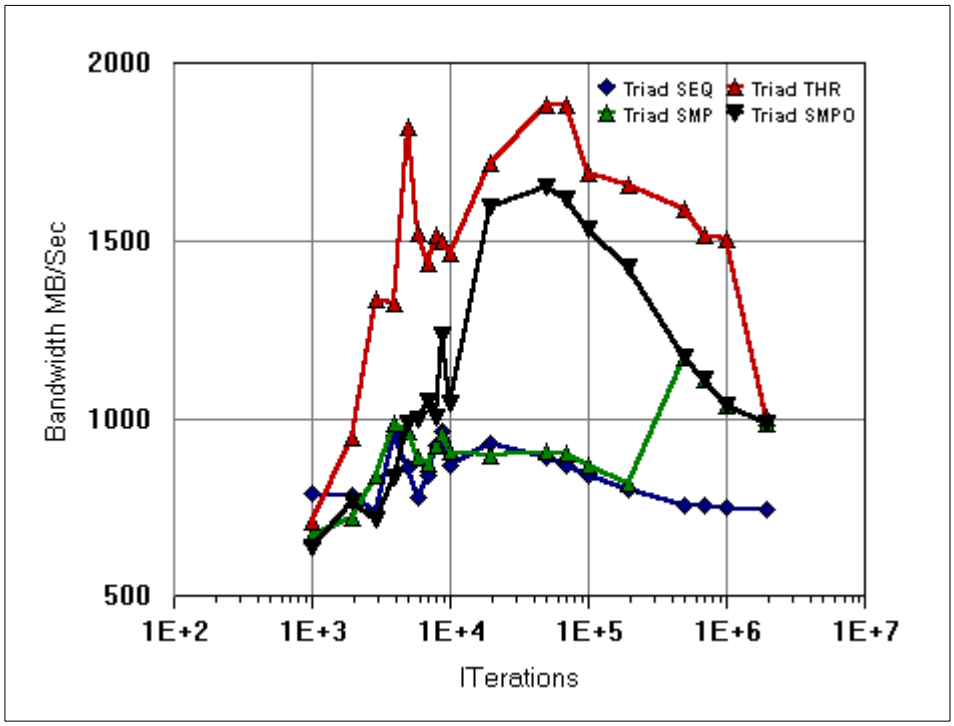


Figure 28. Stream rates for Triad

4.1.10 Granularity and parallelization overhead

When a parallel SMP program is executed, a single (master) thread executes until a parallel Do loop or parallel Section is encountered. At this first encounter of parallel code, the master creates the remaining threads; if the number of threads is two, the master will create one additional thread. At the end of this initial parallel code, the new thread(s) will either be put to sleep or will be placed in a spin loop. Environment variables control how long the thread will be in a spin loop before it goes to sleep. Since it takes longer to wake up a sleeping thread, it is preferred to keeps the threads spinning when they are not in use.

This spinning of threads will utilize 100 percent of the CPU that is running the thread. If the SMP program is running dedicated, this is not an issue. If the operating system is running multi-users simultaneously, the CPUs of the SMP

will not be utilized efficiently. The principal environment variable that controls the sleeping/spinning state of the threads is:

```
export SPINLOOPTIME=5000
```

This variable should be set to the number of times the thread should retry a busy lock before going to sleep.

From the Fortran documentation, two additional environment variables are supplied, which are meant to control the spin time. These environment variables, spins, and yields do not appear to have as positive an effect as SPINLOOPTIME.

In addition to the overhead of the thread entering and leaving the work, there is synchronization overhead. Most of the time, all threads must wait at the end of a DO loop until all threads are complete with the execution of the parallel code. This is what is known as a BARRIER, and it is the major source of synchronization time. Another source of synchronization time is the execution of LOCKS around a reduction function. In an earlier example, the reduction function sum had to be updated outside the parallel DO loop, but inside the parallel region. To keep more than one thread at a time from updating this variable, a lock is imposed around sum's update. The lock consists of a fetch/modify of a memory location. This introduces overhead to synchronize the processors around the summation. This is the main reason for pulling the summation outside the DO loop: Local sums are calculated inside the parallel DO loop, and then each processor needs to execute the locked region only once to update the global sum.

Synchronization within a parallel DO loop should be avoided. Sometimes, this can be extremely difficult, particularly when a summation that must be synchronized is in a subroutine or function called from the parallel DO loop. The only time this synchronization should be used inside such a DO loop is if the granularity of the DO loop is extremely large - large enough to overcome the overhead of the synchronization.

While OpenMP has the notion of an ORDERED region, this should be avoided at all costs. While the reduction function can be performed with a CRITICAL region, that is, only one processor can enter the region at a time in any order, the ORDERED region requires the processors to execute the region in the order specified by the loop index. Whichever thread has the first iteration must go through the region first, then, the thread that has the second iteration, and so on. When an ORDERED region is used, the iterations should be handed out one at a time, which reduces granularity, which cannot overcome the overhead of the ORDERED region.

4.1.11 Parallelization examples

The first example we will investigate is the shallow water benchmark. This is one of the SPEC 95 benchmarks, and it parallelizes fairly well. Consider the following loop from CALC1:

```
C$OMP PARALLELDO
C$OMP&SHARED (FSDY,FSDX,M,N,U,V,P,CU,CV,Z,H)
C$OMP&PRIVATE (I,J)
    DO 100 j = 1, n
    DO 100 i = 1, m
      cu(i + 1, j) = .5 * (p(i + 1, j) + p(i, j)) * u(i + 1, j)
      cv(i, j + 1) = .5 * (p(i, j + 1) + p(i, j)) * v(i, j + 1)
      z(i + 1, j + 1) = (fsdx * (v(i + 1, j + 1) - v(i, j + 1)) -
      .        fsdy * (u(i + 1, j + 1) - u(i + 1, j))) / (p(i, j)
      .        + p(i + 1, j) + p(i + 1, j + 1) + p(i, j + 1))
      h(i, j) = p(i, j) + .25 * (u(i + 1, j) * u(i + 1, j) + u(i, j)
      .        *u(i, j) + v(i, j + 1) * v(i, j + 1) + v(i, j) * v(i, j))
    100 CONTINUE
```

This is a fairly simple DO loop that automatically parallelizes when -qsmp is used. A second case was run where the parallelization was controlled by OpenMP directives. This case was run using -qsmp=omp. The following plot shows the execution of two versions of parallelization: -qsmp, which does automatic parallelization, and -qsmp=omp, which only parallelizes DO loops containing OpenMP directives.

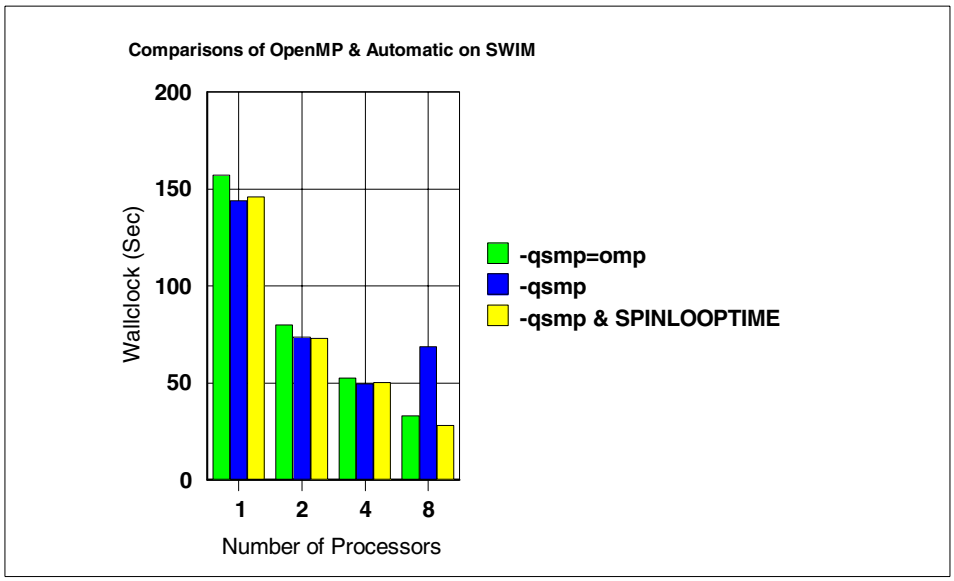


Figure 29. Comparisons of OpenMP and Automatic on SWIM

Notice that the Automatic parallelization beats OpenMP for few numbers of processors. This is because the automatic parallelizer parallelized more DO

loops than were parallelized by placing the OpenMP directives manually. For eight processors, the OpenMP version ran faster. This is due to the granularity of the DO loops parallelized by the automatic parallelizer. When the SPINLOOPTIME environment variable is set to cause the threads to spin all the time, the automatic parallelization runs faster.

Four of the NAS Benchmarks have been parallelized with OpenMP. These represent a range of performance from the EP example that is exactly linear to the MG example that scales fairly poorly. The next figure gives times for the four benchmarks considered.

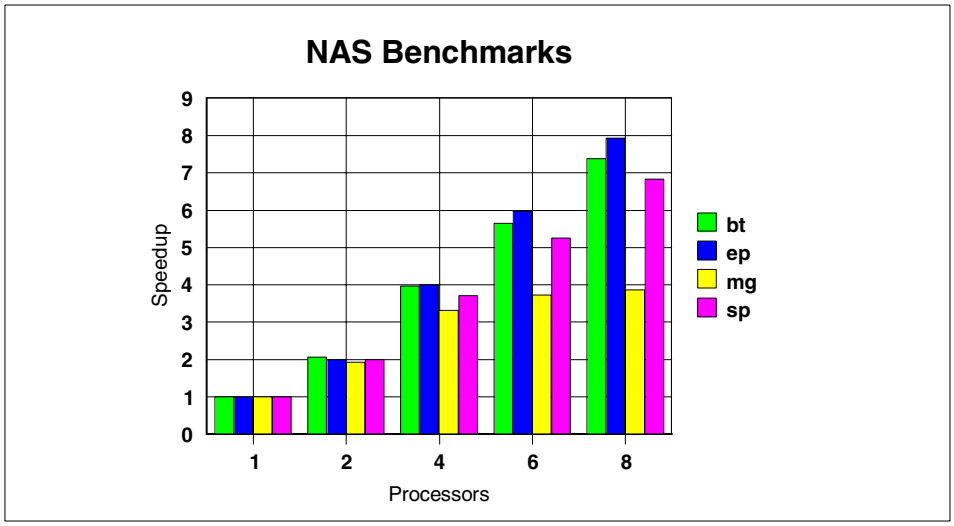


Figure 30. NAS Benchmarks

EP is a simple embarrassingly-parallel program that should run linearly. BT is somewhat more complicated, and it runs very well. SP runs reasonably well; however, we do not get close enough to linear scaling. This is due to the fact that some of the important DO loops have not been parallelized. MGs lack of scalability is primarily due to the granularity of the calculation at eight processors. MG has a range of mesh sizes from the maximum of 256x256 to a much smaller mesh. This variation of grid sizes results in more overhead for the small grids, which degrades the overall performance.

4.1.12 Debugging an OpenMP program

Many of the parallel debuggers work with threads. For those who believe print is the only way to debug, it is important to use the

```
myid = OMP_GET_THREAD_NUM()
```

function to include in the print; so, the individual outputs can be separated. The most common mistake in using OpenMP is not specifying the scoping correctly. Particularly when an outer DO loop, which contains a large number of subroutines and functions, is parallelized. Such a mistake can result in the parallelized code giving correct answers sometimes and incorrect answers at other times.

4.1.13 Compiler switches and environment variables

We have seen many examples using compiler switches. The following is a review of everything that pertains to shared memory parallelization:

- -qsmp
  - Recognizes OpenMP, SMP, and IBMP directives.
  - Performs automatic parallelization.
- -qsmp=omp
  - Recognizes OpenMP directive.
  - Turns off automatic parallelization.

Other options

- auto | noauto
  - Turns automatic parallelization on or off.
- omp | noomp
  - Enforce stricter checking.
- schedule
  - Allows a user to specify default scheduling options at compile time.
- Threshold
  - Allows a user to specify the minimum number of iterations for a DO loop to be automatically parallelized. Default is 100.

Additionally, there are several Environment variables that the user can set to improve performance. These environment variables are set with the XLSMPOPTS variable.

- schedule
  - schedule options discussed earlier in this chapter can be set.
- parthds
  - The number of threads to be used for parallel execution; Default is the number of processors.

- usrthds
  - The number of threads the program will create; Default is 0.
- stack
  - The largest amount of space in bytes that a thread's stack will need. Default is 4194304 bytes.
- spins/yields/delays
  - This was discussed earlier. Controls the busy-waiting state of the threads.
- parthreshold/seqthreshold/profilereq
  - This was discussed earlier. Controls when to run loop parallelized automatically in parallel.

Table 22 is a summary of xlf compliance with the OpenMP 1.0 Standard. In this table, all items marked *future* will be in XLF 7.1.

Table 22. XLF OpenMP summary

OpenMP Feature	XLF Rel.
Directive Sentinels !$omp c$omp *$omp	XLFV5
Conditional compilation	Future
Parallel regions support	XLFV6
Worksharing DO/ENDDO w/NOWAIT	XLFV6
Worksharing SECTIONS/END SECTIONS	Future
SINGLE/END SINGLE	Future
PARALLEL DO clauses (if, private, lastprivate, reduction, schedule, shared)	XLFV5
PARALLEL DO clauses (firstprivate, default)	XLFV6
END PARALLEL DO	XLFV6
THREADPRIVATE directive and copyin clause	Future
ORDERED directive and ordered clause	Future
PARALLEL SECTIONS (if, private, reduction, shared)	XLFV5
PARALLEL SECTIONS clauses (firstprivate, default, lastprivate)	XLFV6
Scheduling algorithms: static, dynamic, guided, runtime	XLFV5

OpenMP Feature	XLF Rel.
MASTER/END MASTER	XLFV6
CRITICAL/END CRITICAL	XLFV5
BARRIER	XLFV6
ATOMIC	Future
FLUSH	Future
omp_get_thread_num_procedure	XLFV6
OMP runtime library (remaining)	Future
Environment variables	Future (much function in "XLSMPOPTS" today)

4.2 Programming with threads

Threads provide a low-level and very flexible method of distributing the work in a given application into separate streams of execution that share a single memory address space. Each thread executes its own function and can be independently controlled by the operating system. This feature is particularly useful for computers whose operating systems have access to multiple processors. On such systems, a program begins to execute as a single thread, and additional threads are created and terminated at will. These threads can be used to concurrently schedule work onto the multiple processors.

There is a standardized application interface for threads called Pthreads (POSIX threads) that is part of the UNIX specification (see item 1 in Section 4.2.9, “References” on page 145). The fact that threads share a common memory address space simplifies access to global memory, but protections are required to ensure that updates to memory locations are properly controlled. In this section, we focus on simple examples of thread programming that we have found to be useful in the context of technical computing with the IBM SP platform. For a more general discussion of Pthreads, we refer the reader to the books about thread programming listed in items 2 - 4 in Section 4.2.9, “References” on page 145).

Programming explicitly with threads is not recommended for the casual user. It requires considerable care to write bulletproof multithreaded code. In many cases, it is possible to get the benefits of multiple threads by using OpenMP directives as discussed in Section 4.1, “Shared memory parallelization with

OpenMP” on page 105, by using SMP-enabled library routines, or by making use of the automatic parallelization capability of compilers. We recommend these simpler approaches in many cases. However, there are times when it is preferable to take direct control of thread management, and, in those situations, the Pthreads interface is required. On IBM systems, a Fortran version of the Pthreads interface is available in addition to the standard C Pthreads interface. This makes it relatively simple to introduce threads into numerically-intensive Fortran applications. We give examples in both C and Fortran, but the reader should recognize that the Fortran implementation is not backed by an industry-wide standard.

4.2.1 Thread creation and termination

Many of the basic features of threads are best discussed within the context of specific examples. We start with a simple "hello threads" program. Although this particular example does not do any useful work, it serves as a good starting point and provides a template for thread creation that can be directly copied into many applications. A C version of "hello threads" is listed in Figure 31 on page 132, and the corresponding Fortran code is listed in Figure 32 on page 133. In this example, the initial thread creates four worker threads, and each worker thread prints a "hello" message and terminates. The same steps would be taken in any multithreaded application where the work is partitioned according to an integer thread identifier. The application should be compiled and linked with cc_r, which defines the symbol _THREAD_SAFE and links with the Pthreads library.

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

void * thfunc(void * arg)
{
    int id;

    id = *((int *) arg);

    printf("hello from thread %d \n", id);
    return NULL;
}

int main(void)
{
    pthread_t thread[4];
    pthread_attr_t attr;
    int arg[4] = {0,1,2,3};
    int i;

    // setup joinable threads with system scope
    pthread_attr_init(&attr)
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    // create N threads
    for (i=0; i<4; i++)
    {
        pthread_create(&thread[i], &attr, thfunc, (void *) &arg[i]);
    }

    // wait for the N threads to finish
    for (i=0; i<4; i++) pthread_join(thread[i], NULL);

    return 0 ;
}
```

Figure 31. C version of "hello threads" - Template for thread creation

Threads are created using the pthread_create function. This function has four arguments: A thread identifier, which is returned upon successful completion, a pointer to a thread-attributes object, the function that the thread will execute, and the argument of the thread function. The thread function takes a single pointer argument (of type void *) and returns a pointer (of type void *). In practice, the argument to the thread function is often a pointer to a structure, and the structure may contain many data items that are accessible to the thread function.

In the "hello threads" example, the argument is a pointer to an integer, and the integer is used to identify the thread. Error checking was not implemented in this example. In real applications, it is advisable to #include <errno.h>, check the return codes from the Pthread functions, and use the perror() routine to print error messages if the return code indicates failure. The thread

attributes that are listed in Figure 31 and Figure 32 are explained in the next section.

The IBM Fortran version of the Pthreads API is similar to the C version, where the function names and data types from C are preceded with f_. Fortran programs that use explicit Pthread routines must have a statement to include the f_pthread module. A number of Fortran routines, including f_pthread_create, have call sequences that differ from the standard C version. For example, the f_pthread_create function takes an additional parameter to specify properties of the argument to the thread function. The IBM Fortran implementation of Pthreads is described in the *XL Fortran for AIX Language Reference*, SC09-2718.

Multithreaded Fortran applications should be compiled and linked with xlf_r, which sets certain compiler options, such as -qthreaded, and links with the Pthreads library. In addition, for Fortran 77 applications, it is usually necessary to turn on the -qnosave compiler option, which ensures that local variables are allocated on the stack and, thus, thread-private.

```
program hello
      use f_pthread
      implicit none
      type(f_pthread_t) thread(4)
      type(f_pthread_attr_t) attr
      external thfunc
      integer i, rc, flag, arg(4)
      data arg/0,1,2,3/

      ! setup joinable threads with system scope
      rc = f_pthread_attr_init(attr)
      rc = f_pthread_attr_setdetachstate(attr, PTHREAD_CREATE_JOINABLE)
      rc = f_pthread_attr_setscope(attr, PTHREAD_SCOPE_SYSTEM)

      ! create N threads
      flag = FLAG_DEFAULT
      do i = 1, 4
         rc = f_pthread_create(thread(i), attr, flag, thfunc, arg(i))
      end do

      ! wait for the N threads to exit
      do i = 1, 4
         rc = f_pthread_join(thread(i))
      end do

      end

      subroutine thfunc(id)
      implicit none
      integer id
      write(6,*) 'hello from thread ', id
      end
```

Figure 32. Fortran version of "hello threads" - Template for thread creation

The time required for thread creation is an important factor in the performance of multithreaded applications. The simplest method of measuring the overhead is probably with the AIX trace facility. This can be done with the following commands:

```
trace -a -o trcfile -j 465,467; mtcode; trcstop trcrpt -o trace.report
-O exe=on,pid=on trcfile
```

where `mtcode` is the multithreaded code that you want to trace. Such measurements indicate a rather wide variance, except with typical thread creation times of about 200 microseconds. One consequence is that it only makes sense to create/terminate threads in routines that execute for longer than about 1 millisecond. The time spent in each routine can be determined by profiling the application; so, the profile provides valuable information about which routines may be good candidates for creating threads within the routine.

Threads terminate implicitly when they complete execution of the thread function. A thread can terminate itself explicitly by calling pthread_exit. It is also possible for one thread to terminate other threads by calling the pthread_cancel function. The initial thread has a special property. If the initial thread reaches the end of its execution stream and returns, the exit() routine is invoked, and, at that time, all threads that belong to the process will be terminated. However, the initial thread can create "detached" threads, and then safely call pthread_exit(). In this case, the remaining threads will continue execution of their thread functions, and the process will remain active until the last thread exits.

4.2.2 Thread attributes

Although the Pthreads application interface is standardized, some thread attributes may have implementation-dependent default values. In AIX version 4.3, the default thread attributes are that the state of a thread is "detached" and that threads have "process" contention scope. These default values may not be the best choices. In many applications, it is useful for the initial thread to create a group of threads and then wait for them to terminate before continuing further. This is best done with threads that are "joinable", not "detached". To ensure that threads are "joinable", it is necessary to initialize a thread-attributes object, and then call pthread_attr_setdetachstate to set the state to "joinable" as shown in Figure 31 and Figure 32. With "joinable" threads, one can use the function pthread_join to suspend the calling thread until the referenced thread has terminated. In the "hello threads" example, the call to pthread_join ensures that the initial thread does not exit until all of the other threads have finished their work. The pthread_join function will return with an error code if the referenced thread is "detached" instead of "joinable";

so, it is necessary to specify "joinable" threads if you want to use the pthread_join function. The thread contention scope is very important from the performance perspective. Threads with "process" scope share a kernel thread with other threads of "process" scope. AIX 4.3 supports a general thread-mapping model, which can map N user threads of "process" scope onto M kernel threads. The default for AIX 4.3.2 is that there are eight threads of "process" scope per kernel thread. The important consideration is that threads with "process" scope do not directly correspond to kernel threads, which are the entities that are scheduled onto processors by the operating system. In contrast, threads with "system" scope are mapped one-to-one onto kernel threads. In many applications, the intended benefit of creating P threads is to ensure that P processors can work concurrently to improve the performance of the application. This requires the creation of threads with "system" scope. One can control the default AIX thread scope with an environment variable, AIXTHREAD_SCOPE. This can be set to P for "process", which is the AIX default, or S for "system", but it is preferable to directly specify the thread scope attribute within the application using the pthread_attr_setscope function as shown in the examples.

The default AIX thread attributes are discussed in detail in *AIX V4 General Programming Concepts: Writing and Debugging Programs*, SC23-2533. In our experience, instead of using the default attributes, it is often desirable to make the threads "joinable" and set the contention scope to "system" as shown in Figure 31 and Figure 32. There are many other thread-attributes that can be customized. For example, the default thread-scheduling policy in AIX is not round robin (RR) or first-in-first-out (FIFO); it is labeled as SCHED_OTHER, and it uses processor affinity and dynamically-adjusted thread priority values.

In AIX, only kernel threads with root authority can use a fixed priority scheduling policy (RR or FIFO). In a normal user program, an attempt to create a thread with the scheduling policy set to SCHED_RR or SCHED_FIFO will fail with an error message indicating wrong ownership. These scheduling policies are available to normal users only if the executable is owned by root with the suitable permission bits set. Threads are, by default, not bound to a specific processor. The operating system will occasionally move threads among the available processors. However, the default scheduling policy, SCHED_OTHER, takes processor affinity into consideration. Our experience has been that the default scheduling attributes are adequate for most purposes including numerically-intensive multithreaded computations.

4.2.3 Programming models

With Pthreads, one can implement different parallel programming models. The "hello threads" examples listed in Figure 31 and Figure 32 illustrate the master-slave model. The initial, or master, thread creates N worker threads and then waits until the workers are done. One can modify the thread creation template to make the master thread work too. For example, the initial thread can create N-1 threads and then execute the thread function before calling pthread_join as shown in Figure 33 on page 136. This technique can significantly reduce the thread creation overhead compared to the lazy-master model. In addition to the master-slave and equal-worker models, one can construct workflow pipelines or other algorithms for distributing work among threads:

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

void * thfunc(void * arg)
{
    int id;

    id = *((int *) arg);

    printf("hello from thread %d \n", id);
    return NULL;
}

int main(void)
{
    pthread_t thread[4];
    pthread_attr_t attr;
    int arg[4] = {0,1,2,3};
    int i;

    // set-up joinable threads with system scope
    pthread_attr_init(&attr)
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    // create N-1 threads
    for (i=1; i<4; i++)
    {
        pthread_create(&thread[i], &attr, thfunc, (void *) &arg[i]);
    }

    // call the thread function
    Function((void *) &arg[0]);

    // wait for all other threads to finish
    for (i=1; i<4; i++) pthread_join(thread[i], NULL);

    return 0 ;
}
```

Figure 33. Version of "hello threads" where initial thread shares the work

So far, our simple examples have specified a fixed number of threads. In many applications, it is useful to have the program decide how many threads to create at run time while providing the ability to override the default behavior by setting an environment variable. For numerically-intensive routines, a sensible default is to ensure that the work is distributed over P threads when there are P processors on line. In AIX, one can get the number of online processors by calling sysconf(). A sample code in C is shown in Figure 34. It includes an option to override the default behavior by checking an environment variable called NUMTHREADS. The sysconf() system call is part of the UNIX specification. However, this call is not required to provide the number of online processors; so, this method may not port to other UNIX flavors. After determining the number of threads to create, the program can use dynamic memory allocation to set up storage for thread data. After that, thread creation follows the templates listed in previous figures.

```
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
...
char * penv;
int ncpus, numthreads;
...
// get the number of online processors
ncpus = sysconf(_SC_NPROCESSORS_ONLN);
if (ncpus < 1) ncpus = 1;

// check the NUMTHREADS environment variable
penv = getenv("NUMTHREADS");
if (penv == NULL) numthreads = ncpus;
else              numthreads = atoi(penv);
...
```

Figure 34. Sample code for setting the number of threads at run time

4.2.4 Synchronization

In many multithreaded applications, it is necessary to ensure that threads update global memory locations in a controlled manner. This is usually done with mutex (mutual exclusion) variables. A mutex object can be locked and unlocked, and the operating system ensures that access to the mutex object is serialized, that is, only one thread can lock the mutex at a given time. Working with mutex variables can be rather tricky. Our experience has been that, in many programs, a barrier synchronization routine for threads would suffice to ensure proper behavior. For example, suppose that multiple threads are working to fill out a table, and, once that is done, each thread needs read access to the table for the next step. A barrier synchronization point would ensure that no thread could proceed to the next step until all threads have finished filling out the table. Instead of working directly with the low-level

pthread_mutex functions, a higher level thread synchronization function is very useful. The construction of a barrier synchronization function provides a good example of how to use mutex variables, and we will describe a thread synchronization routine in some detail.

