IBM AIX Parallel Environment Programming Primer Release 2.0

SH26-7223-00

IBM

IBM AIX Parallel Environment Programming Primer Release 2.0

SH26-7223-00

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page iv.

First Edition, June 1994

This edition applies to Version 1, Release 2 of IBM AIX Parallel Environment, program number 5765-144, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department 52QA, Mail Station 911
1 Enterprise Drive
Kingston, NY 12401-1099
United States of America

FAX: (International Access Code)+1+914+385-0662

IBMLink* (United States customers only): KGNRCF at KGNVMC
IBM Mail Exchange: USIB27BQ at IBMMAIL
Internet: kgnrcf@vnet.ibm.com

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

Copyright International Business Machines Corporation 1993, 1994. All rights reserved. Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.. Contains licensed material — Property of IBM.

Contents

Notices iv

About This Book v
Who Should Use This Book v
How to Use This Book v
Overview of Contents v
Related PE Publications vi
Related IBM Publications vi
Related Non-IBM Publications vi
Ordering Additional Copies of This Book vi
Obtaining This Book's Examples Through Anonymous ftp vii
Accessing Online Help Information vii

Chapter 1. Introduction 1

Chapter 2. The Message Passing Library (MPL) Calls 3
Managing Tasks in the Parallel Partition 4
Creating and Querying Task Groups 5
Sending and Receiving Messages 6
How Tasks Pass Messages 6
Using the dontcare Wildcard When Receiving Messages 8
Non-Blocking and Blocking Routines 9
Overview of the Basic Message-Passing Routines 10
Avoiding Programming Pitfalls in Point-to-Point Message Passing 15
Performing Collective Communication Within Task Groups 22
Performing Collective Message Passing Within a Task Group 22
Performing Collective Compute Operations Within Task Groups 27

Chapter 3. Converting Serial Programs for Parallel Execution 31
Step 1: Tune Serial Code 31
Step 1a: Familiarize Yourself with the Serial Code 32
Step 1b: Identify Program Hot Spots 33
Step 1c: Specify Low-Level Compiler Optimizations 36
Step 1d: Specify or Perform High-Level Tuning 37
Step 2: Parallelize the Application 42
Step 2a: Identify Section(s) of Code to Parallelize 42
Step 2b: Call MPL Routines to Enable Parallel Execution 43
Step 3: Tune Parallel Code 47

Appendix A. Functional Decomposition Example 49
MPLprob2s.f 49
MPLprob2p.f 52

Glossary of Terms and Abbreviations 57

Index 61

© Copyright IBM Corp. 1993

Notices

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:

IBM Corporation
IBM Director of Licensing
208 Harbor Drive
Stamford, Connecticut
United States 06904

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

AIX	RISC System/6000
IBM	Scalable POWERparallel Systems
IBMLink	SP1
InfoExplorer	SP2
POWERparallel	

The following terms, denoted by a double asterisk (**), in this publication, are trademarks of other companies as follows:

Express	Parasoft Corporation
KAP	Kuck and Associates, Inc.
VAST	Pacific-Sierra Research Corporation
X-Windows System	Massachusetts Institute of Technology

The following term, denoted by three asterisks (***) in this publication, is a public domain package.

PVM	developed at Oak Ridge National Laboratory

About This Book

This primer provides guidelines and hints for developing parallel C, C++, and FORTRAN applications with the IBM* AIX* Parallel Environment program product (PE). In particular, it covers the fundamentals of using PE's message passing library (MPL) subroutines to parallelize applications.

While this primer does cover the concepts central to using MPL subroutines, it does not contain complete syntax for the calls. For these, you are instead referred to the on-line PE manual pages or the *IBM AIX Parallel Environment: Programming Subroutine Reference*, (SH26-7228). Also note that while this primer does refer to PE tools designed to help you tune your parallel applications, it does not contain detailed instructions for their use. Detailed instructions appear in *IBM AIX Parallel Environment: Operation and Use*, (SH26-7230).

Who Should Use This Book

This book is intended for experienced programmers who want to write parallel applications in the C, C++, or FORTRAN programming languages. Readers of this book should be familiar with AIX and parallel programming concepts. Knowledge of distributed memory - message passing systems and RISC System/6000* processors is also helpful.

How to Use This Book

Overview of Contents

This book contains an introduction to the PE, an overview of its MPL subroutines, and some general guidelines for converting serial programs for parallel execution. It contains three chapters and an appendix.

- **Chapter 1, "Introduction" on page 1** . This chapter briefly describes the PE and its Message Passing Library (MPL).
- **Chapter 2, "The Message Passing Library (MPL) Calls" on page 3** . This chapter describes all the MPL subroutines you can use to parallelize your programs. It provides a foundation necessary for understanding and using the subroutines, and describes how to avoid some typical programming pitfalls.
- **Chapter 3, "Converting Serial Programs for Parallel Execution" on page 31**. Using a case-study approach, this chapter contains basic steps and hints you can follow to parallelize existing serial applications using the MPL subroutines. The programs and profiler output used throughout this chapter are available via anonymous ftp from **tc.cornell.edu**.
- **Appendix A, "Functional Decomposition Example" on page 49** . This appendix contains some additional programs for illustration. The programs shown here are, like the examples in the previous chapter, available via anonymous ftp from **tc.cornell.edu**.

© Copyright IBM Corp. 1993

Related PE Publications

IBM AIX Parallel Environment: Installation, Administration, and Diagnosis, (SH26-7231)

IBM AIX Parallel Environment: Operation and Use, (SH26-7230)

IBM AIX Parallel Environment: Programming Subroutine Reference, (SH26-7228)

IBM AIX Parallel Environment: General Information, (GH26-7229)

Related IBM Publications

AIX for RISC System/6000: Performance Monitoring and Tuning Guide, (SC23-2365)

IBM RISC System/6000: NIC Tuning Guide for FORTRAN and C, (GG24-3611)

AIX for RISC System/6000: Optimization and Tuning Guide for FORTRAN, C and C++, (SC09-1705)

AIX XL FORTRAN Compiler/6000: Users Guide, (SC09-1610)

AIX XL FORTRAN Compiler/6000: Language Reference, (SC09-1611)

Related Non-IBM Publications

This primer assumes you already understand some basics of parallel computing. For more information about parallel computing, you may want to consult some of the following non-IBM publications:

- Almasi, G., Gottlieb, A. *Highly Parallel Computing*. Benjamin-Cummings Publishing Company, Inc., 1989.
- Bailey, D., Barton, J., Lasinski, T., Simon, H. eds, *NASA Technical Memorandum 103863*, 1993.
- Flynn, M.J. *Very High-Speed Computing Systems* Proc. IEEE, Vol. 29, 1966.
- Hwang, K., Briggs, F.A. *Computer Architecture and Parallel Processing* McGraw-Hill, 1984.
- IEEE Computer Society Technical Committee on Supercomputing Applications and Computer Architecture, *Supercomputing '93 Proceedings*, IEEE Computer Society Press, 1993.
- Perrot, R.H. *Parallel Programming* Addison-Wesley, 1987.

Ordering Additional Copies of This Book

To order additional copies of this book, use order number SH26-7223.

Obtaining This Book's Examples Through Anonymous ftp

Many of the serial and parallel programs used as examples throughout this book are available via anonymous ftp from **tc.cornell.edu** at the Cornell Theory Center. The following instructions describe how to obtain copies of these example programs, as well as other programs, makefiles, shell scripts, include files, and console and profiler output from the *anonymous* user account on **tc.cornell.edu**. From a workstation on the Internet:

ENTER **ftp tc.cornell.edu**

▲ A connection is established between your local machine and **tc.cornell.edu**, which prompts you for the user account.

```
Name(tc.cornell.edu: yourname) :
```

ENTER **anonymous**

▲ **tc.cornell.edu** now prompts you to enter your identity in place of a password.

```
Password (enter ident as password):
```

ENTER your identity – your account name on your local machine. For example, *shu@fort.stre.nyn*.

Once logged in to **tc.cornell.edu**, you will need to change to the directory containing this book's examples:

ENTER **cd pub/aixpe**

The README file in this directory describes all the serial and parallel programs, makefiles, include files, and so on that you can transfer to your local machine. You can selectively **get** just the files you want, or you can transfer the archive file *aixpe_examples.tar* which contains all available files in **tar** format.

Accessing Online Help Information

The Parallel Environment product provides manual pages as well as online help information in InfoExplorer* format. InfoExplorer is the AIX online information facility. You can use InfoExplorer's extensive search capabilities to find help on any part of the Parallel Environment product. When you press **Help** on one of the product's X-Window's applications, the appropriate book in the Parallel Environment library will appear on the screen in InfoExplorer format. In particular, the X-Windows** applications display error messages in their own X-Windows. Pressing **Help** in one of these windows invokes InfoExplorer and opens the appropriate book to the appropriate spot for error resolution.

You can also access the books in InfoExplorer format without using the graphical user interface. To access this book or any other book in this library in InfoExplorer format:

ENTER **info -l pe**

To access a list of manual pages for PE:

ENTER **man pe**

Chapter 1. Introduction

The IBM* AIX* Parallel Environment program product (PE) is a software environment designed for the development and execution of parallel FORTRAN, C, or C++ programs on a distributed memory system. It runs on the RISC System/6000* platform using the AIX operating system (Version 3.2 of later). Specifically, you can use the PE to develop and execute parallel programs on:

- any configuration of the IBM 9076 Scalable POWERparallel Systems* (SP1* and SP2*) as described in the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*. Essentially, a POWERparallel* system is a collection of RISC System/6000 processors grouped into a number of *frames*. Each frame of a POWERparallel system can contain from eight to 16 RISC System/6000 processors.
- a networked cluster of RISC System/6000 processors.
- a *mixed system*. In a mixed system, additional RISC System/6000 processors supplement the processors of a POWERparallel system.

While this book avoids assumptions regarding your specific hardware and configuration, it is assumed that those most interested in performance will be running on a POWERparallel system configured with its optional high-performance switch feature. The optional switch is key to performance as it is designed for optimal message passing on the POWERparallel system – providing low latency and high bandwidth.

The RISC System/6000 processors of your system are called *processor nodes*. A parallel program executes as a number of individual, but related, *parallel tasks* on a number of your system's processor nodes. The group of parallel tasks is called a *partition*. The parallel programming examples in this book follow the Single Program - Multiple Data (or SPMD) model. In this common programming model, the parallel tasks are identical. Each task, in fact, contains the entire parallel program. The processor nodes thus execute the same stored program, but, since conditional statements can vary execution differently from task to task, not necessarily the same instructions.

In a distributed memory system where memory and address space are local to each processor node, data can only be shared among program tasks by passing messages. PE's Message Passing Library, or MPL, provides the necessary application programming interface for communicating data between, and coordinating execution among, the parallel tasks of your program. Similar to the libraries of PVM*** and Express**, the MPL routines enable message passing among parallel tasks. The parallel tasks call the MPL routines, which in turn call Communication Subsystem (CSS) library routines. It is the CSS routines which handle the actual communication among processor nodes.

© Copyright IBM Corp. 1993

Like other message-passing libraries, the MPL offers basic point-to-point message-passing routines. In addition to these, the MPL provides routines for:

- creating and querying logical groups of tasks. Ordering tasks into such *task groups* can provide additional structure and control to the application.
- collective communication within task groups. These routines are designed to meet common collective communication needs among groups of tasks. They eliminate the need for repetitively coding these frequently-needed operations, and simplify them into a single call.

Chapter 2. The Message Passing Library (MPL) Calls

This chapter provides an overview of the MPL subroutine and function calls you can use to parallelize your application. Whether or not you are familiar with other message-passing systems like PVM, this chapter will provide the foundation necessary to write parallel programs using the MPL calls. For complete syntax information, you can then turn to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages. This chapter is divided into the following sections:

- **"Managing Tasks in the Parallel Partition" on page 4** describes the calls that enable your program to manage execution of its tasks in the parallel partition. It describes how you can get and, where appropriate, set the value of certain system variables and constants. It also describes how you can terminate execution of all program tasks to implement your own error-handling routines.
- **"Creating and Querying Task Groups" on page 5** describes the routines you can use to create and query logical groups of tasks called task groups. Such task groups provide a useful way of programming additional structure and control into your program.
- **"Sending and Receiving Messages" on page 6** describes the basic point-to-point message-passing routines. It describes both the non-blocking calls you can use for asynchronous communication between tasks, and the blocking calls you can use to implicitly combine task synchronization with message passing. This section also describes additional calls for sending and receiving non-contiguous messages, and for sorting through, and selectively receiving, messages. Also included is an overview of some common programming pitfalls and how to avoid them. In short, this section describes the flexible capabilities offered by MPL's basic point-to-point message-passing routines.
- **"Performing Collective Communication Within Task Groups" on page 22** describes the collective communication routines that enable communication among all tasks in a task group. It describes the two types of collective communication routines – those for collective message passing among the participating tasks, and those that combine compute operations along with the collective message passing.

Note: There are FORTRAN and C versions of all the routines described in this chapter. The FORTRAN versions are prefixed with **MP_**, while the C versions are prefixed with **mpc_**. For the purpose of easily distinguishing between the routines, FORTRAN names appear in upper case, while C routines appear in lower case. In is important to note that, in actual use, the FORTRAN calls are not case sensitive, while the C calls are. Also, to avoid repetition, the text of this chapter uses the FORTRAN names to refer to both the FORTRAN and C versions.

© Copyright IBM Corp. 1993

Managing Tasks in the Parallel Partition

The parallel partition consists of all the tasks running your program. These tasks are logically ordered by number. The numbering is in ascending order from 0 to one less than the total number of tasks. Each task's number is called its *task identifier* or, simply, its *task id*. A task identifier designates a task's logical position within the parallel partition. You can add a certain degree of structure and control to your program by manipulating tasks by their task identifiers. Conditional processing can, for example, be based on a task's logical position within the partition.

The MPL calls described in this section enable your program to manage execution of its tasks in the parallel partition. They let you get and, where appropriate, set certain system variables and constants. They also let you terminate execution of all program tasks, enabling you to implement your own error-handling routines. Table 1 briefly outlines these *task management subroutines*. For the complete syntax of these calls, refer to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages.

Table 1. Task Management Subroutines

Use this MPL call:	**To:**
MP_ENVIRON mpc_environ	**Query the number of tasks in the parallel partition, and the calling task's logical position relative to the others.** Typically, this routine is called at the top of the application. For a parallel program to be truly scalable, it needs this information in order to decompose and route its data efficiently regardless of the partition size. This routine gives you two integers – one indicating the partition size, and one indicating the task identifier. Conditional processing can then be based on this information. A task, for example, could differentiate its execution based on its task identifier.
MP_STOPALL mpc_stopall	**Terminates all tasks in the parallel partition.** Calling this routine from any one task in the parallel partition terminates execution of all parallel tasks, and an application-supplied integer is also returned to standard error. This enables you to implement your own error-handling routines.
MP_TASK_QUERY mpc_task_query	**Get the value of system variables and constants.** When you call this routine, you supply an integer which specifies the information you are querying. You can use this routine to determine: • whether or not messages will be truncated when the length of a message sent is greater than the size set up for it by the receiving task. Either the message is truncated when this happens, or execution is terminated. You can set the truncation mode by calling MP_TASK_SET. • the lower and upper bound allowed for the optional user message types on the point-to-point message-passing calls. See “How Tasks Pass Messages” on page 6 for more information on message types. • the value of the system wildcards **dontcare**, **allmsg**, **nulltask**, and **allgrp** that you can use on some MPL calls. For more information on system-specific wildcards, see page 8.
MP_TASK_SET mpc_task_set	**Set system-dependent variables.** This is typically called at the top of your program, and can be used to set: • the truncation mode. This mode determines whether or not messages will be truncated when the length of a message sent is greater than the size set up for it by the receiving task. Either the message is truncated when this happens, or execution is terminated. • set the mode of execution to either *run* or *develop*. In *run* mode, only minimal error checking is performed. Because it has less impact on a program's performance, this mode is intended for fully-tuned programs running in production environments. *Develop* mode does more detailed error checking that can effect performance significantly. It is intended for early stages of application development.

