

IBM System Blue Gene Solution: Blue Gene/P Application Development

Understand the Blue Gene/P
programming environment

Learn how to run and
debug MPI programs

Learn about Bridge and
Real-time APIs



Carlos P. Sosa

Redbooks



International Technical Support Organization

**IBM System Blue Gene Solution: Blue Gene/P
Application Development**

December 2007

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (December 2007)

This edition applies to Version 1, Release 1, Modification 1 of IBM System Blue Gene/P Solution (product number 5733-BGP).

© Copyright International Business Machines Corporation 2007. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks	x
Preface	xi
The team that wrote this book	xi
Become a published author	xiii
Comments welcome	xiii
Part 1. Blue Gene/P: System and environment overview	1
Chapter 1. Hardware overview	3
1.1 System architecture overview	4
1.1.1 System buildup	5
1.1.2 Compute and I/O Nodes	5
1.1.3 Blue Gene/P environment	6
1.2 What is new on Blue Gene/P	7
1.3 Microprocessor	8
1.4 Compute Nodes	9
1.5 I/O Nodes	10
1.6 Networks	10
1.7 Blue Gene/P programs	11
1.8 Blue Gene specifications	12
1.9 Host system	13
1.9.1 Service Node	13
1.9.2 Front End Nodes	13
1.9.3 Storage Nodes	13
1.10 Host system software	14
Chapter 2. Software overview	15
2.1 Blue Gene/P software at a glance	16
2.2 Compute Node Kernel	17
2.2.1 Threading support on Blue Gene/P	18
2.3 Message Passing Interface on Blue Gene/P	18
2.4 Memory considerations	18
2.4.1 Memory leaks	21
2.4.2 Memory management	22
2.4.3 Uninitialized pointers	22
2.5 Other considerations	22
2.5.1 Input/output	22
2.5.2 Linking	23
2.6 Compilers overview	23
2.6.1 Programming environment overview	23
2.6.2 GNU Compiler Collection	23
2.6.3 IBM XL compilers	24
2.7 I/O Node software	24
2.7.1 I/O Node Kernel boot considerations	24
2.7.2 I/O Node file system services	24
2.7.3 Socket services for the Compute Node Kernel	25
2.7.4 I/O Node daemons	25

2.7.5 Control system	25
2.8 Management software	26
2.8.1 Midplane Management Control System	26
Part 2. Kernel overview	27
Chapter 3. Kernel functionality	29
3.1 System software overview	30
3.2 Compute Node Kernel	30
3.2.1 Boot sequence of a Compute Node	31
3.2.2 Common Node Services	32
3.3 I/O Node Kernel	32
3.3.1 Control and I/O daemon	33
Chapter 4. Execution process modes	37
4.1 Symmetrical Multiprocessing Node Mode	38
4.2 Virtual Node Mode	38
4.3 Dual Node Mode	39
4.4 Shared memory support	40
4.5 Deciding which mode to use	41
4.6 Specifying a mode	41
Chapter 5. Memory	43
5.1 Memory overview	44
5.2 Memory management	45
5.2.1 L1 cache	45
5.2.2 L2 cache	46
5.2.3 L3 cache	46
5.2.4 Double data RAM	46
5.3 Memory protection	47
Chapter 6. System calls	51
6.1 Introduction to the Compute Node Kernel	52
6.2 System calls	52
6.2.1 Return codes	52
6.2.2 Supported system calls	53
6.2.3 Other system calls	57
6.3 System programming interfaces	57
6.4 Socket support	57
6.5 Signal support	59
6.6 Unsupported system calls	60
Part 3. Applications environment	63
Chapter 7. Parallel paradigms	65
7.1 Programming model	66
7.2 Blue Gene/P MPI implementation	66
7.2.1 High performance network for efficient parallel execution	67
7.2.2 Forcing MPI to allocate too much memory	69
7.2.3 Not waiting for MPI_Test	70
7.2.4 Flooding of messages	70
7.2.5 Deadlock the system	71
7.2.6 Violating MPI buffer ownership rules	71
7.2.7 Interlocking collectives with point-to-point calls	73
7.3 MPI communications	74