The main idea in our barrier synchronization example is that a global counter is used to keep track of how many threads have called the barrier function. A mutex lock is used to ensure that only one thread can update the global counter. After updating the counter, the thread is suspended. Once the prescribed number of threads has incremented the counter, all threads are notified and resume execution. The waiting and notification steps require the use of condition variables. A relatively simple thread synchronization function in C is listed in Figure 35 on page 139. The corresponding Fortran version is shown in Figure 36 on page 140.

There are some tricky parts to this routine. The objective is to synchronize N threads. The call to pthread_mutex_lock ensures that only one thread at a time can increment the counter of blocked threads. After incrementing this counter, each of the first N-1 threads will wait on a condition by calling pthread_cond_wait, which releases the mutex. Notice that the call to pthread_cond_wait is placed inside a "while" loop. The reason for this is that the Pthreads specification explicitly permits pthread_cond_wait to return spuriously. The "while" loop should always test a relationship involving a shared variable, which evaluates to true if the awakened thread should continue to wait when pthread_cond_wait returns. The Nth thread to call the synchronization function resets the shared counter of blocked threads to zero, increments the barrier instance, and calls pthread_cond_broadcast, which wakes up the threads that were suspended in the call to pthread_cond_wait. When pthread_cond_wait returns, the calling thread gets the mutex; so, all threads must call pthread_mutex_unlock. This ensures that, upon exit of the synchronization routine, the mutex object is available for a subsequent barrier synchronization call.

```
#include <pthread.h>

int barrier_instance = 0;
int blocked_threads = 0;

pthread_mutex_t sync_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  sync_cond = PTHREAD_COND_INITIALIZER;

int syncthreads(int nth)
{
    int instance;

    // the calling thread implements a lock, other threads block
    pthread_mutex_lock(&sync_lock);

    // the thread with the lock proceeds
    instance = barrier_instance;
    blocked_threads++;

    if (blocked_threads == nth)
    {
        // notify all threads that the sync condition is met
        blocked_threads = 0;
        barrier_instance++;
        pthread_cond_broadcast(&sync_cond);
    }

    while (instance == barrier_instance)
    {
        // release the lock and wait here
        pthread_cond_wait(&sync_cond, &sync_lock);
    }

    // all threads call the unlock function and return
    pthread_mutex_unlock(&sync_lock);

    return(0);
}
```

Figure 35. Listing for a thread synchronization function (in C)

```
SUBROUTINE SYNCTHREADS(kth)
c This routine will synchronize kth threads
      use f_pthread
      implicit none
      integer kth,rc,blocked,phase,myphase
      type(f_pthread_mutex_t) :: mutex
      type(f_pthread_cond_t) :: cond
save mutex, cond, blocked, phase
data blocked /0/, phase /0/
data mutex /PTHREAD_MUTEX_INITIALIZER/
data cond  /PTHREAD_COND_INITIALIZER/

c one thread gets the lock, other threads block
      rc = f_pthread_mutex_lock(mutex)

c the thread with the lock proceeds
      myphase = phase
      blocked = blocked + 1

      if (blocked .eq. kth)then
         blocked = 0
         phase = 1 - phase
         rc = f_pthread_cond_broadcast(cond)
      endif

c wait until the phase variable is switched.
      do while(phase .eq. myphase)
         rc = f_pthread_cond_wait(cond, mutex)
      end do

c All threads call the unlock function.
      rc = f_pthread_mutex_unlock(mutex)
      return
      END
```

Figure 36. Listing for a Fortran version of the thread synchronization routine

Note

This example must be compiled with xlf_r using the -qnosave option for correct behavior.

The thread synchronization routines listed in Figure 35 and Figure 36 can be generalized to include a group label so that one can synchronize threads belonging to a particular group. This would provide a synchronization capability that is roughly comparable to what is available in other parallel programming libraries, such as MPI. In many cases, one can accomplish the programming goals with general synchronization routines. This can help simplify writing multithreaded code, and it minimizes the number of cases where one has to deal directly with mutex and condition variables.

4.2.5 Local vs. shared variables

The synchronization example also emphasizes the difference between local variables and shared variables. In C, variables that are declared within a function are thread-private unless they are explicitly given the "static" storage specifier. Each thread has its own private stack. In contrast, static variables are shared among all threads. Variables that are declared outside a scoping unit have the static storage attribute by default and, thus, are shared among all threads. In the synchronization example, the mutex lock (sync_lock) and the condition variable (sync_cond) are shared objects. Also, the barrier instance and the count of blocked threads are shared variables. In contrast, the integer variable "instance" is declared inside the synchronization routine and is private to each thread.

In xl Fortran, the default storage attributes depend on the language level. For Fortran 77, the default storage attribute is static. When multithreaded Fortran routines are compiled with xlf_r, it is necessary to specify the -qnosave option to ensure that local variables are placed on the stack and are, thus, thread-private. Variables that are specified in a "save" list are shared, as are variables placed in a common block. For Fortran 90 codes, the default option for xlf90_r is that all local variables are placed on the stack and are, thus, thread-private, but variables in a "save" list or in common blocks are shared.

4.2.6 Ray-tracing example

Consider a ray-tracing program in which an image is constructed by tracing rays backward from each pixel in the image plane. The computational work can be done in parallel using threads. We started with a sequential program and first tried to parallelize it for SMP systems using OpenMP directives. This attempt failed, and it was necessary to resort to explicit multithreading with Pthreads. The algorithm for this ray-tracing code could be described as follows:

1. Read the input data.
2. Dynamically allocate arrays.
3. For each pixel in the image plane, trace a ray backward from the image to the object.
4. Once the image is complete, display it and/or save it

The difficulty with automatic or directive-based parallelization was that each ray needed its own private data-structures for lists. A compiler would need to understand the whole algorithm in order to make the correct choices. In the sequential code, the data-structures were dynamically allocated using malloc near the top of the code. The simplest way to multithread the application was

to convert the loop over pixels into a thread function. In order to ensure that each thread has its own private work areas, the relevant calls to malloc were moved to inside the thread function, and the associated pointer variables were declared inside the thread function to make them thread-private. The key programming challenge was to identify which data-structures needed to be shared among threads and which ones needed to be thread-private. An alternative way of getting thread-private memory allocation is to use the alloca() routine inside the thread function. The alloca() function does dynamic memory allocation on the stack, and each thread has its own stack. In AIX, it is necessary to add a compiler flag (-qma) or a pragma directive (#pragma alloca) in order to use the alloca() routine. With either technique, each thread obtains private working areas, and the image can be constructed in parallel with good parallel efficiency. The performance of the ray-tracing code on an eight-CPU POWER3 SMP High node is shown in Table 23. The main reason for the deviation from perfectly linear scaling is that there is a load imbalance, that is, the work is not quite shared evenly among the worker threads.

Table 23. Ray-tracing performance on an eight-way POWER3 SMP High node

Number of threads	Elapsed time (sec)	Speed-up factor
1	3.20	1.00
2	1.66	1.93
4	0.88	3.64
8	0.52	6.15

The ray-tracing example fits nicely into the master-slave model. The initial or master thread reads input and sets up globally-accessible arrays. The master thread creates worker threads to compute the image, using pthread_create, and then calls pthread_join to wait for the workers to finish. After the workers are done, the master thread goes on to display or save the resulting image. Alternatively, the master could create N-1 threads and share the work before calling pthread_join as shown in the listing in Figure 33 on page 136. For a correct multithreaded code, it is critically important to identify the arrays or lists that need to be thread-private and the data-structures that can be safely shared among threads.

4.2.7 Overlapping communication or I/O with computation

The explicit creation of threads can be very useful in cases where one wants a thread to handle a distinct task, such as communication or I/O while computation is in progress. By overlapping computation with communication or I/O, performance can be improved. We will describe an MPI example where the overall work flow is:

1. Initial Phase (initialize MPI, read input, etc.)
2. For many iterations or time-steps

   Step A. Do some computation
   Step B. Write results
   Step C. Repeat steps A and B
3. Final phase (close files, finalize MPI, etc.)

Without threads, the elapsed time per time-step would be the sum of the times for steps A and B above. With a threaded implementation, the time per iteration could be reduced to approximately the larger of the times for steps A and B, because the two steps can be scheduled concurrently by the operating system. If you create threads at the top level to handle the different tasks, the application template would be:

1. Initialize MPI, read input, etc.
2. Set up thread attributes for joinable threads with system scope.
3. Create threads using pthread_create.
4. Wait for all threads to finish using pthread_join 5.
5. Finalize MPI, close files, and exit.

The thread function would contain the iteration or time-step loop as sketched in Figure 37.

```
Begin Thread Function
   For each iteration or time-step
       if thread id = 0
          synchronize threads
          write output
       else
          do computation
          synchronize threads
       end if
   End For
End Thread Function
```

Figure 37. Sketch of a thread function for overlapping computation with I/O

The thread with `id = 0` first waits in a synchronization routine until the threads that are handling the computation have completed one time-step. Then, thread 0 can write output for the first time-step while computation is in

progress for the next step. The routines listed in Figure 35 and Figure 36 can be used to ensure synchronization of the threads.

It is necessary to ensure that the buffer used by thread 0 for I/O is protected against inappropriate updates by the other threads. This can be done with mutex variables or with a barrier synchronization routine. For example, the template for the thread function could be modified to the form shown in Figure 38.

```
Begin Thread Function
    For each iteration or time-step
        if thread id = 0
           synchronize threads to ensure that all workers are done
           copy shared data into a private buffer
           synchronize threads to wait for the copy by thread 0
           write output from the private buffer
        else
           do computation
           synchronize threads to ensure that all workers are done
           synchronize threads to wait for the copy by thread 0
        end if
   End For
End Thread Function
```

Figure 38. Thread function for overlapping computation and I/O

This thread function includes shared data protection. The additional synchronization point introduces a serial bottleneck that will negatively impact performance, but it is necessary to ensure thread-safety.

4.2.8 Concluding remarks

The threads application interface offers great flexibility for making effective use of nodes with multiple processors, but the flexibility comes at the cost of considerable code complexity. In many cases, the simpler approach of using OpenMP directives, SMP-enabled libraries, or the automatic SMP capabilities of compilers is preferred, and it is not always the case that one can get better performance with the explicit use of threads. Debugging multithreaded applications is awkward at best. When the target platform is the IBM SP, it is possible to mix threads with MPI, but a significant amount of care is required, and things are simplest when only one thread is handling communication. For many MPI applications, the best way to use nodes with multiple processors is to have multiple MPI tasks on each node; so, why use threads at all? The answer is that, in some cases, it is the only viable approach. We hope that the

code fragments and discussions in this section will help those who choose to go where only explicit threads can take them.

4.2.9 References

- *CAE Specification, System Interfaces and Headers, Issue 5: Volume 1*, the Open Group 1997, ISBN 1-8591-2181-0. The UNIX specification including the Pthreads interface is available online at www.opengroup.org.
- *Programming With Posix Threads*, by David R. Butenhof, Addison Wesley 1997.
- *Thread Time: The Multithreaded Programming Guide*, by Scott J. Norton, Mark D. Depasquale, Prentice Hall 1996
- *Pthreads Programming*, by Bradford Nichols, et al, O'Reilly & Associates 1996

Chapter 5. Hybrid programming model

With the availability of SMP (Symmetric Multi-Processor) nodes in the RS/6000 SP system, the parallel application developer now has two choices for exploiting the multiple CPUs of each node. With the first (traditional) approach, an MPI task can be initiated for each CPU on each node. For example, with 64 two-way POWER3 SMP Thin/Wide nodes (200 MHz), a 128-way MPI job can be initiated. With the second (combined) approach, a single MPI task can be initiated on each node, and each MPI task can be threaded to use the multiple CPUs on each node. For example, with 64 POWER3 SMP Thin/High nodes, a 64-way MPI job can be initiated where each MPI task uses two threads.

5.1 OpenMP+MPI

With the traditional approach, interprocessor communication takes place between nodes (internode) and within a node (intranode), and each MPI task occupies a single CPU. With the combined approach, interprocessor communication, in it is simplest form, only takes place between nodes, and the computational (and possibly the communications) portions are threaded to occupy the multiple CPUs. This second combined approach is fully supported by the current system software and is currently being used in several Numerical Weather Prediction applications.

Threading can be accomplished using either the Pthreads API or through the use of OpenMP and #pragma directives to prescribe where thread level parallelism exists. The OpenMP directives for Fortran and the #pragma directives for C result in the generation of subroutine and function calls to the Pthreads library and can be viewed as a form of preprocessing. It is anticipated that most users will chose to use the OpenMP and #pragma directives since the learning curve for the compiler directives is shorter than for using the Pthreads library explicitly.

5.1.1 Motivation

It is anticipated that most MPI applications will use the traditional approach of initiating as many MPI tasks on a node as CPUs since this approach requires no additional effort. There are, however, some applications that can benefit significantly, in a performance sense, from the combined approach. Applications that are interprocessor communication-bound and/or exhibit a high degree of load imbalance as the number of MPI tasks increases are prime candidates for the combined approach.

The nature of the performance advantage of the combined approach stems from the architecture of the SP system and the characteristics of MPI codes. With the current SP interconnect fabric, there is one "adapter" per node that is shared by all the MPI tasks performing communication on each node. The net effect is that there is less interprocessor bandwidth and higher latency, on a per-MPI task basis, with multiple MPI tasks per node when compared with a single MPI task per node. For example, with two-way POWER3 SMP Thin/Wide nodes, the per-MPI task nominal exchange bandwidth between nodes is reduced by approximately one-half when comparing two MPI tasks per node with one MPI task per node; this is not surprising since the single adapter in the interconnect fabric is shared.

With regard to applications, codes that exploit a domain decomposition strategy may be characterized by having the total amount of communication directly proportional to the number of MPI tasks. Since the aggregate interprocessor communication performance of a group of nodes is a constant, a programming approach that results in fewer total MPI tasks on the group of nodes will result in less total communication time. In other words, if an application produces more total interprocessor communication with 128 MPI tasks than with 64 MPI tasks, it will require more interprocessor communication time when run on 64 nodes compared with using 64 MPI tasks on 64 nodes. Of course, the computational portions must be effectively threaded or the total runtime will be greater even though the communication time is less.

5.1.2 Logistical considerations - Using POE

In this section, a quick review of how to use POE to initiate interactive parallel jobs will be discussed followed by a discussion of how to use POE in the context of LoadLeveler, the batch queuing system.

For interactive work, POE will either use command line arguments or environment variables to obtain the needed information for initiating parallel execution. The following subset of the POE environment variables is applicable for the traditional and combined approach:

MP_PROCS= total number of MPI tasks

MP_NODES= total number of nodes

MP_TASKS_PER_NODE= number of tasks per node

For the traditional approach, MP_PROCS should be set to the total number of MPI tasks, which is, essentially, the number of nodes multiplied by the number of CPUs per node. Either MP_NODES or MP_TASKS_PER_NODE should also be set. POE will either divide MP_PROCS by

MP_TASKS_PER_NODE to determine the total number of nodes, or it will divide MP_PROCS by MP_NODES to determine the number of tasks per node. Use MP_NODES if MP_PROCS does not divide evenly into MP_TASKS_PER_NODE.

For the combined approach, MP_PROCS should be set to the total number of nodes. Either MP_NODES can be set to MP_PROCS or MP_TASKS_PER_NODE can be set to one.

When using LoadLeveler in batch processing, the MP_PROCS, MP_NODES, and MP_TASKS_PER_NODE environment variables, as well as the command line analogs, are ignored. POE will obtain this information directly from LoadLeveler, and the user must supply the information via LoadLeveler keywords. The relevant keywords are as follows:

#@ total_tasks = total number of MPI tasks

#@ node = total number of nodes

#@ tasks_per_node = number of tasks per node

For the traditional approach, #@ total_tasks should be set to the same value that would be used for MP_PROCS if running interactively. The same applies to #@ node (refer to MP_NODES) and #@ tasks_per_node (refer to MP_TASKS_PER_NODE). The same logic should be applied when using the combined approach. Refer to *LoadLeveler User's Guide Release 2.1*, SH26-7226, and *Parallel Environment for AIX: Operation and Use, Volume 1*, SC28-1979, for additional information as well as the system administration staff for site-specific details.

5.1.3 Logistical considerations - OpenMP

In this section, a discussion of the logistics of building an application that uses the combined approach is presented. The concepts regarding the use of OpenMP can be found in the *XL Fortran for AIX Language Reference*, SC09-2718, and at the following URL:

```
http://www.openmp.org
```

Assuming that the computational and/or communications portions have been threaded via OpenMP directives for Fortran, use the following compiler and options:

```
mpxlf90_r -qsmp=noauto ...
```

If the code uses fixed form Fortran syntax, add the -qfixed option as mpxlf90_r assumes free form syntax. The "-qsmp=noauto" will only result in the interpretation of OpenMP directives. If you wish to have automatic

detection of loop-level parallelism also take place, specify "-qsmp". The "_r" version of mpxlf instructs the compiler to generate thread-safe code and instructs the linker to link the thread-safe version of the system libraries.

Since the MASS scalar routines are not thread-safe, do not link the MASS library if there are any references to MASS scalar intrinsics in any parallel regions. MASS Version 2.5 does provide thread-safe vector intrinsics. If any ESSL (Engineering and Scientific Subroutine Library) routines are called in parallel regions, specify "-lessl_r" at link time to use the thread-safe version of ESSL.

With respect to controlling the number of threads that are initiated, use the XLSMPOPTS environment variable. For example:

```
export XLSMPOPTS="parthds=2"
```

will instruct the OpenMP runtime library to create two threads in the parallel regions. If not set, "parthds" will default to the number of CPUs on the node. Refer to the *XL Fortran for AIX Language Reference*, SC09-2718, for additional details regarding the XLSMPOPTS environment variable and for additional information on compiler options.

5.1.4 Special hardware considerations

For the new POWER3 SMP High node (eight-way) on the current generation interconnect fabric, the combined approach may be the only way to fully exploit the node for an MPI code. The current node adapters support a maximum of four-MPI tasks in User Space. This capability is adequate for the four-way 332 MHz SMP node and the two-way POWER3 SMP Thin/Wide node. For the eight-way POWER3 SMP High node, there are now more CPUs available than the maximum number of MPI tasks in User Space; eight MPI tasks cannot be initiated in User Space. Eight MPI tasks can be initiated using the IP protocol over the interconnect fabric, but the increase in latency when using IP has negative performance implications. If an MPI application wants to use all eight CPUs and User Space, the combined approach is required. Of course, many permutations are possible.

For example, four MPI tasks can be initiated on each node, and the number of threads per task can be set to 2 (via the XLSMPOPTS environment variable) to create a scenario that uses all eight CPUs and exploits User Space. Similarly, one MPI task per node with eight threads per task will also occupy all eight CPUs.

It is expected that future hardware development for the SP Switch fabric interconnection will increase the number of MPI tasks supported in User

Space. The combined approach (MPI tasks and threads) will be appropriate for some applications since the number of "ports" from a node to the interconnect fabric may be exceeded by the number of CPUs per node. Greater per-MPI task performance can still be achieved by reducing the number of MPI tasks per node. The availability of the multi-protocol MPI (shared memory can be used for intranode communications) should provide for some increase in the internode communication performance by reducing the load on the adapters from MPI tasks that are performing intranode communication.

5.1.5 Some MPI considerations

The most obvious scenario with the combined approach is to thread the computational portions of the code that exist between the MPI calls. With this scenario, the MPI calls are "single-threaded" and, therefore, only use a single CPU. The MPI calls can also be threaded. If the message passing strategy is to perform bidirectional exchanges, there may not be any performance advantage to threading the MPI calls. Unidirectional transfers may exhibit a performance improvement; test this with a simple code as part of your decision making process.

Care must be taken if MPI collective communication routines are threaded. The programmer must make sure that all participating tasks execute collective communications on any given communicator in the same order. For other MPI routines that use "tags", care should be taken to make sure that unique communicators are used for each thread as appropriate.

5.1.6 Performance example

In this section, an example of the performance advantage of the combined approach is presented. Although these results were obtained on four-way 332 MHz SMP nodes, they are representative of other SP node types.

MM5 is a popular Numerical Weather Prediction model that uses the combined approach. The code uses a two-dimensional domain decomposition strategy and the computational portions are very effectively threaded using OpenMP directives. Table 24 on page 152 shows the results of running the code on 16 four-way 332 MHz SMP nodes; the traditional approach uses 64 MPI tasks (four per node), and the combined approach uses 16 MPI tasks (one per node with four threads).

Table 24. MM5 performance on 332 MHz SMP nodes

Approach	Communications Time (secs)	Total Time (secs)
Traditional (64 MPI Tasks)	494	1,755
Combined (16 MPI Tasks)	281	1,505

Note that the total communication time is substantially reduced because the total amount of communication is reduced with fewer total MPI tasks. Specifically, the code produces approximately one-half the total interprocessor communication with 16 MPI tasks as compared with 64 MPI tasks; this factor of two is nearly realized in the measured communication time. In addition, when using the combined approach, the reduced number of MPI tasks results in a reduction in load-imbalance; this can be seen by observing that the reduction in communications time is less than the reduction in the total runtime of the code.

5.1.7 Summary

For certain applications, the combined approach can provide a performance advantage when compared to initiating as many MPI tasks as CPUs on each node. For POWER3 SMP High nodes on the current SP Switch, it may be the only method for fully utilizing the CPUs on the node. In order for the combined approach to be effectively exploited, the computational portions must be efficiently threaded; otherwise, it will make more sense to use the traditional approach of one MPI task per CPU on each node.

5.2 An example of the hybrid programming model

As an example of the hybrid programming model, we are going to look at the matrix-vector multiplication. The example will show how to write a program using the hybrid programming model. It will also shed light on some implementation peculiarities and present some timing results.

The first thing to consider is how the MPI data distribution is going to be. In the following examples, you can see the two basic possibilities to distribute the data. The example shown in Figure 39 distributes the matrix columns, and the example shown in Figure 40 distributes the matrix rows between the different MPI tasks. Since we are going to use Fortran, which stores data columns oriented in memory, we are going to use the first possibility, shown in Figure 39.

For the Problem Ab=c, with Matrix A having the dimension ncols x nrows and the vectors b and c having the dimension nrows, our serial main loop will look like the following:

```
DO j=1,ncols
 DO i=1,nrows
  c(i)=c(i)+a(i,j)*b(i)
 END DO
END DO
```

Figure 39. Matrix columns distributed

Using MPI for the distributed memory programming, there are only two changes:

```
DO j=1,n_loc ! My local part
 DO i=1,nrows
  c(i)=c(i)+a(i,j)*b(i)
 END DO
END DO
CALL MPI_REDUCE_SCATTER(c) ! Update c
```

Figure 40. Matrix rows distributed

The loop length of the outer loop n_loc=ncols/nprocs is the number of columns divided by the number of MPI tasks. For simplicity, it is assumed that this division returns an integer number. The MPI call, MPI_REDUCE_SCATTER, will use the MPI operator, MPI_ADD, to update all local c on the root tasks and then "scatter" the results back to all tasks.

After the distributed memory part of our parallelization is done, we have to decide how to parallelize the shared memory part. Again, as shown in in the following examples, there are two basic ways to do this. The first approach, shown in Figure 41 on page 154, splits each local column onto the threads. The second approach, shown in Figure 42 on page 154, takes the same approach we used for the distributed memory model and assigns whole columns to each thread. We will look into both approaches.