Creating and Querying Task Groups

This section describes the MPL routines your program can use to create and query logical groups of tasks called *task groups*. Ordering tasks into task groups is a useful way of providing additional structure and control to the application. You could, for example, use task groups to build rows in a two-dimensional array. The task groups your program creates will consist of a number of tasks logically ordered from 0 to one less than the total number of tasks in the group. Each task's number designates its logical position within the group and is called its *rank*.

You can create task groups by:

- specifically naming the tasks you want to include by their task identifiers.
- dividing an existing task group into subgroups.

For identification, each task group has both an integer label supplied by the application and a task group identifier defined by the system. These identifiers are used by other MPL routines associated with task groups. In particular, a task group identifier must be specified on all the MPL collective communication routines described in "Performing Collective Communication Within Task Groups" on page 22.

There is one standard, pre-defined task group. This system-defined group consists of all tasks sorted according to their task identifiers. In this group, a task's rank is therefore the same as its task identifier. The integer label for this pre-defined task group is 0. Its task group identifier is equal to the system specific parameter, or *wildcard* value, **allgrp**. To obtain the value of **allgrp** and other system-specific wildcards, you can use the MP_TASK_QUERY routine described in Table 1 on page 4. For more information on system-specific wildcards, see page 8.

Table 2 describes the two routines for creating new task groups.

Table 2. Routines For Creating Task Groups

Use this MPL call:	To:
MP_GROUP mpc_group	**Create a new task group by explicitly identifying the tasks to be grouped.** All tasks to be included in the new group must issue corresponding MP_GROUP calls. Each task that calls this routine must specify: • an array of task identifiers indicating the tasks to be grouped. • an integer indicating the number of tasks in the new group. • an integer label identifying the group. This routine gives you the system-defined task group identifier.
MP_PARTITION mpc_partition	**Create two or more new task groups by partitioning an existing task group into subgroups.** All the tasks in the already existing task group must issue corresponding MP_PARTITION calls in order to create the new task groups. Each task that calls this routine must specify: • the task group identifier of the original, parent, task group. • an integer indicating the calling task's rank within the new child task group. • an integer label identifying the new child task group to which this task will belong. This routine gives you the system-defined task group identifier for the new child task group to which the calling task belongs.

Table 3 on page 6 describes the MPL routines for querying information about task groups and their member tasks.

Table 3. Routines For Querying Information Regarding Task Groups	
Use this MPL call:	**To:**
MP_GETSIZE mpc_getsize	**Get the size of an existing task group.** When you call this routine, you specify a task group identifier. The routine gives you an integer indicating the size of that group.
MP_GETMEMBERS mpc_getmembers	**Get an array of task identifiers indicating the members of an existing task group.** When you call this routine, you specify a task group identifier. The routine gives you an integer array containing the ordered membership list for that group.
MP_GETLABEL mpc_getlabel	**Get the label of an existing task group**. When you call this routine, you specify a task group identifier. The routine gives you that group's label integer.
MP_GETRANK mpc_getrank	**Get the rank of a task in an existing task group.** When you call this routine, you specify a task group identifier. The routine gives you that task's rank within the indicated group.
MP_GETTASKID mpc_gettaskid	**Get the task identifier for a task of a certain rank within an existing task group.** When you call this routine, you specify a task group identifier and an integer indicating a particular task's rank within that group. The routine gives you the task identifier for the task of that rank.

Notes:

1. The task group routines are especially useful when used in conjunction with the collective communication routines. For a description of the collective communication routines, see “Performing Collective Communication Within Task Groups” on page 22.
2. For the complete syntax of the calls described in the previous two tables, refer to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages.

Sending and Receiving Messages

This section describes the basics of message passing. Specifically, it describes the point-to-point message-passing routines for sending messages from one task to another. Additional routines for collective communication are then discussed in “Performing Collective Communication Within Task Groups” on page 22. This section contains descriptions of:

- how messages are passed from one task to another.
- the *system specific parameters*, or *wildcard values* you can use on certain routines for added flexibility in message passing. In particular, the **dontcare** wildcard for flexibility in receiving messages is described.
- the two types of basic message passing operations – non-blocking and blocking.
- an overview of the point-to-point message-passing routines.
- some common programming pitfalls encountered when using the point-to-point message-passing routines, and how to avoid them.

How Tasks Pass Messages

The tasks of your parallel program communicate data with each other via message passing. A *message* is a sequence of bytes sent as one logical unit from one task to another. The sending task identifies the application buffer that contains the message to send, and the receiving task identifies the application buffer into which the message will be received. An application buffer is simply a program array or a structure – it's any area in memory defined by your application.

To send a message, the sending task identifies the application buffer containing the message, its size in bytes, and its destination – the receiving task. The sending task can optionally provide an integer indicating a message's type. The integer is used to provide the receiving task with some additional information about the message. For example, it might indicate the priority of a message. Or if you are sending elements of an array to different tasks, it might indicate the column or row number of the array.

Note: If you plan to use message types, note that there is a lower and upper bound on allowable user message types. This is because the system itself uses a certain number of types to implement its own internal message passing. To determine the lower and upper bound for user message types, you must call the MP_TASK_QUERY routine described in Table 1 on page 4.

On the receiving side, the task identifies the application buffer for the incoming message and its size in bytes. Again, the application buffer is just an array or structure. For consistency, it is a good programming practice to identify the same kind of array or structure as the application buffer on both the sending and receiving side. A task can select the message it wants to receive into the application buffer by source and type.

Note: Using a non-restrictive wildcard value for source or type enables a task to receive a message by its source or type alone. Using this non-restrictive wildcard value in place of both the source and the type enables a task to receive the next message available. See “Using the dontcare Wildcard When Receiving Messages” on page 8 for more information.

The process of receiving a message is often referred to as “posting a receive” because it may be a pending operation. A task can call a receive routine and continue processing before a message meeting its specifications is actually received into the application buffer. In fact, a receive can be posted for a message before it has been sent.

When a call is made to send a message, the MPL routine transfers it as a number of packets into a temporary communication buffer on the sending side. It is important that the task does not modify the array or structure identified as the application buffer before the message has been transferred into the communication buffer. If the task does modify the array or structure, this could corrupt the message. The message that eventually arrives at the receiving task will not be what you had intended to send. Once the message is transferred to the communication buffer, the send operation is considered complete. There are then no further constraints on modifying the application buffer. It is important to note, however, that the send and receive operations are local, and the “completion” of these operations refers only to the local view. A completion of a send operation, therefore, does not mean that the message passing is concluded. It only indicates that the message has been copied into the communication buffer on the sending side. The message is then transferred in a number of packets to a communication buffer on the receiving side provided there is enough available buffer space. The message passing is not concluded until a matching receive is completed by the receiving task.

On the receive side, this operation is reversed. A call is made to post a receive. As soon as a message meeting the posted specifications is available in the communication buffer on the receiving side, it is transferred in a number of packets

to the receiving task's application buffer. Just as the sending task should not modify the application buffer until the send operation completes, the receiving task should not try to work with it until the receive operation completes – until the message has actually been transferred into the array or structure identified as the receiving application buffer.

Using the dontcare Wildcard When Receiving Messages

A *wildcard value*, also known as a *system specific parameter*, is a value you can use to represent one or more variables. There are three of these wildcard values you can use on select point-to-point MPL routines. They are the **dontcare**, **allmsg**, and **nulltask** wildcards. There is also one wildcard, discussed in "Creating and Querying Task Groups" on page 5, associated with task groups and collective communication routines. Your program can obtain the system-specific value of any of the wildcards by calling the MP_TASK_QUERY routine described in Table 1 on page 4. The routine returns the requested wildcard value.

The most common wildcard is the **dontcare** wildcard, which can be used on several of the basic point-to-point routines associated with receiving messages. If you do not use this wildcard, the receive call must specify the source and type of the message it wants to receive. For additional flexibility in receiving messages, however, you can set a variable to the **dontcare** wildcard value and use it in place of:

- *the source*. In this case, the task "receives by type." The next message of the posted type available in the receiving side's communication buffer is received.
- *the type*. In this case, the task "receives by source". The next message from the posted source available in the receiving side's communication buffer is received.
- *both the source and type*. In this case, the next available message is received regardless of its type and source.

In addition, the variable set to the **dontcare** wildcard will, once the message arrives, then contain the type or source of the message received. This enables the program task to check the new value and base conditional processing on it.

Using the **dontcare** wildcard variable enables a program task to receive messages more flexibly since a single posted receive could be satisfied by a variety of sources and types. This allows the program task to receive messages as they become available rather than possibly having to wait for a more-specific posted receive to be satisfied. In order to avoid message truncation when using **dontcare** wildcard variables, the receive buffer must be large enough for any potential message.

You can use a variable set to the **dontcare** wildcard value in the MP_RECV, MP_BRECV, and MP_BVRECV calls for receiving messages. You can also use a **dontcare** wildcard variable on the MP_WAIT and MP_PROBE calls associated with receiving messages. All these routines are discussed later in this section. You can also refer to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages for more information.

Notes:

1. Although the **dontcare** wildcard is the most common, there are two other wildcards that you can use on the basic point-to-point message-passing routines. They are:
   - the **nulltask** wildcard value for use on the MP_BSENDRECV routine. See the description of this routine in Table 5 on page 13, or consult its on-line manual page for more information.
   - The **allmsg** wildcard value for use on the MP_WAIT routine. See the description of this routine in Table 4 on page 11, or consult its on-line manual page for more information.
2. A number of programming errors can result from incorrect use of the **dontcare** wildcard. See "Avoiding Programming Pitfalls in Point-to-Point Message Passing" on page 15 for more information.

Non-Blocking and Blocking Routines

There are two types of send and receive operations – *non-blocking* and *blocking*.

- *non-blocking* routines operate asynchronously. Non-Blocking routines return as soon as they complete initialization, and processing continues. It is your program's responsibility not to corrupt the message in the application buffer on the sending side. On the receiving side, it is your program's responsibility to ensure that it does not attempt to use the message before it is actually available.
- *blocking* routines do not return until the message has been safely transferred to, or from, the communication buffer. On the sending side, blocking routines protect the application buffer from being corrupted. You cannot overwrite or delete the message data because no other work can be done until the call completes. On the receive side, the blocking routines ensure that you do not try to use the message data before it is actually available. You can also use the blocking subroutines to combine task synchronization with the message transfer. If you use blocking routines on both the send and receive sides, the calls form loose and asymmetric synchronization points between the two tasks. In other words, the receiving task is guaranteed that the sending task has passed its send point. Real synchronization can be achieved using a blocking receive on both tasks.

While less safe in terms of message integrity, the non-blocking routines afford you more flexibility in coding your program, thus enabling you to more easily tune your program for peak performance. You generally obtain better performance using the non-blocking routines, because a program task can continue working while the system takes care of the communication. And while the non-blocking sends and receives do not themselves wait for the operation to complete, there are additional routines you can use for this. Your program can react to an incomplete non-blocking call by querying its status, and either continuing with non-dependent work, or forcing the program to wait. In short, the price for the improved performance afforded by non-blocking routines is extra care on your part.

The following table summarizes the advantages and disadvantages of the non-blocking and blocking routines.

	non-blocking routines	blocking routines
advantages	Greater flexibility in optimizing program for peak performance.	• Safe in terms of maintaining message integrity. Avoids the possibility of overlaying the application buffer on the sending side or trying to use it on the receiving side before the message is available. • Loosely-synchronous nature of calls makes it easier to conceptualize, in time, what the program tasks are doing. Debugging is, therefore, easier.
disadvantages	• Does not protect you from using the application buffers before the non-blocking operation is complete. • Lack of forced synchronization points can make it harder to conceptualize, in time, what the program tasks are doing. This can make the program harder to debug.	Peak performance can be degraded if tasks wait for send and receive operations to complete while they could be doing other work.

Overview of the Basic Message-Passing Routines

The descriptions of the basic MPL routines you can use for point-to-point message passing are divided under the following headings.

- **"Using Non-Blocking Message-Passing Routines"** describes the non-blocking send and receive routines. Also covered are needed routines for checking the status of non-blocking operations and for making a task wait until a specified non-blocking operation completes.
- **"Using Blocking Message-Passing Routines" on page 12** describes the basic blocking send and receive calls. Also covered is a routine that combines a blocking send and receive into one operation.
- **"Sending and Receiving Non-Contiguous Messages" on page 14** describes the MPL calls you can use for sending messages consisting of bytes not contained in a consecutive series of storage locations.
- **"Probing Through Messages Available to be Received" on page 14** describes an MPL call that enables you to probe to see what messages are available to be received, but do not yet have a matching receive posted.

Keep in mind as you read through the summary of these routines that they are divided under these different headings for presentation only. A message could, for example, be sent using a blocking routine, while its posted receive could be non-blocking.

Using Non-Blocking Message-Passing Routines

Table 4 on page 11 shows you the four basic MPL calls for non-blocking message passing. If these were the only routines available, they would be enough to parallelize your applications. They can be viewed as forming the foundation of the MPL, because most of the other routines can be seen as building on these four. For the complete syntax of these calls, refer to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages.

Table 4. Non-Blocking Message-Passing Routines

Use this MPL call:	To:
MP_SEND mpc_send	**Send a message and continue processing** This routine identifies the application buffer containing the message to send and continues processing. It does not wait for the send operation to complete. It is therefore your program's responsibility not to corrupt the outgoing message before it has been transferred out of the application buffer. Before modifying the array or structure identified as the sending application buffer, the task should check that the send operation is complete using MP_STATUS. If the task cannot continue useful work without modifying the send buffer, it should block further execution using MP_WAIT until the send operation completes. When you call this routine, you identify a particular array or structure as the application buffer, its size in bytes, and its destination – the receiving task. To provide the receiving task with additional information on the message being sent, you can optionally specify an integer indicating a message type. This routine gives you a message identifier that can be used by the MP_STATUS or MP_WAIT routine to check or wait for the completion of the send operation. The send operation is complete once the message is transferred from the application buffer to the communication buffer.
MP_RECV mpc_recv	**Post a receive for a message and continue processing.** This routine does not wait for the receive operation to complete. It is therefore your program's responsibility not to attempt to use the message data before it is actually available – before it has been transferred into the array or structure identified as the receiving application buffer. Before using the message data, the task should check that the send or receive operation is complete using MP_STATUS. If the task cannot continue useful work without the message data, it should block further execution using MP_WAIT until the receive operation completes. When you call this routine, you identify the application buffer for the incoming message and its size in bytes. You can select the message to receive by specifying both its source and type or, using a **dontcare** wildcard variable, by source or type alone. Using **dontcare** wildcard variables for both source and type enables the task to receive the next available message. If a **dontcare** wildcard variable was used in place of source or type, the actual source or type of the message received is reported when the receive completes. The actual number of bytes received is reported by the MP_STATUS or MP_WAIT routine.
MP_STATUS mpc_status	**Check to see if a specified non-blocking send or receive operation has completed.** This routine checks the status of the operation and continues processing. When you call this routine, you provide the message identifier returned by a non-blocking send or receive routine. This routine returns a value of -1 if the specified operation has not completed. If the specified operation has completed, it returns the number of bytes either sent or received by the operation.
MP_WAIT mpc_wait	**Block further execution of the calling task until a specified non-blocking send or receive operation has completed.** When you call this routine, you provide either: • the message identifier returned by a non-blocking send or receive routine. Execution is then blocked until that operation has completed. • a variable set to the **dontcare** wildcard value. This specifies the next non-blocking send or receive operation. Execution is blocked only until the next non-blocking send or receive operation completes. • a variable set to the **allmsg** wildcard value. This wildcard can only be used on this routine. Execution is blocked until all pending send and receive operations complete. This routine gives back the number of bytes in the message sent or received. If you used the **allmsg** wildcard, this is the number of bytes in the last of the messages sent or received. If either wildcard was used, **MP_WAIT** also returns a message id.

Notes:

1. Your program can obtain the system-specific value of the **dontcare** or **allmsg** wildcards mentioned in the previous table by calling the MP_TASK_QUERY routine. This routine is described in Table 1 on page 4. For additional information about the **dontcare** wildcard, see "Using the dontcare Wildcard When Receiving Messages" on page 8.
2. There is an additional routine – MP_PROBE – you can use on the receive side to check for the arrival of messages for which no matching receive has yet been posted. Because you can use this routine to check for the arrival of messages sent by both non-blocking and blocking routines, it is not discussed here. See "Probing Through Messages Available to be Received" on page 14 for more information.