7.3.1	Blue Gene/P MPI extensions	74
7.4	MPI functions	76
7.5	Compiling MPI programs on Blue Gene/P	77
7.6	MPI communications performance	79
7.6.1	MPI point-to-point	80
7.6.2	MPI collective	81
7.7	OpenMP	83
7.7.1	OpenMP implementation for Blue Gene/P	83
7.7.2	Selected OpenMP compiler directives	83
7.7.3	Selected OpenMP compiler functions	86
7.7.4	Performance	86
Chapter 8.	Developing applications with IBM XL compilers	91
8.1	What is new.	92
8.2	Compiling and linking applications on Blue Gene/P	92
8.3	Default compiler options	93
8.4	Unsupported options	94
8.5	Support for threads, OpenMP, and SMP	94
8.6	XL runtime libraries	95
8.7	Mathematical Acceleration Subsystem libraries	96
8.8	Engineering and Scientific Subroutine Library libraries	96
8.9	Tuning your code for Blue Gene/P	96
8.9.1	Using the compiler optimization options	96
8.9.2	PowerPC 450 processor parallel double-precision floating point multiply add unit	97
8.9.3	Using Single Instruction Multiple Data instructions in applications	98
8.10	Tips for optimizing constructs	100
8.10.1	Structuring data in adjacent pairs	100
8.10.2	Using vectorizable basic blocks	101
8.10.3	Using inline functions	101
8.10.4	Removing possibilities for aliasing (C/C++)	102
8.10.5	Structure computations in batches	103
8.10.6	Checking for data alignment	104
8.10.7	Using XL built-in floating-point functions for Blue Gene/P	106
8.10.8	Complex type manipulation functions	109
8.10.9	Load and store functions	111
8.10.10	Move functions	113
8.10.11	Arithmetic functions	114
8.10.12	Select functions	123
8.10.13	Examples of built-in functions usage	124
Chapter 9.	Running and debugging applications	129
9.1	Running applications	130
9.1.1	MMCS console	130
9.1.2	mpirun	131
9.1.3	LoadLeveler	131
9.1.4	Other scheduler products	132
9.2	Debugging applications	132
9.2.1	General debugging architecture	132
9.2.2	GNU Project debugger	133
9.2.3	Core Processor debugger	138
9.2.4	Starting the Core Processor tool	138
9.2.5	Attaching running applications	139
9.2.6	Saving your information	145

9.2.7	Debugging live I/O Node problems	145
9.2.8	Debugging core files	146
9.2.9	The addr2line utility	148
Chapter 10.	Checkpoint and restart support for applications	151
10.1	Checkpoint and restart	152
10.2	Technical overview	152
10.2.1	Input/output considerations	153
10.2.2	Signal considerations	153
10.3	Checkpoint API	155
10.3.1	Checkpoint library API	155
10.4	Directory and file naming conventions	157
10.5	Restart	157
10.5.1	Determining the latest consistent global checkpoint	157
10.5.2	Checkpoint and restart functionality	158
Chapter 11.	Control system (Bridge) APIs	159
11.1	API requirements	160
11.1.1	Configuring environment variables	160
11.1.2	General comments	161
11.2	APIs	162
11.2.1	API to the Midplane Management Control System	162
11.2.2	Asynchronous APIs	163
11.2.3	State sequence IDs	163
11.2.4	Bridge API return codes	163
11.2.5	Blue Gene hardware resource APIs	164
11.2.6	Partition-related APIs	166
11.2.7	Job-related APIs	171
11.2.8	Field specifications for the rm_get_data() and rm_set_data() APIs	178
11.2.9	Object allocator APIs	189
11.2.10	Object deallocator APIs	189
11.2.11	Messaging APIs	190
11.3	Small partition allocation	191
11.3.1	Subdivided busy base partitions	192
11.4	API examples	192
11.4.1	Retrieving base partition information	192
11.4.2	Retrieving node card information	193
11.4.3	Defining a new small partition	194
11.4.4	Querying a small partition	195
Chapter 12.	Real-time Notification APIs	197
12.1	API support overview	198
12.1.1	Requirements	198
12.1.2	General comments	199
12.2	Real-time Notification APIs	200
12.3	Real-time callback functions	201
12.4	Real-time Notification API status codes	207
12.4.1	Status code specification	208
12.5	Sample real-time application code	209
Chapter 13.	mpirun	217
13.1	mpirun implementation on Blue Gene/P	218
13.2	mpirun setup	219
13.2.1	User setup	219

13.2.2	System administrator setup	219
13.3	Invoking mpirun	220
13.4	Environmental variables	224
13.5	Return codes	225
13.6	Examples	227
13.7	mpirun application program interfaces	234
Chapter 14. Dynamic Partition Allocator APIs		237
14.1	Overview of API support	238
14.1.1	Requirements	238
14.2	API details	239
14.2.1	APIs	239
14.2.2	Return codes	240
14.3	Sample program	241
Part 4. Applications		243
Chapter 15. Performance overview of engineering and scientific applications		245
15.1	Blue Gene/P system from an applications perspective	246
15.2	Selected Chemistry and Life Sciences applications	247
15.2.1	Classical Molecular Mechanics and Molecular Dynamics applications	248
15.2.2	Molecular Docking applications	252
15.2.3	Electronic structure (Ab Initio) applications	253
15.2.4	Bioinformatics applications	254
15.2.5	Performance kernel benchmarks	256
15.2.6	MPI point-to-point	257
Part 5. Appendixes		263
Appendix A. Blue Gene/P hardware naming convention		265
Appendix B. Header files and libraries		271
	Blue Gene/P applications	272
	Resource management APIs	273
Appendix C. Files on architectural features		275
	Personality of Blue Gene/P	276
	Example of running personality on Blue Gene/P	276
Appendix D. Porting applications		279
Appendix E. Mapping		281
Appendix F. Statement of completion		285
References		287
Related publications		291
	IBM Redbooks	291
	Other publications	291
	Online resources	293
	How to get IBM Redbooks	294
	Help from IBM	294
Index		295

Notices

This information was developed for products and services offered in the U.S.A.

IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service might be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right might be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM might have patents or pending patent applications covering subject matter described 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 Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM might make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM might use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.