The code fragment for the first example would look like:

```
DO j=1,n_loc ! My local part
!$OMP do parallel
!$OMP shared(a,b)
!$OMP private(i)
!$OMP reduction(c)
 DO i=1,nrows
  c(i)=c(i)+a(i,j)*b(i)
 END DO
END DO
call MPI_REDUCE_SCATTER(c) ! Update c
```

Figure 41. Each local column split onto threads

The second code fragment would look like:

```
!$OMP do parallel
!$OMP shared(a,b)
!$OMP private(i,j)
!$OMP reduction(c)
DO j=1,n_loc ! My local part
 DO i=1,nrows
  c(i)=c(i)+a(i,j)*b(i)
 END DO
END DO
CALL MPI_REDUCE_SCATTER(c) ! Update c
```

Figure 42. Whole columns assigned to each thread

There are some problems with this code and OpenMP. The main problem is that an OpenMP reduction variable has to be a scalar; so, we cannot use the REDUCTION directive, but we have to program the reduction ourselves by introducing a c_loc with the same dimension as c.

```
c=0.0
!$OMP parallel shared (c), private (c_loc)
c_loc=0.0
DO j=1,n_loc
!$OMP DO PRIVATE (i)
 DO i=1,nrows
  c_loc(i)=c_loc(i)+a(i,j)*b(j)
 END DO
!$OMP END DO NOWAIT
END DO
!$OMP CRITICAL
DO i=1,nrows
 c(i)=c(i)+c_loc(i)
```

```
END DO
!$OMP END CRITICAL
!$OMP END PARALLEL
CALL MPI_REDUCE_SCATTER(c)
```

The code for the next example looks exactly like the preceding example, except the $OMP DO statement moves before the j-loop.

Using the IBM extension to the OpenMP standard allows us to use vectors in a reduction statement.

```
c=0.0
!SMP$ PARALLEL REDUCTION(+:c)
c=0.0
DO j=1,n_loc
!SMP$ DO PRIVATE (i)
 DO i=1,nrows
  c(i)=c(i)+a(i,j)*b(j)
 END DO
!SMP$ END DO NOWAIT
END DO
!SMP$ END PARALLEL
CALL MPI_REDUCE_SCATTER(c)
```

At first glance, it seems strange that the variable c is initialized twice, but comparing it with the OpenMP code shows why this is necessary. The first initialization sets the global variable c to zero, and the second sets the local variable c, which corresponds to c_loc in the OpenMP example, to zero. Both initializations are needed to get a properly-running program. Note that the some compilers might initialize the reduction variables depending on the reduction function.

We run both examples on two POWER3 SMP High nodes using the shared memory implementation of MPI. The compilation was done as described earlier, using optimization level -O4 -qtune=pwr3 -qarch=pwr3. For SMP, the following options were used: -qsmp=noauto -qnosave. The results are documented in Table 25 and Table 26 on page 156. As can be seen, the preceding example not only scales better but is also faster than the first example. This is a hint that, when you are programming for shared memory, you should still think "distributed". Also, note that there was no measurable

performance difference between the OpenMP and IBM extension versions of the code.

Table 25. Results in seconds on two NH1 nodes for Example 1

#(MPI tasks)	Threads			
	1	2	4	8
1	218	127	90	173
2	114	64	46	90
4	57	33	26	--
8	31	19	--	--

Table 26. Results in seconds on two NH1 nodes for Example 2

#(MPI tasks)	Threads			
	1	2	4	8
1	207	105	58	32
2	105	53	30	17
4	53	18	17	--
8	28	16	--	--

In conclusion, it should be said that this is just an example. Even more, it is an example with very moderate MPI communication. Thus, it is not surprising that the timings are favorable for MPI. The situation is likely to change if the amount of MPI communication increases and several MPI tasks sharing the same switch adapter generate a serious bottleneck. In this situation, it might be strongly advisable to use mixed mode programming to relieve the adapter from its workload.

5.3 Mixed-mode MPI

PSSP 3.1.1 supersedes the previous release (3.1.0) and provides MPI with shared memory access between processes on the same node while still using Userspace or IP space for communication between processes on different nodes.

In order to enable the shared memory capability of PSSP 3.1.1, the environment variable, MP_SHARED_MEMORY, must be set to YES. The default is NO. In order to get good performance, the environment variable MP_WAIT_MODE must be set to POLL. When PSSP 3.1.1 is referred to in

this section, it implies that the above environment variables are set for good performance.

Operations can be performed either in Userspace or IP space (referred to in this section as US and IP respectively). US gives the best performance, but uses 100 percent CPU while waiting for MPI blocking operations to complete.

To be strictly accurate, some PSSP 3.1.0 measurements were made using 3.1.1 without setting MP_SHARED_MEMORY to YES. This is considered equivalent to PSSP 3.1.0.

5.3.1 Point-to-point operations

The latency and transmission rates that can be achieved between two processes on the same node were measured using the MPI_SENDRECV function. Each process issued MPI_SENDRECV concurrently so as to exchange data.

Figure 43 shows the Latency obtained using US and IP on the same node for PSSP 3.1.1 and PSSP 3.1.0. The time taken for each MPI_SENDRECV call was halved to give the one-way latency. For comparison, the latency for processes on different nodes is also shown. The latency obtained with PSSP 3.1.1 is approximately the same for processes on the same node for both US and IP. Clearly, the latency is much less than that previously obtained with PSSP 3.1.0.

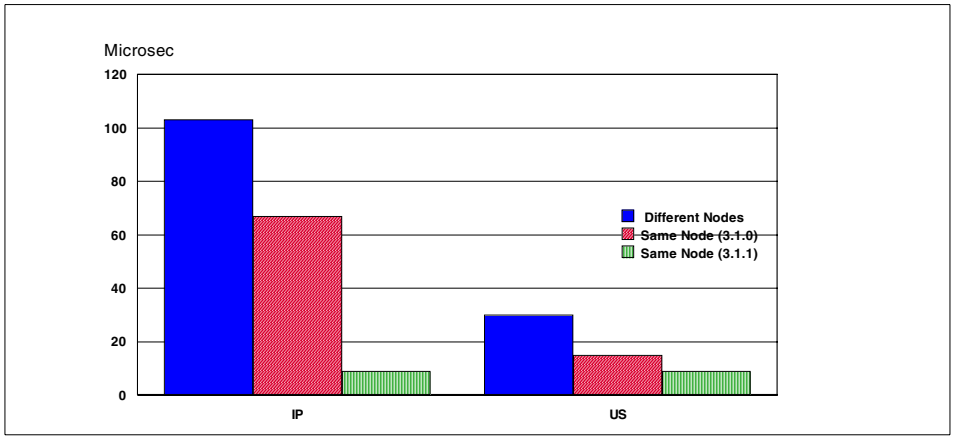


Figure 43. MPI_Sendrecv latency

Figure 44 on page 158 shows the MBps rate obtained (adding data rates in both directions) for processes on the same node. Again, results achieved

with PSSP 3.1.1 are similar for US and IP and are much greater than those achieved previously with PSSP 3.1.0.

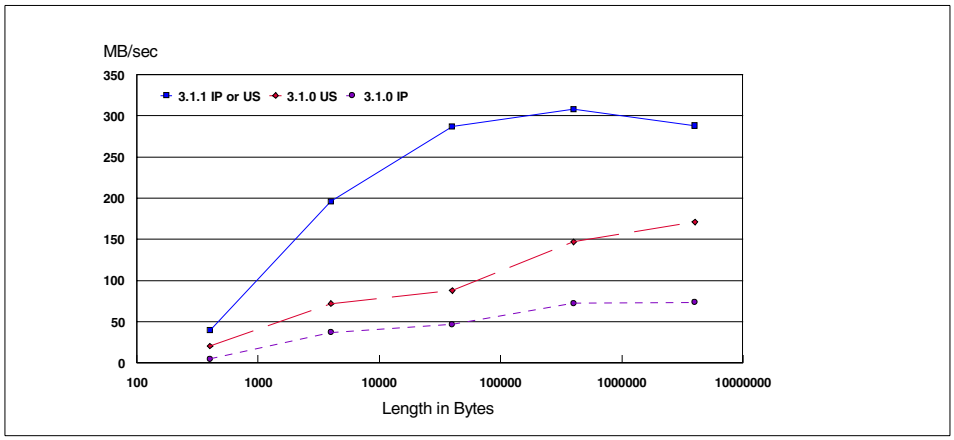


Figure 44. MPI_Sendrecv (same node)

5.3.2 Collective communication

The collective communication operations in MPI involve multiple processes. These processes may be on the same node or different nodes.

Because the entire pattern of communication between the processors is known for each call, there is an excellent opportunity with these calls to make the best use of shared memory and distributed memory.

Parallel programs using point-to-point calls between processes on the same node and between processes on different nodes will probably be limited by the time taken for communication between processes on different nodes. For example, if each process exchanges data with its neighbor, the neighboring processes on the same node will exchange data quickly, but the overall program will be limited by the time the neighboring processes are on different nodes.

For many programs, however, the largest amount of communication time occurs in collective operations, such as MPI_REDUCE, MPI_ALLREDUCE, and MPI_ALLTOALL.

In general, a significant time savings can be achieved by using shared memory to perform as many of these calls as possible between processes on the same node and by performing the minimum amount of communication

between nodes. This is explained in more detail in the following sections for the following specific calls.

- MPI_BARRIER
- MPI_BCAST
- MPI_REDUCE
- MPI_ALLTOALL

The time to execute these calls using the new PSSP 3.1.1 version of PSSP has been measured for different message lengths and different numbers of nodes. The times have been compared with the times taken by the previous PSSP 3.1.0 version. In all cases involving processes on the same node, the PSSP 3.1.1 version is significantly faster.

Measurements were made using both IP and US. When US is used, there is a limitation that permits only four processes per node to participate in the communication, even though the POWER3 SMP High node has eight processors. Thus, when 16 processes participate in the communication, there are four processes on each of the four nodes.

In the case of IP, there is no such limitation; so, for 16 processes, there are eight processes on each of the two nodes. This discrepancy does lead to some interesting comparisons, which are explained in more detail in the following sections.

In the standard IBM implementation of MPI, which supports multiple threads, MPI calls are implemented with a number of locks. The overall design of MPI is explained in a paper by Richard Treumann on the ACTC Web site. The URL for this paper is:

`http://www.research.ibm.com/actc/Tools/MPI_Threads.htm`

In order to improve performance even further, some code has been written for certain specific cases of the calls discussed below. The code uses Shared Memory segments, spin loops, and no locks to reduce the overhead to an even smaller amount than in PSSP 3.1.1.

The code is written mostly in Fortran, with a few C routines to handle the shared memory allocation. When one of the routines is called for the first time, communication groups are created for all processes on the same node and for all lowest process IDs on different nodes. This information is stored in a common block. Specific collective communication calls can then be written as required, making use of the common block information. The routines are

prefixed by MPJ_ (instead of MPI_) so that the application can call them specifically.

This code is entirely unofficial, and results are shown here to indicate what further performance enhancements might be achieved. In the following sections, the enhancements are referred to as the "Turbo" version.

5.3.2.1 Barrier

The time taken (by process 0) for repeated MPI_BARRIER calls was measured for two to 32 processes. The results are shown in Figure 45 on page 160. The results are actually shown as Latency/N (where N is the number of processes), simply to aid representation on a single chart.

As expected, PSSP 3.1.1 US is always lower than PSSP 3.1.0 US. This is particularly so for two and FOUR processes when all the processors are on the same node and all PSSP 3.1.1 communication is using shared memory.

PSSP 3.1.1 IP takes the same time as PSSP 3.1.1 US with two and four processes, and take significantly less time for more than eight processes. For eight processes, IP takes less time than US because all eight processes are on the same node for IP, but are on two nodes for US.

Turbo IP and Turbo US are faster still, particularly when all processes are on the same node. This is the case for two to four processors using US, and two to eight processors using IP.

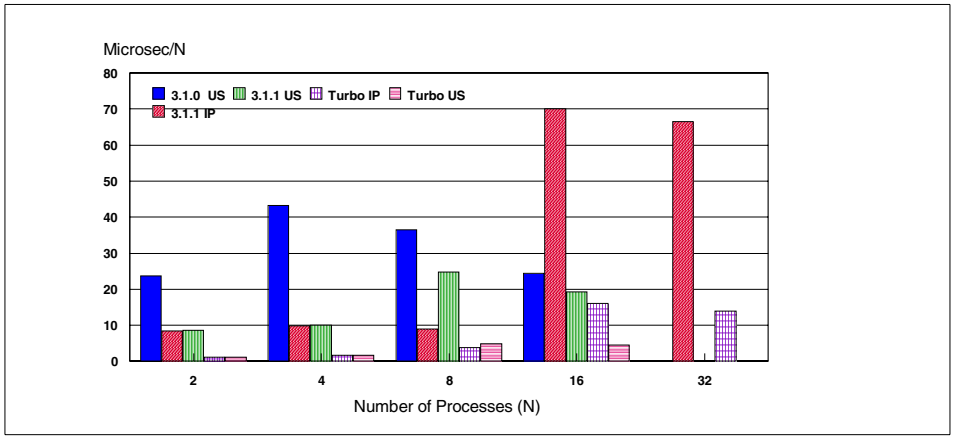


Figure 45. POWER3 SMP High node: MPI_Barrier latency

5.3.2.2 Reduce plus broadcast

It is somewhat difficult to measure the time for repeated MPI_BCAST calls or repeated MPI_REDUCE calls, because, for short message lengths (below the MP_EAGER_LIMIT value), they are not blocked. Consequently, the available buffer space fills up resulting in unpredictable timing.

One way around this problem is to follow each MPI_REDUCE call with an MPI_BCAST call so that they block each other. This combination is actually equivalent to a single MPI_ALLREDUCE call (although the time taken is longer).

The time taken for very short messages is shown in Figure 46. Not surprisingly, these times follow a similar pattern to those of MPI_BARRIER.

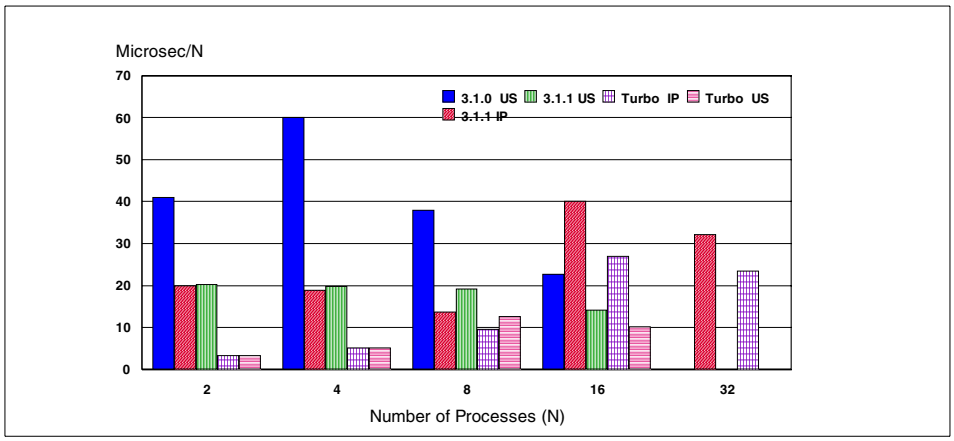


Figure 46. POWER3 SMP High node: MPI_Reduce+MPI_Bcast latency

Figure 47 on page 162 through Figure 50 on page 163 show the MBps rate (for the combined calls) for two to 16 processes for various message lengths.

It is particularly noticeable that, for all numbers of processes and for all message lengths, the PSSP 3.1.1 US rates are much faster than the PSSP 3.1.0 US rates. In fact, except for fairly short message lengths on 16 processors, PSSP 3.1.1 IP is also faster than PSSP 3.1.0 US. Exceptionally, for eight processes, the PSSP 3.1.1 IP rates are faster than the PSSP 3.1.1 US rate. This is because, in the IP case, all processes are on the node.

Finally, note that the Turbo times are, usually, more than twice as fast as the standard times.

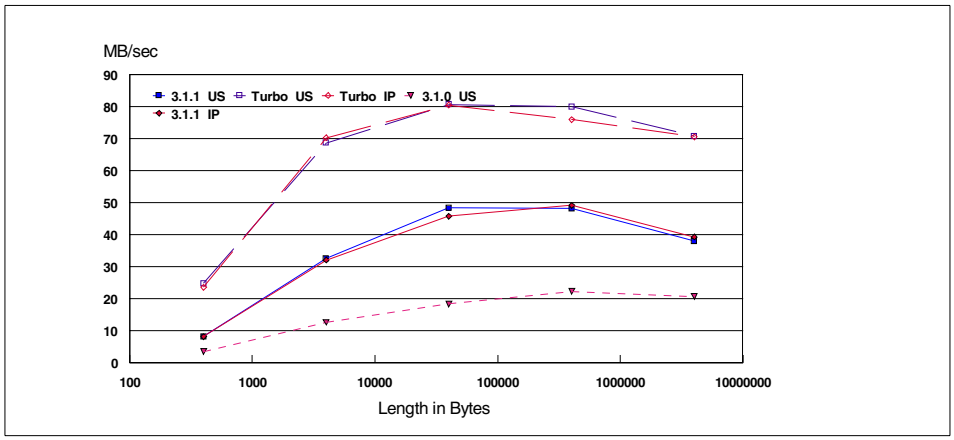


Figure 47. POWER3 SMP High node: MPI_Reduce+MPI_Bcast for two processes

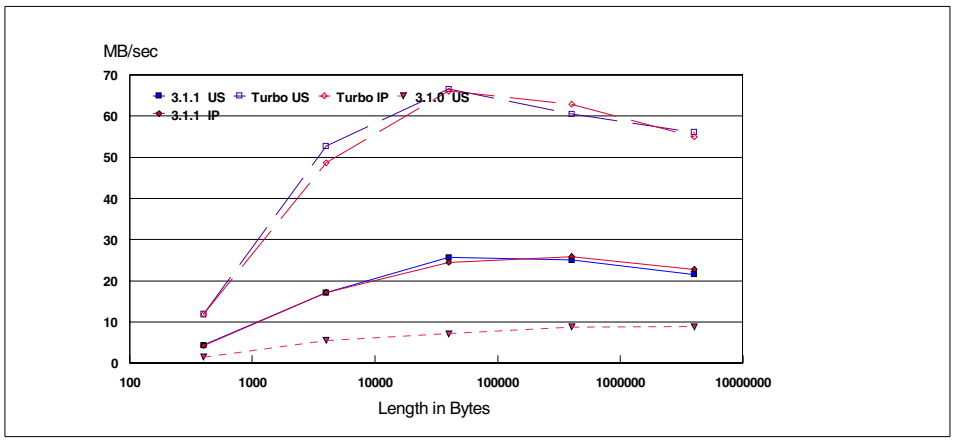


Figure 48. POWER3 SMP High node: MPI_Reduce+MPI_Bcast for four processes

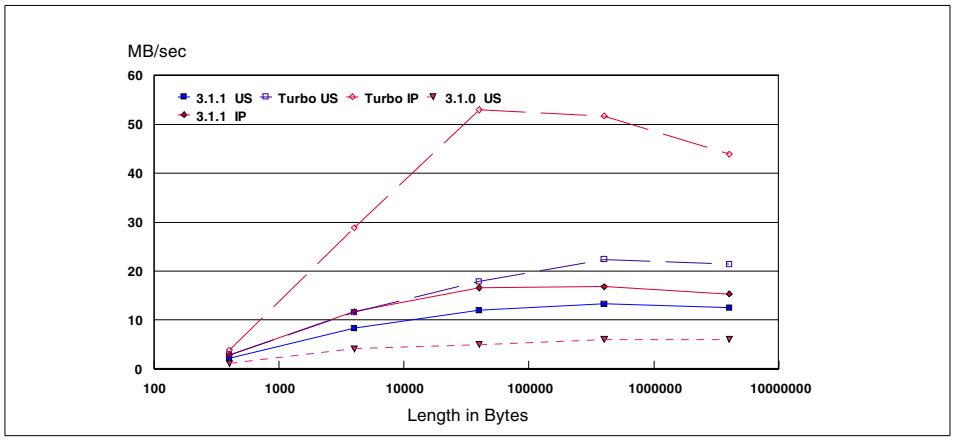


Figure 49. POWER3 SMP High node: MPI_Reduce+MPI_Bcast for eight processes

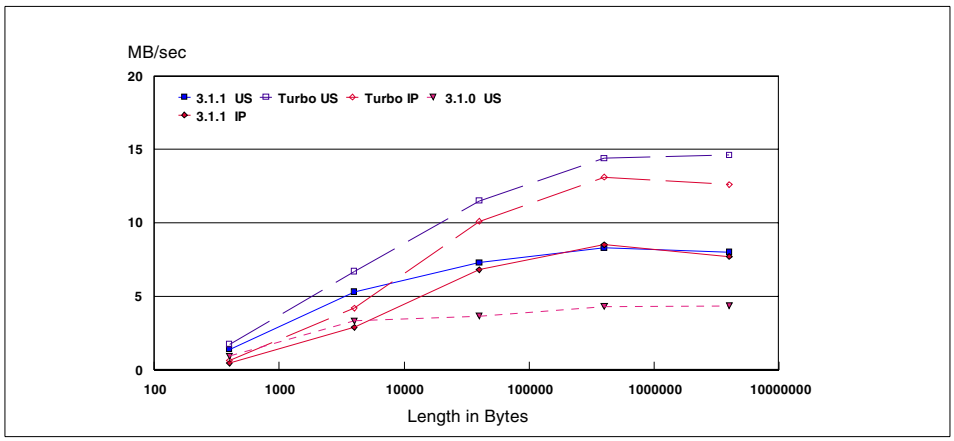


Figure 50. POWER3 SMP High node: MPI_Reduce+MPI_Bcast for 16 processes

5.3.2.3 Alltoall

In Alltoall communication, each processor sends a different message to every other processor. There are a number of algorithms for implementing this procedure.

One way of implementing this algorithm is for each process to start by sending a message to the next highest numbered process and receiving a message from the next lowest numbered process. Wraparound is assumed. Each process then repeats the operation with the second next higher and

second next lower process and so on. For N processors, this results in N-1 latencies for each process.

Frequently, the message length exchanged between processes decreases inversely with N so that, for large numbers of processes, the alltoall communication time is dominated by the latency time.

For the Turbo implementation, the following modification to the above procedure is used. The lowest numbered process on each node uses shared memory to collect all messages from other processes on the same node. The lowest numbered processes on each node then perform alltoall communication between themselves. Finally, the lowest numbered process on each node distributes messages to the other processes on the same node.

The amount of data exchanged between nodes remains the same, but the number of latencies involved is only N-1, where N is the number of nodes rather than the number of processes.

The time taken for very short messages is shown in Figure 51 on page 165. Once again, the times for PSSP 3.1.1 US and IP are very much less than the times for PSSP 3.1.0 US. For eight processors, the PSSP 3.1.1 IP time is slightly less than the PSSP 3.1.1 US time because all processes are on the same node for the IP case but on two nodes for the US case.

The figures on pages 1698 through 170 show the MBps rate for each process for two to 16 processes for various message lengths (N). In this case, N is the total length of all messages sent by each process. Rates for PSSP 3.1.1 US markedly outperform rates for PSSP 3.1.0 US, except for 16 processors, where the improvement is only slight. In fact, for 16 processors, there is a marked improvement in the US times (for both PSSP 3.1.0 and PSSP 3.1.1) when for N=40000. For eight processors, the PSSP 3.1.1 IP rate is greater than the PSSP 3.1.1 US rate because processors are on the same node for the IP case but on two nodes for the US case.

For a single node or for fairly small message lengths, the Turbo implementation results in significant improvements, but, for larger message lengths on more than one node, the overhead of collecting and distributing messages on the same node sometimes exceeds the savings. More work is required on this algorithm!