Using Blocking Message-Passing Routines

Table 5 on page 13 shows the three basic blocking calls for point-to-point message passing. These calls can be seen as building on the non-blocking calls described in Table 4 on page 11. A blocking send, for example, can be seen as a call to MP_SEND immediately followed by a call to MP_WAIT. For the complete syntax of these calls, refer to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages.

Table 5. Blocking Message-Passing Routines

Use this MPL call:	To:
MP_BSEND mpc_bsend	**Send a message and block.** This routine identifies the application buffer containing the message to send and blocks execution of the calling task until the message is transferred out of the application buffer. This routine is, therefore, logically equivalent to calling **MP_SEND** followed by **MP_WAIT**. It ensures that the task does not unintentionally corrupt the message before it is sent. When you call this routine, you identify a particular array or structure as the application buffer, its size in bytes, and its destination – the receiving task. To provide the receiving task with additional information on the message being sent, you can optionally specify an integer indicating a message type.
MP_BRECV mpc_brecv	**Post a receive and block.** This routine identifies the application buffer for the incoming message, and blocks execution of the receiving task until the message is transferred into that buffer. This routine is, therefore, logically equivalent to calling **MP_RECV** followed by **MP_WAIT**. It ensures that the task does not attempt to use the message data before it is actually available. When you call this routine, you identify the application buffer for the incoming message and its size in bytes. You can select the message to receive by specifying both its source and type or, using a **dontcare** wildcard variable, by source or type alone. Using **dontcare** wildcard variables for both source and type enables the task to receive the next available message. This routine gives the number of bytes received and, if a **dontcare** wildcard variable was used in place of source or type, the actual type or source of the message received.
MP_BSENDRECV mpc_bsendrecv	**Send a message and post a receive before blocking.** This routine combines in one call the sending of a message to one task and the receiving of another message from that same task or another. Specifically, you can use this call to: • exchange messages between two tasks • execute a shift operation across a number of tasks. This might be done, for example, to mimic different topologies such as a ring structure. You can only use the MP_BSENDRECV routine to pass messages with another MP_BSENDRECV routine. You cannot mix MP_BSENDRECV calls on one task with ordinary non-blocking or blocking send and receive calls on another task. When you call this routine, you identify two application buffers – one containing the message to send, and one containing the message to receive – and their size in bytes. It is important that these buffers are distinct; they must not overlap in memory. You must specify a message type on this call and cannot use a **dontcare** wildcard variable. The type specified applies to both the outgoing and incoming messages. You specify the destination for the message you are sending and the source of the message to receive. You can use a **dontcare** wildcard variable in place of source in which case the task will receive by type. You can also use the wildcard **nulltask** in place of specifying either the source or the destination. This allows a task to only participate in one half of the send/receive operation. This is particularly useful when executing shift operations. If mimicking a ring topology and shifting data “up”, for example, you might not wish task 0 to receive a message from the last task.

Notes:

1. Your program can obtain the system-specific value of the **dontcare** or **nulltask** wildcards mentioned in the previous table by calling the MP_TASK_QUERY routine. This routine is described in Table 1 on page 4. For additional information on the **dontcare** wildcard, see “Using the dontcare Wildcard When Receiving Messages” on page 8.
2. There is an additional routine – MP_PROBE – you can use on the receive side which checks for the arrival of messages for which a receive has not been posted. Because you can use this routine to check for the arrival of messages sent by both blocking and non-blocking routines, it is not discussed here. See “Using the dontcare Wildcard When Receiving Messages” on page 8 for more information.

Sending and Receiving Non-Contiguous Messages

Table 6 shows the four basic calls associated with sending and receiving non-contiguous messages. Many applications will need to transmit data stored in non-contiguous memory locations. These non-contiguous message-passing routines are needed, for example, if your application deals with sparse data structures. You can send non-contiguous data by packing it into a contiguous buffer on the sending side, and unpacking it again on the receiving side. Or you can use the special blocking send and receive designed for sending non-contiguous messages. For the complete syntax of these calls, refer to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages.

Table 6. Routines For Sending and Receiving Non-Contiguous Messages

Use this MPL call:	To:
MP_PACK mpc_pack	**Pack data from a non-contiguous buffer into a contiguous buffer for sending.** When you call this routine, you specify the starting location of the buffer containing the initial non-contiguous data and the buffer into which the packed data is to be placed. For the original non-contiguous buffer, you must also specify the number of blocks it contains, their size in bytes, and the offset space, in bytes, between each.
MP_UNPACK mpc_unpack	**Unpack data from a contiguous buffer into a non-contiguous buffer on the receive side.** When you call this routine you specify the buffer containing the data to be unpacked and the starting location of the buffer into which the unpacked data is to be placed. For the non-contiguous buffer, you must also specify the number of blocks it will contain, their size in bytes, and the offset space, in bytes, between each.
MP_BVSEND mpc_bvsend	**Send a non-contiguous message and block.** This routine identifies the application buffer containing the non-contiguous message to send, and blocks execution of the calling task until the message is transferred out of that buffer. When you call this routine, you specify the starting location of the buffer containing the message to send and its destination. You must specify the number of blocks contained in the message, their size in bytes, and the offset space, in bytes, between each. To provide the receiving task with additional information on the non-contiguous message being sent, you can optionally specify an integer indicating a message type.
MP_BVRECV mpc_bvrecv	**Post a receive for a non-contiguous message and block.** This routine identifies the application buffer for the incoming non-contiguous message, and blocks execution of the calling task until the message is transferred into that buffer. When you call this routine, you specify the starting location of the buffer containing the message to receive. You must specify the number of blocks contained in the message, their size in bytes, and the offset space, in bytes, between each. You can select the non-contiguous message to receive by specifying both its source and type or, using a **dontcare** wildcard variable, by source or type alone. Using **dontcare** wildcard variables for both source and type enables the task to receive the next available non-contiguous message.

Note: Your program can obtain the system-specific value of the **dontcare** wildcard mentioned in the previous table by calling the MP_TASK_QUERY routine. This routine is described in Table 1 on page 4. For additional information on the **dontcare** wildcard, see “Using the dontcare Wildcard When Receiving Messages” on page 8.

Probing Through Messages Available to be Received

MP_PROBE is a useful routine you can call on the receive side to learn about the messages that are available to be received. MP_PROBE is a way of checking for messages that have arrived in the communication buffer on the receive side, but do not yet have a matching receive posted. The routine does not receive a message, but it does determine that a message matching particular specifications has arrived.

When you call this routine, you probe for a message by its source and type. Using a **dontcare** wildcard variable enables a task to probe for a message by its source or type alone. Using **dontcare** wildcard variables for both source and type enables the task to check if any messages, regardless of source or type, are available to be received.

The routine gives the number of bytes contained in the first queued message satisfying the probe and, if a **dontcare** wildcard variable was used in place of source or type, the actual type or source of the message. This information makes MP_PROBE particularly useful when the task needs to receive a message but does not know the message's length. MP_PROBE is also useful in an event-driven application where the action taken by a program task depends on the type of message to be received.

For the complete syntax of the call, refer the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages. For more information on the **dontcare** wildcard variable, see “Using the dontcare Wildcard When Receiving Messages” on page 8.

Avoiding Programming Pitfalls in Point-to-Point Message Passing

This section discusses some typical pitfalls you can run into when programming with the point-to-point MPL routines. There are two classes of programming pitfalls – those that make the program incorrect, and those that make the program unsafe.

- An *incorrect* program is one that will either not run to completion or will not produce accurate results.
- An *unsafe* program can run to completion and produce accurate results, but makes unnecessary demands on the system that could, on some runs, lead to abnormal termination, deadlock conditions, or inaccurate results.

The Remainder of this section provides some tips for ensuring that your programs are both safe and correct.

“Avoiding Program Deadlock and Buffer Overflow Conditions” describes incorrect and unsafe programming that could cause communication buffer overflow and program deadlock.

“Avoiding dontcare Wildcard Pitfalls” on page 19 describes the correct and safe way to use **dontcare** wildcard variables when receiving messages.

Avoiding Program Deadlock and Buffer Overflow Conditions

The following SPMD program excerpt (Example A-1) illustrates an incorrect program that causes deadlock. The program has arranged tasks in a logical ring, and is attempting a circular shift of data around the ring. Each task is sending a message to the task on its right, and receiving a message from the task on its left. The program is incorrect because the order of the send and receive operations was inverted by mistake. A blocking receive call is made before the message is actually sent. Therefore, all the tasks block on the receive call and no message is ever sent.

Example A-1 (Incorrect)

```
CALL MP_ENVIRON (NUMPE, ME)
IF (ME .LT. NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
CALL MP_BRECV(INMSG1, MSGLEN, SOURCE, TYPE, NBYTES)
CALL MP_BSEND(OUTMSG1, MSGLEN, RIGHT, TYPE)
```

The program is incorrect because it will never complete.

The following two examples are correct versions of this same circular shift program. Both of the following examples are unsafe, however, because they make unnecessary assumptions and demands on the amount of available system buffer space.

The first unsafe but correct example (Example A-2) solves the deadlock problem by replacing the blocking receive call with a non-blocking receive call. The tasks are not blocked at the call, and so can continue on to send the message.

Example A-2 (Correct but Unsafe)

```
CALL MP_ENVIRON (NUMPE, ME)
IF (ME .LT. NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
CALL MP_RECV(INMSG1, MSGLEN, SOURCE, TYPE)
CALL MP_BSEND(OUTMSG1, MSGLEN, RIGHT, TYPE)
```

The second correct but unsafe solution to the circular-shift problem (Example A-3) simply reverses the order in which it calls the blocking routines.

Example A-3 (Correct but Unsafe)

```
CALL MP_ENVIRON (NUMPE, ME)
IF (ME .LT. NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
CALL MP_BSEND(OUTMSG1, MSGLEN, RIGHT, TYPE)
CALL MP_BRECV(INMSG1, MSGLEN, SOURCE, TYPE, NBYTES)
```

Both of these corrected versions of the circular-shift program are still unsafe because they both make unnecessary assumptions regarding the amount of available system buffer space. The success of the program depends on the size of the messages sent and on the amount of memory available. If there is insufficient buffer space, this will result in program deadlock or abnormal termination caused by buffer overflow. The problem is that buffer space is required for both incoming and outgoing messages and all tasks are sending and receiving at the same time. Each node has to be able to hold the information being sent to it, as well as the information coming from it. If the messages being shifted in this example are just slightly larger than half the size of the available buffer space for any one node, buffer overflow will result.

The following two examples are correct and safe versions of the circular-shift example. There is no difference in the function of the code. It has simply been reorganized to make more efficient use of buffer space.

The first correct and safe solution (Example A-4) codes the circular shift so that odd and even tasks alternate send and receive operations. The only way this example could cause a buffer overflow is if a single message is larger than the available buffer space. This is hardly an unreasonable demand on buffer space.

Example A-4 (Correct and Safe)

```
CALL MP_ENVIRON (NUMPE, ME)
IF (ME .LT. NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
IF ((ME \ 2) .EQ. 0)
   CALL MP_BSEND(OUTMSG1, MSGLEN, RIGHT, TYPE)
   CALL MP_BRECV(INMSG1, MSGLEN, SOURCE, TYPE, NBYTES)
ELSE
   CALL MP_BRECV(INMSG1, MSGLEN, SOURCE, TYPE, NBYTES)
   CALL MP_BSEND(OUTMSG1, MSGLEN, RIGHT, TYPE)
ENDIF
```

The second safe and correct solution (Example A-5) codes the circular shift using a combined blocking send and receive call. This is the both the easiest and safest version of the circular shift, because the routine itself has internal buffer management schemes that minimize the demands on system-buffer space.

Example A-5 (Correct and Safe)

```
CALL MP_ENVIRON (NUMPE, ME)
IF (ME .LT. NUMPE-1)
   THEN RIGHT = ME+1
   ELSE RIGHT = 0
END IF
CALL MP_BSENDRECV(OUTMSG1, MSGLEN, RIGHT, TYPE, INMSG1, MSGLEN, SOURCE, NBYTES)
```

To further illustrate programming pitfalls that can cause deadlock and buffer overflow, consider the following example:

Example B-1 (Correct but Unsafe)

```
CALL MP_ENVIRON (NUMPE,ME)
IF (ME.EQ.1)
   THEN
     DO I = 1, LARGE
       DEST=2
       TYPE=I
       CALL MP_SEND(OUTMSG(I), MSGLEN, DEST, TYPE)
     END DO
     CALL MP_WAIT(ALLMSG)
   ELSE IF (ME.EQ.2)
     DO I = LARGE, 1, -1
       SOURCE=1
       TYPE=I
       CALL MP_BRECV(INMSG(I), MSGLEN, SOURCE, TYPE, NBYTES)
     END DO
END IF
```

The previous example (Example B-1) is correct, but unsafe. It will probably cause a buffer overflow because of the unrealistic assumptions it makes regarding available buffer space. The problem in this case is introduced by the unsafe way the messages are received by type. The messages are sent with types starting at 1 and, with each loop iteration, incrementing their way up to the value of `LARGE`. The messages are received by type in reverse order – starting at `LARGE` and, with each loop iteration, decrementing its way down to 1. This means that all the messages must be sent before any of them can be receive.

This next example (Example B-2) makes the program safe by reversing the order in which messages are received.

Example B-2 (Correct and Safe)

```
CALL MP_ENVIRON (NUMPE,ME)
IF (ME.EQ.1)
   THEN
     DO I = 1, LARGE
       DEST=2
       TYPE=I
       CALL MP_BSEND(OUTMSG(I), MSGLEN, DEST, TYPE)
     END DO
   ELSE IF (ME.EQ.2)
     DO I = 1, LARGE
       SOURCE=1
       TYPE=I
       CALL MP_BRECV(INMSG(I), MSGLEN, SOURCE, TYPE, NBYTES)
     END DO
END IF
```

To Avoid Deadlock and Buffer Overflow Conditions:

- Do not call a blocking routine that can never be satisfied.
- Guarantee that you have a receive for every message sent.
- Do not allow a number of messages to accumulate without being received. Attempt to receive messages as soon as they are available in the communication buffer on the receiving side. If possible, have the receive posted before the message arrives.

Avoiding dontcare Wildcard Pitfalls

Several programming pitfalls are the result of incorrect use of the **dontcare** wildcard. As described in “Using the dontcare Wildcard When Receiving Messages” on page 8, you can set a variable to the value of this system specific parameter. You can then use a **dontcare** wildcard variable on the MP_RECV, MP_BRECV, and MP_BVRECV calls in place of specifying either the type or source of the message to receive. You can also use **dontcare** wildcard variables on the MP_PROBE call to give you similar flexibility in message probing. On the MP_WAIT call, you can use a **dontcare** wildcard variable so a task blocks only until the next non-blocking send or receive operation completes. In any case, the variable used as the **dontcare** wildcard will be overwritten. On the receive calls, the variable will then contain either the type or source of the message received. On the MP_PROBE call, the variable will contain either the type or source of the message that satisfied the probe operation. On MP_WAIT, the variable will then contain the message identifier that satisfied the wait operation. This allows your program to check the value of the variable and know what message has been received or has satisfied the probe or wait operation.

If you use the **dontcare** wildcard on the MP_RECV routine, there are some safety issues you must consider. Assume the variable *msg_src* is set to the **dontcare** wildcard value, and used to specify the source in the MP_RECV routine. The MP_RECV routine cannot overwrite the value of *msg_src* until the message arrives. It will, however, record the address of *msg_src* so it can eventually be updated with the task identifier of the sending task.