This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You might copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®	General Parallel File System™	POWER5™
Blue Gene/L™	GPFS™	POWER6™
Blue Gene/P™	IBM®	Redbooks®
Blue Gene®	LoadLeveler®	Redbooks (logo)  ®
DB2 Universal Database™	PowerPC®	System p™
DB2®	POWER™	Tivoli®
eServer™	POWER4™	

The following terms are trademarks of other companies:

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names might be trademarks or service marks of others.

Preface

This IBM® Redbooks® publication is one in a series of IBM books written specifically for the IBM System Blue Gene/P Solution. The Blue Gene/P system is the second generation of a massively parallel supercomputer from IBM in the IBM System Blue Gene® Solution series. This book provides an overview of the application development environment for the Blue Gene/P system. It is intended to help programmers understand the requirements to develop applications on this high-performance massively parallel supercomputer.

In this book, we explain instances where the Blue Gene/P system is unique in its programming environment. We also attempt to look at the differences between the IBM System Blue Gene/L™ Solution and the Blue Gene/P Solution. This book does not delve into great depth about the technologies that are commonly used in the supercomputing industry, such as Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) nor tries to teach parallel programming. References are provided in those instances for you to find more information if desired.

Prior to reading this book, you must have a strong background in high-performance computing (HPC) programming. The high-level programming languages that are used throughout this book are C/C++ and Fortran95. Previous experience using the Blue Gene/L system can help you understand better some concepts in this book that we do not extensively discuss. However, several IBM Redbooks publications about the Blue Gene/L system are available for you to obtain general information about the Blue Gene/L system. We recommend that you refer to “IBM Redbooks” on page 291, for a list of those publications.

The team that wrote this book

This book was produced in collaboration with the IBM Blue Gene developers at IBM Rochester, Minnesota, and IBM Blue Gene developers at the IBM T. J. Watson Center in Yorktown Heights, N.Y. The information presented in this book is direct documentation of many of the Blue Gene/P hardware and software features. This information was published by the International Technical Support Organization, Rochester, MN.

Carlos P. Sosa is a Senior Technical Staff Member in the Blue Gene Development Group of IBM, where he has been the team lead of the Chemistry and Life Sciences high-performance effort since 2006. For the past 18 years, he has focused on scientific applications with emphasis in Life Sciences, parallel programming, benchmarking, and performance tuning. He received a Ph.D. degree in Physical Chemistry from Wayne State University and completed his post-doctoral work at the Pacific Northwest National Laboratory. His areas of interest are future IBM POWER™ architectures, Blue Gene, Cell Broadband, and cellular molecular biology.

We thank the following people and their teams for their contributions to this book:

- ▶ Tom Liebsch for being the lead source for hardware information
- ▶ Harold Rodakowski for software information
- ▶ Thomas M. Gooding for kernel information
- ▶ Michael Blocksome for parallel paradigms
- ▶ Michael T. Nelson and Lynn Boger for their help with the compiler
- ▶ Thomas A. Budnick for his assistance with APIs
- ▶ Brant L. Knudson and Paul Allen for their extensive contributions

We also thank the following people for their contributions to this project:

Gary Lakner
Gary Mullen-Schultz
ITSO, Rochester, MN

Dino Quintero
ITSO, Poughkeepsie, NY

Paul Allen
Mike Blocksome
Lynn Boger
Thomas A. Budnik
Ahmad Faraj
Thomas M. Gooding
Nicholas Goracke
Todd Inglet
Brant L. Knudson
Tom Liebsch
Mark Megerian
Sam Miller
Mike Mundy
Tom Musta
Mike Nelson
Jeff Parker
Joseph Ratterman
Richard Shok
Brian Smith
IBM Rochester

Philip Heidelberg
Sameer Kumar
Martin Ohmacht
James C. Sexton
Robert E. Walkup
Robert Wisniewski
IBM Watson Center

Mark Mendell
IBM Toronto

Ananthanaraya Sugavanam
Enci Zhong
IBM Poughkeepsie

Kirk Jordan
IBM Waltham

Jerrold Heyman
IBM Raleigh

Subba R. Bodda
IBM India

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

- ▶ Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- ▶ Send your comments in an e-mail to:

redbooks@us.ibm.com

- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400



Part 1

Blue Gene/P: System and environment overview

The IBM System Blue Gene Solution is the next generation on massively parallel systems produced by IBM. It follows on the tradition established by the IBM Blue Gene/L Solution in challenging our thinking to take advantage of this innovative architecture. This next generation of supercomputers follows the winning formula provided as part of the Blue Gene/L Solution, that is, orders of magnitude in size and substantially more efficient in power consumption.

In this part, we present an overview of the two main topics of this book: hardware and software environment. This part includes the following chapters:

- ▶ Chapter 1, “Hardware overview” on page 3
- ▶ Chapter 2, “Software overview” on page 15



Hardware overview

In this chapter, we provide a brief overview of hardware. This chapter is intended for programmers who are interested in learning about the Blue Gene/P system. This chapter is also an overview for programmers who are already familiar with the Blue Gene/L system and want to understand the differences between the Blue Gene/L and Blue Gene/P systems.

It is important to understand where the Blue Gene/P system fits within the multiple systems that are currently available in the market. To gain a historical perspective as well as a perspective from an applications point of view, we recommend that you read the first chapter of the book *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Although this book is written for the Blue Gene/L system, these concepts apply to the Blue Gene/P system.