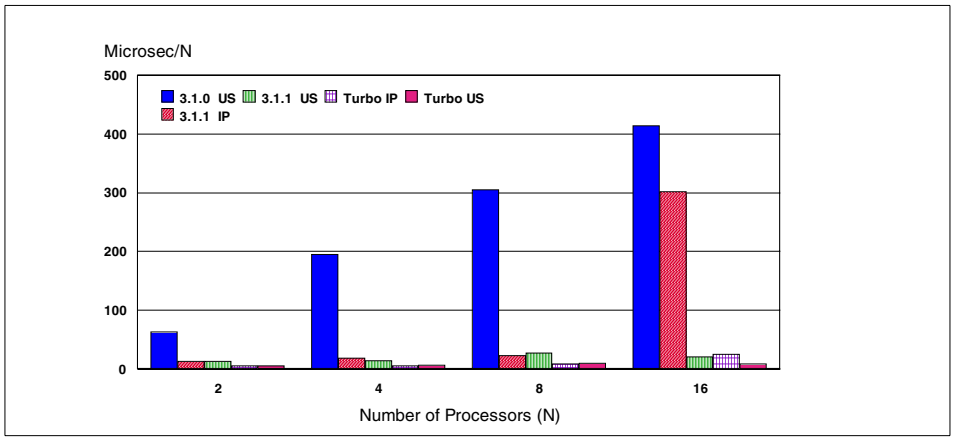


Figure 51. POWER3 SMP High node: MPI_Alltoall Latency

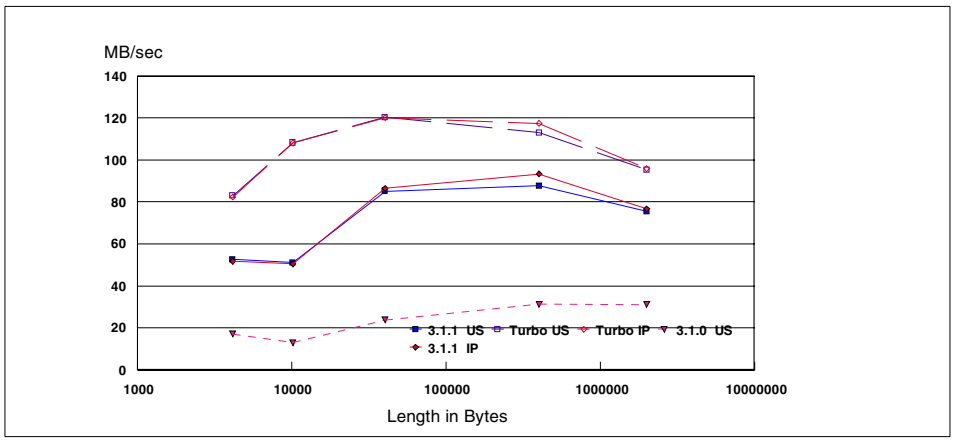


Figure 52. POWER3 SMP High node: MPI_Alltoall for two processes

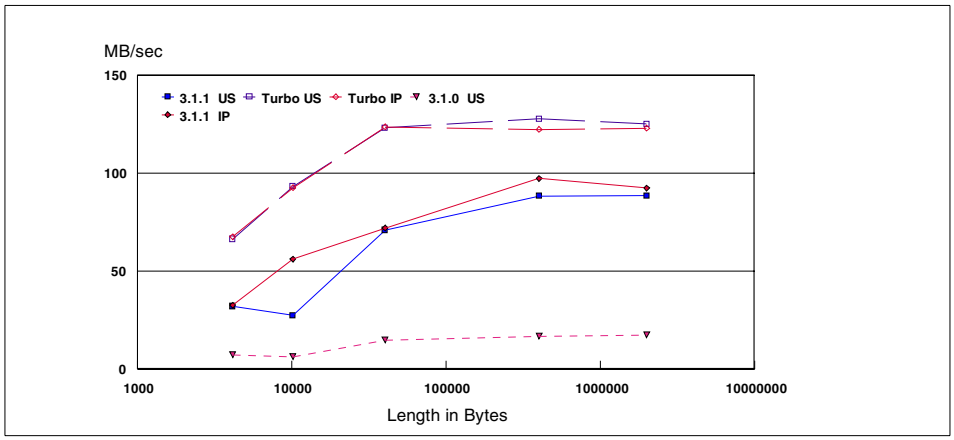


Figure 53. POWER3 SMP High node: MPI_Alltoall for four processes

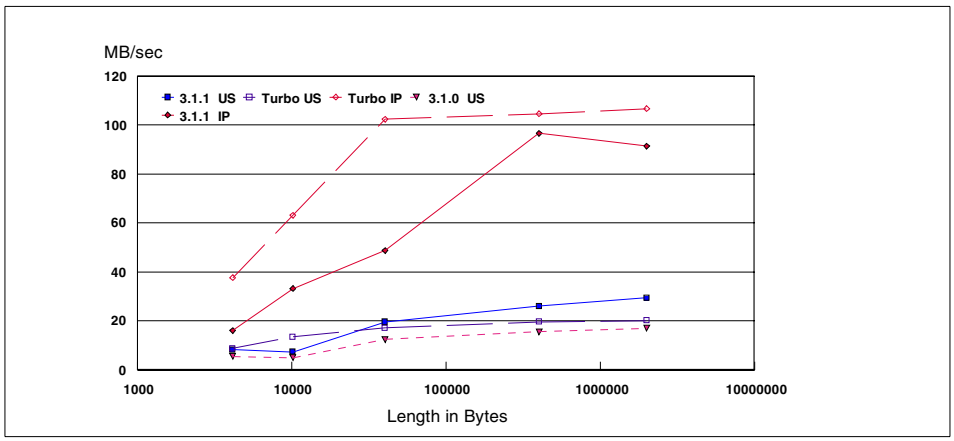


Figure 54. POWER3 SMP High node: MPI_Alltoall for eight processes

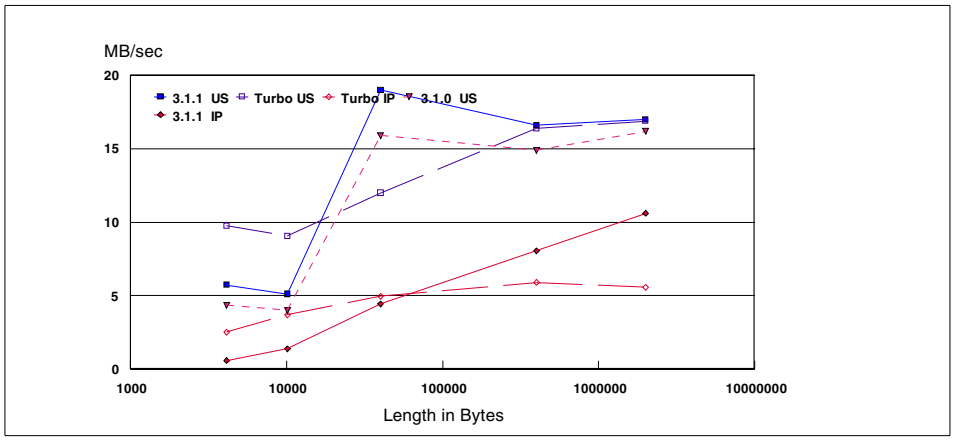


Figure 55. POWER3 SMP High node: MPI_Alltoall for 16 processes

Chapter 6. Input/output

Most program packages execute input and output (I/O) operations, examples of which are interactions with users, exchanges of data with other program packages, and the reading and writing of disk files. In situations in which the performance of such operations significantly influences the performance of a package as a whole, the optimization of I/O is an important consideration. In actual practice, however, this optimization is frequently given less than due consideration not only because I/O can be difficult to characterize, but also because modern computer systems (including the RS/6000 SP) provide an almost bewildering variety of I/O options. Difficulties in the characterization of I/O arise from the basic nature of time-sharing operating systems, which only maintain accounting information for active processes and system processes executing on their behalf and, so, do not reflect I/O wait time; the variety of I/O options is the result of the combinations of facilities provided by the designers of both hardware and system software to mitigate the delays imposed primarily by the electromechanical or distance communication components of I/O hardware.

The processor clock rate of the POWER3 SMP High Node is 222 MHz. See Table 27. Therefore, the CPU cycle time is 4.505 ns (1 ns = 10-9 s). When compared with the highest level in the memory hierarchy, transferring data between RAM and disk requires approximately four orders of magnitude more time per byte of data (1 ms = 10-3 s); this performance gap is likely to grow rapidly in the near future because microprocessor performance improvements significantly outpace those of disk drives.

Table 27. Data transfer rates

Transfer type	Block size (bytes)	Transfer time	
		CPU cycles	Time
Register ← L1 cache	16	1	4.5 ns
Register → L1 cache	8	1	4.5 ns
L1 cache ↔ L2 cache	128	9	40.5 ns
L2 cache ↔ RAM	128	80	360 ns
RAM ↔ disk	4096	$\sim 2.2 \times 10^6$	~10 ms

In this chapter, the presentation will emphasize the I/O of data that are either destined for or being recovered from disk storage. This will permit a treatment of I/O within a single chapter, but it will also address the type of I/O that is of greatest interest to the majority of scientific users of RS/6000 SP systems.

An overview of the I/O hardware subsystem of the POWER3 SMP High Node has already been presented in Section 1.1.1, “Hardware architecture” on page 3. In Section 6.1.1, “POWER3 SMP High Node I/O subsystem” on page 171, we revisit this subsystem in more detail and, in Section 6.1.2, “Disk subsystems” on page 172, we extend the overview to peripherals (disk drives and adapters) that may be attached to the subsystem. The POWER3 SMP High Node is, of course, an element of an RS/6000 SP system. The collective aspect of an RS/6000 SP system in the context of disk I/O is manifested in its network file systems, each of which exploits a communication subsystem to carry data between file server and client nodes; to this extent, an understanding of the communication subsystems is vital to an appreciation of I/O on the RS/6000 SP system; also, for this reason, the communication subsystem of the RS/6000 SP is revisited in Section 6.1.3, “Communication subsystems” on page 172.

File systems are the topic of Section 6.2, “File systems” on page 173. An overview of fixed-disk storage management in AIX is taken up in Section 6.2.1, “AIX file systems” on page 173. Applications that perform I/O operations to files that exceed 2 GB in size may require special attention; this is the topic of Section 6.2.2, “Large file support” on page 176.

The optimization of I/O is treated in Section 6.3, “I/O optimization” on page 177, which is, by far, the largest section of the present chapter and describes proven methods for the improvement of I/O performance.

The IBM general parallel file system for AIX (GPFS) can provide enhanced I/O performance on the RS/6000 SP. All XL Fortran applications as well as POSIX-compliant C and C++ applications may be executed unchanged from a GPFS file system; most will require less I/O time. A brief overview of the GPFS file system from the user’s perspective appears in Section 6.4, “GPFS” on page 211.

Parallel applications that exploit the MPI or LAPI libraries for communication may be rewritten to use I/O facilities provided by the MPI-2 standard for enhanced I/O performance. IBM has implemented a subset of the MPI-IO specification; this is described in Section 6.5, “MPI-IO” on page 214.

There is some interdependency among the topics presented in this chapter and its many suggested references. In many cases, more than one described technique can be brought to bear to improve the performance of a program package. The reader may, therefore, derive benefits from multiple passes through the presented and cited material.

6.1 I/O hardware

Peripherals, such as disk drives, are connected to computer systems through shared communication links known as I/O buses. Some common I/O bus standards are MicroChannel, PCI (peripheral component interconnect), and SCSI (small computer system interface). Bus adapters interconnect I/O buses with memory subsystems and I/O buses with other buses.

The most common scheme by which a CPU addresses an I/O device is known as memory mapping. Portions of the address space are associated with I/O devices. The transfer of data by a CPU to and from these addresses may cause data to be transferred between real memory and a disk subsystem; the actual transfer is typically determined by the amount of memory that can be assigned to serve as a "file cache" and the transfer policy implemented by the operating system.

An I/O bus is physically a single set of wires usually shared by more than one peripheral device. There is, therefore, contention among peripheral devices for the shared link represented by the bus. Moreover, most buses do not support simultaneous read (from memory) and write (to memory) operations.

6.1.1 POWER3 SMP High Node I/O subsystem

The interface between the memory subsystem of the POWER3 SMP High Node and its I/O subsystem is a 6XX data bus; the I/O subsystem as a whole is essentially a peer to each pair of POWER3 microprocessors. Communication and storage devices may be attached to the I/O subsystem through two high-performance interfaces, each of which supports bidirectional communication, one at a peak rate of 2×500 MBps, the other at 2×250 MBps; two ports of the former type and seven of the latter are available; an eighth port of the latter type drives the node's on-board MX ('mezzanine') and PCI buses. An Ultra2 SCSI bus is attached to the PCI bus; the former services both on-board disk bays.

Under the POWER3 SMP High Node's memory-mapped AIX operating system, a CPU's request for I/O data is serviced, when possible, from a file cache in real memory; if this fails, a request is made for data resident on devices attached to the I/O subsystem; data transferred between on-board SCSI disk drives and memory traverses an Ultra2 SCSI bus, a bus adapter to a PCI bus, another bus adapter to one of the 2×250 MBps interconnections, and a switched 6XX data bus into a memory bus that delivers the data into the region of memory corresponding to the I/O device; Serial Storage Architecture (SSA) peripherals connect through an SSA controller to the PCI bus bypassing the Ultra2 SCSI subsystem. The aggregate latency of the

longest hardware path described is much smaller than the latency associated with the electromechanical components of currently available disk drives, and, thus, it does not constitute a significant performance penalty.

6.1.2 Disk subsystems

Disk drives with (unformatted) capacities of 9.1 GB or 18.2 GB are available for attachment to a node's on-board SCSI adapter; in most cases, identical disks occupying the node's two on-board storage bays will be employed as a mirrored pair.[1]

IBM SSA storage subsystems can yield significant improvements in performance over the POWER3 High Node's integrated Ultra2 SCSI interface, which provides a peak bandwidth of 80 MBps. The SSA 6225 adapter is capable of transferring data between IBM 7133 Advanced Models D40 and T40 and an RS/6000 SP node at a rate of up to 160 MBps.[2]

The bandwidth of a disk subsystem is governed by the bandwidth of the adapter, the number of disk drives attached to adapter, the media transfer rates of the disk drives, and the scheme under which data are stored on the drives. Essentially, the latency of a disk subsystem is governed by the disk drives' moving mechanical components, that is, the "seek time" to position the magnetic head and the "rotational latency", after which the required disk sector moves beneath the head.

6.1.3 Communication subsystems

An RS/6000 SP configured for scientific computing is typically equipped with at least two complete and independent communication subsystems: An Ethernet network and an SP Switch network. Ethernet networks comprise adapters that are either those integrated into the system "planar" or supplementary adapters occupying one or more PCI bus slots and interconnection media; these support both the standard 10 megabits/s (10×10^6 bps, or 1.25×10^6 Bps) and Fast 100 Mbps variants. The SP Switch network comprises SPSMX adapters that occupy one MX slot and interconnection media that join adapters and SP Switch ports.

All CPUs on a POWER3 SMP High Node share the communication "channels" effectively provided by the adapters; this implies contention among processors when more than one simultaneously performs communication operations; contention is manifested as an increase in

[1] Other schemes under which data may be stored on drives for AIX systems include striping and compression

[2] Simultaneous read and write operations yield an aggregate bandwidth of 160 MBps. The peak bandwidth for read operations is 85 MBps. Write operations proceed at up to 85 MBps in non RAID mode and up to 35 MBps in RAID 5 mode.

per-CPU effective latency and a decrease in per-CPU effective bandwidth, with the degradation in each case increasing with the number of processors; the trends are monotonic, that is, in a simple linear model, the per-CPU effective latency when *N* CPUs are simultaneously performing communication operations, L_N, is related to the latency when only one CPU is performing such operations, L_1,

$$L_N = L_1 + (N+1) \times \Delta L \qquad 6.1$$

typically ΔL is substantially smaller than L_1; however, when *(N-1)* is sufficiently large, the second term may be comparable with the first. The per-CPU effective bandwidth, in an analogous notation, is

$$B_N = B_1 / N \qquad 6.2$$

In situations where communication channels are simultaneously exploited for interprocess communication and file I/O, some degradation due to contention in both of these functions is to be expected.

6.2 File systems

The file system plays a central role in the AIX operating system[3]. An overview of the AIX file system as it pertains to the organization of fixed-disk storage appears in Section 6.2.1, “AIX file systems” on page 173. A relatively-recent[4] enhancement of the AIX file system is support for large files; this is discussed in Section 6.2.2, “Large file support” on page 176.

6.2.1 AIX file systems

Fixed-disk storage is organized in a hierarchical fashion in AIX; this facilitates its management for both system administrators and users. We shall discuss these from the "bottom up" in this subsection.

An individual fixed-disk (the lowest level in the hierarchy) is a "physical volume". A brief summary of AIX physical volumes may be generated by issuing the `lspv` command at the command line; typical output is as follows:

```
hdisk0  0004400100025d1b  rootvg
hdisk1  00000720c3a7dc2c  rootvg
hdisk2  00002976ebbbdb12  vgssa
hdisk3  00040542e8435291  vgssa
hdisk4  00002976ebbc2469  vgssa
```

[3] "Everything in the UNIX system is a file. This is less of an oversimplification than you might think.'' — B. W. Kernighan and R. Pike, The UNIX Programming Environment.

[4] Large file support first appeared in AIX 4.2.1.

The entries in the first column are physical volume names. Additional device information can be obtained by issuing the `lscfg | grep -i hdisk` command. On the same system, one obtains the following:

```
+ hdisk0  10-68-00-0,0  16 Bit SCSI Disk Drive (9100 MB)
+ hdisk1  10-68-00-1,0  16 Bit SCSI Disk Drive (9100 MB)
* hdisk2  10-80-L       SSA Logical Disk Drive
* hdisk3  10-80-L       SSA Logical Disk Drive
* hdisk4  10-80-L       SSA Logical Disk Drive
```

Physical volumes are combined in "volume groups". Volume group names can be determined by issuing the `lsvg` command. On our prototype system, the generated list is

```
rootvg
vgssa
```

The volume group to which a physical volume belongs is given in the last column of the output of the `lspv` command (see above).

A volume group comprises one or more "logical volumes". Data belonging to a logical volume may be distributed over more than one physical volume within the same volume group. A summary of logical volume information is obtained by issuing `lsvg -o | lsvg -i -l`. On the system used thus far, we obtain:

```
vgssa:
LV NAME  TYPE    LPs   PPs   PVs  LV STATE      MOUNT POINT
bench7   jfs     1599  1599  3    open/syncd    /bench7
loglv00  jfslog  1     1     1    open/syncd    N/A
rootvg:
LV NAME  TYPE    LPs   PPs   PVs  LV STATE      MOUNT POINT
hd5      boot    1     1     1    closed/syncd  N/A
hd6      paging  280   280   1    open/syncd    N/A
hd8      jfslog  1     1     1    open/syncd    N/A
hd4      jfs     2     2     1    open/syncd    /
hd2      jfs     134   134   2    open/syncd    /usr
hd9var   jfs     3     3     1    open/syncd    /var
hd3      jfs     2     2     1    open/syncd    /tmp
lv01     jfs     2     2     2    open/syncd    N/A
lv00     jfs     1     1     1    open/syncd    /var/adm/csd
lv02     jfs     32    32    1    open/syncd    /scratch
benchlv  jfs     300   300   2    open/syncd    /bench1
```

A block of entries that occupies a row in the preceding output is seen to be associated with each volume group. The name of the logical volume is the first item, and the type is the second. The AIX "logical volume manager"

component supports enhanced disk storage performance ("striping"), capacity ("compression"), and availability ("mirroring"). Issuing the `lslv` command against a logical volume's name provides a summary of the current status of the logical volume; for instance, issuing `lslv benchlv` on our prototype system yields the following:

```
LOGICAL VOLUME: benchlv              VOLUME GROUP: rootvg
LV IDENTIFIER:  00000720c3a7df79.11  PERMISSION:   read/write
VG STATE:       active/complete      LV STATE:     opened/syncd
TYPE:           jfs                  WRITE VERIFY: off
MAX LPs:        512                  PP SIZE:      16 megabytes
COPIES:         1                    SCHED POLICY: striped
LPs:            300                  PPs:          300
STALE PPs:      0                    BB POLICY:    relocatable
INTER-POLICY:   maximum              RELOCATABLE:  no
INTRA-POLICY:   middle               UPPER BOUND:  2
MOUNT POINT:    /bench1              LABEL:        /bench1
MIRROR WRITE CONSISTENCY: on
EACH LP COPY ON A SEPARATE PV ?: yes
STRIPE WIDTH:   2
STRIPE SIZE:    64K
```

This indicates that blocks of data are "striped" across two disks in this logical volume, each stripe consisting of 64 KB of data on each disk. Striping can significantly improve disk I/O performance, especially for large amounts of data that are read or written sequentially. Compression invariably imposes a performance penalty due to the increase in processing effort associated with the encoding and decoding of data into and out of more compact representations. Mirroring, where redundant copies (COPIES > 1) of data are written to protect against corruption of one or more copies, may, in some cases, improve read performance because data are read from the least busy disk; mirroring typically reduces write performance, and mirrored volumes cannot be striped.

Data organized in the manner familiar to users of any UNIX-like file system (directories and files) are managed by the journaled file system (JFS) associated with a logical volume. Journaling is a database technique that ensures recoverability if a file system is halted abnormally. Each JFS comprises a pool of blocks of the size of a memory "page"; this is, currently, 4096 bytes or fragments of 512, 1024, or 2048 bytes. One or more such blocks is allocated to each file in a JFS.

AIX supports the Network File System (NFS), which allows access to files on network-connected systems as if they were local. NFS file systems may be mounted over any network for which support for Remote Procedure Call

(RPC) and eXternal Data Representation (XDR) or comparable functionality are available. It is noteworthy that NFS file systems can be mounted over the SP Switch on RS/6000 SP systems so equipped.

Under the AIX operating system, disk files, when accessed in any manner, are mapped into virtual memory. The "virtual memory manager" component of the operating system caches file data in memory and actually transfers data to disk storage only when this memory must be allocated to some other service. This scheme can provide significant improvements in I/O performance when it is possible to reuse file data that reside in memory.

6.2.2 Large file support

An AIX "large file" is one that exceeds 2 GB in length: Such a file has offsets[5] that can exceed 2^{31}-1, which is the maximum (positive) value of a signed integer in a 32-bit two's complement representation. An AIX journaled file system must be built by the system administrator to be large-file-enabled if it is to contain large files[6]; an existing file system that is not large-file-enabled cannot be converted.

Input and output operations executed from within Fortran code (and, thus, handled by the Fortran runtime environment) require no modification to use large files. The off_t datatype defined by the ISO C standard is a 32-bit (signed) long (int) in AIX, unless _LARGE_FILES is defined, whereupon it is a 64-bit (signed) long long (int)[7]. I/O data structures and library calls are also selected to be 32-bit offset or 64-bit offset versions based upon whether _LARGE_FILES is defined or not. Programmers may exploit the large-file capability of AIX and yet retain portability in 32-bit environments by following some simple guidelines:

- Ensure that #include <fcntl.h> defines the system I/O call prototypes: These prototypes should not be "hand coded".
- Use the off_t datatype for offsets; arithmetic manipulations that yield offsets should employ datatypes that will not overflow; the appropriate format specifier should be used to convert between strings and off_t datatypes: Format selection can be based on whether sizeof(off_t) == sizeof (int) or not; the format specifier for the long long datatype is %lld.
- Replace the ISO C standard invocations of fseek and ftell with the POSIX standard invocations fseeko and ftello: The former expect offsets to be of datatype long, while the latter expect them to be of datatype off_t.

[5] The relative location in bytes, measured from the beginning, current location, or end of a file.
[6] AIX Version 4.3 Commands Reference, Volume IV: see the -a bf flag of the crfs command. The maximum size of a large file is 64 GB on AIX 4.3.2 systems.
[7] See /usr/include/sys/types.h.

- Set the file size resource limit to RLIM_INFINITY; this will permit the use of large files.

In this context, we remind the reader that the ISO C standard specifies that the return type of a function with no prototype be assumed to be an int, and that C programs that do not comply with the ISO C standard may exhibit unexpected behaviors: For instance, the prototype of the malloc storage allocation function appears in <stdlib.h> in an ISO C-compliant environment; if the prototype is assumed to reside in <malloc.h>, pointers, which may have 64-bit integer values, may overflow when truncated to the default int type.[8]

6.3 I/O optimization

We shall base our discussion of I/O optimization on simple mathematical models of the time, T, an application spends performing disk I/O. We begin with the simplest model,

$$T = \eta \cdot \tau + V/B \qquad 6.3$$

Here, η is the number of disk I/O requests; τ is the average "latency" of a disk I/O request, which is the average time interval that must elapse before any data are actually transferred and is typically measured in units of ms (10^{-3} seconds) when data are transferred to or from disk media; V is the total volume of data transferred, that is, the number of bytes input and the number of bytes output, and B is the average "bandwidth" , that is, the average rate of data transfer, which is typically measured in MBps (10^6 bytes per second), again, when data are transferred to or from disk media. The rationale for the majority of optimization strategies can be understood in terms of this simplest of I/O models.

Techniques for I/O optimization may be placed in two categories: Those that require no program modifications on the part of the user and those that do; we shall refer to these as "non-intrusive" and "intrusive" optimizations and treat each in turn in the remainder of this section.

6.3.1 Characterizing I/O for non-intrusive optimization

The `iostat` utility provides summaries of I/O activity for physical volumes that reside on the AIX system[9] into which a user is logged. The syntax is:

```
iostat [-d|-t] [PhysicalVolume ...] [Interval[Count]]
```

[8] This is, perhaps, the opportune location to mention that the AIX free call does not release paging-space slots: These are released upon process termination or by the AIX disclaim system call (see /usr/include/sys/shm.h and the subprogram freeiobuffer in Section 6.3.10, “Intrusive optimizations: Exploiting high-performance FS” on page 193).

[9] In the present context, a POWER3 SMP High node is an AIX system.