To Avoid dontcare Wildcard Pitfalls with MP_RECV:

- In this example, keep in mind that the storage for *msg_src* must be persistent from the point of the MP_RECV call to the return from the corresponding call to MP_WAIT. If *msg_src* is on the stack, then a subroutine which issues a wildcard MP_RECV must not return until it confirms the message is received using the MP_WAIT or MP_STATUS routines. This is true even if it has no interest in the final value of *msg_src*.
- In this example, keep in mind that the application should not use *msg_src* for any other purpose between the wildcard MP_RECV and the corresponding MP_WAIT or MP_STATUS. If simultaneous wildcard MP_RECV calls are pending, each must have its own variable for the wildcard parameter.
- In optimized C code, consider declaring *msg_srv* as volatile to guarantee that future reference to *msg_src* is reloaded from storage. Although FORTRAN does not have a data attribute like C's volatile, you could employ some trick like calling a dummy subroutine with *msg_src* as a parameter after the calls to MP_WAIT. This would ensure that any future reference would see the updated storage value of *msg_src*. A program which makes no reference to *msg_src* between the call to the MP_RECV passing *msg_src* and the return from MP_WAIT is probably safe even without this precaution.

Note: The same considerations outlined in the previous list also apply when you use a **dontcare** wildcard variable for message type.

Other programming pitfalls can occur with **dontcare** wildcard variables when you do not remain attentive to the fact that the variable will be overwritten upon completion of the receive, probe, or wait operation. The following example (Example C-1) represents such a pitfall.

Example C-1 (Unsafe, Probably Incorrect)

```
ntype = dontcare
source = 5
call MP_BRECV(inmsg,msglen,source,ntype,nbytes)
call MP_BRECV(inmsg,msglen,source,ntype,nbytes)
```

The flaw in the previous example is that the variable `ntype` is used as the type in both blocking receives with the assumption that its value is the **dontcare** wildcard. This is only true for the first blocking receive. It will receive the next message of any type from task 5. Upon completion of the first blocking receive, however, the variable `ntype` is overwritten with the type associated with the message received. The second call will block until a message of that same type is available to be received. This could deadlock the program, or result in a buffer overflow condition. At any rate, it is not what the programmer had intended, and the program will probably not complete successfully. This next example (Example C-2) corrects the problem by resetting the variable `ntype` back to the system-specific **dontcare** value before its subsequent use.

Example C-2 (Correct and Safe)

```
source = 5
ntype = dontcare
call MP_BRECV(inmsg,msglen,source,ntype,nbytes)
ntype = dontcare
call MP_BRECV(inmsg,msglen,source,ntype,nbytes)
```

If your program will later need to know the type of both the messages received, you could first keep the type of the first message received in a separate variable. For example:

Example C-3 (Correct and Safe)

```
source = 5
ntype = dontcare
call MP_BRECV(inmsg,msglen,source,ntype,nbytes)
nrecvtype = ntype
ntype = dontcare
call MP_BRECV(inmsg,msglen,source,ntype,nbytes)
```

The following example (Example D-1) illustrates another case where the programmer did not remain attentive to the fact that the variable set to the **dontcare** value will be overwritten. In order to receive the next available message of any type from any source, the programmer uses the **dontcare** value for both type and source.

Example D-1 (Unsafe)

```
idontcare = dontcare
call MP_BRECV(inmsg,msglen,idontcare,idontcare,nbytes)
```

The blocking receive will complete. The problem is that the variable `idontcare` is replaced by either the type or the source of the message received. You cannot tell which. If the program truly does not care what it receives, there is no problem here. If, however, it checks the new value of `idontcare` after completion of the operation expecting to learn the type or source of the message received, an error could result. The safe way to code the previous example is to set two separate variables to the **dontcare** wildcard value.

Example D-2 (Safe)

```
ntype = dontcare
nsource = dontcare
call MP_BRECV(inmsg,msglen,nsource,ntype,nbytes)
```

To Avoid dontcare Wildcard Pitfalls Involving Overwriting:

- Do not use the same **dontcare** wildcard variable more than once, expecting it to retain the **dontcare** value. If you do use the same variable more than once, make sure you reset it to the **dontcare** value for each subsequent use.
- Avoid using the same **dontcare** wildcard variable as both **type** and **source** in a receive call.

Performing Collective Communication Within Task Groups

This section describes the higher-level MPL routines you can use for collective communication among all the tasks in a participating task group. These collective communication routines are implemented using the point-to-point message-passing routines already described, and are designed for high-performance and programming flexibility. These routines can, using the pre-defined task group consisting of all tasks, operate over all the tasks in your parallel partition. They can also, using a task group you have defined, operate over a subset of your partition. Before reading this section, you should be familiar with task groups as described in "Creating and Querying Task Groups" on page 5.

The collective communication routines are designed to meet common collective communication needs that arise frequently in parallel computing. They eliminate the need for repetitively coding these frequently-needed operations, and simplify them into a single call. While you could implement the same functionality these routines provide using the point-to-point message-passing routines, it is easier and safer to take advantage of the collective communication routines. The collective communication routines are safer, for example, because they have internal buffer-management schemes which minimize the demands on buffer space to avoid buffer overflow conditions. In addition, all collective communication routines are blocking so they are safer in terms of message integrity.

There are two basic types of collective communication routines – those for collective message passing among tasks in a task group, and those that combine compute operations with the collective message passing. "Performing Collective Message Passing Within a Task Group" summarizes the collective communication routines for collective message passing. The routines that preform compute operations in addition to message passing are summarized in "Performing Collective Compute Operations Within Task Groups" on page 27.

Performing Collective Message Passing Within a Task Group

The MPL collective communication routines for collective message passing enable you to:

- Broadcast a message from one task in the group to all other tasks in the group.
- Distribute distinct messages from one task in the group to all the tasks in the group or from each task in the group to all the tasks in the group.
- Gather distinct messages from each task in the group to a single destination task.
- Create barriers to synchronize the processing of all the tasks in the group.
- Shift data up or down within the group.

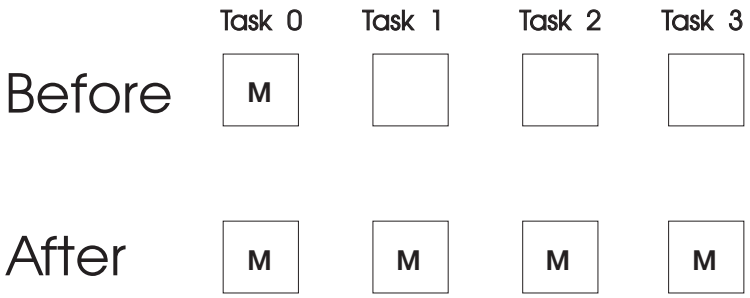
Table 7 on page 23 describes the collective communication routines for collective message passing within a task group. Conceptual illustrations of these routines follow the table. For the complete syntax of these calls, refer to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages.

Table 7. Collective Message-Passing Routines

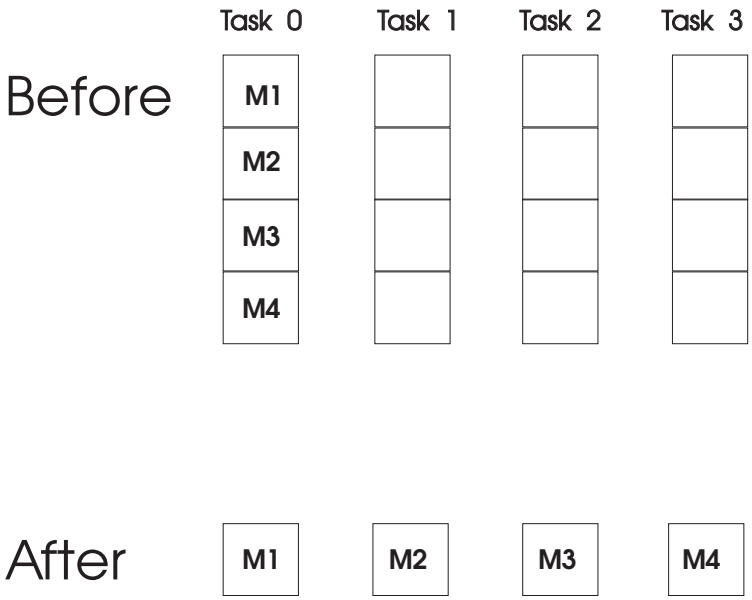
Use this MPL call:	To:
MP_BCAST mpc_bcast	**Send a message from one task to all tasks in the group.** This operation is called a broadcast. All tasks in the participating task group must issue corresponding calls to MP_BCAST. Each task in the group specifies the group identifier, indicates which task within the group is sending the message, and gives the message size in bytes. Each task also identifies an application buffer. For the sending task, this is the application buffer containing the message to send. For all the receiving tasks, this is the application buffer into which the message will be received. See "Broadcast Operation (MP_BCAST)" on page 24 for illustration of the broadcast operation.
MP_SCATTER mpc_scatter	**Distribute distinct messages from one task in the group to each task in the group.** This operation is called a scatter. All tasks in the participating task group must issue corresponding calls to MP_SCATTER. The reverse of this operation is called a gather and can be performed using the MP_GATHER routine. Each task in the group specifies the group identifier, indicates which task within the group will be sending the distinct messages, and gives the message size in bytes. Each task also identifies one or two application buffers. For the one sending task, there is the application buffer containing the messages to send, while for all receiving tasks, there must be a corresponding scratch buffer. For all the receiving tasks, there is also an application buffer into which its distinct message will be received. See "Scatter Operation (MP_SCATTER)" on page 24 for illustration of the scatter operation.
MP_GATHER mpc_gather	**Gather distinct messages from each task in the group and collect them in a single destination task.** This operation is called a gather. All tasks in the participating task group must issue corresponding calls to MP_GATHER. The reverse of the gather operation is called a scatter and can be performed using the MP_SCATTER routine. Each task in the group specifies the group identifier, indicates which task within the group will be receiving all the messages, and gives the message size in bytes. Each task also identifies one or two application buffers. For each sending task, there is the application buffer containing the outgoing message to be gathered, as well as a scratch buffer of a size adequate for the gathered messages. For the one receiving task, there is the application buffer into which all the messages will be placed. See "Gather Operation (MP_GATHER)" on page 25 for illustration of the gather operation.
MP_SYNC mpc_sync	**Synchronize execution of all tasks in the group.** All tasks in the participating task group must issue corresponding calls to MP_SYNC. The calls themselves then act as a barrier to execution. A call to MP_SYNC blocks execution of a task until all tasks in the group have made the corresponding call.
MP_SHIFT mpc_shift	**Perform a circular-shift operation.** Tasks in the group form a logical ring, and the routine rotates, or shifts, messages a number of steps "up" or "down" according to task rank. Each task in the group specifies the group identifier, indicates application buffers for outgoing and incoming messages, and gives the message size in bytes. Tasks also indicate the degree and direction of the shift. See "Shift Operation (MP_SHIFT)" on page 25 for illustration of the shift operation.
MP_CONCAT mpc_concat	**Take one distinct message from each, and send the concatenated result to all, tasks in a group.** This operation is called a concatenation or a complete broadcast, and is logically equivalent to a gather operation followed by a broadcast operation. All tasks in the participating task group must issue corresponding calls to MP_CONCAT. Each task in the group specifies the group identifier, indicates application buffers for outgoing and incoming messages, and gives the outgoing message size in bytes. See "Concatenation or Complete Broadcast Operation (MP_CONCAT)" on page 26 for illustration of this operation.
MP_INDEX mpc_index	**Distribute distinct messages from each task in the group to all other tasks.** This operation is called an index and is logically equivalent to each task performing a separate scatter. Each task in the group specifies the group identifier, indicates application buffers for outgoing and incoming messages, and gives the outgoing message size in bytes. See "Index Operation (MP_INDEX)" on page 26 for illustration of this operation.

The following figures provide conceptual illustration of the MPL collective communication routines outlined in Table 7 on page 23. In the figures, the word "Before" refers to the contents of the application buffers before the call to the collective communication routine. The word "After" refers to the contents of the application buffer once the routine completes.

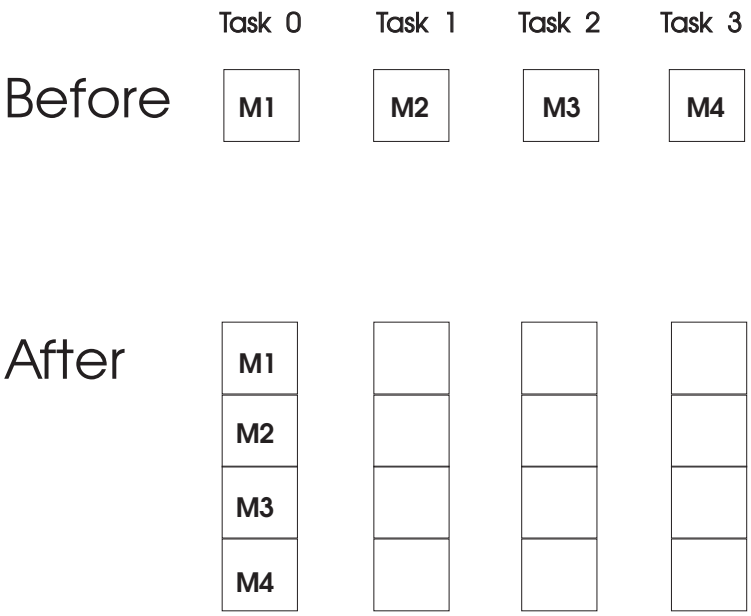
Broadcast Operation (MP_BCAST): This operation sends a message from one task to all tasks in the group. In this example, task 0 sends the message.



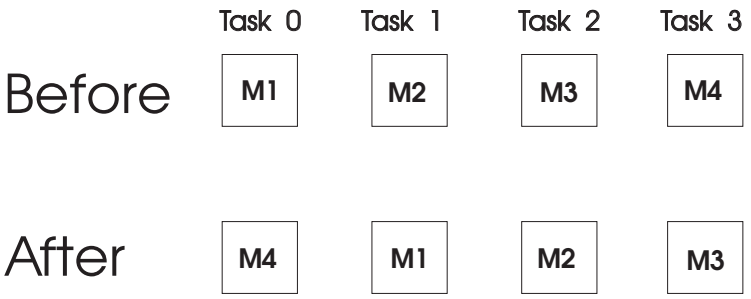
Scatter Operation (MP_SCATTER): This operation distributes distinct messages from one task in the group to each task in the group. In this example, task 0 sends the distinct messages.



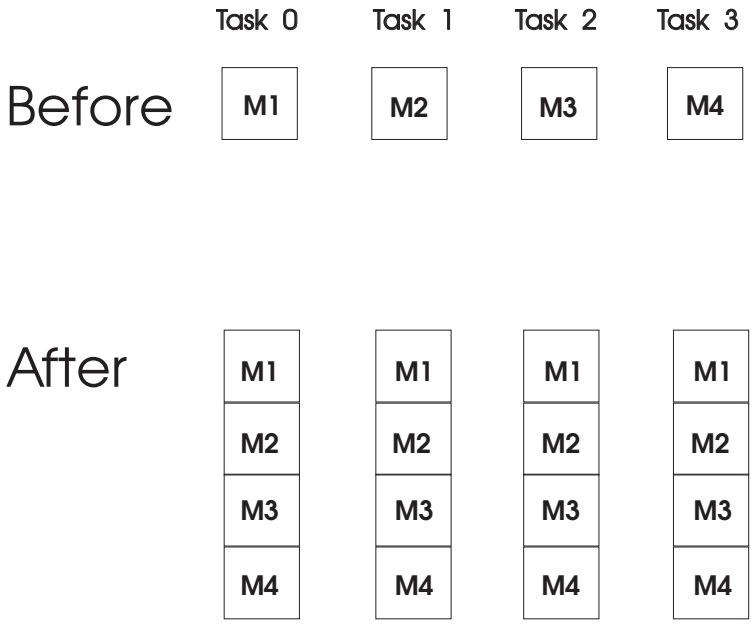
Gather Operation (MP_GATHER): This operation gathers distinct messages from each task in the group and collects them in a single destination task. In this example, task 0 is the destination task.