In this chapter, we describe the Blue Gene/P architecture. We also provide an overview of the machine with a brief description of some of the components. Specifically we address the following topics:

- ▶ “System architecture overview” on page 4
- ▶ “What is new on Blue Gene/P” on page 7
- ▶ “Microprocessor” on page 8
- ▶ “Compute Nodes” on page 9
- ▶ “I/O Nodes” on page 10
- ▶ “Networks” on page 10
- ▶ “Blue Gene/P programs” on page 11
- ▶ “Blue Gene specifications” on page 12
- ▶ “Host system” on page 13
- ▶ “Host system software” on page 14

1.1 System architecture overview

The IBM System Blue Gene Solution is a revolutionary and important milestone for IBM in the high-performance computing arena. The Blue Gene/L system has been the fastest supercomputer in the last few years as noted by the TOP500 organization.¹ Now IBM has introduced the Blue Gene/P system as the next-generation of massively-parallel supercomputers, based on the same successful architecture in the Blue Gene/L system.

The Blue Gene/P system includes the following key features and attributes among others:

- ▶ Dense number of cores per rack: 4096 cores per rack
- ▶ PowerPC®, Book E compliant, 32-bit microprocessor, 850 MHz
- ▶ Double precision, dual pipe floating point acceleration on each core
- ▶ 24-inch/42U server rack air cooled
- ▶ Low power per flop ratio on Blue Gene/P™ compute application-specific integrated circuit (ASIC), 1.8 watts per GFlop/sec. per SOC
- ▶ Includes memory controllers, caches, network controllers, and high-speed input/output (I/O)
- ▶ Linux® Kernel running on I/O Nodes
- ▶ Message Passing Interface (MPI)² support between nodes via MPI library support
- ▶ Open Multi-Processing (OpenMP)³ application programming interface (API)
- ▶ Scalable control system based on external Service Node and Front End Node
- ▶ Standard IBM XL family of compilers⁴ support with XLC/C++, XLF, and GNU Compiler Collection⁵
- ▶ Software support for LoadLeveler®,⁶ General Parallel File System™ (GPFS™),⁷ and Engineering and Scientific Subroutine Library (ESSL)⁸

Figure 1-1 illustrates the Blue Gene/P system architecture. It provides an overview of the multiple system components, from the microprocessor to the full system.

The system contains the following components:

Chip	The Blue Gene/P base component is a quad-core chip (also referred throughout this book as a <i>Compute Node or node</i>). The frequency of a single core is 850 MHz.
Compute card	One chip is soldered to a small processor card, one per card, together with memory (DRAM), to create a compute card (one node). The amount of DRAM per card is 2 GB.
Compute Node card	The compute cards are plugged on a node card. These are two rows of sixteen compute cards on the card (planar). From zero to two I/O Nodes per Compute Node card can be added to the Compute Node card.
Rack	A rack holds a total of 32 Compute Node cards.
System	A full petaflop system consists of 72 racks.

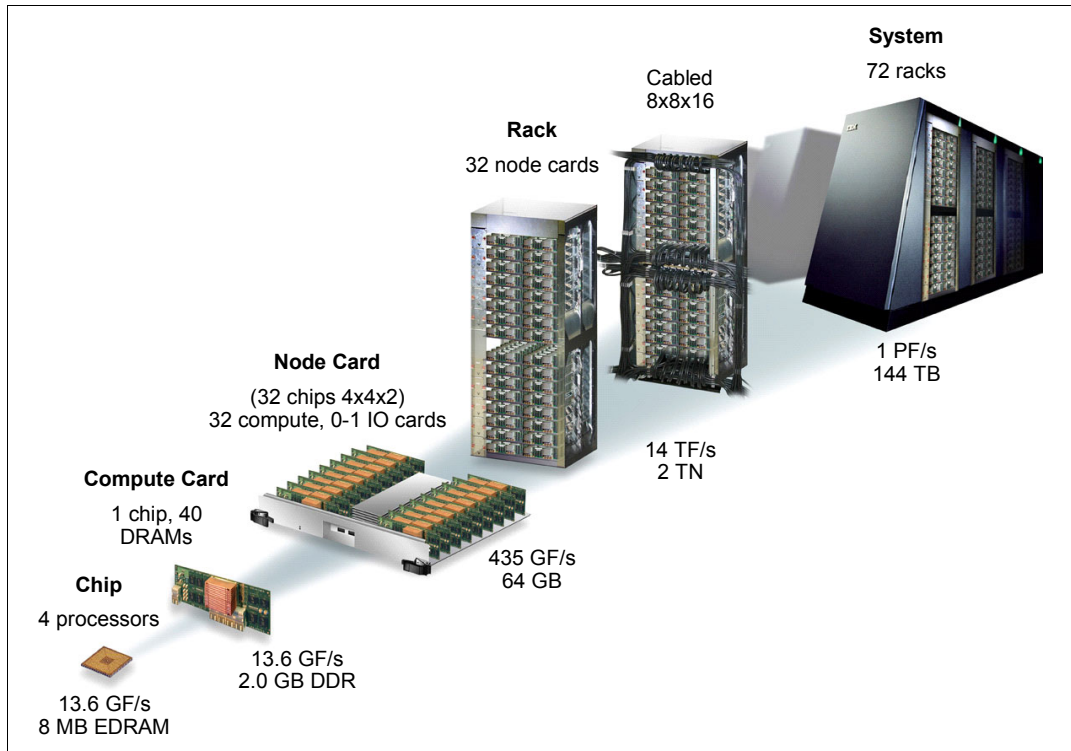


Figure 1-1 Blue Gene/P system overview from the microprocessor to the full system

1.1.1 System buildup

Similar to the Blue Gene/L system, the number of cores in a system can be computed as follows:

Number of cores = (number of racks) x (number of node cards per rack) x (number of compute cards per node card) x (number of cores per compute card)

This equation corresponds to cores and memory. However, I/O is carried out through the I/O Node that is connected externally via a 10 Gigabit Ethernet network. This network corresponds to the functional network. I/O Nodes are not considered in the previous equation.