The iostat command cannot be executed with any options in which case statistics are gathered system-wide; on a lightly-loaded AIX uniprocessor system, such an invocation yields

```
tty: tin tout   avg-cpu: % user  % sys  % idle  % iowait
     0.0 2.3               0.6    0.4    98.6      0.4

Disks:  % tm_act  Kbps  tps   Kb_read  Kb_wrtn
hdisk0     0.4    1.9   0.4    268973  20139673
hdisk1     0.0    0.1   0.0     11917    781264
hdisk2     0.0    0.3   0.0    443598   3072524
hdisk3     0.0    0.3   0.0    435410   2875620
hdisk4     0.0    0.3   0.0    427410   2833836
hdisk5     0.0    0.3   0.0    426026   2820732
cd0        0.0    0.0   0.0         0         0
```

The report comprises two parts that may be selected using either of the mutually exclusive -t or -d options. The first part of the report is a tty and CPU utilization summary; this information is updated at regular intervals by the kernel (typically sixty times per second). tin and tout are the total number of characters read and written by the system for all ttys; % user and % sys are the fractions of CPU utilization at the user and system levels; % idle is the fraction of time that the system was idle and did not have an outstanding disk I/O request; finally, % iowait is the fraction of time during which the CPU(s) were idle and the system had an outstanding disk I/O request. The second part of the report is a disk and CD-ROM utilization summary; the statistics apply to the interval between system boot and command invocation; % tm_act is the fraction of time a storage device was active; Kbps is the rate of transfer in kilobytes per second; tps is the number of transfers (potentially merged multiple logical requests of unspecified size) per second; finally, Kb_read and Kb_wrtn are the total number of kilobytes read and written.

The aggregate volume of disk I/O (measured in kilobytes) for the myexecutable application may be determined by differencing the appropriate entries from disk utilization summaries generated before and after the application is executed:

```
iostat -d
myexecutable < myinput > myoutput
iostat -d
```

We remind the reader that system-wide (not process-specific) data are generated by the iostat command; this data is approximately equal only when the resource utilization of myexecutable dominates that for any other process. In most cases, this can be ensured by granting the user of myexecutable exclusive access to the system.

The "sampling" mode of the `iostat` command is obtained by specifying a positive value for `Interval`; for instance, the sequence of keystrokes

```
iostat 3 &
myexecutable < myinput > myoutput
fg
^C
```

executed under the Korn or any other shell will cause `iostat` to generate a report on stdout every three seconds interval from the background, execute `myexecutable` in the foreground, return the `iostat` process to the foreground, and, finally, interrupt the latter's execution. An alternative that may be preferable in some circumstances is to invoke `iostat` in one window and `myexecutable` in another. In sampling mode, the second and all subsequent reports apply to `Interval` *not* to the interval between system boot and the generation of the most recent report; the number of reports generated by `iostat` may be limited to `Count` instances by specifying the additional positive integer.

Network file systems do not fall under the purview of the `iostat` utility because they reside on physical volumes that are remote. The hostname or the IP address of the remote server may be determined by issuing the `df` command. A partial output from `df` on an arbitrarily selected system, which we shall refer to as the current host, is

```
Filesystem    512-blocks      Free   %Used    Iused  %Iused  Mounted on
/dev/hd4           98304     20696     79%     4351     18%  /
/dev/hd2         4390912    474584     90%    55829     11%  /usr
/dev/hd9var       229376    120568     48%      739      3%  /var
/dev/hd3           65536     63224      4%       55      1%  /tmp
/dev/lv02        1048576   1015568      4%       16      1%  /scratch
/dev/benchlv     8192000   4038728     51%      303      1%  /bench1
v07l12:/aba60   32768000  24624800     25%   115947      3%  /aba60
fsserver1.vendor.pok.ibm.com:/export/fs1fs9/parpia
                16777216   3889200     77%    78060      4%  /u/parpia
```

file systems with names containing a string immediately followed by a colon, such as the last two file systems in the list generated by df above, are network file systems; the hostname of the corresponding server *connection* is the string that precedes the colon. The type of the connection `v07l12` may be determined as follows:

Issue

```
traceroute v07l12
```

to obtain a response of the form

```
traceroute to v07l12.vendor.pok.ibm.com (129.40.1.172)
  from 129.40.1.161 (129.40.1.161), 30 hops max
outgoing MTU = 32768
trying to get source for v07l12
source should be 129.40.1.161
 1 v07l12 (129.40.1.172) 2 ms 2 ms 2 ms
```

indicating that a single "hop" is required to transmit a data packet from IP address `129.40.1.161` to IP address `129.40.1.172`, no "routing" is done. Next, issue

```
rsh v07l12 netstat -r | grep v07l12 | awk '{ print $6 }'
```

to obtain the corresponding interface device name

```
 css0
```

A more complete description of the interface is obtained by issuing

```
 rsh v07l12 lscfg -l css0
```

which yields

```
DEVICE   LOCATION      DESCRIPTION
css0     00-f1000000   SP Switch Communications Adapter (Type 6-A)
```

so that file system `/aba60` is mounted across the SP Switch network.

Data written to and read from the file system `/u/parpia` requires two hops:

```
traceroute fsserver1
```

yields

```
trying to get source for fsserver1
source should be 129.40.16.161
traceroute to fsserver1.vendor.pok.ibm.com (129.40.4.231)
  from 129.40.16.161 (129.40.16.161), 30 hops max
outgoing MTU = 1500
 1 wft7 (129.40.16.190) 4 ms 2 ms 2 ms
 2 fsserver1 (129.40.4.231) 3 ms 3 ms 3 ms
```

indicating that data are routed by `wft7` to server `fsserver1`. Executing

```
rsh fsserver1 netstat -r | grep fsserver1 | awk '{ print $6 }'
```

yields

```
fi0
```

which, per

```
rsh fsserver1 lscfg
```

does not exist, although fddi0

```
DEVICE   LOCATION    DESCRIPTION
fddi0    00-07       FDDI Primary Card, Single Ring Fiber
```

does. Thus, all data are transferred between `fsserver1` and `wft7` over a Fiber Distributed Data Interface (FDDI) 100 MBps fiber-optic network. Host `wft7` does not respond to the command

```
rsh wft7 netstat -r
```

indicating that it is, in fact, a router, not a general-purpose machine performing routing functions. The output of the command

```
netstat -r | grep wft7 | awk '{ print $6 }'
```

executed on the current host yields

```
en1
```

indicating that the router and the current host are connected through an Ethernet network. Data in the file system `/u/parpia` are, thus, transferred between the fileserver `fsserver1` and the router `wft7` through an FDDI network, and between `wft7` and the current host through an Ethernet network.

Clearly, utilities that can monitor network traffic must be employed to characterize I/O for network file systems. Networks are, of course, rarely used exclusively for file I/O; so, care must be taken to ensure that network traffic that is not associated with file I/O is somehow either controlled so as to be negligible or explicitly accounted for. The command `netstat -ni`, when issued before the transfer shows the following output:

```
Name  Mtu   Network     Address          Ipkts    Ierrs Opkts    Oerrs   Coll
lo0   16896 link#1                        518291 0       523696       0 0
lo0   16896 127         127.0.0.1         518291 0       523696       0 0
lo0   16896 ::1                           518291 0       523696       0 0
en0   1500  link#2      0.60.94.e9.e.29   1780781 0      1610592      0 0
en0   1500  129.40.16.6 129.40.16.66      1780781 0      1610592      0 0
en1   1500  link#3      0.60.94.9d.6e.9b 11184736 0     12459433 2190233 0
en1   1500  129.40.16.1 129.40.16.161    11184736 0     12459433 2190233 0
css0  65520 link#4                       6178062 0      7315527      0 0
css0  65520 129.40.1    129.40.1.161     6178062 0      7315527      0 0
```

and, after copying a file of length 29,768,419 bytes between two remote file systems, the `netstat -ni` command shows the following output:

```
Name  Mtu   Network     Address          Ipkts    Ierrs Opkts    Oerrs   Coll
lo0   16896 link#1                        518337 0       523742       0 0
lo0   16896 127         127.0.0.1         518337 0       523742       0 0
lo0   16896 ::1                           518337 0       523742       0 0
```

```
en0   1500  link#2      0.60.94.e9.e.29   1780885 0      1610685       0 0
en0   1500  129.40.16.6 129.40.16.66      1780885 0      1610685       0 0
en1   1500  link#3      0.60.94.9d.6e.9b 11207817 0     12482938 2190233 0
en1   1500  129.40.16.1 129.40.16.161    11207817 0     12482938 2190233 0
css0  65520 link#4                       6178170 0      7315693       0 0
css0  65520 129.40.1    129.40.1.161      6178170 0      7315693       0 0
```

This shows that the great bulk of network traffic in this period was carried by en1 adapter. The differences between the values of `Ipkts` (input packets) and `Opkts` (output packets) in the two outputs of `netstat -ni` are 23,081 and 23,505. The actual volume of data transferred cannot be determined because the size of a packet is not a fixed quantity; other AIX utilities, such as `entstat`, `fddistat`, and `tokstat`, may be used to gather more detailed information for a particular connection.

6.3.2 Non-intrusive optimizations exploiting high-performance FS

Data transferred to or from files in virtual file systems "mounted" on all nodes, such as `$HOME`, generally traverse several hardware "layers": The memory hierarchy of the "client" node on which the application is executing, the I/O subsystem of the client node on which the application is executing, the network that connects the client node with a file "server", and the I/O subsystem and memory hierarchy of the server. In some instances, there is more than one traversal of an I/O subsystem; for instance, data arriving at an Ethernet adapter at a server node is copied to memory and then again to a disk subsystem. System software overheads, such as those connected with the processing of a "protocol stack" or a file system (operations, such as journaling, token management, or instance) may dominate hardware delays in some situations.

High-performance file systems reduce I/O delays by eliminating one or more hardware layers, employing hardware with smaller delays (lower latencies and higher bandwidths) or exploiting "lighter" protocols (those with reduced processing overheads); more than one of these strategies can be combined for greater benefit.

A disk subsystem is said to be "local" to a node of an RS/6000 SP system when data destined for or recovered from this disk subsystem does not traverse a communication subsystem. Since all overheads associated with communication are eliminated, latencies [τ in Eq. (6.3)] are invariably reduced; it is also frequently the case in a multiuser production computing environment that local disk subsystems support higher average bandwidths [*B* in Eq. (6.3)] than most network file systems because there is more contention for network and fileserver resources. Local disk subsystems are typically easily exploited by sequential applications or shared-memory parallel applications; if relative file names are used, the application package

need only be copied to the local file system; if pathnames are used, but set in input files, it is only necessary to modify the pathnames in the input files suitably; if pathnames are "hard coded", a system administrator may be able to remount the local file system to the required path. The exclusive use of local file systems by distributed-memory parallel or hybrid (distributed- and shared-memory) parallel applications frequently requires additional setup or modification of source code. These topics are taken up in Section 6.3.2, “Non-intrusive optimizations exploiting high-performance FS” on page 182, and Section 6.3.10, “Intrusive optimizations: Exploiting high-performance FS” on page 193.

Striping, which is touched upon in Section 6.2.1, “AIX file systems” on page 173, improves performance by eliminating the hardware bottleneck presented to a disk I/O adapter by a single disk drive: Two or more disk drives are simultaneously used to transfer a single "logical" block of data. A striped disk subsystem is exploited in the same manner as a local disk subsystem[10], that is, program packages are relocated to the striped disk subsystem; pathnames are suitably modified, or mount points are appropriately changed. Buffers used in I/O operations should be aligned on page boundaries to obtain maximum performance for striped logical volumes. The XL Fortran compiler option `-qalign=4k` will ensure such alignment for large data objects[11]; the explicit page alignment of I/O buffers in Fortran and of any buffer in C requires modifications to source code and is, therefore, discussed in Section 6.3.10, “Intrusive optimizations: Exploiting high-performance FS” on page 193.

Disk subsystems based on IBM SSA (Serial Storage Architecture) provide performance enhancements at the hardware level through increased adapter throughput for both reading and writing operations and simultaneous reading and writing. An SSA disk subsystem is used in the same manner as a local disk subsystem or a striped disk subsystem: Program packages are relocated to the SSA disk subsystem, pathnames are suitably modified, or mount points are appropriately changed.

The use of the GPFS file system constitutes a non-intrusive optimization technique that is discussed in Section 6.4, “GPFS” on page 211.

6.3.3 Non-intrusive optimizations: Obviating contention

Parallel applications frequently require read-only access to one or more files; if these files are replicated or scattered appropriately onto local file systems, I/O subsystem contention between parallel tasks on different nodes can be eliminated; also, communication subsystem contention between internode

[10] Local disk subsystems are often striped.
[11] The reader is referred to *XL Fortran for AIX User’s Guide* for details.

task messaging and switch mounted I/O is eliminated; these advantages accrue in addition to those described for local file systems in the preceding subsection. Many parallel applications perform I/O to files that are specific to the rank of a parallel task; in such cases, local file systems may be used with the same benefits as read-only files.

The `mcp` command may be used to replicate a file `infile` from a globally-mounted file system onto local file systems on all nodes in the partition allocated to the user[12]. The syntax is as follows:

```
mcp infile [TargetFile|TargetDirectory] [POEOptions]
```

Only one `infile` can be specified per invocation of the command (although the name of this file may be determined by shell pattern-matching), and this file must reside on a globally-mounted file system. The `mcp` command will also fail if `TargetFile` or `TargetDirectory/infile` already exists on any node. To delete all local copies of a file `LocalPathname`, execute:

```
poe "/bin/rm -f LocalPathname" [POEOptions]
```

To delete `LocalDirectory` together with its contents, execute:

```
poe "/bin/rm -rf LocalDirectory" [POEOptions]
```

Omitting the `-f` option may lead to premature termination of parallel tasks on one or more nodes as a result of the failure to find `LocalPathname` or `LocalDirectory` on one or mode nodes. The `mcp` command is likely to fail if more than one parallel task executes on a node. All tasks, except the first to open the file on the local file system, encounter a file that already exists. Poor performance of the `mcp` command will result from a failure to initialize PE to use the user-space protocol over the SP Switch fabric or from contention for file server access; in most cases, the latter may be eliminated by staging the file to a globally-mounted high-performance file system.

A more general command is `mcpscat`, for which the syntax is as follows:

```
mcpscat [-i] InFile_0 [InFile_1] ... TargetDirectory [POEOptions]
```

or

```
mcpscat [-i] -f FileList TargetDirectory [POEOptions]
```

or

```
mcpscat [-i] SourceDirectory TargetDirectory [POEOptions]
```

[12] The reader is reminded that the assignment of nodes to parallel tasks can only be explicitly controlled by using a hostfile in interactive mode. Consecutive jobs that do not make use of hostfiles are unlikely to make use of the same partition; this may explain the absence of files the user may expect to have prepared in a previous job.

The different forms of the command allow lists of files to be specified in different ways; all files in the list must reside on globally-mounted file systems. If the list comprises only one element, this one file is copied to TargetDirectory by all tasks; in all other cases, the files are copied in "round robin" fashion to the nodes executing the parallel tasks: The first file in the list is copied to the TargetDirectory on the node on which task 0 is executing, the second on the same directory on the node on which task 1 is executing, and so on. In the first form, the list of files is determined from the file names (including "wildcard characters" for shell pattern matching if desired) as entered on the command line; in the second form, a list of files (one per line with no wildcard characters) is read from the file FileList, and, in the last syntax, all files in the directory are determined by the `ls` command. The `-i` option activates checking for duplicated or missing files; if such file names are detected, the copy operation is skipped and execution continues with the next file and next parallel task.

The `mcpgath` command performs a "gather" operation, which, in some sense, is the opposite of the `mcpscat` command, which "scatters" files. The syntax is either

```
mcpgath [-ai] InFile_1 [InFile_2] ... TargetDirectory [POEOptions]
```

or

```
mcpgath [-ai] SourceDirectory TargetDirectory [POEOptions]
```

A list of files, such as `InFile_1 [InFiFile_2] ...`, may be explicitly entered or generated using shell pattern-matching; the corresponding files are copied with the same file names to the directory `TargetDirectory` on task 0. Files with these names must be "visible" to all parallel tasks; to prevent overwriting, the `-a` option should be used: its effect is to append a task number to the file name. If the second form of the command is used, all files in the directory `SourceDirectory` are copied to `TargetDirectory` on task 0, and all caveats that apply to the first form of the command continue to apply to the second.

6.3.4 Non-intrusive optimizations: The `vmtune` utility

At the end of Section 6.2.1, “AIX file systems” on page 173, we mentioned that file data are mapped into virtual memory by the AIX operating system. The performance of the AIX virtual memory manager (VMM) component, thus, influences the performance of I/O operations. The VMM is "tunable" in that some of its parameters may be modified to advantage when workload characteristics are well known. The `vmtune` command (in directory `/usr/samples/kernel`) may be used to this end; use of `vmtune` requires

root privileges because incorrect settings can lead to system performance degradation and even failure[13].

Virtual memory is partitioned into 256 MB "segments", which, in turn, are partitioned into "pages" of 4 KB size. Real memory (RAM) is partitioned into "page frames", also of 4 KB size. The VMM component of AIX manages the mapping of data and instructions to pages, the mapping of pages to page frames, the location of pages in RAM, disk paging, and file volumes. A "page fault" occurs when data or instructions required by a CPU are not present in its RAM, and, so, must be retrieved from disk paging or file volumes. We have already noted that disk access is some orders of magnitude slower than RAM access; much effort has been expended in minimizing these delays for the VMM. Well-known strategies, such as caching, coalescing transfers, and speculative transfer (read-ahead, write-behind), are employed to this end. The syntax of the associated command is:

```
vmtune [ParameterFlag ParameterValue] ...
```

that is, `vmtune` followed by zero or more pairs of flags and values; the latter are collected with summary information in Table 28. A list of current parameter values is generated if `vmtune` is issued with no arguments. Parameters revert to their default values when an AIX system is booted. Administrators can conveniently set parameters to other values by making the appropriate /etc/inittab entries.

Table 28. Flags and parameters for the `vmtune` *command*

Flag	Parameter	Explanation; constrains	Range	Dflt
`-b`	`numfsbuf`	File system bufstructs	≥ 64	64
`-B`	`numpbuf`	LVM `pbuf`s	≤ 128	
`-c`	`numclust`	Number of 16 KB clusters processed by write behind	≥ 1	1
`-f`	`minfree`	Minimum number of frames on free list	[8-204800]	
`-F`	`maxfree`	Number of frames on free list at which page "stealing" stops; `maxfree` >= `minfree` + `maxpageahead`	[16-204800]	
`-k`	`npskill`	Start killing processes when free paging-space pages are fewer than this parameter		128

[13] Other system parameters are modified with privileged modes of commands, such as `bosboot`, `chdev`, `chnfs`, `chps`, `mkps`, `nfso`, `no`, `odmadd`, `odmdelete`, `schedtune` (in directory `/usr/samples/kernel`), and `syncd`.

Flag	Parameter	Explanation; constrains	Range	Dflt
-M	maxpin	Maximum percentage of real memory pinned kernel requires 4 MB unpinned	(0-100)	80
-p	minperm	Percentage of real memory page frames below which file pages are protected from 'stealing'	[5,100)	
-P	maxperm	Percentage of real memory pages frames above which only file pages are 'stolen'	[5,100)	
-r	minpageahead	Number of pages with which sequential read ahead starts	$0,1<2^n\leq4096$	
-R	maxpage ahead	Maximum number of pages read ahead; maxpageahead >= minpageahead	$0,1\leq2^n\leq4096$	
-w	pswarn	Number of free pages at which SIGDANGER sent to processes		512

Some defaults are configuration-dependent and are not listed above; the user may determine them by issuing the vmtune command (with no options) immediately following system boot (provided there are no vmtune entries in /etc/inittab). A page is said to be "stolen" when it is transferred from real memory to paging disk storage in order to free page frames for reuse. Additional information on these parameters may be found in the *AIX Version 3.2 and V4 Performance Monitoring and Tuning Guide*, SC23-2365.

More efficient data transfer to and from high-performance disk subsystems (striped logical volumes or disk arrays) is generally obtained by increasing the numfsbuf[14] and numclust parameters .

Speculative read-ahead for a minimum of minpageahead file pages is triggered when sequential file access is deemed to have been detected by the VMM; read-ahead continues for maxpageahead file pages. The performance of applications that involve extensive sequential disk-RAM data transfers may be improved by increasing the minpageahead and maxpageahead parameters. It is suggested that minpageahead be set to twice the number of disk drives and maxpageahead to 16 times the number of disk drives; a concomitant increase of maxfree may be required; in some instances, it may also be necessary to increase numpbuf.

[14] struct buf bufstruct stores a device request; see /usr/include/sys/buf.h.

The diagnostic information obtained from the iostat utility should be supplemented by that from the vmstat utility in estimating suitable values for the parameters shown in Table 28 on page 186.

6.3.5 Non-intrusive optimizations: Reorganizing a file system

File space in AIX is allocated entirely in blocks of 4096 bytes when the size of a file equals or exceeds 32 KB, in blocks of 4096 bytes plus one or more "fragments" when filesize is less than 32 KB but greater than 4096 bytes, and entirely in fragments when the filesize is less than 4096 bytes. The default fragment size is 4096 bytes, but it can also be set to 512 bytes, 1024 bytes, or 2048 bytes at the time a file system is created. Although the utilization of storage capacity is generally more efficient when fragment sizes are small[15], 'space fragmentation' (the scattering of logically-contiguous blocks across physical disk space) is more likely to occur when the fragment size is small. Access to a fragmented file is generally slower because of the increased activity of the electromechanical components of disk drives. File allocations can be rendered more contiguous by the execution of the `defragfs` command by a system administrator; the syntax of the command is:

```
defragfs [-q|-r] [Device|Filesystem]
```

The presence of either of the mutually-exclusive options `-q` and `-r` precludes the actual defragmentation of the named `Device` or `Filesystem` in favor of generating reports on current statistics (`-q`) and both current and potentially-reorganized file system statistics.

The frequency of JFS logging operations or access to such log information is sometimes comparable to that of user data storage operations; a great deal of time may then be expended in the constant repositioning of disk arms between tracks associated with user data and those storing JFS logs. In such cases, I/O performance can be significantly improved by placing the JFS log on a logical volume that resides on drives different than those storing user data.

By way of example, to create a JFS logging device, /dev/jfslognew, on a newly-created volume group, `newvg`, first create a logical volume of type `jfslog`:

```
mklv -t jfslog -y jfslognew newvg 1
```

In this case, the size of the logical volume, `jfslognew`, is 1 logical partition; the logical volume is formatted by executing `logform /dev/jfslognew`, after which it is ready to be used as a JFS log device. The association between the

[15] A well known exception is a file system so highly fragmented that the paucity of blocks of size 4096 bytes prevents the storage of files in excess of 4096 bytes.

new JFS, `/fastfs`, and log device, `/dev/jfslognew`, is established by issuing the appropriate form of the `chfs` command:

```
chfs -a log=/dev/jfslognew /fastfs
```

6.3.6 Non-intrusive optimizations: Reorganizing an LV or an LVG

Fragmentation (see the preceding section) can also occur at the logical volume level; this can be diagnosed using

```
lslv -p PhysicalVolume [LogicalVolume]
```

to generate an allocation map for PhysicalVolume or, more specifically, if it is desired for its comprised LogicalVolume and is rectified using

```
reorgvg [-i] VolumeGroup [LogicalVolume ...]
```

6.3.7 Non-intrusive optimizations: The `sync` daemon interval

File data stored in I/O buffers in RAM are periodically written to disk to maintain a high degree of consistency between RAM and disk copies. The `sync` system call queues modified block buffers for such writing but does not force the write in the manner of fsync. The sync daemon issues sync calls with a period of `Interval` (typically 60 seconds) established in /sbin/rc.boot with a record of the form

```
nohup /usr/sbin/syncd Interval > /dev/null 2>&1 &
```

The `rc.boot` file can only be modified by a user with root privileges.

An AIX sync system call consumes a small amount of CPU effort and always leads to some disk activity; write-behind is, at least, partially defeated by its hastening of data transfers to disk; it is, therefore, rarely beneficial to decrease `Interval`, and the only situation in which this may yield some performance benefit is when fsync is called very frequently in the course of a system's operation.

6.3.8 Non-intrusive optimizations: Tuning the SCSI device driver

The SCSI device driver can coalesce disk I/O requests to decrease latency overheads [*n* in Eq. (6.3)]. By default, the largest request that can be assembled is 64 KB (`0x10000` in hexadecimal notation). The performance of I/O operations to and from striped logical volumes and disk arrays may be improved by increasing the value of max_coalesce in the PdAt (predefined attribute object class) stanza in the ODM database. As an illustration, the procedure to double the value of max_coalesce is as follows:

Recover the current version of the stanza (if it exists) into the file foo:

```
odmget -q "uniquetype=disk/scsi/osdisk AND \
attribute=max_coalesce" PdAt > foo
```

and modify all occurrences of `0x10000` to `0x20000` in foo; otherwise, create `foo` from scratch to contain the following records

```
PdAt:
      uniquetype = "disk/scsi/osdisk"
      attribute = "max_coalesce"
      deflt = "0x20000"
      values = "0x20000"
      width = ""
      type = "R"
      generic = ""
      rep = "n"
      nls_index = 0
```

Now, replace the old stanza

```
odmdelete -o PdAt \
  -q "uniquetype=disk/scsi/osdisk AND attribute=max_coalesce"
odmadd < foo
```

and rebuild the kernel and reboot

```
bosboot -a -d hdisk0
shutdown -Fr
```

Local disk subsystems of the RS/6000 SP are frequently SCSI subsystems.