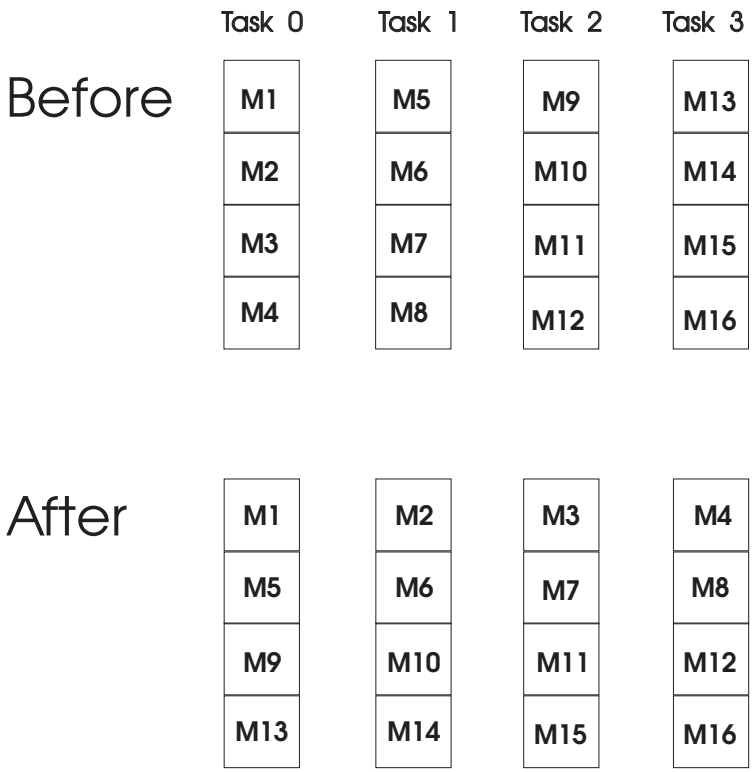
Shift Operation (MP_SHIFT): This operation performs a circular-shift of messages. In this example, the message is shifted “up” by one task.



Concatenation or Complete Broadcast Operation (MP_CONCAT): This operation takes one distinct message from each, and sends the concatenated result to all, tasks in the group.



Index Operation (MP_INDEX): This operation distributes distinct messages from each task in the group to all other tasks in the group.



Performing Collective Compute Operations Within Task Groups

The routines that combine compute operations with message passing enable you to:

- Apply a reduction operation to data from all the tasks in the group and send the result to one task, or all tasks, in the group.
- Shift data while applying a reduction operation across all tasks in the group.

The routines call external functions which implement the desired reduction operation. This can be a user-supplied function, or you can use the functions provided with PE to perform one of the following reduction operations:

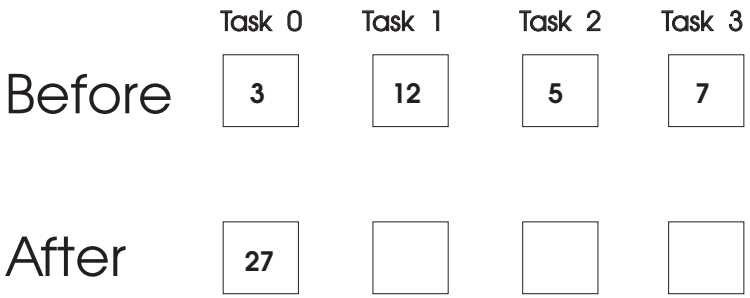
- integer vector addition
- integer vector multiplication
- integer vector minimum
- integer vector maximum
- single-precision floating-point vector addition
- double-precision floating-point vector addition
- single-precision floating-point vector multiplication
- double-precision floating point vector multiplication
- single-precision floating-point vector minimum
- double-precision floating-point vector minimum
- single-precision floating-point vector maximum
- double-precision floating-point vector maximum
- bitwise vector AND
- bitwise vector OR
- bitwise vector XOR
- logical (Boolean) vector AND
- logical (Boolean) vector OR

Table 8 on page 28 describes the collective communication routines for collective compute operations within a task group. Conceptual illustrations of these routines follow the table. For the complete syntax of these calls, refer to the *IBM AIX Parallel Environment: Programming Subroutine Reference* or the on-line PE manual pages.

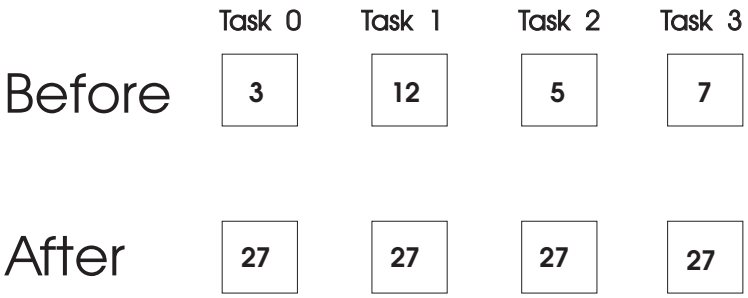
Table 8. Collective Compute Routines	
Use this MPL call:	**To:**
MP_REDUCE mpc_reduce	**Apply a reduction operation on message data from all tasks in the group and send the result to one task.** This operation is called a reduce. All tasks in the participating task group must issue corresponding calls to MP_REDUCE. Each task in the group specifies the group identifier, indicates which task in the group will be receiving the result, and gives the name of the external function that will implement the reduction operation. Each task also identifies an application buffer. For all the sending tasks, this is the application buffer containing their share of the data. For the one receiving task, this is the application buffer into which the result of the reduction operation will be placed. See "Reduce Operation (MP_REDUCE)" on page 28 for illustration of the reduce operation.
MP_COMBINE mpc_combine	**Apply a reduction operation on message data from, and send the result to, all tasks in the group.** This operation is called a combine and is logically equivalent to a reduce operation followed by a broadcast operation. All tasks in the participating task group must issue corresponding calls to MP_COMBINE. Each task in the group specifies the group identifier and gives the name of the external function that will implement the reduction operation. Each task also identifies application buffers for both the outgoing and incoming messages. See "Combine Operation (MP_COMBINE)" on page 29 for illustration of the combine operation.
MP_PREFIX mpc_prefix	**Shift data while applying a reduction operation across all tasks in the group.** The reduction operation is performed at each step and the corresponding and cumulative result is placed in each task along the way. This operation is called a prefix or a scan. All tasks in the participating task group must issue corresponding calls to MP_PREFIX. Each task in the group specifies the group identifier and gives the name of the external function that will implement the reduction operation. Each task also identifies application buffers for both the outgoing and incoming messages. See "Prefix or Scan Operation (MP_PREFIX)" on page 29 for illustration of the prefix operation.

The following figures provide conceptual illustration of the MPL collective communication routines outlined in Table 8. In the figures, the word "Before" refers to the contents of the application buffers before the call to the collective communication routine. The word "After" refers to the contents of the application buffer once the routine completes.

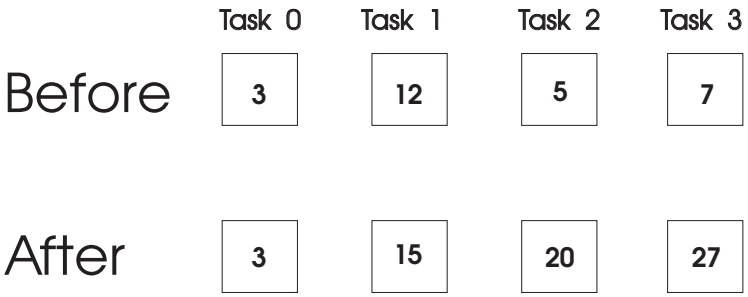
Reduce Operation (MP_REDUCE): This operation applies a reduction operation on message data from all tasks in the group and sends the result to one task. This example shows an integer addition reduction operation. Task 0 is the destination task.



Combine Operation (MP_COMBINE): This operation applies a reduction operation on message data from, and sends the result to, all tasks in the group. This example shows an integer addition reduction operation.



Prefix or Scan Operation (MP_PREFIX): This operation shifts data while applying a reduction operation across all tasks in the group. The reduction operation is performed at each step and the corresponding and cumulative result is placed in each task along the way. This example shows an integer addition reduction operation. Task 1 gets the sum of 3 + 12, task 2 gets the sum of 3 + 12 + 5, and task 3 gets the sum of 3 + 12 + 5 + 7.



Chapter 3. Converting Serial Programs for Parallel Execution

This chapter demonstrates some basic steps you can follow to convert serial programs for execution as a number of parallel tasks. This is referred to as *parallelizing* your application. Since improved performance is the motivation behind parallel programming, this chapter is not simply on enabling the serial program to run in parallel. Instead its focus is on steps for optimizing the code to take full advantage of parallel execution.

To demonstrate the steps involved in parallelizing an application, this chapter takes a case-study approach using a common but non-trivial numerical method for illustration. This approach is taken in order to be meaningful to, without specifically addressing, a number of parallel programming applications. Because parallel programming involves such varied fields as computational chemistry, seismic modelling, aircraft design, and census data analysis, it becomes impossible to address specifics for all fields. However, the numerical method parallelized here, a matrix multiply, is common enough to be meaningful to all these fields.

The numerical method we parallelize in this chapter provides a good example of *data decomposition*, where parallelism involves dividing program data up among parallel tasks. This chapter favors this method of decomposing a problem as it is generally easier to scale, and simplifies load balancing. An example of *functional parallelism* (where each program task performs unique work) is provided for contrast in Appendix A, “Functional Decomposition Example” on page 49. For a complete description of these concepts, refer to the parallel computing books listed in “Related Non-IBM Publications” on page vi.

All the examples used in this chapter are available via anonymous ftp from **tc.cornell.edu**. See “Obtaining This Book's Examples Through Anonymous ftp” on page vii for more information.

Step 1: Tune Serial Code

Before converting a serial program to run in parallel, you should first tune the serial version of the code. Only then can you identify which section(s) of the code to parallelize, or in fact whether the program would benefit at all from being run in parallel. If you can get a serial version of your program to run real problems in under a minute, there is not much point in parallelizing it. Tuning the serial code should always be your first step, because only then can the full benefits of parallel performance be realized. This step outlines the serial tuning we performed on our matrix-multiply example before parallelizing it. What follows, however, is not a detailed description of performance analysis and tuning. The purpose here is to stress the importance of tuning the serial code before parallelizing it. For a more detailed discussion of performance analysis and tuning, refer to the *Optimization and Tuning Guide for FORTRAN, C, and C++*.

© Copyright IBM Corp. 1993

Step 1a: Familiarize Yourself with the Serial Code

The following example shows `matmul1s.f` – the initial serial version of our matrix multiply program. It initializes matrices `A` and `B` and does a matrix multiply to produce matrix `C`.

```
c
c       matmul1s.f - Matrix multiply
c
        program matmul1s
        parameter (L=320, M=320, N=320)
        real*8    a(L,M), b(M,N), c(L,N)
        integer   i,j,k

c       initialize A and B matrices

        do j=1,M
           do i=1,L
              a(i,j) = i + j
           enddo
           do i=1,N
              b(j,i) = i + j
           enddo
        enddo

c       perform the matrix multiply

        do i=1,L
           do j=1,N
              c(i,j)=0.0
              do k=1,M
                 c(i,j) = c(i,j) + a(i,k) * b(k,j)
              enddo
           enddo
        enddo

c       print results

        write(6,*) "First 16x16 elements of solution matrix in Matrix Multiply"
        do i=1,8
           do j=1,8
              write(6,FMT='(8F10.0)') (c(i,j), j=1,16)
           enddo
        enddo
        stop
        end
```

Before you can tune and parallelize a program, you need, if you have not written the program yourself, to take the time to familiarize yourself with it. The `matmul1s.f` program is pretty straight-forward. The program:

1. initializes its data. Specifically, it sets the row and column size for the three matrices to 320 elements.

```
parameter (L=320, M=320, N=320)
real*8    a(L,M), b(M,N), c(L,N)
integer   i,j,k
```

You should do your serial optimization on a data set that is smaller than your production data set, so that the runs will complete in a reasonable amount of time. The data set should not be so small, however, as to mask the benefits of optimization and parallelization. Here we parameterized the size of the three matrices. If during serial or parallel tuning, this size proves too small to demonstrate useful profiler output, we can then easily increase the problem size.

2. initializes matrices `A` and `B`. The program uses two nested loops to set up its own input instead of reading it in.

```
do j=1,M
   do i=1,L
      a(i,j) = i + j
   enddo
   do i=1,N
      b(j,i) = i + j
   enddo
enddo
```

3. performs the actual computing – the matrix multiply – using a triple-nested loop. The rows of matrix A are multiplied by the columns of matrix B to produce matrix C.

```
do i=1,L
   do j=1,N
      c(i,j)=0.0
      do k=1,M
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      enddo
   enddo
enddo
```

4. prints the results of the matrix multiply.

```
write(6,*) "First 16x16 elements of solution matrix in Matrix Multiply"
do i=1,8
   do j=1,8
      write(6,FMT='(8F10.0)') (c(i,j), j=1,16)
   enddo
enddo
stop
end
```

Step 1b: Identify Program Hot Spots

Once you are familiar with the serial version of the program, you next want to do some form of performance analysis to determine where the bulk of the work is being done. This is because the greatest performance benefits can be obtained by tuning within the subroutines, loops, and blocks accessed most frequently. These frequently accessed routines are known as *hot spots*, because they take up a percentage of processor time disproportionate to their size within the program.

In a simple program like `matmul1s.f`, hot spots can be located visually by looking for the innermost loops and determining the number of times the statements within each such loop will be executed at run time. In `matmul1s.f`, it looks as if the bulk

of the processing time will probably be spent performing the matrix multiply itself. This is because the new matrix C is computed using a triple-nested loop. In many programs, however, the hot spots are not so apparent. Complex or elaborate coding, for example, could easily obscure the location of hot spots. You will then have to turn to some AIX tool or utility designed for performance analysis.

We decided to do performance analysis using **prof** – the most common AIX profiling utility. You might prefer to do your own performance analysis using profilers such as **gprof** and **tprof**. Although some prefer to use timing calls such as **MCLOCK** and **gettimer**, we choose **prof** because it is the most straight-forward and accessible performance analysis tool. We recommend either the **prof** or **gprof** profilers for performance analysis, because both have been enabled for use on parallel programs. This allows you to continue using them to tune the parallel version of your program.

For a discussion of the AIX profilers and timers you can use for performance analysis and tuning, refer to the *Optimization and Tuning Guide for FORTRAN, C, and C++*.

To enable profiling, we compiled `matmul1s.f` using the **-p** compiler option on the **xlf** command:

```
xlf matmul1s.f -p -o matmul1s
```

Executing the program then produced a data file which we processed using the **prof** command. For more information on the **xlf** command and its options, refer to *AIX XL FORTRAN Compiler/6000: User's Guide*. For more information on the **prof** command, see the *AIX for RISC System/6000: Commands Reference*.

The **prof** utility reads the data file of the program's run and writes its profile to standard output. The profile of `matmul1s` was:

```
 Name              %Time     Seconds     Cumsecs  #Calls   msec/call
 .main             100.0       47.77       47.77       1     47770.
 .catopen            0.0        0.00       47.77       1          0.
 .catclose           0.0        0.00       47.77       2          0.
 .setlocale          0.0        0.00       47.77       1          0.
 .free               0.0        0.00       47.77       2          0.
 .free_y             0.0        0.00       47.77       2          0.
 .write              0.0        0.00       47.77       1          0.
 .exit               0.0        0.00       47.77       1          0.
 .moncontrol         0.0        0.00       47.77       1          0.
 .monitor            0.0        0.00       47.77       1          0.
```

The columns in the profile show, from left to right, the:

- name of each routine.
- the percent of the total run-time spent in the routine.
- the total number of seconds spent in the routine.
- the cumulative number of seconds spent running the program.
- the number of times the routine was called.
- the total number of seconds spent in the routine divided by the number of times the routine was called.

This profile shows us the total running time of `matmul1s`, but it does not help us identify any hot spot. That is because **prof**, like **gprof**, displays a subroutine profile. The work is broken down by subroutine, and currently all the work is being done in the main program. A number of system routines were called, but that is all. All we can tell from this profile is that it took `47.77` seconds to run `matmul1s`.

To get a useful profile from **prof** or **gprof**, you need to break the components of a program into called subroutines. This requires only a small amount of additional coding.