Finally, the Compute and I/O Nodes are connected externally (to the outside world) via the following peripherals:

- ▶ One Service Node
- ▶ One or more Front End Nodes
- ▶ Global file system

1.1.2 Compute and I/O Nodes

Nodes are made of one quad-core with 2 GB of memory. These nodes do not have a local file system. Therefore, they must route I/O operations to an external device. In order to reach this external device (outside the environment), a Compute Node sends data to an I/O Node, which in turn, carries out the I/O requests.

The hardware for both types of nodes is virtually identical. They only differ in the way that they are used. For example, there might be also extra RAM on the I/O Nodes, and the physical connectors are different. A Compute Node runs a light, UNIX@-like proprietary kernel,

referred as *Compute Node Kernel*. The Compute Node Kernel ships all network bound requests to the I/O Node.

The I/O Node is connected to the external device through an Ethernet port to the 10 Gigabit functional network and can perform file I/O operations.

In the next section, we provide an overview of the Blue Gene environment, including all the components that fully populate the system.

1.1.3 Blue Gene/P environment

The Blue Gene/P environment consists of all the components that form part of the full system. Figure 1-2 illustrates the multiple components that form the Blue Gene/P environment.

The Blue Gene/P system consists of the following key components:

- Service Node** This node provides control of the Blue Gene/P system.
- Front End Node** This node provides access to the users to submit, compile, and build applications.
- Compute Node** This node runs applications. Users cannot login to this node.
- I/O Node** This node provide access to external devices, and I/O requests are all routed through this node.
- Functional network** This network is used by all components of the Blue Gene/P system except the Compute Node,
- Control network** This network is the service network for specific system control functions between the Service Node and the I/O Node.

In the remainder of this chapter, we describe these key components.