6.3.9 Characterizing I/O for intrusive optimization

Program packages of any complexity tend to exhibit unexpected behaviors in the context of the breakdown of time among their component subprograms; this is especially true when applications are employed in the new ways that more powerful processors permit. Therefore, an indispensable step in optimization is the characterization of an application's behavior by empirical means. Profiling utilities, such as prof[16], tprof[17], gprof[18], and xprofiler[19]

[16] This utility provides a "flat profile", that is, a report of user CPU time expended in each subprogram traversed in the course of an application's execution. This information is obtained by linking with the `-p` option. The number of entries into each subprogram is also available when the `-p` option is added to the compiler invocation. Many, but not all, system libraries have counterparts that are enabled in this manner. The definitive reference for the AIX `prof` utility is the *AIX Commands Reference*.

[17] This tool is capable of providing a "flat line-by-line profile" based on user CPU time in addition to "flat profile" information, such as that provided by the `prof` tool. Applications must be compiled and linked with the `-g` compiler option and must be executed "under" the `tprof` utility: for instance, `myprog < myinput > myoutput` would be executed under `tprof` with `tprof myprog < myinput > myoutput`. The assignment of "ticks" (1 tick =10^{-2} s of user CPU time) to lines is only approximate when optimizations that entail code movement are present. Use of `tprof` requires root privileges in certain phases of its operation, and system administration policies may, therefore, preclude its general use. The definitive reference for the AIX `tprof` tool is the *AIX Commands Reference*.

provide information about the expenditure of "user" CPU time by an application, not the "real" time spent waiting for I/O requests to complete.

At the present time, the characterization of I/O must be accomplished by a more tedious method, namely, the insertion of timer calls surrounding each I/O call and the subsequent interpretation of the timing information thus obtained[20]. This process may be facilitated to some extent by the use of a simple library `librtp.a` (RTP: real-time profile); the use of this library with Fortran applications is described here; the source will be made available at the URL cited in the References section of this chapter. Four routines comprise the interface to the library:

- SUBROUTINE RTP_INITIALIZE() must be invoked prior to any other RTP call;
- SUBROUTINE RTP_START_CLOCK(IRTPA) is invoked to start or restart the clock associated with the real-time accumulator IRTPA (this is a sequence number 1, 2, 3, and so on, that is, a Fortran INTEGER; up to 1024 accumulators are available, and this can be increased by suitably modifying rtp.include.);
- SUBROUTINE RTP_STOP_CLOCK(IRTPA) is called to stop the clock associated with the real-time accumulator IRTPA; the accumulator is updated when the latter subprogram is executed;
- SUBROUTINE RTP_FINALIZE() is called to dump all profiling data including the elapsed time between the invocations of RTP_INITIALIZE() and RTP_FINALIZE() to stdout.

All elapsed times are measured in the RTP package using the XL Fortran timer utility, `RTC()`, which is expected to have a resolution of better than 1 μs.

The burdens of keeping track of the accumulator tags IRTPA so that the wrong tag is not accidentally used where it is not intended as well as the locations of the matched calls to RTP_START_CLOCK(IRTPA) and

[18] This utility is capable of providing a "call-tree profile" of an application in addition to "flat profile" information, such as that generated by the `prof` tool: A subprogram may be invoked by more than one subprogram constituting an application; gprof separates the user CPU time expended in a subprogram by its calling subprogram; a by-product of this sorting procedure is the call tree for the entire execution; this information is frequently useful in developing parallelization schemes. Complete information is generated by gprof only when the -pg option is included in compilation as well as link operations. The definitive reference for the AIX gprof utility is the *AIX Commands Reference*.

[19] The author's prejudice is that this is the tool of choice for application optimization based on user CPU time: the graphical user interface of xprofiler provides a complete and readily-assimilated visual representation of the call-tree profile (as would be generated by the gprof utility); line-by-line profile information (as would be generated by the `tprof` tool) is available through a simple set of menus. Applications should be compiled and linked with both the -g and -pg options to exploit the full functionality of xprofiler. The definitive reference for the xprofiler tool is *PE Operation and Use Vol. 2, Part 2 — Profiling*.

[20] The forthcoming availability of the Dynamic Probe Class Library (DPCL) and tools based upon DPCL is likely to improve this situation substantially. Potential users of DPCL should be aware of the -qdpcl compiler option.

STOP_CLOCK(IRTPA), which must be known to interpret the dump generated by RTP_FINALIZE(), lie with the user.

The explicit instrumentation of every subprogram that constitutes an application to determine I/O wait time "hot spots" may be difficult; it is also unnecessary: An iterative "winnowing" procedure may be adopted instead: The main program is instrumented to determine which subprograms called at this level manifest the greatest discrepancy between real time and CPU time (CPU time is easily obtained with a tool, such as gprof); these subprograms become candidates for instrumentation in the next step, and so on through a few branches of the call tree. Finally, when the discrepancy is isolated to a few subprograms, greater "resolution" is obtained by instrumenting sectors of these subprograms (typically blocks of code). The following hypothetical example illustrates the procedure:

Using a fairly obvious notation where indentation is proportional to calling "depth", let the call tree be

```
MAIN
  MYSUB1
    MYSUB11
    MYSUB12
  MYSUB2
    MYSUB21
    MYSUB22
      MYSUB221
      MYSUB222
    MYSUB23
    MYSUB24
      MYSUB241
      MYSUB242
      MYSUB243
    MYSUB25
  MYSUB3
    MYSUB31
  MYSUB4
    MYSUB41
```

A typical code fragment that includes instrumentation in `MAIN` would be as follows:

```
PROGRAM MAIN
...
CALL RTP_INITIALIZE ()
...
CALL RTP_START_CLOCK (1)
```

```
CALL MYSUB1 (ARG1, ARG2)
CALL RTP_STOP_CLOCK (1)
...
CALL RTP_START_CLOCK (2)
CALL MYSUB2 (ARG3)
CALL RTP_STOP_CLOCK (2)
...
CALL RTP_START_CLOCK (3)
CALL MYSUB3 (ARG4, ARG5, ARG6)
CALL RTP_STOP_CLOCK (3)
...
CALL RTP_START_CLOCK (4)
CALL MYSUB4 ()
CALL RTP_STOP_CLOCK (4)
...
CALL RTP_START_CLOCK (5)
CALL MYSUB4 ()
CALL RTP_STOP_CLOCK (5)
...
CALL RTP_FINALIZE ()
END
```

Note that `MYSUB4` is called twice; accumulator 4 tracks the real time in one case and 5 tracks the real time in another; the aggregate corresponding user time information may be inferred from a profile generated by gprof or xprofiler. The dump generated by `RTP_FINALIZE()` will indicate which call of `MYSUB1`, `MYSUB2`, `MYSUB3`, and `MYSUB4` contains the hotspots and, thus, which of these subprograms should be instrumented with calls to `RTP_START_CLOCK` and `RTP_STOP_CLOCK`. If, for instance, it were determined that the hotspot were in `MYSUB2`, the calls to the clock routines would surround calls to `MYSUB21`, `MYSUB22`, `MYSUB23`, `MYSUB24`, and `MYSUB25` in `MYSUB2`; the hotspot could then be traced to the next level of ramification.

6.3.10 Intrusive optimizations: Exploiting high-performance FS

Local file systems have been defined in Section 6.3, “I/O optimization” on page 177. Programs that use local file systems require modification if a parallel task should require I/O to a file on a local file system that is "off node". The following are some options that are available in such situations:

- Modify the underlying algorithm so that off-node I/O is not required.
- Reorganize I/O so that only shared data resides in a file system that is mounted on all nodes; employ a high-performance file system for the shared data.

- Program the intertask communication that is required to access off-node I/O: Currently, LAPI provides facilities for one-sided communication, which may be preferable to message passing in the present context; soon, the IBM implementation of MPI-2 will provide facilities for one-sided communication as well.

As described in Appendix 6.3.2, “Non-intrusive optimizations exploiting high-performance FS” on page 182, the alignment of I/O buffers on page boundaries improves performance for striped logical volumes. Explicit page alignment of an I/O buffer of length `nbytes` may be achieved by invoking the following function:

```
#include <stdlib.h>
#include <errno.h>
char *allociobuffer (int nbytes, void **lstart)
{
  char *iobuffer;
  if (!(*lstart = malloc (nbytes + 4096)));
    perror ("allociobuf: malloc");
  iobuffer = (char *)(((int) *lstart + 4096) & ~0xfff);
  return iobuffer;
}
```

The declaration of lstart in the argument list of allociobuffer permits the latter to return the value of lstart to a calling Fortran subprogram: Data are passed by reference in Fortran, and the calling program would, thus, be passing the address of a pointer, which is the address of an address. A counterpart for allociobuffer to release storage allocated to an I/O buffer is

```
#include <stdlib.h>
#include <sys/shm.h>
#include <errno.h>
void freeoibuffer (int nbytes, void **lstart)
{
  free (*lstart);
  if (disclaim (*lstart, nbytes+4096, ZERO_MEM))
    perror ("freeiobuffer: disclaim");
}
```

The following code fragment illustrates the invocation of allociobuffer and freeiobuffer from an XL Fortran application:

```
...
INTEGER LBUFF
PARAMETER (LBUFF = 2000)
DOUBLE PRECISION BUFER
POINTER (PIOBUFFER,BUFFER(1:LBUFF))
```

```
INTEGER LSTART, NBYTES
INTEGER ALLOCIOBUFFER
EXTERNAL ALLOCIOBUFFER
...
NBYTES = 8*LBUFF
PIOBUFFER = ALLOCIOBUFFER (%VAL (NBYTES), LSTART)
...
CALL FREEIOBUFFER (%VAL (NBYTES), LSTART)
...
```

Note that `ALLOCIOBUFFER` should be declared to be INTEGER*8 if 64-bit addressing is used. The exploitation of MPI-IO is an intrusive I/O optimization; this topic is treated in Section 6.5, “MPI-IO” on page 214.

6.3.11 Intrusive optimizations: Obviating I/O

One strategy for the reduction of I/O delays is the elimination of some I/O. In terms of Eq. (6.3), this is the simultaneous reduction of *n* and *V*; a side-effect is the elimination of some contention: In terms of Equations (6.1) and (6.2), this is the reduction of *N*. A great variety of techniques may be brought to bear in these contexts; only a few are touched upon here.

Programs have sometimes been developed on machines with severely limited real memory (RAM) but are eventually employed in production use on better-endowed systems. This is especially true of older applications, that is, those developed before *any* computer had real memory comparable to a modern system; Such "legacy" programs can sometimes be relatively easily rewritten so that most, if not all, frequently-accessed data remain resident in real memory for the entire course of execution and, so, do not suffer repeated transfer between real memory and disk. Some care must be exercised in the process of revision: The AIX operating system supports virtual memory that may be well in excess of real memory, especially in 64-bit addressing mode; so, it is relatively easy for a programmer to provoke disk activity associated with paging in the attempt to eliminate disk activity associated with data transfer under his or her direct control.

Algorithmic improvements can provide substantial reductions in I/O traffic: For instance, an iterative algorithm with improved convergence properties will typically reduce the number of iterations and, therefore, the number of associated transfers of data to or from disk; similarly, an algorithm that sweeps through disk-based data only once to perform setup for many related computational steps is likely to be superior to one that performs a sweep for the setup of every step. There are rapidly-growing bodies of literature in the computational sciences and engineering disciplines; application developers who keep abreast of advances in their fields can often reap very substantial

benefits from the straightforward implementation of new algorithms or minor modifications thereof.

Improved data structures can reduce the volume of data transferred between real memory and disk storage by permitting a greater fraction of frequently-accessed data to remain in memory or obviating storage for data of known or repeated value or those that are never accessed. It is, however, quite generally true that the use of data structures that are more efficient in terms of demands placed on memory size also increases the complexity of a program and incurs additional processing. Dimensions of large arrays should be tailored to the size required by a problem; this is, typically, only an important consideration in programs originally coded to the FORTRAN 77 standard, which provides no facility for dynamic storage allocation; in production code of this type, which generally cannot be repeatedly redimensioned and rebuilt, the dynamic allocation facilities provided by the Fortran 90 and Fortran 95 standards or the INTEGER POINTER extension, all supported by XL Fortran, may be employed[21]; a common by-product of this type of revision is reduced stride, which frequently yields improved performance because of improved data locality in the memory hierarchy. Separate versions of an application should be maintained if problems are very diverse and can have similarly diverse resource requirements.

Contention between multiple parallel tasks performing I/O operations to one set of files may be reduced or eliminated in many cases simply by relegating the processing of different I/O streams to different tasks; when possible, an additional advantage accrues: All I/O operations can be performed using local file systems, which are, ideally, on different nodes. The most common optimization of this type is to relegate all I/O processing for stdin and stdout to the root task (MPI task 0), which performs I/O to these streams from a local file system.

6.3.12 Formatted and unformatted I/O

All data is represented as strings of binary digits in computers[22], but must be translated into some other form[23] to be readily intelligible. The translation of numerical data from a machine's internal representation[24] into a string of base-10 digits, separated, if necessary, by delimiters for the decimal and exponent parts, requires processing and generally increases the volume of data: For instance, an eight-byte Fortran DOUBLE PRECISION or C (or C++)

[21] Users of the INTEGER POINTER extension should be aware of the -qddim compiler option.

[22] This is, of course, already an abstraction.

[23] Anothre form, such as complex images, juxtapositions of simple images, such as the letters, numbers, and punctuation that constitute this page, sounds, or, even, motion.

[24] An excellent discussion of the IEEE floating-point standard can be found in *Computer Arithmetic by David Goldberg*, and *Appendix* A of *Computer Architecture: A Quantitative Approach*.

double datum requires up to twenty-two characters for its accurate representation in "scientific notation": The sign `-`, a decimal point (or a comma), fifteen decimal digits for the numerical portion of the mantissa, one of the characters `D`, `E`, or `e`, the sign `+` or `-`, and three decimal digits for the exponent. Each character requires a byte of storage so that the factor for the increase in storage is 22/8 = 2.75. The increase in processing time required to convert data from the internal representation to the character representation is generally insignificant compared with the increase in time associated with the greater volume of data transferred between disk storage and memory.

Numerical data expressed in "scientific notation" or as a simple string of decimal digits as would be appropriate to the integer datatypes is said to be "formatted"; their binary counterparts are (perhaps misleadingly) referred to as "unformatted". In view of the increased time associated with the reading or writing of formatted data, it is advisable to employ formatted I/O as rarely as possible. Quite generally, data that will not be examined by typical users of an application should remain unformatted.

Formatting is often adopted as an expedient for the generation of "portable" data, that is, data that is generated on one machine and then transferred to another for further processing. In such instances, it may, however, be preferable to convert all data to a standard, such as the eXternal Data Representation (XDR), which is likely to provide the necessary portability yet also retain a relatively compact representation[25].

Formatted I/O in Fortran is most efficient when the format specification can be determined at compile time; thus, the `WRITE` statement in the fragment

```
      WRITE (*, 200) I1, R2, R3
  200 FORMAT (I10, F10.0, F2.2)
```

is likely to execute faster than the `WRITE` statement in the fragment

```
      CHARACTER*18 FMT
      ...
      FMT = '(I10, F10.0, F2.2)'
      WRITE (*, FMT) I1, R2, R3
```

which, in turn, is likely to execute faster than the `WRITE` statement in the fragment

```
      WRITE (*, 200) I1, R2, R3
  200 FORMAT (I<2+J>, F10.0, F2.2)
```

[25] The reader is referred to *Communications Programming Concepts* for a detailed presentation of IBM facilities for exploiting XDR; only a C language binding is available for this library; Fortran programmers may wish to consult the chapter on interlanguage calls in the *XL Fortran User's Guide*.

which employs a frequently convenient XL extension to the Fortran language standards. The simplification of FORMAT statements into more compact forms can also improve performance: For instance,

```
FORMAT (3I6, 2F3.2)
```

is likely to execute faster than

```
FORMAT (3(1X,I5),2(1X,F2.2))
```

which, in turn, is faster than

```
FORMAT (1X,I5,1X,I5,1X,I5,1X,F2.2,1X,F2.2)
```

It is important to note, however, that only the last two of the above three FORMAT statements are strictly equivalent in the output they generate: The ranges of data that will generate strings of asterisks (*...) indicating field overflow are different for the descriptors I5 and I6 and the descriptors F2.2 and F3.2. It is more efficient to incorporate strings into FORMAT statements than to store them as CHARACTER constants and print the latter out: for instance, the form

```
      WRITE (*,300) INDEX1, INDEX2
300 FORMAT ('Index_1 = ',1I3,', Index_2 = ',1I3)
```

should be preferred to the form

```
      CHARACTER*10 SI1
      CHARACTER*12 SI2
      ...
      SI1 = 'Index_1 = '
      SI2 = ', Index_2 = '
      WRITE (*,300) SI1, INDEX1, SI2, INDEX2
300 FORMAT (A10,1I3,A12,1I3)
```

when the performance of the associated sector of code is an important consideration.

Fortran NAMELIST processing incurs a significant per-datum processing overhead and should be avoided in performance-sensitive I/O.

In the C language, input and output operations and conversions between various representations of numerical data are performed by explicitly invoking library subprograms. The standard C library includes functions for the formatting of data that are for applications I/O as well as interfaces for the conversion of numerical data between internal representations, such as int, float, and double, and character strings corresponding to other representations, which are controlled by user-specified control strings. An

overhead is associated with every function invocation; additional processing overheads are associated with the parsing of control strings.

Optimization strategies that follow immediately from these observations are that I/O functions should be invoked as infrequently as possible, and that control specifications should be as simple as possible. Testing reveals that the sprintf function, as implemented in recent versions of the C library, requires less processing effort for the conversion of double data into their full precision string counterparts than the gcvt utility in AIX 4.3.2; the opposite conversion (from string to double) is faster when the atof function is used rather than the more complex and versatile sscanf function.

6.3.13 I/O Blocksize

The efficiency of I/O operations generally increases with the size of the quantum of data transfer: I/O bandwidths are actually an approximately monotonic function of the size of the quantum (this is not made explicit in Eq. 6.3) and closely approach the asymptotic value even when this quantum is relatively small; further, the latency overhead is incurred every time a synchronous I/O request is made; finally, in Fortran I/O, 32-bit record-length specifiers precede and follow every record written in sequential mode[26]. An obvious corollary of these observations is that I/O requests should be consolidated to the extent possible; for instance, the Fortran fragment

```
      COMPLEX*16 TEGRAL
      INTEGER*1 IA, IB, IC, ID
      ...
      DO I = 1,MANY
*
*  Calculate IA, IB, IC, ID, and TEGRAL
*
        ...
        WRITE (OSTREAM) IA, IB, IC, ID, TEGRAL
      ENDDO
```

causes the file attached to the output stream, `OSTREAM`, to store `MANY` records of 28 bytes length, only 20 bytes (1\times `COMPLEX*16`+4 \times `INTEGER*1`) of which are user data, incurring a storage volume overhead of 8/20 = 40%; rewriting the application to employ larger buffers in the form

```
      INTEGER LBUFF
      PARAMETER (LBUFF = 2000)
      COMPLEX*16 TEGRAL, TEGRALB(LBUFF)
      INTEGER*1 IA, IB, IC, ID, NBUFF,
     :          IAB(LBUFF), IBB(LBUFF), ICB(LBUFF), IDB(LBUFF)
```

[26] The default value of the uwidth option (otherwise, specified using the XL Fortran `XLRTEOPTS` environment variable or the `SETRTEOPTS` service procedure) is 32, but it may be set to 64 to permit record lengths in excess of 2 GB.

```
      ...
      NBUFF = 0
      DO I = 1,MANY
*
*  Calculate IA, IB, IC, ID, and TEGRAL
*
        ...
*
*  Buffer the output
*
        NBUFF = NBUFF+1
        IAB(NBUFF) = IA
        IBB(NBUFF) = IB
        ICB(LBUFF) = IC
        IDB(NBUFF) = ID
        TEGRALB(NBUFF) = TEGRAL
*
*  Flush the buffer if it is full
*
        IF (NBUFF .EQ. LBUFF) THEN
          WRITE (OSTREAM) IAB, IBB, ICB, IDB, TEGRALB
          NBUFF = 0
        ENDIF
*
      ENDDO
*
*  Flush any unwritten buffer data to disk
*
      IF (NBUFF .NE. 0) THEN
        WRITE (OSTREAM) (IAB(I), I = 1,NBUFF),
     :                  (IBB(I), I = 1,NBUFF),
     :                  (ICB(I), I = 1,NBUFF),
     :                  (IDB(I), I = 1,NBUFF),
     :                  (TEGRALB(I), I = 1,NBUFF)
        NBUFF = 0
      ENDIF
```

is much more efficient in spite of the additional bookkeeping overhead associated with ensuring that buffers `IAB`, `IBB`, `ICB`, `IDB`, and `TEGRALB` are flushed appropriately: Now, the storage volume overhead is approximately 8/(20*2000) = 0.02 percent and limited only by the storage made available for the I/O buffers through the value of PARAMETER LBUFF; indeed, if LBUFF can be set to MANY, output to disk is most efficient in the form implied by the following fragment:

```
      COMPLEX*16 TEGRAL(MANY)
```

```
      INTEGER*1 IA(MANY), IB(MANY), IC(MANY), ID(MANY)
      ...
      DO I = 1,MANY
*
*  Load arrays IA, IB, IC, ID, and TEGRAL
*
        ...
      ENDDO
*
      WRITE (OSTREAM) IA, IB, IC, ID, TEGRAL
```

No extra data, such as the record length specifiers in sequential FortranI/O, are written to disk files by the I/O functions of the standard C library. As in Fortran, it is much more efficient to perform I/O operations with large blocks of data. A very small penalty is paid when block sizes are much larger than the optimal size for the read, write, fread and fwrite functions

6.3.14 Intrusive optimizations: Memory-mapped I/O

A disk file or a portion thereof can be mapped into memory using the AIX mmap function[27]. Once mapped in this fashion, file data may be manipulated directly using their memory addresses, thus, entirely bypassing I/O library subprograms, such as read, write, etc., and their associated data copying overheads.

Memory-mapped I/O is limited by the virtual address space that can be made available for file data. We will now discuss this in some detail.

- 32-bit address space model - The 4 GB 32-bit address space of AIX is divided into 16 segments, `0x0` through `x0F`, each 256 MB in size. Segment `0x2`, beginning at virtual address 0x20000000, is the *process private segment*, and stores most process-specific information including user data, user stack, kernel stack, and user block. In the *large address space model*[28], the segments 0x3 through 0xA can be made available to user data. Memory-mapped file data can occupy the ten segments 0x3 through 0xC for a total of 2.5 GB of data in early versions of AIX; in AIX 4.2.1 and later, segment 0xE is also available for this purpose allowing a total of 2.75 GB of mapped data.
- *64-bit address space model* - Now, 36 address bits (compared with four in the 32-bit model) are used to select segments permitting, in principle, segments 0x00000000 through 0xFFFFFFFF (a total of 68,719,476,736 segments); the prohibitive cost of very large amounts of RAM and disk

[27] The current version of the GPFS file system does not support memory-mapped files.
[28] This is activated when the o_maxdata field is set in the XCOFF header of the program; one way to set this field is with the -bmaxdata option of the `ld` command.

storage constrain practically achievable address spaces to much smaller sizes. Segments 0x07000000 through 0x07FFFFFF (16,777,216 segments and 4096 TB of address space) are available for variable-location mappings (the only type discussed in this subsection).

A portion of size `len` bytes beginning at offset `off` bytes of a data file with descriptor `filedes` may be mapped into a location in virtual memory decided by the system with a call with the following synopsis:

```
#include <sys/mman.h>
void *mmap(0, size_t len, int prot, int flags, int filedes, int off);
```

The return value of the call is the address of the first byte of the memory region into which the file is mapped. Depending on the access mode of the file, prot is one of PROT_READ, PROT_WRITE or their bitwise inclusive "or": PROT_READ|PROT_WRITE. Setting flags to MAP_SHARED associates the region of memory with the disk file such that store operations are reflected in the file, whereas setting it to MAP_PRIVATE restricts the modification to a copy of the file in memory.

The storage of data into memory locations that are beyond the end of the file associated with the descriptor specified to the mmap call will lead to the generation of a SIGBUS signal; one method of avoiding application failure associated with this signal is to ensure that the end of file is suitably relocated using, for instance, the lseek call.

A region mapped by invoking mmap is implicitly unmapped when the calling process terminates; explicit unmapping may be performed by invoking

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

The `addr` argument should be set to the return value of the call to `mmap`, while the value of `len` should be the same as that in the invocation of `mmap`.

Additional information on virtual memory and the `mmap` and `shmat` families of calls is available in *:* and *AIX V4.3 General Programming Concepts; Writing and Debugging Programs*, SC23-4128.

6.3.15 Intrusive optimizations: Asynchronous I/O

The execution of standard Fortran I/O statements, such as READ, WRITE, and PRINT statements, ISO C library I/O functions, such as fread, fscanf, fwrite, and fprintf, and ISO C++ stream operations is "synchronous": execution of the remainder of an application is blocked until the I/O operation completes. An important technique for "hiding" the delay associated with I/O

operations is to exploit the parallelism permitted by "asynchrony": Execution of an application continues after I/O requests are queued (not completed) so that I/O operations can execute simultaneously with computation to the extent permitted by the algorithm and system.