The example below is the second serial version of our matrix multiply example – `matmul1ss.f`. Although we could have broken all of its parts into separate subroutines, we chose to make a separate subroutine call only for the matrix multiply itself. That is where we guessed the bulk of the work was being done.

```
c
c       matmul1ss.f - Matrix multiply
c
        program matmul1ss
        parameter (L=320, M=320, N=320)
        real*8    a(L,M), b(M,N), c(L,N)
        integer   i, j, k

c       initialize A and B matrices

        do j=1,M
           do i=1,L
              a(i,j) = i + j
           enddo
           do i=1,N
              b(j,i) = i + j
           enddo
        enddo

        call mmkernel(a, b, c, L, M, N)

c       print the results - advisable on small values of L, M, N (or to file)

        write(6,*) "First 8x8 elements of resultant in Matrix Multiply"
        do i=1,8
              write(6,FMT='(8F8.0)') (c(i,j), j=1,8)
        enddo

        stop
        end


c       mmkernel subroutine implements the matrix multiply

        subroutine mmkernel(aa, bb, cc, L, M, N)
        real*8  aa(L,M), bb(M,N), cc(L,N)
        integer i, j, k

c       perform the matrix multiply
```

```
do i=1,L
   do j=1,N
      cc(i,j) = 0.0
      do k=1,M
         cc(i,j) = cc(i,j) + aa(i,k) * bb(k,j)
      enddo
   enddo
enddo
return
end
```

With the matrix multiply now broken into the separate subroutine `mmkernel`, we compiled `matmul1ss.f`, ran it, and issued the **prof** command as before. The **prof** command read the data file of this new run and wrote the following profile to standard output.

```
Name                %Time     Seconds     Cumsecs  #Calls   msec/call
.mmkernel            99.7       64.66       64.66       1     64660.
.matmul1s             0.3        0.18       64.84       1       180.
.catopen              0.0        0.00       64.84       1         0.
.catclose             0.0        0.00       64.84       2         0.
.setlocale            0.0        0.00       64.84       1         0.
.free                 0.0        0.00       64.84       2         0.
.free_y               0.0        0.00       64.84       2         0.
.write                0.0        0.00       64.84       1         0.
.exit                 0.0        0.00       64.84       1         0.
.moncontrol           0.0        0.00       64.84       1         0.
.monitor              0.0        0.00       64.84       1         0.
```

This profile is more informative than the first and confirms our belief that the bulk of the program's time is spent doing the actual matrix multiply. The profile shows in fact that `99.7` of the total `64.84` second run was spent computing matrix C. The subroutine `mmkernel` is the hot spot, and should be the focus of our serial-tuning efforts. The `mmkernel` routine may or may not remain a hot spot after it has been tuned for serial efficiency. If still a hot spot after serial tuning, and if we feel performance improvements can still be achieved through parallelizing the program, this is where we will also concentrate those efforts.

Step 1c: Specify Low-Level Compiler Optimizations

The easiest form of program tuning is to simply specify a compiler option for optimization. Optimization requires additional compile time, but usually results in reduced run time. Best of all, it requires no additional work on your part. The **xlf** compiler has two optimization levels you can specify using the **-O** flag followed by a digit signifying the level – either **2** or **3**.

When you enable compiler optimization, it performs control and data optimizations for the entire program unit during compilation. This analysis allows low-level optimizations such as:

- Value numbering
- branch straightening
- common expression elimination
- code motion
- reassociation and strength reduction
- global constant propagation

- store motion
- dead store elimination
- dead code elimination
- global register allocation
- instruction scheduling

For more information on the **xlf** compiler and its optimizations, refer to the *AIX XL FORTRAN Compiler/6000: User's Guide* or the *AIX for RISC System/6000: Commands Reference*.

To demonstrate compiler optimization, we compiled `matmul1ss.f` again, this time adding the **-O3** flag for optimized output.

```
xlf matmul1ss.f -p -03 -o matmul1s
```

We specified the **-03**-level because it performs more optimizations than the **-02**-level. However, the additional **-03**-level optimizations are memory and compile-time intensive. For this reason, the **-03**-level optimizations are only recommended when the desire for run-time speed improvements outweigh the concern for limiting compile-time resources. In our case, limiting compile-time resources was not a concern. If this is a concern for you, specify the **-02**-level optimizations instead. The **-02**-level optimizations offer improved run-time performance without a great increase in the time or storage required for compilation.

After compiling `matmul1ss.f` for optimized output, we ran it and issued the **prof** command as before. The **prof** command read the data file of this new run and wrote the following profile to standard output.

```
Name                %Time     Seconds     Cumsecs  #Calls   msec/call
 .mmkernel            99.6       22.71       22.71       1     22710.
 .matmul1s             0.4        0.10       22.81       1       100.
 .catopen              0.0        0.00       22.81       1         0.
 .catclose             0.0        0.00       22.81       2         0.
 .setlocale            0.0        0.00       22.81       1         0.
 .free                 0.0        0.00       22.81       2         0.
 .free_y               0.0        0.00       22.81       2         0.
 .write                0.0        0.00       22.81       1         0.
 .exit                 0.0        0.00       22.81       1         0.
 .moncontrol           0.0        0.00       22.81       1         0.
 .monitor              0.0        0.00       22.81       1         0.
```

The profile demonstrates the easy performance improvements gained by simply requesting low-level compiler optimization. The total run time of the program has been reduced from `64.84` seconds to `22.81` seconds. The bulk of the work is still being done in the `mmkernel` routine, but the time required to run this has gone from `64.66` seconds down to `22.71` seconds.

Step 1d: Specify or Perform High-Level Tuning

While the compiler optimizations do improve performance significantly, they only work at the loop and data-placement level – reorganizing data in order to manage memory. Further performance improvements can be realized through higher-level reordering of the source code itself. You can hand tune the code, or use either the KAP** or VAST-2** preprocessors. The KAP and VAST-2 preprocessors are programs that modify the source code before it is processed by the compiler. They attempt to transform source-level algorithms into algorithms that can take full

advantage of the optimizing capabilities of the compiler. They restructure the source code itself to better utilize the RISC System/6000 processing unit resources and memory hierarchy. The KAP and VAST-2 preprocessors perform the same kinds of high-level transformations to improve performance. For example, both preprocessors perform:

- division rotation
- stride minimization
- array padding
- blocking
- data-structure transformations
- alignment
- loop fusion, distribution, and interchange
- loop peeling, unrolling, and elimination
- invariant IF code floating
- induction variable simplification
- semantic IF merging
- GOTO to block IF
- inlining
- interprocedural analysis
- lifetime analysis

The KAP and VAST-2 preprocessors used to be known as the **-Pk** and **-Pv** preprocessors and were supplied with the **xlf** compiler. In the current version of XL FORTRAN, they are separate products available from third-party vendors. You, therefore, might not have either of these preprocessors. If you do not, you should know that any of the transformations performed by the preprocessors can also be accomplished through hand tuning. Depending on the program, this can require a significant amount of your time, but can be done. Instructions for hand-tuning all the optimizations in the above list appear in *AIX Version 3.2 for RISC System/6000: Optimization and Tuning Guide for FORTRAN, C, and C++*.

If you do not already have a preprocessor, we recommend that you make the investment. In many cases, the preprocessors yield programs that perform as efficiently as, or more efficiently than, their hand-tuned equivalents, without a significant investment of your time. If you use the preprocessors, you also may not require as thorough an understanding of tuning techniques or RISC System/6000 architecture. In addition, some of the transformations done by the preprocessors involve considerable expansion of source code. Implementing these same expansions through hand-tuning increases the likelihood of algorithmic and typographical errors, reduces the readability of the source code, and makes maintenance of the program more difficult. In short, although it is possible to tune your programs by hand to as great a level of efficiency as the preprocessors do, it will require a significant investment of your time and some of the more complicated transformations can lead to hard to read code and coding errors. Guidelines for choosing the preprocessor that best meets your needs appear in *AIX for RISC System/6000: Optimization and Tuning Guide for FORTRAN, C, and C++*. You can also get information on the preprocessors by contacting the companies that offer them as described in the following table.

For orders and information about the KAP preprocessor software and its manuals, contact:	For orders and information about the VAST-2 preprocessor software and its manuals, contact:
Kuck and Associates Attn: Sales Dept. 1906 Fox Drive Champaign, IL 61820 USA (217) 356-5184 (217) 356-5199 (facsimile) You can also send inquiries electronically to sales@kai.com.	Pacific-Sierra Research Corp. Computer Products Group 2901 28th Street Santa Monica, CA 90405 USA (310) 314-2300 Extension 343 (310) 314-2323 (facsimile) You can also send inquiries electronically to info@psrv.com.

Because the preprocessors are extra-cost options and may might not be available to you, this step shows both methods of tuning at the source code level. We restructure our matrix multiply example:

- by calling the KAP preprocessor which restructures the code for us. Although both preprocessors just mentioned are comparable, we used the KAP preprocessor as a version of it is also available for restructuring C programs. VAST-2 is only available for FORTRAN.
- by hand-tuning the code ourselves.

Calling a Preprocessor For High-Level Tuning

If you have a preprocessor available, this step is easy – you simply recompile your program using the flag to call the preprocessor. The flag for the KAP preprocessor is **-Pk**, so we entered:

```
xlf matmul1ss.f -p -Pk -03 -o matmul1s
```

After compiling `matmul1ss.f` with the **-Pk** flag, we ran it and issued the **prof** command as before. The **prof** command read the data file of this new run and wrote the following profile to standard output.

```
 Name                %Time      Seconds      Cumsecs  #Calls   msec/call
 .mmkernel            98.9         2.69         2.69       1      2690.
 .matmul1s             1.1         0.03         2.72       1        30.
 .catopen              0.0         0.00         2.72       1         0.
 .catclose             0.0         0.00         2.72       2         0.
 .setlocale            0.0         0.00         2.72       1         0.
 .free                 0.0         0.00         2.72       2         0.
 .free_y               0.0         0.00         2.72       2         0.
 .write                0.0         0.00         2.72       1         0.
 .exit                 0.0         0.00         2.72       1         0.
 .moncontrol           0.0         0.00         2.72       1         0.
 .monitor              0.0         0.00         2.72       1         0.
```

This profile shows that the code restructuring done by the KAP preprocessor was successful in tuning the program. The run time has gone from `22.81` seconds (with just **-03**-level optimization) down to `2.72` seconds by first calling the preprocessor. Compared with the initial run time of `64.84` seconds, this is excellent speedup considering all we did was add two extra flags when compiling the program.

Hand-Tuning the Source Code

If you do not have the KAP or VAST-2 preprocessor, you will need to hand-tune the source code yourself. The *AIX for RISC System/6000: Optimization and Tuning Guide for FORTRAN, C, and C++* contains instructions for performing the hand-tuned equivalents of all the preprocessor optimization techniques. You need to be familiar enough with all the tuning techniques described in order to spot where they can be applied.

As always in performance tuning, concentrate your efforts in the program's hot spots. For us, the hot spot is the `mmkernel` routine:

```
do i=1,L
   do j=1,N
      cc(i,j) = 0.0
      do k=1,M
         cc(i,j) = cc(i,j) + aa(i,k) * bb(k,j)
      enddo
   enddo
enddo
```

The `mmkernel` uses a triple-nested loop to perform the matrix multiply itself. It takes the rows of matrix A, and multiplies them by the columns of matrix B to produce matrix C. Upon examining the matrix multiply, we saw that one of the most productive loop optimizations – stride minimization – could be applied. *Stride* refers to the layout of an array's elements in memory, and the order in which these elements are accessed by the program. A stride of 1 means that memory-adjacent array elements are accessed on successive iterations of an array-processing loop. A stride of *N* means that for each array element accessed, *N*-1 memory-adjacent elements are skipped over before the next accessed element. Stride minimization is especially important when the stride can be brought down to a value smaller than the number of elements of an array that can be stored in a single 128-byte cache line. An array of type REAL(8) consists of values 8 bytes in length, so 16 such values can fit into a single cache line. If the stride of such an array cannot be reduced to less than 16, no two successively referenced elements will be close enough together to fit into the same cache line, and a cache miss will occur at each reference. Particularly large strides could also cause TLB misses and page faults. For more information on stride minimization, see *AIX for RISC System/6000: Optimization and Tuning Guide for FORTRAN, C, and C++*.

Our matrix multiply was coded to our algorithm without any consideration of its use of cache. We realized that by changing the nesting order of the loops, we could minimize the stride in which array elements would be accessed on successive iterations. This optimization technique is called *loop interchange*.

The following example shows our hand-tuned version of the matrix multiply – `matmul2ss.f`. The only difference is the nesting order of the loops in the `mmkernel` routine.

```
c
c       matmul2ss.f - Matrix multiply, serial example with optimized
c                     matrix multiply kernel in subroutine mmkernel
c                     MS IBM PPS
c
        program matmul2ss
        parameter (L=320, M=320, N=320)
```

```
        real*8    a(L,M), b(M,N), c(L,N)
        integer   i, j, k

c       initialize A and B matrices

        do j=1,M
           do i=1,L
              a(i,j) = i - 1
           enddo
           do i=1,N
              b(j,i) = i - 1
           enddo
        enddo

        call mmkernel(a, b, c, L, M, N)

c       print the results - advisable on small values of L, M, N (or to file)
        write(6,*) "First 8x8 elements of resultant in Matrix Multiply"
        do i=1,8
              write(6,FMT='(8F8.0)') (c(i,j), j=1,8)
        enddo

        stop
        end


c       mmkernel subroutine implements the matrix multiply

        subroutine mmkernel(aa, bb, cc, L, M, N)
        real*8 aa(L,M), bb(M,N), cc(L,N)
        integer i, j, k, L, M, N

c       perform the matrix multiply. Middle product approach
c       is faster than matmul1ss with only -O3
c       comparable to -O3 -Pk on matmul1ss.

        do j=1,N
           do i=1,L
              cc(i,j) = 0.0
           enddo
           do k=1,M
              do i=1,L
                 cc(i,j) = cc(i,j) + aa(i,k) * bb(k,j)
              enddo
           enddo
        enddo

        return
        end
```

To see how our hand-implemented loop interchange improved performance, we compiled `matmul2ss` with the **-p** flag for profiling and the **-O3** flag for optimized output.

```
xlf matmul2ss.f -p -03 -o matmul1s
```

We then ran `matmul2ss` and issued the **prof** command as before. The **prof** command read the data file of this new run and wrote the following profile to standard output.

```
Name              %Time     Seconds     Cumsecs  #Calls   msec/call
.mmkernel          96.3        2.63        2.63       1      2630.
.matmul2ss          3.7        0.10        2.73       1       100.
.catopen            0.0        0.00        2.73       1         0.
.catclose           0.0        0.00        2.73       2         0.
.setlocale          0.0        0.00        2.73       1         0.
.free               0.0        0.00        2.73       2         0.
.free_y             0.0        0.00        2.73       2         0.
.write              0.0        0.00        2.73       1         0.
.exit               0.0        0.00        2.73       1         0.
.moncontrol         0.0        0.00        2.73       1         0.
.monitor            0.0        0.00        2.73       1         0.
```

This profile shows a speedup comparable to that achieved by calling the KAP preprocessor. This is simply because we did the same stride minimization through loop interchange that the KAP preprocessor preforms. The profile of `matmul1ss` compiled with **-Pk** and **-03** showed the `mmkernel` routine running in `2.69` seconds. The profile of our hand-tuned code compiled only with **03**-level optimization shows that same routine running in `2.63` seconds. Similarly, while the KAP compiler was able to get the total run time down to `2.72` seconds, our hand-tuning got it down to `2.73` seconds.

Step 2: Parallelize the Application

Now that we have an optimized serial version of our matrix multiply example, we can convert it to run in parallel. Parallelizing an application involves identifying the section(s) of code to parallelize, and inserting the appropriate MPL calls to enable parallel execution.