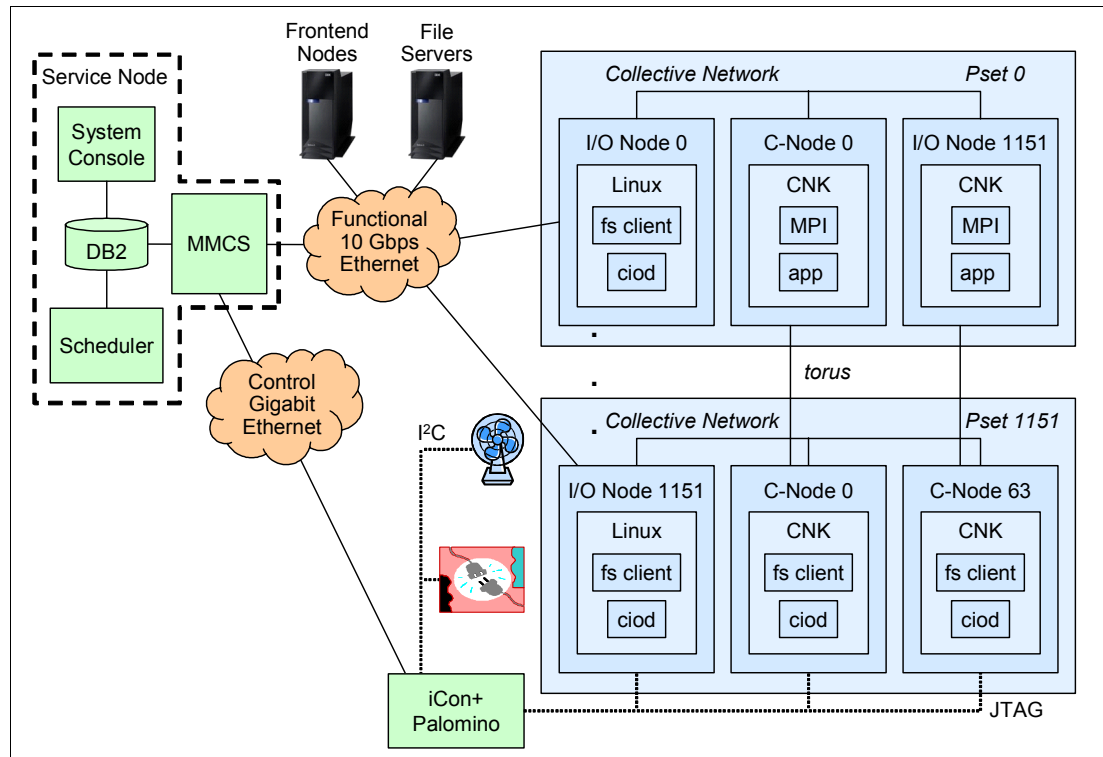


Figure 1-2 Blue Gene/P environment

1.2 What is new on Blue Gene/P

The Blue Gene/P Solution is a highly scalable multi-node supercomputer. Table 1-1 shows key differences between the Blue Gene/L and Blue Gene/P systems. Each node consists of a single ASIC and forty 512 Mb SDRAM-DDR2 memory chips. The nodes are interconnected through six networks, one of which connects the nearest neighbors into a three-dimensional (3D) torus or mesh. A system with 72 racks has a (x, y, z) 72 x 32 x 32 3D torus. The ASIC that is powering the nodes is in IBM CU-08 (CMOS9SF) system-on-a-chip technology and incorporates all of the compute and communication functionality that is needed by the core Blue Gene/P system. It contains 8 MiB of high-bandwidth embedded DRAM that can be accessed by the four cores in approximately 20 cycles for most L1 cache misses.

MiB: 1 MiB = 220 bytes = 1,048,576 bytes = 1,024 kibibytes

The scalable unit of Blue Gene/P packaging consists of 512 Compute Nodes on a doubled-sided board, called a *midplane*, with dimensions of approximately 20 inches x 25 inches x 34 inches.

Note: This is the smallest unit that supports the full 3D torus.

Each node operates at Voltage Drain Drain (VDD) = 1.1v or 1.2v or 1.3v, Temp_{junction} <70C, and a frequency of 850 MHz. Using an IBM PowerPC 450 processor and a single-instruction, multiple-data (SIMD), double precision floating point multiply add unit (double floating point multiply add (FMA)), it can deliver four floating point operations per cycle, or a theoretical maximum of 7.12 teraflops/sec. at peak performance for a single midplane. Two midplanes are contained within a single cabinet.

A midplane set of processing nodes, from a minimum of 16 to a maximum of 128, can be attached to a dedicated quad-processor I/O Node for handling I/O communications to and from the Compute Nodes. The I/O Node is assembled using the same ASIC as a Compute Node. Each Compute Node has a separate light-weight kernel, the Compute Node Kernel, which is designed for high performance scientific and engineering code. With help from the I/O Node kernel, the Compute Node Kernel provides Linux-like functionality to user applications. The I/O Nodes run an embedded Linux operating system that is extended to contain additional system software functionality to handle communication with the external world and other services.

The I/O Nodes of the Blue Gene/P system are connected to an external 10 Gigabit Ethernet switch, as previously mentioned, which provides I/O connectivity to file servers of a cluster-wide file system as illustrated in Figure 1-2. The 10 Gigabit Ethernet switch connects the Blue Gene/P system to the Front End Node and other computing resources. The Front End Node supports interactive logins, compiling, and overall system management.

Table 1-1 compares selected features between the Blue Gene/L and Blue Gene/P systems.

Table 1-1 Feature comparison between the Blue Gene/L and Blue Gene/P systems

Feature	Blue Gene/L	Blue Gene/P
Node		
Cores per node	2	4
Core clock speed	700 MHz	850 MHz
Cache coherency	Software managed	SMP
Private L1 cache	32 KB per core	32 KB per core
Private L2 cache	14 stream prefetching	14 stream prefetching
Shared L3 cache	4 MB	8 MB
Physical memory per node	512 MB - 1 GB	2 GB
Main memory bandwidth	5.6 GBps	13.6 GBps
Peak performance	5.6 GFlop/sec. per node	13.6 GFlop/sec. per node
Network topologies		
Torus		
Bandwidth	2.1 GBps	5.1 GBps
Hardware latency (nearest neighbor)	200 ns (32B packet) and 1.6 μ s (256B packet)	100 ns (32B packet) and 800 ns (256B packet)
Tree		
Bandwidth	700 MBps	1.7 GBps
Hardware latency (round trip worst case)	5.0 μ s	3.0 μ s
Full system		
Peak performance	410 TFlop/sec. (72 racks)	1 PFlop/Sec. (72 racks)
Power	1.7 MW (72 racks)	2.1 MW (72 racks)

Appendix A, “Blue Gene/P hardware naming convention” on page 265, provides an overview of how the Blue Gene/P hardware locations are assigned. You will find that the naming is used consistently throughout both the hardware and software chapters. Understanding the naming convention is particularly useful when running applications on the Blue Gene/P system.

1.3 Microprocessor

The microprocessors is a PowerPC 450, Book E compliant, 32-bit microprocessor with a clock speed of 850 MHz. The PowerPC 450 microprocessor, with double-precision floating point multiply add unit (double FMA), can deliver four floating point operations per cycle with 3.4 GFlop/sec. per core.

1.4 Compute Nodes

The Compute Node contains four PowerPC 450 processors with 2 GB of shared RAM and run a lightweight kernel to execute user-mode applications only. Typically all four cores are used for computation either in Dual Node Mode, Virtual Node Mode, or symmetrical multiprocessing. (Chapter 4, “Execution process modes” on page 37, covers these different modes.) Data is moved to and from the I/O Nodes over the global collective network. Figure 1-3 illustrates the components of a Compute Node.