Asynchronous I/O must be explicitly enabled: A sequence of messages, such as

```
XL Fortran (Asynchronous I/O Initialization):
  AIX asynchronous I/O is disabled.
IOT/Abort trap(coredump)
```

or

```
Could not load program asyncio
Symbol kaio_rdwr in ksh is undefined
Symbol listio in ksh is undefined
Symbol acancel in ksh is undefined
Symbol iosuspend in ksh is undefined
Error was: Exec format error
```

indicates that such intercession by a system administrator is necessary; the required commands are

```
chdev -l aio0 -P -a autoconfig='available'
mkdev -l aio0
```

The optimal configuration of asynchronous I/O is governed to a large extent by the number of "servers", that is, kernel processes that service asynchronous I/O requests[29]. The default maximum and minimum number of servers, are 1 and 10, which is equivalent to issuing

```
chdev -l aio0 -a minservers='1' -a maxservers='10'
```

These settings are likely to be inadequate for systems in which heavy use is made of asynchronous I/O; the symptom of such a deficiency is long intervals between the queueing of a request and its servicing; the delay is exacerbated by the reduced opportunity for the operating system's seek optimization algorithm; in such situations it is recommended that the maximum number of servers be set to 10 times the number of disk drives accessed asynchronously and that the minimum number of servers be set to half the maximum number of servers.

Fortran asynchronous I/O was first made available on RS/6000 platforms as an extension to XL Fortran Version 5 Release 1. We introduce its use by improving the performance of the timed portion of a simple (albeit contrived) example that uses synchronous direct I/O:

[29] Asynchronous I/O against raw logical volumes do not use kproc server processes.

```
      PROGRAM SYNCIO
*
      IMPLICIT NONE
*
      INCLUDE 'param.h'
*
      REAL*8 ARRAYI(NELEM), ARRAYO(NELEM),
     :       RTC, TREAD, TSTART, TWRITE                    !
      INTEGER IELEM, IREC, ISTREAM, OSTREAM
      EXTERNAL RTC
*
      ISTREAM = 11
      OPEN (ISTREAM, ACCESS = 'DIRECT', RECL = LREC,
     :      ASYNCH = 'NO', FORM = 'UNFORMATTED')
      OSTREAM = 12
      OPEN (OSTREAM, ACCESS = 'DIRECT', RECL = LREC,
     :      ASYNCH = 'NO', FORM = 'UNFORMATTED')
*
      TREAD = 0.0D 00                                      !
      TWRITE = 0.0D 00                                     !
*
      DO IREC = 1,NREC
*
        TSTART = RTC ()                                    !
        READ (ISTREAM, REC = IREC) ARRAYI
        TREAD = TREAD+RTC ()-TSTART                        !
*
        DO IELEM = 1,NELEM
          ARRAYO(IELEM) = GAMMA (ARRAYI(IELEM))
        ENDDO
*
        TSTART = RTC ()
        WRITE (OSTREAM, REC = IREC) ARRAYO
        TWRITE = TWRITE+RTC ()-TSTART                      !
*
      ENDDO
*
      PRINT *, 'READ time: ',TREAD                         !
      PRINT *, 'WRITE time:',TWRITE                        !
      PRINT *, '  Total:   ',TREAD+TWRITE                  !
*
      END
```

PROGRAM SYNCIO reads NREC records of length LREC bytes from stream ISTREAM (disk file fort.11 by default), performs numerically-intensive operations on the NELEM REAL*8 data items in each of these records, and

writes the results to stream OSTREAM (disk file fort.12 by default); file `param.h`, which prescribes `NREC` and `LREC`, consists of the following two records

```
INTEGER LREC, NELEM, NREC
PARAMETER (LREC = 2**20, NELEM = LREC/8, NREC = 2**10)
```

Calls to the XL Fortran high-resolution real-time clock (`RTC()`) utility surround the READ and WRITE statements to obtain profiling information for I/O operations. (The invocations of RTC() and the XL Fortran intrinsic GAMMA(X), which computes the (complete) real gamma function, $\Gamma(x)$, compromise the portability of PROGRAM SYNCIO.) The input file for the PROGRAM SYNCIO is generated by PROGRAM SETUP:

```
      PROGRAM SETUP
*
      IMPLICIT NONE
*
      INCLUDE 'param.h'
*
      REAL*8 ARRAY(NELEM)
      INTEGER IELEM, IREC, ISTREAM
*
      ISTREAM = 11
      OPEN (ISTREAM, ACCESS = 'DIRECT', RECL = LREC,
     :      ASYNCH = 'NO', FORM = 'UNFORMATTED')
*
      DO IREC = 1,NRE
        DO IELEM = 1,NELEM
          ARRAY(IELEM) = LOG (0.5D 00*DBLE (IREC+IELEM))
        ENDDO
        WRITE (ISTREAM, REC = IREC) ARRAY
      ENDDO
*
      END
```

The sequential variants of PROGRAMS SYNCIO and SETUP are obtained by modifying ACCESS = 'DIRECT', RECL = LREC to ACCESS = 'SEQUENTIAL', in the OPEN statements and eliminating REC = IREC from the READ and WRITE statements.

The user times for the execution of both variants of PROGRAM SYNCIO are significantly larger than the total of the sys, READ, and WRITE times; this is the gross characteristic of the necessary (but insufficient) conditions for the successful exploitation of asynchronous I/O; `PROGRAM SYNCIO` has, of course, deliberately been designed to exhibit the sufficient condition: adequate non-I/O processing interleaved between I/O requests.

The scheme we select for the exploitation of asynchronous I/O is applied to read and write operations in the same manner: The set of all READ or WRITE requests is split into even and odd components based on the order in which they are issued; an even component is prepared (computed, buffered, and queued) while the preceding odd component is in transit from (or to) an I/O device and vice versa, that is, with odd and even interchanged. PROGRAM SYNCIO is rewritten to yield PROGRAM ASYNCIO with a view to maintaining the numerically-intensive portion, which is, invariably, more complex than the I/O sector in a "real" scientific application, in unchanged form; portability is not a primary concern because no extant Fortran standard supports asynchronous I/O.

```
      PROGRAM ASYNCIO
*
      IMPLICIT NONE
*
      INCLUDE 'param.h'
*
      REAL*8 ARRAYI, ARRAYO, ARIE, ARIO, AROE, AROO,
     :       RTC, TSTART, T(8)                              !
       POINTER (PARI,ARRAYI(1)), (PARO,ARRAYO(1)),
     :        (PARIE,ARIE(1:NELEM)), (PARIO,ARIO(1:NELEM)),
     :        (PAROE,AROE(1:NELEM)), (PAROO,AROO(1:NELEM))
       INTEGER I,IDREVN, IDRODD, IDWEVN, IDWODD,
     :        IELEM, IREC, ISTREAM, MALLOC, NBYTES, OSTREAM
       EXTERNAL MALLOC, RTC
*
       NBYTES = 8*NELEM
       PARIE = MALLOC (%VAL (NBYTES))
       PARIO = MALLOC (%VAL (NBYTES))
       PAROE = MALLOC (%VAL (NBYTES))
       PAROO = MALLOC (%VAL (NBYTES))
*
       ISTREAM = 11
       OPEN (ISTREAM, ACCESS = 'DIRECT', RECL = LREC,
     :      ASYNCH = 'YES', FORM = 'UNFORMATTED')
       OSTREAM = 12
       OPEN (OSTREAM,  ACCESS = 'DIRECT', RECL = LREC,
     :      ASYNCH = 'YES', FORM = 'UNFORMATTED')
*
       DO I = 1,8                                             !
         T(I) = 0.0D 00                                       !
       ENDDO                                                  !
*
       TSTART = RTC ()                                        !
       READ (ISTREAM, REC = 1, ID = IDRODD) ARIO
```

```
      T(2) = T(2)+RTC ()-TSTART                                !
*
      DO IREC = 1,NREC
*
        IF (MOD (IREC,2) .EQ. 1) THEN
          IF (IREC .LT. NREC) THEN
            TSTART = RTC ()                                    !
            READ (ISTREAM, REC = IREC+1, ID = IDREVN) ARIE
            T(1) = T(1)+RTC ()-TSTART                          !
          ENDIF
          TSTART = RTC ()                                      !
          WAIT (ID = IDRODD)
          T(4) = T(4)+RTC ()-TSTART                            !
          PARI = PARIO
          IF (IREC .GT. 1) THEN
            TSTART = RTC ()                                    !
            WAIT (ID = IDWODD)
            T(8) = T(8)+RTC ()-TSTART                          !
          ENDIF
          PARO = PAROO
        ELSE
          IF (IREC .LT. NREC) THEN
            TSTART = RTC ()                                    !
            READ (ISTREAM, REC = IREC+1, ID = IDRODD) ARIO
            T(2) = T(2)+RTC ()-TSTART                          !
          ENDIF
          TSTART = RTC ()                                      !
          WAIT (ID = IDREVN)
          T(3) = T(3)+RTC ()-TSTART                            !
          PARI = PARIE
          IF (IREC .GT. 2) THEN
            TSTART = RTC ()                                    !
            WAIT (ID = IDWEVN)
            T(7) = T(7)+RTC ()-TSTART                          !
          ENDIF
          PARO = PAROE
        ENDIF
*
        DO IELEM = 1,NELEM
          ARRAYO(IELEM) = GAMMA (ARRAYI(IELEM))
        ENDDO
*
        IF (MOD (IREC,2) .EQ. 1) THEN
          TSTART = RTC ()                                      !
          WRITE (OSTREAM, REC = IREC, ID = IDWODD) AROO
          T(6) = T(6)+RTC ()-TSTART                            !
          IF (NREC-IREC .LE. 1) THEN
```

```
              TSTART = RTC ()                                    !
              WAIT (ID = IDWODD)
              T(8) = T(8)+RTC ()-TSTART                          !
            ENDIF
          ELSE
            TSTART = RTC ()                                      !
            WRITE (OSTREAM, REC = IREC, ID = IDWEVN) AROE
            T(5) = T(5)+RTC ()-TSTART                            !
            IF (NREC-IREC .LE. 1) THEN
              TSTART = RTC ()                                    !
              WAIT (ID = IDWEVN)
              T(7) = T(7)+RTC ()-TSTART                          !
            ENDIF
          ENDIF
*
        ENDDO
*
        PRINT *, 'Operation            Time (in seconds)'    !
        PRINT *, 'Read request (even) ',T(1)                 !
        PRINT *, 'Read request (odd)  ',T(2)                 !
        PRINT *, 'Read wait (even)    ',T(3)                 !
        PRINT *, 'Read wait (odd)     ',T(4)                 !
        PRINT *, 'Write request (even)',T(5)                 !
        PRINT *, 'Write request (odd) ',T(6)                 !
        PRINT *, 'Write wait (even)   ',T(7)                 !
        PRINT *, 'Write wait (odd)    ',T(8)                 !
        PRINT *, '  Total             ',T(1)+T(2)+T(3)+T(4)+ !
       :                               T(5)+T(6)+T(7)+T(8)  !
*
        END
```

Much of the complexity of PROGRAM ASYNCIO can be attributed to code for the collection and reporting of timing data(): Such code is tagged with an exclamation point (!) in column 60. The XL Fortran "integer POINTER" datatype is used to alternately associate the name ARRAYI with the arrays ARIE and ARIO, and the name ARRAYO with the arrays AROE and AROO. Although the dimension of ARIE, ARIO, AROE, and AROO is declared to be NELEM, storage is not allocated to each array until the appropriate MALLOC call is issued; the dimension specification only serves to simplify the structures of the READ and WRITE statements: For instance, the form

```
WRITE (OSTREAM, REC = IREC, ID = IDWODD) AROO
```

is equivalent to the more cumbersome but explicit form

```
 WRITE (OSTREAM, REC = IREC, ID = IDWODD)
:  (AROO(IELEM), IELEM = 1,NELEM)
```

The sequential variant of PROGRAM ASYNCIO is readily created by making the same modifications that are required to create the sequential variant of PROGRAM SYNCIO. Execution of the sequential version of PROGRAM ASYNCIO confronts us with a surprise: Whereas the direct variant of PROGRAM ASYNCIO is significantly faster than the direct variant of PROGRAM SYNCIO, the opposite is true of the sequential variants. The location of a sequential data record depends upon the location of the previous record, which causes the serialization of I/O operations in most cases. Synchronous I/O is performed regardless of the mode specified in the I/O statement when the Fortran compiler must allocate temporary storage for an I/O expression that is resolved at run time.

The POSIX.1b asynchronous I/O facility comprises the "include file" <aio.h> and the following C language bindings:

int aio_cancel(int FileDescriptor, struct aiocb *aiocbp): requests cancellation of asynchronous I/O requests defined by the aiocb control block pointed to by aiocbp pending against the file with descriptor FileDescriptor; return value AIO_CANCELED indicates successful cancelation of all requests defined by the arguments; AIO_ALLDONE indicates that all I/O defined by the arguments completed before the request for cancellation; the return value AIO_NOTCANCELED indicates that all requests defined by the arguments were not successfully canceled.

int aio_error(const struct aiocb *aiocbp) returns the error status associated with the asynchronous I/O requests defined by the aiocb control block pointed to by aiocbp; error statuses are the same as those returned by the (synchronous) read, write, and fsync functions.

int aio_fsync(int opcode, struct aiocb *aiocbp) is an asynchronous request to write all modified I/O block buffers associated with the aiocb control block pointed to by aiocbp so that disk and RAM resident copies are identical; this is not implemented in AIX 4.3.2.

int aio_read(struct aiocb *aiocbp) queues the read request defined by the aiocb control block pointed to by aiocbp; the corresponding AIX 4.3.2 call is aio_read(int FileDescriptor, struct aiocb *aiocbp).

int aio_return(struct aiocb *aiocbp) returns the return status associated with the asynchronous I/O requests defined by the aiocb control block pointed to by aiocbp; return statuses are the same as those returned by the (synchronous) read, write, and fsync functions; the corresponding AIX 4.3.2 call is aio_return(aio_handle_t handle).

int aio_suspend(const struct aiocb * const list[], int count, const struct timespec *timeout) suspends the calling thread until one or more of the asynchronous I/O requests defined by the aiocb control block array list has completed or is interrupted by a signal, or if timeout is satisfied; it is implemented in AIX 4.3.2 as aio_suspend(const struct aiocb * const list[]).

int aio_write(struct aiocb *aiocbp) queues the write request defined by the aiocb control block pointed to by aiocbp; the corresponding AIX 4.3.2 call is aio_write(int FileDescriptor, struct aiocb *aiocbp).

int lio_listio(int mode, struct aiocb * const list[], int count, struct event *eventp) queues the asynchronous I/O requests defined by the aiocb control block array list with count entries with behavior mode; the setting mode to LIO_WAIT; the last argument is ignored in the AIX 4.3.2 implementation.

The _LARGE_FILES counterpart of the asynchronous I/O interface as defined by AIX 4.2.1 is also available in AIX 4.3.3.

We remind the user that writing code to nonstandard interfaces to gain performance typically results in reduced portability; this latter failing can usually be greatly mitigated by isolating non-portable code in a few ‘wrapper’ subprograms that provide the required functionality; porting efforts can then be focussed on the wrapper routines rather than on non-portable code scattered throughout an application.

An alternative to asynchronous I/O as described above is the use of familiar synchronous I/O that is under the control of one or more spawned threads.

6.3.16 Intrusive optimizations: Raw disk I/O

All overheads associated with the use of the journaled file system (JFS) may be eliminated by performing I/O directly to a logical volume; however, the burden of many storage management operations is now placed upon the user. The benefits of logical volume striping remain available. The system administrator's intercession is required to permit direct reading from or writing to a logical volume. Logical volumes that have been enabled in this manner should not simultaneously be used for JFS file systems; indeed, existing JFS file systems on such a volume will be destroyed by raw I/O. Access to a suitably-prepared logical volume is gained by opening the corresponding special file (/dev/...) as a single direct-access file with a record length that is an integer multiple of the sector size (almost invariably 512 bytes). Operations that attempt to access data outside the boundaries of the logical volume may not be detected by the I/O support provided by languages or libraries. Mechanisms for the protection of files from simultaneous access by

different processes are not provided at the logical volume level and must be explicitly prepared if required. In XL Fortran applications, the STATUS='SCRATCH' or STATUS='DELETE' specifiers are ignored, and the maximum volume of data that can be stored is that of the logical volume less one stripe, which is reserved by the runtime library. Optimal performance requires that buffers be aligned on 64-byte boundaries; such alignment may be obtained as described in the latter portion of Section 6.3.10, “Intrusive optimizations: Exploiting high-performance FS” on page 193[30].

6.4 GPFS

The interface presented by the General Parallel File System (GPFS) to the user is essentially that of a standard AIX "virtual file system" (JFS or NFS); in particular, the great majority of POSIX file system calls[31] are available; large file support is provided as described in Section 6.2.2, “Large file support” on page 176. Scalability results from the distribution of file data across storage devices attached to multiple nodes, each of which is equipped with a high-speed communication subsystem; the "availability" of a GPFS file system (its usability in the face of component failure) depends on the degree of replication of data (in particular, the mirroring of file "metadata") and connectivity (the "multitailing" of disk subsystems).

An application package that makes use of relative file names only (this is true of the majority) can exploit GPFS simply by being relocated to a GPFS file system; if, on the other hand, pathnames are "hard coded" in a package, a system administrator may be able to remount a GPFS file system to the required path; more typically, some input data, source code, shellscript, or other editing may be required to reconfigure an application to the pathname of the available GPFS file system.

Applications that are communication-intensive are likely to suffer significant performance degradation if large volumes of data are transferred between the nodes on which they are executing and a GPFS file system because of contention for the communication subsystem between interprocess communication and file data transfer; this may be eliminated by the exclusive use of local file systems (see Section 6.3.2, “Non-intrusive optimizations exploiting high-performance FS” on page 182, Section 6.3.3, “Non-intrusive optimizations: Obviating contention” on page 183, and Section 6.3.10, “Intrusive optimizations: Exploiting high-performance FS” on page 193) or mitigated by the use of such file systems for some, but not all, file I/O. In most

[30] The subprograms allociobuffer and freeiobuffer force alignment on page boundaries, which is more restrictive than is required in the present context; to obtain alignment on 64-byte boundaries, every occurrence of 4096 and 0xfff in these functions should be replaced by 64 and 0x3f.

[31] The few exceptions are listed in detail towards the end of the present subsection.

RS/6000 SP production environments, GPFS file systems are shared among the users of the system. Some reduction in file system performance will result from contention for disk and communication subsystem resources on VSD server nodes. Again, local file systems may be exploited to improve performance.

The GPFS Token Manager provides a byte-range locking mechanism to provide a parallel task exclusive access to non-overlapping file blocks. Application I/O performance (and, thus, overall application performance) can deteriorate if the token management overhead is significant in comparison to the I/O time; this situation typically arises when I/O requests are small compared to stripe size.

The performance of an application that uses a GPFS file system can be sensitive to aspects of the file system that are under the control of system administrators.

Memory-mapped files (see Section 6.3.15, “Intrusive optimizations: Asynchronous I/O” on page 202) are not supported in GPFS Version 1 Release 2. In particular, the calls mmap, munmap, msync, shmat are not supported by GPFS 1.2; IBM intends to support these calls in a future release of GPFS. Some system software, such as the XL Fortran compiler, may be programmed to switch from file-based I/O to memory-mapped I/O in certain circumstances; this switching will fail if I/O is to a GPFS 1.2 file system.

The accuracy of the components time_t st_atime, time_t st_ctime, and time_t st_mtime of struct stat (see /usr/include/sys/stat.h) as returned by the stat, fstat, and lstat functions, is only guaranteed when a GPFS file is closed. The maintenance of similar accuracy for these members at other times imposes an unacceptable performance penalty; it follows that applications that depend upon the accurate reporting of these fields at other times may not work as expected.

The process filesize resource limit (file(blocks), for instance, as returned by the `ulimit` command is not enforced in GPFS 1.2; IBM intends to provide support in future releases of GPFS.

The GPFS programming interface currently comprises one subprogram with a C language binding; the syntax specification is available in the header file `/usr/include/gpfs.h`:

```
int gpfs_prealloc (int fileDesc, offset_t StartOffset,
offset_t BytesToPrealloc);
```

Applications incorporating this call should be linked with the GPFS library, /usr/lib/libgpfs.a. The file with descriptor fileDesc must be opened prior to the invocation of gpfs_prealloc. The return value is 0 for a successful invocation; otherwise, the return value is -1 and the global errno variable is set. Preallocation can result in faster I/O as compared with an incremental file enlargement.

References

- AIX Commands Reference, A set of manuals with this title has been issued with every release of the AIX operating system for some time; for AIX Version 4.3 (Version 4 Release 3), the IBM publication number is SBOF-1877
- Communications Programming Concepts: A manual with this title has been issued with every release of the AIX operating system for some time; for AIX Version 4.3, the IBM Publication number is SC23-4124. The information in this book can also be found in HTML format on the AIX Version 4.3 Base Documentation CD. This online documentation is designed for use with an HTML version 3.2 compatible Web browser.
- Computer Architecture: A Quantitative Approach, Second Edition, J. L. Hennessy and D. A. Patterson, published by Morgan Kaufman, 1996
- General Programming Concepts: Writing and Debugging Programs: A manual with this title has been issued with every release of the AIX operating system for some time; for AIX Version 4.3 (Version 4 Release 3), the IBM Publication number is SC23-4128.
- Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification: The Open Group
- GPFS: A Parallel File System, IBM Redbook SG24-5165
- IBM General Parallel File System for AIX: Installation and Administration Guide, IBM Publication Number SA22-7278
- Optimization and Tuning Guide for Fortran, C, and C++: for AIX Version 4, the IBM Publication number is SC09-1705 PE Operation and Use Vol. 2, Part 2 — Profiling, IBM Publication SC28-1980.
- Performance Tuning Guide, AIX Versions 3.2 and 4, IBM Publication SC23-2365
- The UNIX Programming Environment, B. W. Kernighan and R. Pike, Prentice-Hall, Inc. 1984.
- XL Fortran for AIX User's Guide: A manual with this title has been issued with every release of the XL Fortran Compiler; for XL Fortran Version 66 Release 1, the IBM Publication number is SC09-2719.

6.5 MPI-IO

The MPI-1 standard was a big step forward in defining an expressive and portable interface for developing message passing programs. Some of the most useful tools that MPI-1 adds to the basic send/receive model have

already been discussed in this book. These include collective communication, communicators, MPI datatypes, and nonblocking send/receive. One longstanding problem in developing good message passing parallel programs has been file IO. Parallel codes are used to solve large problems. Often, this means that large data sets must be read and written and that each task will be concerned only with distinct pieces of these large sets. MPI-1 did not deal with IO even though it was clear that moving data between files and tasks should be similar in many ways to moving data among tasks. In what is arguably the most valuable addition to MPI, the MPI-2 standard has defined an interface for reading and writing files that extends MPI's expressive power and portability to doing parallel IO.

POSIX IO is quite portable because it is supported by most platforms on which MPI is implemented. However, traditional POSIX operations for reading and writing files have serious limitations in large parallel applications. POSIX read/write does not give a program any way to express that more than one task is interested in a particular file; so, the program must be designed to work around this limitation. There are several techniques, and none of them is entirely satisfactory. For example, a parallel program in which each task has generated some portion of the whole output picture might do one of the following:

- Use message passing to collect the data at one task, which then does a write()
- Have each task write() its data to a unique file and either keep the set of files or post process to make a single file
- Have each task write() to different parts of a common file and let the file system sort it out

Similar techniques can apply to large input data sets. The third technique is much more likely to give satisfactory performance for reading than for writing. Not all shared or distributed files systems can guarantee to sort out the scattered writes of option three correctly.

IBM GPFS does provide a POSIX read/write interface that can make the third approach safe for writing and, in cases where each task writes to its own large contiguous section of the file, quite efficient. When tasks write smaller interleaved file fragments, the token management that GPFS or any parallel file system must do to assure correct results will have a serious performance cost. The problem is inherent in the write() function because it offers no way of indicating what any other task will do or what the calling task will do when it makes additional write() calls. When the goal is to have a single output file, the first and second approaches take extra programming and serialize much

of the actual output. The use of distinct output files that are kept distinct can allow writing to proceed in parallel and with high efficiency, but an output set from an eight task job is probably only useful as input to another eight task job that wants the data exactly as the previous job partitioned it. Post processing the distinct files into a single file with a serial job is likely to be too time-consuming to consider.

Parallel processing with MPI is most useful for solving a problem structured in a way that allows it to be subdivided among several tasks and big enough to be worth the effort. Any MPI job, whether it has two or 1000 tasks typically involves cooperation among the tasks to solve one problem, and, from this perspective, there can be one set of input data and one set of output data. Assume a user wants to produce one output file representing an 400x400x400 matrix of C ints and uses eight tasks with each task producing a 200x200x200 corner cube. If each task tries to distribute its 8,000,000 ints directly to the right places in the file with POSIX write() calls, there will be thousands of write calls per task and much contention. If the output operation is viewed globally, its intent is a single 8*8,000,000*sizeof(int) write from one job to one file. Without MPI-IO, the programmer probably will choose to write additional MPI code to shuffle the data among tasks and into large contiguous blocks that can be written more efficiently.