Step 2a: Identify Section(s) of Code to Parallelize

To identify the section(s) of code to parallelize, you want to do some form of performance analysis on your tuned serial code to determine where the bulk of the work is being done. In other words, you want to concentrate your parallelization efforts in the program hot spots. To determine where we were going to concentrate our efforts, we only needed to examine the profiler output for `matmul2ss.f` – our final, hand-tuned, serial version.

```
Name              %Time     Seconds     Cumsecs  #Calls   msec/call
.mmkernel          96.3        2.63        2.63       1      2630.
.matmul2ss          3.7        0.10        2.73       1       100.
.catopen            0.0        0.00        2.73       1         0.
.catclose           0.0        0.00        2.73       2         0.
.setlocale          0.0        0.00        2.73       1         0.
.free               0.0        0.00        2.73       2         0.
.free_y             0.0        0.00        2.73       2         0.
.write              0.0        0.00        2.73       1         0.
.exit               0.0        0.00        2.73       1         0.
.moncontrol         0.0        0.00        2.73       1         0.
.monitor            0.0        0.00        2.73       1         0.
```

While the run time for `mmkernel` was now down to `2.63` seconds, the matrix multiply itself was clearly still the program hot spot.

Step 2b: Call MPL Routines to Enable Parallel Execution

Once you have identified the sections of code to parallelize, you can call MPL routines to enable parallel execution of the program as a number of tasks. The following example shows `matmul2ssp.f` – the parallel version of our matrix multiply program.

```
c
c       matmul2ssp.f - Matrix multiply, parallel example based in matmul2ss.f
c                      with intensive work folded into subroutine mmkernel
c                      Data is decomposed and work done in SPMD fashion.
c                      For simplicity, l/ntasks, m/ntasks, and n/ntasks
c                      should be integers
c                      MS IBM PPS
c
        program matmul2ssp
        parameter (L=1000, M=1000, N=1000)
        real*8    a(L,M), b(M,N), c(L,N), cc(L,N)
        integer   i, j, k, ntasks, myid
        integer   nasize, nbsize, L0, L1
        include "MPL.inc"

c       fetch our MPL environment characteristics
        call mp_environ(ntasks, myid)


c       initialize A and B matrices (could initialize in parallel, but
c         chose to illustrate data decomposition instead)

        if (myid.eq.0) then
           do j=1,M
              do  i=1,L
                 a(i,j) = i + j
              enddo
              do i=1,N
                 b(j,i) = i + j
              enddo
           enddo

        end if  ! end of 0 node initialization

c       broadcast all of B to nodes, first compute size of blocks to send
        nasize = 8 * L * M
        nbsize = 8 * M * N
        call mp_bcast(b, nbsize, 0, ALLGRP)
c       simple test, just broadcast A to all nodes
        call mp_bcast(a, nasize, 0, ALLGRP)

c       compute bounds on L dimensions (rows of A and C) in each task
c        (again for simplicity, set this up so that L0, L1 are integers
c         remember, myid is 0 origin, not 1)
```

```
        L0 = 1 + ( myid * L / ntasks )
        L1 = (myid + 1) * L / ntasks
        LL = L1 - L0

c       on each node, myid, perform Matrix Multiply for subset of B
d
d       do i=L0,L0+7
d          write(6,FMT='(I4," ROW",I4,8F6.0)') myid,i,(a(i,j), j=1,8)
d       enddo
        call mmkernel(a, b, cc, L0, L1, LL, M, N, myid)

c       gather results back on node 0 and print
        nccsize = 8 * LL * N
        call mp_gather(cc, c, nccsize, 0, ALLGRP)

        if (myid.eq.0) then

c       print the results - advisable on small values of l, m,n (or to file)

        write(6,*) "First 8x8 elements of resultant from Matrix Multiply"
        do i=1,8
              write(6,FMT='(8F6.0)') (c(i,j), j=1,8)
        enddo

        end if  ! end 0 node results print
        stop
        end


c       mmkernel subroutine implements the matrix multiply
c        in this version, use only the space needed, not full matrix.

        subroutine mmkernel(aa, bb, cc, L0, L1, LL, mm, nn, myid)
        real*8 aa(ll,mm), bb(mm,nn), cc(LL,nn)
        integer i, j, k, L0, L1, LL, mm, nn, myid

d       write(6,FMT='("MYID=",4I4)') myid, L0, L1, LL

c       perform the matrix multiply, middle product approach

        do j=1,nn
           do i=1,LL
              cc(i,j) = 0.0
           enddo
           do k=1,m
              do i=1,LL
                 cc(i,j) = cc(i,j) + aa(i,k) * bb(k,j)
              enddo
           enddo
d
d          write(6,FMT='(I4," COL",I4,9F6.0)') myid, j ,(cc(i,j), i=L0,L0+2),
d    +                 (aa(j,k), k=1,3), (bb(k,j), k=1,3)
        enddo
```

```
      return
      end
```

Like the earlier serial versions, `matmul2ssp.f` is pretty straight-forward. The program:

1. initializes the data. Notice that, in addition to declaring some new variables, we also increased the problem size.

   ```
   program matmul2ssp
   parameter (L=1000, M=1000, N=1000)
   real*8    a(L,M), b(M,N), c(L,N), cc(L,N)
   integer   i, j, k, ntasks, myid
   integer   nasize, nbsize, L0, L1
   ```

   We increased the problem size simply because we felt that the one we had used for the serial versions would prove too small to demonstrate useful profiler output when the program was run in parallel. The problem size is now about three-times larger in each dimension. The problem itself is now scaled 27-times larger.

2. queries the parallel partition by including the file `MPL.inc` and calling the MP_ENVIRON routine.

   ```
   include "MPL.inc"

c       fetch our MPL environment characteristics
   call mp_environ(ntasks, myid)
   ```

   The include file `MPL.inc` is available via anonymous ftp from **tc.cornell.edu**. It is a convenient file that picks up the MPL wildcard values using the MP_TASK_QUERY routine. The `MPL.inc` include file consists of the following lines of code.

   ```
   integer DONTCARE, ALLMSG, NULLTASK, ALLGRP
   integer nbuf(4), nelem, nqtype

c       set up arguments to return wild card types, nelem=4 returns all
   nqtype = 3
   nelem  = 4
   call mp_task_query(nbuf, nelem, nqtype)

   DONTCARE = nbuf(1)
   ALLMSG   = nbuf(2)
   NULLTASK = nbuf(3)
   ALLGRP   = nbuf(4)
   ```

   Note: If you plan to use `MPL.inc` in your own programs, note that its placement in your program is important. It must be included, as it is here, immediately after variable declarations and before the first executable statement.

3. initializes matrices `A` and `B`. These are the same lines of code used in the serial versions of the matrix multiply. Note, however, that only task 0 actually initializes the matrices.

```
      if (myid.eq.0) then
         do j=1,M
            do  i=1,L
               a(i,j) = i + j
            enddo
            do i=1,N
               b(j,i) = i + j
            enddo
         enddo
```

4. distributes the matrices from task 0 to all tasks using two MP_BCAST calls. It first computes the size of the blocks to send.

```
      nasize = 8 * L * M
      nbsize = 8 * M * N
      call mp_bcast(b, nbsize, 0, ALLGRP)
c     simple test, just broadcast A to all nodes
      call mp_bcast(a, nasize, 0, ALLGRP)
```

5. computes, for each task, the bounds of the L dimension. This determines the portion of the matrix multiply that each task will compute, and depends on the size of the parallel partition. The data will be decomposed differently depending on the number of program tasks.

```
      L0 = 1 + ( myid * L / ntasks )
      L1 = (myid + 1) * L / ntasks
      LL = L1 - L0
```

6. has each task performs a portion of the actual computing – the matrix multiply – and then gathers the results (using the MP_GATHER routine) back to task 0.

```
d     do i=L0,L0+7
d        write(6,FMT='(I4," ROW",I4,8F6.0)') myid,i,(a(i,j), j=1,8)
d     enddo
      call mmkernel(a, b, cc, L0, L1, LL, M, N, myid)

c     gather results back on node 0 and print
      nccsize = 8 * LL * N
      call mp_gather(cc, c, nccsize, 0, ALLGRP)
```

The `mmkernel` routine called here uses the middle-product approach used in our final, hand-tuned, serial version. Notice, however, that each task only works on its portion of the problem.

```
      subroutine mmkernel(aa, bb, cc, L0, L1, LL, mm, nn, myid)
      real*8 aa(ll,mm), bb(mm,nn), cc(LL,nn)
      integer i, j, k, L0, L1, LL, mm, nn, myid

d     write(6,FMT='("MYID=",4I4)') myid, L0, L1, LL

c     perform the matrix multiply, middle product approach

      do j=1,nn
         do i=1,LL
            cc(i,j) = 0.0
         enddo
         do k=1,m
            do i=1,LL
               cc(i,j) = cc(i,j) + aa(i,k) * bb(k,j)
            enddo
         enddo
d
d        write(6,FMT='(I4," COL",I4,9F6.0)') myid, j ,(cc(i,j), i=L0,L0+2),
d    +                (aa(j,k), k=1,3), (bb(k,j), k=1,3)
      enddo

      return
      end
```

7. has task 0 print the results of the matrix multiply.

```
      if (myid.eq.0) then

c     print the results - advisable on small values of l, m,n (or to file)

      write(6,*) "First 8x8 elements of resultant from Matrix Multiply"
      do i=1,8
            write(6,FMT='(8F6.0)') (c(i,j), j=1,8)
      enddo

      end if  ! end 0 node results print
      stop
      end
```

Step 3: Tune Parallel Code

Once you have a parallel version of your program, you run it in PE's Parallel Operating Environment as described in *IBM AIX Parallel Environment: Operation and Use*. Further performance tuning, if necessary, can be done at this time. Typically, you begin by tuning on a small partition of processor nodes, with successive runs on larger partitions. For tuning parallel programs, PE offers:

- **prof and gprof enablement.** The AIX profiler commands **prof** and **gprof** have been enabled for use on parallel programs. One of the reasons we recommend using **prof** and **gprof** for performance analysis of your serial programs, is that you can continue using them to tune the parallel versions.

  The difference between profiling serial and parallel programs with the AIX profilers, is that serial programs generate a single profile data file, while a parallel program produces one for each parallel task. The system gives unique names to the data files by appending each tasks's identifying number to `mon.out` or `gmon.out`. For **-p** profiling, the data files produced take the form:

`mon.out.task_id`

For **-pg** profiling, the data files produced take the form:

`gmon.out.task_id`

You can use the **prof** or **gprof** utility to write a profile to standard output by reading one, some, or all of the data files produced.

- **The Visualization Tool (VT)** This X-Windows tool consists of a group of displays which show unique performance characteristics of your program and system. VT enables you to play back statistical and event records called trace records generated during a parallel program's execution. The VT views then enable you to visualize:
  - communication among processor nodes
  - the type and duration of communication events
  - CPU utilization of processor nodes
  - a parallel program's source code as it relates to the executable's run
  - the number of times a program's tasks are swapped in and out of active execution on a processor node
  - the number of disk reads, disk writes, and disk transfers for a processor node
  - the number of TCP/IP packets sent and received by a processor node
  - the number of times a processor node has to load a page of virtual memory back into real memory
  - the number of times a processor node invokes a kernel subroutine
  - a summary of system activity.

For a complete description of profiler enablement and the Visualization Tool, refer to *IBM AIX Parallel Environment: Operation and Use*.

Appendix A. Functional Decomposition Example

The sample serial and parallel program in this appendix shows a functional decomposition of a problem. These programs are provided here as a contrast to the data decomposition example (the matrix multiply) described throughout Chapter 3, “Converting Serial Programs for Parallel Execution” on page 31. The two sample programs shown here are available via anonymous ftp from **tc.cornell.edu** at the Cornell Theory Center. See “Obtaining This Book's Examples Through Anonymous ftp” on page vii for more information.

“MPLprob2s.f” is the original serial program. It is a contrived application which tabulates statistics from a set of test scores.

“MPLprob2p.f” on page 52 is the parallel version decomposed along functional lines. It must run as 6 tasks, and is not scalable. Task 0 handles I/O, while the remaining tasks each calculate one of five independent pieces of the test score problem.

MPLprob2s.f

```
C
C   MPLprob2s.f - adapted by Ron Mraz, IBM Watson Research Center 1992
C
C             Problems to Test Parallel and Vector Languages
C             CSD-TR 516, Computer Science, Purdue University
C             JOHN R. RICE, May 1, 1985
C
C             Revised by John R. Rice AND J. Jing, OCT. 1, 1990
C
      PROGRAM MPLprob2s
      PARAMETER (NTDIM=50,NSDIM=1000,KASES=5)
      DIMENSION SCORES(NTDIM,NSDIM),TOP(NSDIM),M(KASES)
      DIMENSION N(KASES),TAVER(KASES)
      DIMENSION TLOWABO(KASES)
      LOGICAL ABOVE(NTDIM,NSDIM),GENIUS,TGENIUS(KASES),G(NSDIM)
      REAL LOWABO
      DIMENSION TARRAY(2)
      DATA N / 5,20,30,40,50 /,M / 12,30,200,400,1000 /
      TIME1 = DTIME(TARRAY)
      DO 10 I = 1, NTDIM
        DO 10 J = 1, NSDIM
          SCORES(I,J) = 60.+40.*SIN(I*J*63.21)
   10 CONTINUE
      DO 100 K = 1, KASES
        NABOVE = 0
        LOWABO = 9.9E10
        NT = N(K)
        NS = M(K)
        DO 20 I = 1, NT
          TOP(I) = XMAX(SCORES,NTDIM,NSDIM,I)
   20   CONTINUE
        SUM = 0.0
```

© Copyright IBM Corp. 1993

```
      DO 30 I = 1, NT
        DO 30 J = 1, NS
          SUM = SUM+SCORES(I,J)
 30    CONTINUE
      AVER = SUM/(NS*NT)
      DO 50 I = 1, NT
        DO 40 J = 1, NS
C
C               INCREASE SCORES ABOVE AVERAGE
C
          IF (SCORES(I,J).GT.AVER) THEN
            SCORES(I,J) = 1.1*SCORES(I,J)
          ENDIF
C
C               SET SWITCH FOR SCORES ABOVE AVERAGE
C
          IF (SCORES(I,J).GT.AVER) THEN
            NABOVE = NABOVE+1
            ABOVE(I,J) = .TRUE.
          ENDIF
 40       CONTINUE
 50    CONTINUE
C
C       LOWABO = MASKMN(SCORES, ABOVE)
C
      DO 60 I = 1, NT
        DO 60 J = 1, NS
          IF (ABOVE(I,J)) THEN
            IF (SCORES(I,J).LT.LOWABO) THEN
              LOWABO = SCORES(I,J)
            ENDIF
          ENDIF
 60    CONTINUE
C
C               FIND THE GENIUSES
C         G(I) = ALL(ABOVE(*,I))
C
      DO 80 I = 1, NT
        G(I) = .TRUE.
        DO 70 J = 1, NS
          IF (.NOT.(ABOVE(I,J))) THEN
            G(I) = .FALSE.
          ENDIF
 70       CONTINUE
 80    CONTINUE
C
C       GENIUS = ANY(G)
C
      GENIUS = .FALSE.
      DO 90 I = 1, NT
        IF (G(I)) THEN
          GENIUS = .TRUE.
        ENDIF
 90    CONTINUE
```

```
        TAVER(K) = AVER
        TGENIUS(K) = GENIUS
        TLOWABO(K) = LOWABO
  100 CONTINUE
      TIME2 = DTIME(TARRAY)
      DO 110 K = 1, KASES
        PRINT 105, N(K), M(K)
  105   FORMAT('PROBLEM 5 WITH NT, NS =', I6,2X,I6)
        PRINT *,'GIVES AVERAGE, GENIUS =',TAVER(K),TGENIUS(K)
        PRINT *,'     AND LOW-ABOVE    =',TLOWABO(K)

C
C         WRITE(6,30) (I,TOP(I), G(I), I=1, NT)
C  30     FORMAT(5(I4,F9.4,L2))
C
  110 CONTINUE
      PRINT 120,TIME2,TARRAY(1),TARRAY(2)
  120 FORMAT ('TIME ',F8.4,' SECONDS, (',F8.4,' USER, ',F8.4,' SYSTEM)')
      STOP
      END

      FUNCTION XMAX (A,N,M,I)
C
C            MAX FUNCTION TO HANDLE FINDING THE MAXIMUM
C            VALUE IN A ROW OF AN ARRAY
C
      DIMENSION A(N,M)
      XXMAX = 1.0E-10
      DO 10 JJ = 1, M
        IF (A(I,JJ).GT.XXMAX) THEN
          XXMAX = A(I,JJ)
        ENDIF
   10 CONTINUE
      XMAX = XXMAX
      RETURN
      END
```