Other chapters in this book have illustrated some of the ways an MPI program might use collective communication and datatypes to express communications or interactions among tasks. Such tools do not solve any problem that the programmer could not also solve with enough extra code. What they really do is allow the programmer to express the intent rather than write the extra code and let the MPI library choose a way to carry out this intent. MPI-IO has the same goal: Let the programmer specify the intent of each task at a higher level and with collective operations where appropriate. Let MPI carry it out efficiently. A POSIX write() at some task can only identify one fragment of data from that task. It cannot identify whether other tasks will contribute other fragments or even what fragment the next local write() will contribute. The single task call, MPI_FILE_WRITE_AT(), depends on MPI_Datatypes to identify a set of data fragments at the call. The collective call, MPI_FILE_WRITE_AT_ALL(), not only identifies a set of file fragments the calling task wishes to write but also declares that every other task in the group will also make the call and identify a set of fragments. The collective operation is most appropriate when fragments from each task fit with those from the other tasks in a way that allows MPI to coalesce them into a few larger fragments.

6.5.1 IBM implementation

Parallel Environment 2.4 is the first release to provide features of MPI-IO on the SP. IBM is developing its own implementation of MPI-IO, which is tightly-integrated with the rest of the MPI implementation. The long-range intent is that this will provide the best performance because it can directly utilize features of the IBM underlying communication subsystem and can provide the best integration with GPFS. The public domain version of MPI (MPICH) does provide an implementation of MPI-IO (ROMIO) that is nearly complete but is layered on top of other MPI functions. IBM had to choose between porting ROMIO and having a complete MPI-IO quickly vs. developing an integrated MPI-IO implementation and staging its delivery. PE Release 2.4 contains a subset of the MPI-IO API, and work on completing the interface as well as improving performance will continue.

The subset of MPI-IO to be provided in the first release was selected by surveying technical users who were already familiar with the MPI-2 standard. They were asked what parts of the MPI-2 standard they thought would be needed to make MPI-IO useful to the applications which concerned them. There was general agreement on the subset which PE 2.4 provides. This includes the information displayed in Table 29.

Table 29. MPI-2 subset included in PE 2.4

MPI Functionality	MPI routines
File manipulation	MPI_FILE_OPEN MPI_FILE_CLOSE MPI_FILE_DELETE MPI_FILE_SET_SIZE MPI_FILE_GET_SIZE
File views	MPI_FILE_SET_VIEW MPI_FILE_GET_VIEW
File hints	MPI_FILE_SET_INFO MPI_FILE_GET_INFO
Blocking single task I/O	MPI_FILE_READ_AT MPI_FILE_WRITE_AT
Blocking collective I/O	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
Non blocking single task I/O	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT
Inquiry	MPI_FILE_GET_GROUP MPI_FILE_GET_AMODE MPI_FILE_GET_ATOMICITY

MPI Functionality	MPI routines
File consistency	MPI_FILE_SYNC
Filetype constructors	MPI_TYPE_CREATE_SUBARRAY MPI_TYPE_CREATE_DARRAY
Error handler management	MPI_FILE_CREATE_ERRHANDLER MPI_FILE_SET_ERRHANDLER MPI_FILE_GET_ERRHANDLER
MPI Info functions	MPI_INFO_CREATE MPI_INFO_SET MPI_INFO_DELETE MPI_INFO_GET MPI_INFO_GET_VALUELEN MPI_INFO_GET_NKEYS MPI_INFO_GET_NTHKEY MPI_INFO_DUP MPI_INFO_FREE

There is often confusion about what MPI-IO is. It is not a file system; rather, it is an application interface to a file system. MPI-IO does not deal with disk drives, RAID arrays, striping across drives, or hardware failure recoverability. That is all left to lower-level systems. The implementation of MPI-IO in PE 2.4 is designed to work with the GPFS file system. MPI lets GPFS/VSD take care of the details that make a GPFS file visible to all SP nodes on which the file system is mounted and ensures that the data committed to it by an MPI program is reliably delivered to disk drives. The MPI-IO implementation requires that a file opened via MPI_FILE_OPEN() be visible to every task in the MPI job and presumes that GPFS will maintain the file's integrity when operations, such as write() or fsync(), occur from within MPI on more than one node. A single call to MPI_FILE_WRITE_AT() by one task can result in many tasks making concurrent (but never conflicting) write() calls to GPFS. Even when only a subset of MPI_COMM_WORLD tasks opens a file or make MPI-IO calls to access it, any task in the job may do read() or write() calls to GPFS in support of the access. That is why every task must be on a node that has the GPFS file system mounted.

GPFS manages access to a file distributed across nodes in terms of blocks. The block size is configured by the administrator with 256K being fairly typical. GPFS uses a token management scheme to control which node has current authority for each block. When a task on some node touches a block with a read() or write(), there are three possibilities: The best case is when the node has previously touched the block and still holds the token (no other node has stolen it). The second best case is when no node has touched the block so far, and the current node can be quickly granted a token. The worst

case by far is when some other node has been writing to the block, and GPFS must wait until it can complete and commit any pending activity at that node, reclaim the token, and grant it to the new node.

The MPI-IO implementation is designed with a data shipping strategy to ensure that, for each GPFS block, there is only one node that will issue reads and writes to GPFS. This is done by defining a file partition size within MPI to be a multiple of the GPFS block size and then associating each MPI file partition with one task of MPI_COMM_WORLD. The partition size for PE 2.4 is set to 1 meg and the partitions are mapped to tasks in round-robin rank order. In a four-task job, rank 0 is responsible for the first, fifth, ninth, and so on 1 meg partitions. A write to the file made through any MPI-IO call at any task is done by determining which parts of the output data fall within each MPI file partition and sending a command (descriptor) and data to the nodes that own each affected file partition. A mechanism (called a “responder”) is waiting at each task to asynchronously accept a command and assign it to a background thread to be executed. The task and thread on which a user program call to an MPI-IO read or write occurs plans both the actions it must take and the complementary actions the responder must take. The command built at the task where the call occurs details the steps for the responder to take. For a single task (non collective) write operation, responder execution involves receiving the data and doing one or more write() calls on the responder thread. For a collective write operation, the responder expects a command and, probably, data from every task in the group. The data sent by each task is merged in a memory buffer with data from other tasks, and, after all tasks have contributed, the minimum number of write() calls is used to transfer the coalesced data to the file. Once a responder thread finishes a command, it sends a report to the task at which the MPI-IO call was made. When every command that the operation requires has been carried out and reported back, the MPI-IO call is complete and can return either MPI_SUCCESS or an error to the calling task. An MPI-IO read from a file uses a similar approach, except that the responders do read() calls and distribute the data rather than accept and write() it.

6.5.2 Using MPI-IO effectively

The MPI data shipping approach described above is a trade-off, and the first release of MPI-IO uses it for all IO done by MPI-IO calls. When the fragments of data to be read or written are small and scattered across tasks, this MPI data shipping is a clear winner. The savings from eliminating GPFS token thrashing far outweigh the cost of shipping the data. When the data to be read or written by each task is in large fragments or when each task accesses different large regions of the file without touching any region accessed by other tasks, there is little or no GPFS token thrashing to avoid. In such cases,

using GPFS directly via read() or write() calls will outperform MPI-IO. In later releases, MPI-IO is expected to use data shipping when it can pay off and bypass it when it cannot. For this release, the performance-sensitive user of the current version may want to consider whether to use MPI-IO or direct read()/write() in each case.

Opening a file is done with MPI_FILE_OPEN(), which is a collective operation; this means that it must be called by every task of the input communicator. Often, the input communicator will be MPI_COMM_WORLD, but any communicator may be used. MPI_FILE_OPEN() returns an MPI file handle (MPI_File) to each calling task, and all further MPI-IO operations on the file use this file handle as an identifier. An MPI_File has much in common with an MPI_Comm. It is defined across some group of tasks; all collective operations on an MPI_File involve that group, tasks outside the group have no access to it, and, finally, operations on one MPI_File are semantically independent of operations on another MPI_File. MPI-IO does allow more than one MPI_FILE_OPEN() to be done on a file; so, more than one MPI_File handle may represent the same actual file. MPI does not define how conflicting operations on the two MPI_Files behave. MPI_FILE_SYNC() on an MPI_File commits all preceding operations on the file and ensures all subsequent operations see any actions committed before the sync. By using MPI_FILE_SYNC() properly, you can enforce a well-defined semantic because you can guarantee that file operations on a single file via different MPI_File handles are not conflicting. When a file is opened, each task has the same default view of it: An array of MPI_BYTE beginning at the first byte of the actual file. Any open MPI_File has an associated view, either this default view or some other view defined by the program to replace the default. In this default view, the elementary type or etype of the file is MPI_BYTE. In any file view, offsets are counted by etype. In the default case, every task sees byte n of the actual file as offset n in its view. An MPI-IO file read or write call at a task specifies an MPI_File and MPI_Offset (in etype units) as well as a buffer address, datatype, and count. The buffer, datatype, and count dictate how much data is to be read or written and the memory locations for that data. The byte location in the actual file is determined by MPI based on the etype offset within the view. With the default file view, MPI_Offset is equivalent to byte displacement, and, for some applications, this is enough.

The real expressive power of MPI-IO in supporting efficiency is exploited by using MPI_FILE_SET_VIEW() at each task to replace the default view with one that fits that task's role in the application. Each view begins at an explicit byte displacement from the beginning of the file and is defined by etype and filetype. The etype is an MPI_Datatype that the programmer selects as the elementary type from which the file is built. It can be either a predefined or

user-defined datatype. The filetype of the view is an MPI_Datatype made of etype components, and it normally has gaps to account for the portions of the file that belong to other tasks. Each gap must be the size of an integral number of etypes. The view of the file as seen from each task is as if the filetype was tiled contiguously across the file, that is, as if by MPI_TYPE_CONTIGUOUS() with the filetype datatype and an arbitrarily large count. Usually, each task will have a different view, perhaps at a different displacement, but the set of views should be designed to complement each other. MPI_FILE_SET_VIEW() is collective; so, all tasks in the MPI_File group must participate in changing views, but views may be changed as often as necessary. To illustrate a use of file views, we will take a look at how the problem above with eight tasks can be solved with file views and a collective write. The following example program is scaled down to produce a 4×4×4 matrix in the file with each task contributing a 2×2×2 corner sub-matrix.

Sample program:

```
#include "mpi.h"
#define CHECK(X) if (X) \
                 {printf("OOPS at line %d.\n",__LINE__); MPI_Abort(MPI_COMM_WORLD,0);}
/* dims of corner each task will tackle */
#define TASK_X 2
#define TASK_Y 3
#define TASK_Z 2
/* dims of matrix in file with all 8 contributions */
#define FILE_X (2*TASK_X)
#define FILE_Y (2*TASK_Y)
#define FILE_Z (2*TASK_Z)
void main( ) {
    int i, j, k, numtasks, me;
    int rem, disp, etype_ext;
    MPI_Datatype file_row, file_slice, corner, file_corner, aot[2];
    MPI_File fh;
    MPI_Status status;
    int gsizes[3] = {FILE_X, FILE_Y, FILE_Z};
  int distribs[3] = {MPI_DISTRIBUTE_BLOCK, MPI_DISTRIBUTE_BLOCK,
MPI_DISTRIBUTE_BLOCK};
   int dargs[3] = {MPI_DISTRIBUTE_DFLT_DARG, MPI_DISTRIBUTE_DFLT_DARG,
MPI_DISTRIBUTE_DFLT_DARG};
    int psizes[3] = {2, 2, 2};
    int matrix[TASK_Z][TASK_Y][TASK_X]; /* C is row major - let X coord be contiguous */
    MPI_Init( 0,0 );
    /* MPE_ERRORS_WARN is an IBM extension, not standard MPI. */
    MPI_File_set_errhandler(MPI_FILE_NULL,MPE_ERRORS_WARN);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (numtasks != 8){
      if (me == 0) printf("This sample program requires 8 tasks.\n");
      MPI_Abort(MPI_COMM_WORLD, 0);
    }
    /* Init task array with values we can take apart later to illustrate the effect */
    for (i=0; i<TASK_Z; i++)
      for (j=0; j<TASK_Y; j++)
        for (k=0; k<TASK_X; k++)
          matrix[i][j][k] = me*1000000 + i*10000 + j*100 + k;
```

```
    /* create filetype MPI_Datatype */
    MPI_Type_create_darray(numtasks,me,3,gsizes,distribs,dargs,psizes,MPI_ORDER_C,
                           MPI_INT, &file_corner);
    MPI_Type_commit(&file_corner);
    CHECK(MPI_File_open(MPI_COMM_WORLD, "./iotest",
                  MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh))
    CHECK(MPI_File_set_view(fh, 0, MPI_INT, file_corner, "native", MPI_INFO_NULL))

    /* We could build an MPI_Datatype to describe all or parts of matrix as well but
       this example is about using file views. We take a shortcut in picking up matrix
      from memory as a flat array based on knowing how C stores the matrix.          */
    CHECK(MPI_File_write_at_all(fh,0,matrix,TASK_X*TASK_Y*TASK_Z,
                  MPI_INT, &status))
    CHECK(MPI_File_close(&fh))
    /* We read back file with one task just to format and print the file content */
    if (me==0) {
      int file_matrix[FILE_Z][FILE_Y][FILE_X];
      int temp, t, x, y, z;
      CHECK(MPI_File_open(MPI_COMM_SELF, "./iotest",MPI_MODE_RDONLY,
                    MPI_INFO_NULL, &fh))
      CHECK(MPI_File_read_at(fh, 0, file_matrix, FILE_X*FILE_Y*FILE_Z,
                    MPI_INT, &status))
      CHECK(MPI_File_close(&fh))
      printf("File matrix dims, in row major order (Z Y X) are: [%d][%d][%d]\n",
                  FILE_Z, FILE_Y, FILE_X);
      printf("Each slice is one XY plane so there are Z slices\n");
      printf("Each cell shows task# and matrix position within that task\n");
      for (i=0; i<FILE_Z; i++) {
        printf("========= File Slice %d ============\n", i);
        for (j=0; j<FILE_Y; j++) {
          for (k=0; k<FILE_X; k++) {
            t = file_matrix[i][j][k] / 1000000;
            temp = file_matrix[i][j][k] % 1000000;
            z = temp / 10000;
            temp %= 10000;
            y = temp /100;
            x = temp % 100;
            printf("%d[%d][%d][%d]   ", t, z, y, x);
          }
          printf("\n");
        }
       }
      }
    MPI_Finalize();
    exit(0);
}
```

There are a few points about the sample program that are worth discussing.

First, we will briefly discuss the simplifications used in the program. An MPI_FILE_WRITE_AT_ALL() or other MPI-IO write can be visualized as an MPI_SEND() to the file with the appropriate MPI_RECV() already waiting. The etype and filetype used in the MPI_FILE_SET_VIEW() call provide the information MPI uses to create this implicit MPI_RECV().

The sample program's FILE_WRITE treated the three dimensional matrix in task memory as a contiguous array of MPI_INT because the way C stores a matrix in memory allows such a shortcut, and all of the data was to go to the

file in its natural order. Other sections of this book deal with user-defined datatypes for describing fragmented data in memory.

Any user-defined datatype may be used in an MPI-IO read or write as long as its type signature matches the type signature of the filetype. The default view in which both etype and filetype are MPI_BYTE is the only exception to the type signature rule and allows the simple file read back; so, results can be printed. No matter what datatype is used for the MPI-IO read or write, each operation must represent on integral number of etypes.

A comment in the program mentions that it may be run as eight tasks on a single workstation without GPFS. When MPI-IO is used from a single node or workstation, the file systems available on the RS/6000 provide the level of file consistency protection required by MPI-IO. This includes at least JFS, NFS, AFS, and DFS. Only GPFS can provide this consistency across multiple nodes. The option of running MPI-IO on a single node is not expected to be useful in production but can be useful for learning or experimentation.

The sample program also illustrates some points about MPI-IO and errors. An important difference between MPI-IO functions and all other MPI functions is that, for all MPI-IO calls, the default error handler is MPI_ERRORS_RETURN. Many of the errors that occur in doing IO are predictable, and well-written programs will try to recover (for example, *file not found* or *permission denied*). For all other MPI functions, the default is MPI_ERRORS_ARE_FATAL because attempting recovery would usually not be safe anyway. It has become common for MPI programmers to assume any error will result in job termination and not bother to check return codes. With MPI-IO, it is essential that the programmer take responsibility for detecting errors and deciding what to do about them. One option is to use MPI_FILE_SET_ERRHANDLER() to replace the default ERRORS_RETURN with another errorhandler. In the sample, the IBM-defined errorhandler MPE_ERRORS_WARN is attached to MPI_FILE_NULL, which has the effect of making it the default error handler for any MPI_File (this errorhandler is an IBM extension to standard MPI. It behaves like MPI_ERRORS_RETURN except that it prints an error message before returning). The sample also puts each MPI-IO call within a macro that provides a return code check and terminates the job on error. Notice that the calls that are not part of MPI-IO do not check return codes. Errors are fatal by default for these calls; so, checking return codes is pointless unless their default errorhandler is replaced.

The sample program has some characteristics that make file views and collective write especially appropriate. One is the complementary roles each task plays in generating the full result; another is that the relatively fine granularity of the fragments coming from each task MPI-IO is able to combine

the many small contributions into contiguous buffers that can be written efficiently. When granularity is small but task contributions to a file do not come along in a reasonably synchronized way, the use of MPI-IO with file views may still make sense, but use of the collective calls probably does not. A collective read or write that does not offer MPI any opportunity to coalesce contributions from different tasks into larger contiguous fragments for read() or write() is unlikely to perform as well as the non-collective operations. A collective read or write is synchronizing (although this is not guaranteed by the MPI standard). When tasks can make independent progress and would otherwise suffer from non-essential synchronizations, the non-collective versions are, again, likely to be better.

Files created by MPI-IO and files created by POSIX are fully-interchangeable. MPI-IO provides powerful features for defining the layout of data in a file but never adds any hidden data of its own. MPI does not create self-defining files. MPI-IO will make use of much the same file metadata POSIX does (for example, file size). Beyond this, all interpretation of the data in a file, whether via POSIX or MPI-IO, depends on the program that reads the file and understands the format in which it was written.

Appendix A. Special notices

This publication is intended for developers of numerically-intensive code, parallel programmers for the RS/6000 SP, business partners and sales specialists wanting supporting metrics for the POWER3 performance potentials, and technical specialists who require detailed product information to help demonstrate IBM industry-leading technology. See the PUBLICATIONS sections of the *IBM Programming Announcement for Parallel System Support Programs for AIX* and *IBM Parallel Environment for AIX* for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only the IBM product, program, or service may be used. Any functionally-equivalent program that does not infringe any IBM intellectual property rights may be used instead of the IBM product, program, or service.

Information in this book was developed in conjunction with the use of the equipment specified and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently-created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including, in some cases, the payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor, and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer

responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX®
AIX/6000®
AS/400®
DB2®
IBM ®
LoadLeveler®
Power PC 603®
Power PC 604®
PowerPC 601®
PowerPC 603®
PowerPC 601e®
POWER2 Architecture®
POWER3 Architecture®
RISC System/6000®
RS/6000®
SP®
System/390®

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries. (For a complete list of Intel trademarks see www.intel.com/tradmarx.htm)

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through The Open Group.

SET and the SET logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Appendix B. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

B.1 IBM Redbooks publications

For information on ordering these publications, see “How to get IBM Redbooks” on page 233.

- *AIX Version 4.3 Differences Guide*, SG24-2014
- *AIX 64-Bit Performance in Focus*, SG24-5103
- *RS/6000 SP: Practical MPI Programming*, SG24-5380

B.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at http://www.redbooks.ibm.com/ for information about all the CD-ROMs offered, updates, and formats.

CD-ROM Title	Collection Kit Number
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
RS/6000 Redbooks Collection (BkMgr Format)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694
Netfinity Hardware and Software Redbooks Collection	SK2T-8046

B.3 Other resources

These publications are also relevant as further information sources:

- *AIX Commands References for AIX Version 4.3*, SBOF-1877
- *Communication Programming Concepts*, SC23-4124

- *Optimization and Tuning Guide for Fortran, C, and C++ for AIX Version 4*, SC09-1705
- *Parallel Environment Operation and USE Vol. 2, Part 2 - Profiling*, SC28-1980
- *AIX Version 3.2 and V4*, *Performance Monitoring and Tuning Guide*, SC23-2365
- *XL Fortran for AIX User's Guide*, SC09-2719
- *Computer Architecture: A Quantitative Approach*, Second Edition, J. L. Hennessy and D.A. Patterson, Morgan Kaufman, 1996
- *The UNIX Programming Environment*, B.W. Kernighan and R. Pike, Prentice-Hall Inc., 1984
- *Parallel Environment for AIX:Operation and Use*, SC28-1979
- *AIX V4 General Programming Concepts: Writing and Debugging Programs*, SC23-2533
- *LoadLeveler User's Guide Release 2.1*, SH26-7226
- *XL Fortran for AIX Language Reference V6.1*, SC09-2718
- *AIX V3.4 General Programming Concepts; Writing and Debugging Programs*, SC23-4128
- *Using MPI*, ISBN 0-2625-7104-8
- *Designing and Building Parallel Programs*, ISBN 0-2015-7594-9
- *MPI: The Complete Reference - 2nd Edition: Volume 2 - The MPI-2 extensions*, MIT Press
- *Parallel Programming With MPI*, Morgan Kaufmann publishers
- *CAE Specification, System Interfaces and Headers, Issue 5: Volume 1*, the Open Group 1997, ISBN 1-8591-2181-0
- *Programming With Posix Threads*, Addison Wesley publishers
- *Thread Time: The Multithreaded Programming Guide*, Prentice Hall
- *Pthreads Programming*, O'Reilly & Associates

B.4 Referenced Web sites

The following Web sites are also relevant as further information sources:

- `http://www.mpi-forum.org/docs/docs.html`
- `ftp://ftp.mcs.anl.gov/pub/mpi/using`

- http://www.openmp.org
- www.opengroup.org
- http://www.openmp.org
- http://www.research.ibm.com/actc/Tools/MPI_Threads.htm

How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** http://www.redbooks.ibm.com/

  Search for, view, download or order hardcopy/CD-ROM Redbooks from the Redbooks web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the IBM Redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl/

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl/

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl/

This information was current at the time of publication, but is continually subject to change. The latest information for customer may be found at http://www.redbooks.ibm.com/ and for IBM employees at http://w3.itso.ibm.com/.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at http://w3.itso.ibm.com/ and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at http://w3.ibm.com/ for redbook, residency, and workshop announcements.

IBM Redbooks fax order form

Please send me the following:

Title	Order Number	Quantity

First name	Last name	
Company		
Address		
City	Postal code	Country
Telephone number	Telefax number	VAT number

☐ Invoice to customer number _____

☐ Credit card number _____

Credit card expiration date	Card issued to	Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Glossary

API	Application Program Interface
ASCI	Accelerated Strategic Computing Initiative
BCT	Branch on Count
BHT	Branch History Table
BLAS	Basic Linear Algebra Subroutines
BLACS	Basic Linear Algebra Communications Subroutines
BT	Block Tridiagonal
BTAC	Branch Target Address Cache
CCR	Condition-Code Register
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
DASD	Direct Access Storage Device
DFL	Divide Float
DIMM	Dual Inline Memory Modules
DOE	Department of Energy
ESSL	Engineering and Scientific Subroutine Library
FMA	Floating-point Multiply-Add
FPR	Floating-Point Register
FPU	Floating Point Unit
GAMESS	General Atomic and Molecular Electronic Structure System
GPR	General-Purpose Register
IBM	International Business Machines
ITSO	International Technical Support Organization
LFD	Load Float Double
LLNL	Lawrence Livermore National Laboratory
LRU	Least Recently Used
MASS	Mathematical Acceleration Subsystem
MFLOPS	Millions of Floating-Point Operations per Second
MPI	Message Passing Interface
MTU	Maximum Transmission Unit
NUS	Numerical Aerodynamic Simulation
NWP	Numerical Weather Prediction
P2SC	POWER2 Single/Super Chip
PBLAS	Parallel Basic Linear Algebra Subroutines
PPM	Piecewise Parabolic Method
PSSP	Parallel System Support Programs

RISC	Reduced Instruction-Set Computer
RSC	RISC Single Chip
SPEC	System Performance Evaluation Cooperative
SOI	Silicon-on-Insulator
SMP	Symmetric Multiprocessing

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

X

IBM Redbooks evaluation

Scientific Applications in RS/6000 SP Environments
SG24-5611-00

Your feedback is very important to help us maintain the quality of IBM Redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com/
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
_ Customer _ Business Partner _ Solution Developer _ IBM employee
_ None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

_____

_____

_____

_____

What other Redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

_____

_____

_____

_____

SG24-5611-00
Printed in the U.S.A.

Scientific Applications in RS/6000 SP Environments

SG24-5611-00