MPLprob2p.f

```
C
C   Test Problem -- ** Np tasks = 6 **
C
C   Program Name -- MPLprob2p.f --
C
C   Modified for MPL testing by Ron Mraz, July, 1992
C
C   This test program tabulates statistics of a set of test scores.
C   A separate processor is used to calculate one of 5 databases.
C   The program test the environments WAIT synchronization.
C
C   Problems to test Parallel and Vector Languages
C   CSD-TR-1016, Computer Science, Purdue University
C   John R. Rice and Jin Jing, October 1, 1990
C
C   This program in parallel form requires the following call(s)
C            MP_ENVIRON
C            MP_WAIT
C            MP_SEND
C            MP_RECV
C
       program main

       PARAMETER ( NTDIM=50,NSDIM=1000,KASES=5)
       DIMENSION SCORES(NTDIM,NSDIM),TOP(NSDIM)
       DIMENSION M(KASES),N(KASES),TAVER(KASES)
       DIMENSION TLOWABO(KASES)
       LOGICAL ABOVE(NTDIM,NSDIM),TGENIUS(KASES),G(NSDIM)
       LOGICAL*1 GENIUS
       REAL*4    AVER
       REAL*4    LOWABO
       INTEGER buf, len, nbytes, src, dest, type, msgid
       DATA N / 5,20,30,40,50 /,M / 12,30,200,400,1000 /

       call MP_ENVIRON(nworkers,kwhoami)
       if( nworkers .NE. 6 ) then
       PRINT 990,nworkers,kwhoami
 990   FORMAT ('Usage NP = 6, You selected',I6,' This is node ',I6)
       GOTO 999
       end if

       PRINT 101,nworkers,kwhoami
 101   FORMAT ('Environment allocated ', I6,' and we are node ', I6)
C
C    Divide the problem into 5 independent pieces.
C    Each piece sends their results to node zero for printout
C
       if (kwhoami .EQ. 0)then
C
C      Receive all messages
C
C     messages are either type 1, 2, or 3
```

```
C       Type 1 messages are for AVER, REAL*4
C       Type 2 messages are for GENIUS, LOGICAL*1
C       Type 3 messages are for LOWABO, REAL*4
C      CALL MP_RECV(buf, len, src, type, msgid)
C
      msgid = 0
      DO 111 K = 1, KASES
      src = K
        nbytes = 4
        ntype = 1
      CALL MP_RECV(AVER, nbytes, src, ntype, msgid)
      CALL MP_WAIT(msgid, nbytes)
        TAVER(K) = AVER
        msgid = msgid + 1
        nbytes = 1
        ntype = 2
      CALL MP_RECV(GENIUS, nbytes, src, ntype, msgid)
      CALL MP_WAIT(msgid, nbytes)
        TGENIUS(K) = GENIUS
        msgid = msgid + 1
        nbytes = 4
        ntype = 3
      CALL MP_RECV(LOWABO, nbytes, src, ntype, msgid)
      CALL MP_WAIT(msgid, nbytes)
        TLOWABO(K) = LOWABO
 111   CONTINUE
C
      DO 110 K = 1, KASES
        PRINT 105, K, N(K), M(K)
 105      FORMAT('PROBLEM 5 CASE',I3,' WITH NT, NS =', I6,2X, I6)
        PRINT *,'GIVES AVERAGE, GENIUS = ',TAVER(K),TGENIUS(K)
        PRINT *,'      AND LOW-ABOVE    = ',TLOWABO(K)

 110   CONTINUE
C
C
      else
C
C    This is a compute node
C
      K = kwhoami
      msgid = 3*(kwhoami -1)
      dest = 0
      DO 10 I = 1,NTDIM
        DO 10 J = 1, NSDIM
          SCORES(I,J) = 60.+40.*SIN(I*J*63.21)
  10   CONTINUE
        NABOVE = 0
        LAWABO = 9.9E10
        NT = N(K)
        NS = M(K)
        DO 20 I = 1, NT
          TOP(I) = XMAX(SCORES,NTDIM,NSDIM,I)
  20   CONTINUE
```

```
      SUM = 0.0
      DO 30 I = 1, NT
        DO 30 J = 1, NS
          SUM = SUM+SCORES(I,J)
   30   CONTINUE
      AVER = SUM/(NS*NT)
        nbytes = 4
        type = 1
      CALL MP_SEND(AVER, nbytes, dest, type, msgid)
C
C     Send average to node zero
C
      DO 50 I = 1,NT
        DO 40 J = 1, NS
C
C     Increase Scores Above Average
C
          IF (SCORES(I,J).GT.AVER) THEN
            SCORES(I,J) = 1.1*SCORES(I,J)
          ENDIF
C
C     Set Switch for Scores Above Average
C
          IF (SCORES(I,J).GT.AVER) THEN
              NABOVE = NABOVE+1
              ABOVE(I,J) = .TRUE.
          ENDIF
   40     CONTINUE
   50   CONTINUE
      print *, 'NABOVE = ', NABOVE
C
C     Set Switch for Scores Above Average
C
      LOWABO = 100.0
      DO 60 I = 1, NT
        DO 60 J = 1, NS
          IF (ABOVE(I,J)) THEN
            IF (SCORES(I,J).LT.LOWABO) THEN
              LOWABO = SCORES(I,J)
            ENDIF
          ENDIF
   60   CONTINUE
C
C     Send LOWABO to node zero
C
      CALL MP_WAIT(msgid, nbytes)
        msgid = msgid +1
        type = 3
      CALL MP_SEND(LOWABO, nbytes, dest, type, msgid)
C
C     Find the Geniuses
C       G(I) = ALL(ABOVE(*,I))
C
      DO 80 I = 1, NT
```

```
          G(I) = .TRUE.
          DO 70 J = 1, NS
             IF (.NOT.(ABOVE(I,J))) THEN
              G(I) = .FALSE.
             ENDIF
   70      CONTINUE
   80    CONTINUE
C
C       GENIUS = ANY(G)
C
          GENIUS = .FALSE.
          DO 90 I = 1, NT
             IF (G(I)) THEN
               GENIUS = .TRUE.
             ENDIF
   90      CONTINUE
C
C       Send GENIUS to node zero
C
        CALL MP_WAIT(msgid, nbytes)
          msgid = msgid +1
          type = 2
          nbytes = 1
        CALL MP_SEND(GENIUS, nbytes, dest, type, msgid)
        CALL MP_WAIT(msgid, nbytes)

        ENDIF

  999   CONTINUE
C       STOP
        END
C

        FUNCTION XMAX (A,N,M,I)
C
C         MAX Function to handle finding the Maximum
C         Value in a row of an array
C
C
        DIMENSION A(N,M)
        XXMAX = 1.0E-10
        DO 10 JJ = 1, M
          IF (A(I,JJ).GT.XXMAX) THEN
            XXMAX =  A(I,JJ)
          ENDIF
   10   CONTINUE
        XMAX = XXMAX
        RETURN
        END
```

Glossary of Terms and Abbreviations

This glossary includes terms and definitions from:

- The *Dictionary of Computing*, SC20-1699.
- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies can be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- The *ANSI/EIA Standard - 440A: Fiber Optic Terminology*, copyright 1989 by the Electronics Industries Association (EIA). Copies can be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue N.W., Washington, D.C. 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

This section contains some of the terms that are commonly used in the Parallel Environment books and in this book in particular.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard *Vocabulary for Information Processing* (Copyright 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3K5 on Terminology and Glossary of the American National Standards Committee X3. ANSI definitions are preceded by an asterisk (*).

Other definitions in this glossary are taken from *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems* (GC20-1699) and *IBM DATABASE 2 Application Programming Guide for TSO Users* (SC26-4081).

© Copyright IBM Corp. 1993

A

address. A character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high function graphics and floating point computations.

AIXwindows Environment/6000. A graphical user interface (GUI) for the RISC System/6000. It has the following components:

- A graphical user interface and toolkit based on OFS/Motif
- Enhanced X-Windows, an enhanced version of the MIT X Window System
- Graphics Library (GL), a graphical interface library for the applications programmer which is compatible with Silicon Graphics' GL interface.

application. The use to which a data processing system is put; for example, a payroll application, an airline reservation application.

bandwidth. The total available bit rate of a digital channel.

C

C. A general purpose programming language. It was formalized by ANSI standards committee for the C language in 1984 and by Uniforum in 1983.

cluster. A group of processors interconnected through a high speed network that can be used for high performance computing. It provides excellent price/performance.

D

data parallelism. Refers to situations where parallel tasks perform the same computation of different sets of data.

F

file. A set of related records treated as a unit, for example, in stock control, a file could consist of a set of invoices.

functional parallelism. Refers to situations where parallel tasks specialize in particular work.

H

hot spot. The portions of a program that take the most time to run.

K

kernel. The core portion of the UNIX operating system which controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in "kernel mode" and is protected from user tampering by the hardware.

L

latency. The time interval between the instant as which an instruction control unit initiates a call for data transmission, and the instant at which the actual transfer of data begins. Latency is related to the hardware characteristics of the system and to the different layers of software that are involved in initiating the task of packing and transmitting the data.

M

message passing. Refers to the process by which parallel tasks explicitly exchange program data.

MPMD (Multiple Program Multiple Data). A parallel programming model in which different, but related, programs are run on different sets of data.

N

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

node. In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network.

P

parallelize. To convert a serial program for parallel execution.

parameter. * (1) A variable that is given a constant value for a specified application and that may denote the application. (2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. (3) A name in a procedure that is used to refer to an argument that is passed to the procedure. (4) A particular piece of information that a system or application program needs to process a request.

process. * (1) A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions. * (2) Any operation or combination of operations on data. * (3) A function being performed or waiting to be performed. * (4) A program in operation. For example, a daemon is a system process that is always running on the system.

R

RISC. Reduced Instruction Set Computing (RISC), the technology for today's high performance personal computers and workstations, was invented in 1975.

S

SPMD (Single Program Multiple Data). A parallel programming model in which different processors execute the same program on different sets of data.

stride. The relationship between the layout of an array's elements in memory, and the order in which those elements are accessed. A stride of 1 means that memory-adjacent array elements are accessed on successive iterations of an array-processing loop. A stride of N means that for each array element accessed, N-1 memory-adjacent elements are skipped over before the next accessed element.

T

task. A unit of computation analogous to an AIX process.

TLB. See *translation lookaside buffer*.

TLB miss. A memory delay that occurs when a memory location is referenced and the page that contains that memory location does not have an entry in the appropriate lookaside buffer (instruction or data).

translation lookaside buffer (TLB). A table in the processor that contains cross-references between the virtual and real addresses of recently referenced pages of memory.

X

X. See the X window system.

X Window System. The UNIX industry's graphics windowing standard that provides simultaneous views of several executing programs or processes on high resolution graphics displays.

Index

A

allgrp wildcard 5
  obtaining system-specific value of 4
allmsg wildcard 11
  obtaining system-specific value of 4

B

blocking routines
  defined 9
  summary of 13
buffer overflow, avoiding 15

C

collective communication routines
  for collective message passing within a task group 22—26
  for combining compute operations with collective message passing 27—29

D

dontcare wildcard 8
  avoiding programming pitfalls related to 19
  obtaining system-specific value of 4

I

incorrect program, defined 15

M

messages, probing for available 14
messages, sending and receiving 6—21
MP_BCAST and mpc_bcast 23
  conceptual illustration 24
MP_BRECV and mpc_brecv 13
MP_BSEND and mpc_bsend 13
MP_BSENDRECV and mpc_bsendrecv 13
MP_BVRECV and mpc_bvrecv 14
MP_BVSEND and mpc_bvsend 14
MP_COMBINE and mpc_combine 28
  conceptual illustration 29
MP_CONCAT and mpc_concat 23
  conceptual illustration 26
MP_ENVIRON and mpc_environ 4
MP_GATHER and mpc_gather 23
  conceptual illustration 25
MP_GETLABEL and mpc_getlabel 6
MP_GETMEMBERS and mpc_getmembers 6
MP_GETRANK and mpc_getrank 6
MP_GETSIZE and mpc_getsize 6
MP_GETTASKID and mpc_gettaskid 6
MP_GROUP and mpc_group 5
MP_INDEX and mpc_index 23
  conceptual illustration 26
MP_PACK and mpc_pack 14
MP_PARTITION and mpc_partition 5
MP_PREFIX and mpc_prefix 28
  conceptual illustration 29
MP_RECV and mpc_recv 11
MP_REDUCE and mpc_reduce 28
  conceptual illustration 28
MP_SCATTER and mpc_scatter 23
  conceptual illustration 24
MP_SEND and mpc_send 11
MP_SHIFT and mpc_shift 23
  conceptual illustration 25
MP_STATUS and mpc_status 11
MP_STOPALL and mpc_stopall 4
MP_SYNC and mpc_sync 23
MP_TASK_QUERY and mpc_task_query 4
MP_TASK_SET and mpc_task_set 4
MP_UNPACK and mpc_unpack 14
MP_WAIT and mpc_wait 11
MPL routines
  blocking routines 9, 13
  for collective communication within a task group 22—29
  for point-to-point message passing 6, 21
  for task management 4—6
  MP_BCAST and mpc_bcast 23, 24
  MP_BRECV and mpc_brecv 13
  MP_BSEND and mpc_bsend 13
  MP_BSENDRECV and mpc_bsendrecv 13
  MP_BVRECV and mpc_bvrecv 14
  MP_BVSEND and mpc_bvsend 14
  MP_COMBINE and mpc_combine 28, 29
  MP_CONCAT and mpc_concat 23, 26
  MP_ENVIRON and mpc_environ 4
  MP_GATHER and mpc_gather 23, 25
  MP_GETLABEL and mpc_getlabel 6
  MP_GETMEMBERS and mpc_getmembers 6
  MP_GETRANK and mpc_getrank 6
  MP_GETSIZE and mpc_getsize 6
  MP_GETTASKID and mpc_gettaskid 6
  MP_GROUP and mpc_group 5
  MP_INDEX and mpc_index 23, 26
  MP_PACK and mpc_pack 14
  MP_PARTITION and mpc_partition 5
  MP_PREFIX and mpc_prefix 28, 29
  MP_RECV and mpc_recv 11

© Copyright IBM Corp. 1993

MPL routines *(continued)*
MP_REDUCE and mpc_reduce 28
MP_SCATTER and mpc_scatter 23, 24
MP_SEND and mpc_send 11
MP_SHIFT and mpc_shift 23, 25
MP_STATUS and mpc_status 11
MP_STOPALL and mpc_stopall 4
MP_SYNC and mpc_sync 23
MP_TASK_QUERY and mpc_task_query 4
MP_TASK_SET and mpc_task_set 4
MP_UNPACK and mpc_unpack 14
MP_WAIT and mpc_wait 11
non-blocking routines 9, 11
non-contiguous message-passing routines 14

N

non-blocking routines
defined 9
summary of 11
non-contiguous messages, sending and receiving 14
nulltask wildcard 13
obtaining system-specific value of 4

P

parallelizing applications (case study) 31—48
partition, managing tasks in 4
program deadlock, avoiding 15

T

task groups 5
creating and querying 5

U

unsafe program, defined 15

W

wildcards
allgrp 5
allmsg 11
avoiding programming pitfalls related to 19
dontcare 8
nulltask 13
obtaining system-specific value of 4

Readers' Comments — We'd Like to Hear from You

IBM AIX Parallel Environment
Programming Primer
Release 2.0

Publication No. SH26-7223-00

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

Use this form to tell us what you think about this manual. If you prefer, you may send comments electronically to your IBM branch office, **Attention: Dept. 52QA-911, Kingston, NY**. Or you may fax your comments to us directly at: **FAX (914) 385-0662**.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

Company or Organization _____

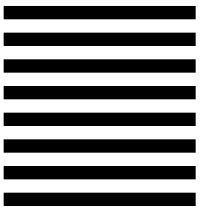
Phone No. _____

Readers' Comments — We'd Like to Hear from You
SH26-7223-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 52QA, Mail Station 911
1 Enterprise Drive
Kingston, NY 12401-1099

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in U.S.A.

SH26-7223-00