Compute Nodes consist of the following components:

- ▶ Four 850 MHz PowerPC 450 cores
- ▶ 2 GB RAM per node
- ▶ Six connections to the torus network at 3.4 Gbps per link
- ▶ Three connections to the global collective network at 6.8 Gbps per link
- ▶ Four connections to the global interrupt network
- ▶ One connection to the control network (JTAG)

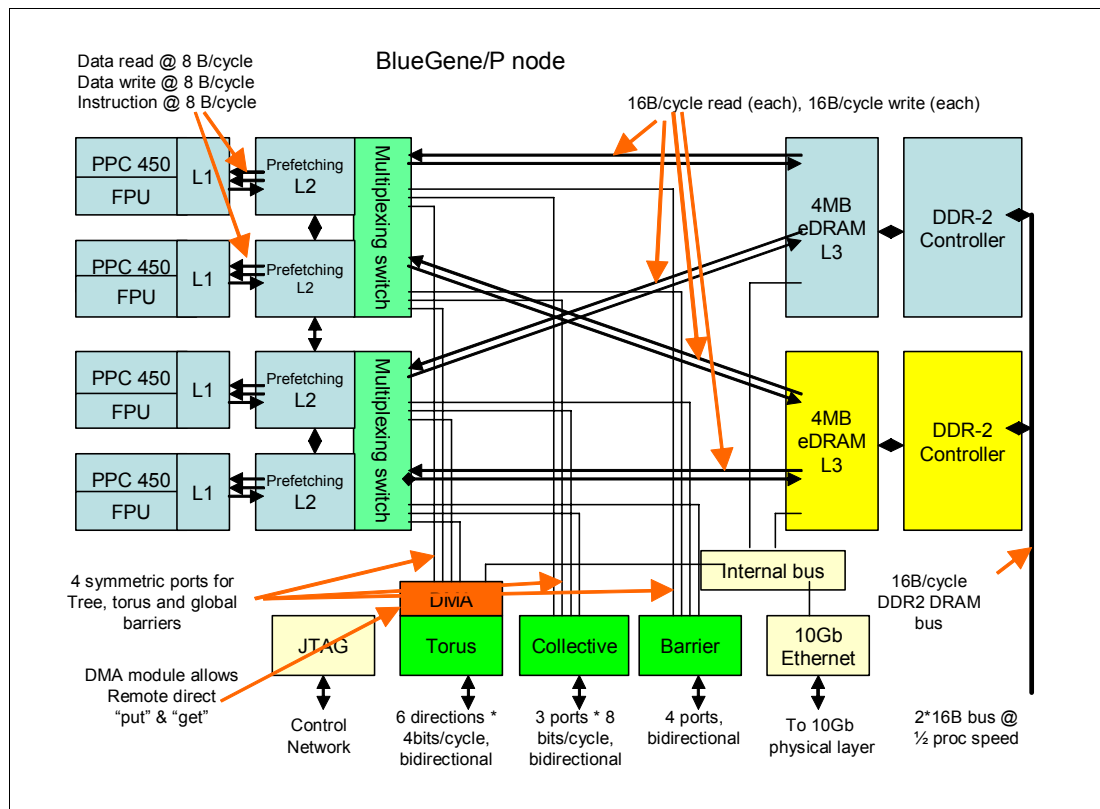


Figure 1-3 Blue Gene/P ASIC

1.5 I/O Nodes

I/O Nodes run an embedded Linux kernel with minimal packages required to support an Network File System (NFS) client and Ethernet network connections. They act as a gateway for the Compute Nodes in their respective rack to the external world (see Figure 1-4). The I/O Nodes present a subset of standard Linux operating interfaces to the user. The 10 Gigabit Ethernet interface of the I/O Nodes is connected to the core Ethernet switch.

The node cards have the following components among others:

- ▶ 850 MHz PowerPC 450 cores
- ▶ 2 GB DDR2 SDRAM
- ▶ One 10 Gigabit Ethernet adapter connected to the 10 Gigabit Ethernet network
- ▶ Three connections to the global collective network at 6.8 Gbps per link
- ▶ Four connections to the global interrupt network
- ▶ One connection to the control network (JTAG)

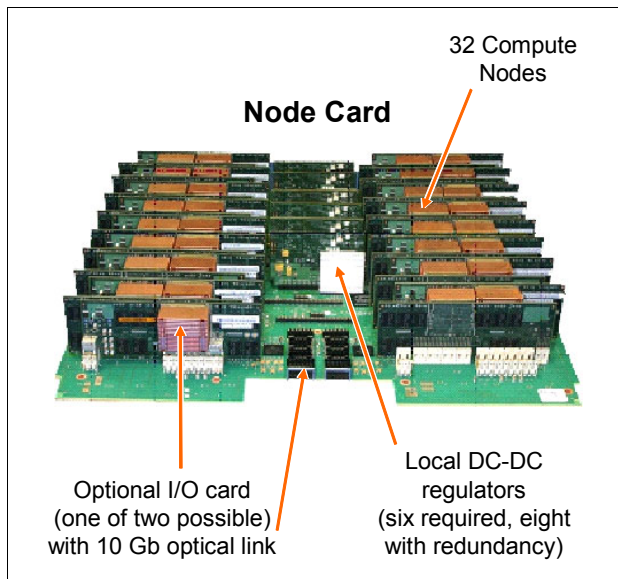


Figure 1-4 Blue Gene/P I/O Node card

1.6 Networks

Five networks are used for various tasks on the Blue Gene/P system:

- ▶ Three-dimensional torus: Point-to-point

The torus network is used for general-purpose, point-to-point message passing and multicast operations to a selected “class” of nodes. The topology is a three-dimensional torus constructed with point-to-point, serial links between routers that are embedded within the Blue Gene/P ASICs. Therefore, each ASIC has six nearest-neighbor connections, some of which can traverse relatively long cables. The target hardware bandwidth for each torus link is 425 MBps in each direction of the link for a total of 5.1 GBps bidirectional bandwidth per node. The three-dimensional torus network supports the following features:

- Interconnection of all Compute Nodes (73,728 for a 72 rack system)
- Virtual cut-through hardware routing
- 3.4 Gbps on all 12 node links (5.1 GBps per node)

- Communications backbone for computations
- 1.7/3.8 TBps bisection bandwidth, 67 TBps total bandwidth
- ▶ Global collective: Global operations

The global collective network is a high-bandwidth, one-to-all network that is used for collective communication operations, such as broadcast and reductions, and to move process and application data from the I/O Nodes to the Compute Nodes. Each Compute and I/O Node has three links to the global collective network at 850 MBps per direction for a total of 5.1 GBps bidirectional bandwidth per node. Latency on the global collective network is less than 2 μ s from the bottom to top of the Collective, with an additional 2 μ s latency to broadcast to all. The global collective network supports the following features:

 - One-to-all broadcast functionality
 - Reduction operations functionality
 - 6.8 Gbps of bandwidth per link; latency of network traversal 2 μ s
 - 62 TBps total binary network bandwidth
 - Interconnects all compute and I/O Nodes (1088)
- ▶ Global interrupt: Low latency barriers and interrupts

The global interrupt network is a separate set of wires based on asynchronous logic, which forms another network that enables fast signaling of global interrupts and barriers (global AND or OR). Round-trip latency to perform a global barrier over this network for a 72 K node partition is approximately 1.3 microseconds.
- ▶ 10 Gigabit Ethernet: File I/O and host interface

The 10 Gigabit Ethernet (optical) network consists of all I/O Nodes and discrete nodes that are connected to a standard 10 Gigabit Ethernet switch. A Cisco switch is typically used because it can be configured as a non-blocking switch and is scalable from a one-frame to many-frame configuration without requiring additional switches. The Compute Nodes are not directly connected to this network. All traffic is passed from the Compute Node over the global collective network to the I/O Node and then onto the 10 Gigabit Ethernet network.
- ▶ Control: Boot, monitoring, and diagnostics

The control network consists of a JTAG interface to a 1 Gigabit Ethernet interface with direct access to shared SRAM in every Compute and I/O Node. The control network is used for system boot, debug, and monitoring. It allows the Service Node to provide runtime non-invasive reliability, availability, and serviceability (RAS) support as well as non-invasive access to performance counters.

1.7 Blue Gene/P programs

The Blue Gene/P software for the Blue Gene/P core rack includes the following programs:

- ▶ Compute Node Kernel, MPI support for hardware implementation and abstract device interface, control system, and system diagnostics
- ▶ Compute Node Kernel and services

Provides an environment for execution of user processes. The services that are provided are process creation and management, memory management, process debugging and RAS management.
- ▶ I/O Node Kernel and services

Provides file system access and sockets communication to applications executing in the Compute Node.

- ▶ GNU Compiler Collection Toolchain Patches (Blue Gene/P changes to support GNU Compiler Collection)

The system software that is provided with each Blue Gene/P core rack or racks includes the following programs:

- ▶ DB2® Universal Database™ Enterprise Server Edition: System administration and management
Provide overall control and monitoring of the Blue Gene/P system. These services are performed from the Service Node.
- ▶ Compilers: XL C/C++ Advanced Edition for Linux with OpenMP support and XLF (Fortran) Advanced Edition for Linux

1.8 Blue Gene specifications

Table 1-2 lists the features of the Blue Gene/P Compute Nodes and I/O Nodes.

Table 1-2 Blue Gene/P node properties

Node properties	
Node processors (Compute and I/O)	Quad 450 PowerPC
Processor frequency	850 MHz
Coherency	symmetrical multiprocessing
L1 Cache (private)	32 KB per core
L2 Cache (private)	14 stream prefetching
L3 Cache size (shared)	8 MB
Main store memory/node	2 GB
Main store memory bandwidth	16 GBps
Peak performance	13.6 GFlop/sec. (per node)
Torus network	
Bandwidth	6 GBps
Hardware latency (nearest neighbor)	64 ns (32 B packet), 512 ns (256 B packet)
Hardware latency (worst case)	3 μs (64 hops)
Global collective network	
Bandwidth	2 GBps
Hardware latency (round trip worst case)	2.5 μs
System properties (for 73,728 Compute Nodes)	
Peak performance	1 PFlop/sec.
Average/peak total power	1.8 MW/2.5 MW (25 kW/34 kW per rack)

1.9 Host system

In addition to the Blue Gene/P core racks, the host system shown in Figure 1-5 is required for a complete Blue Gene/P system. There is generally one host rack for the core Ethernet switch, Service Node, and Front End Node. It might also house the Hardware Management Console (HMC) control node, monitor, keyboard, KVM switch, terminal server, and Ethernet modules.

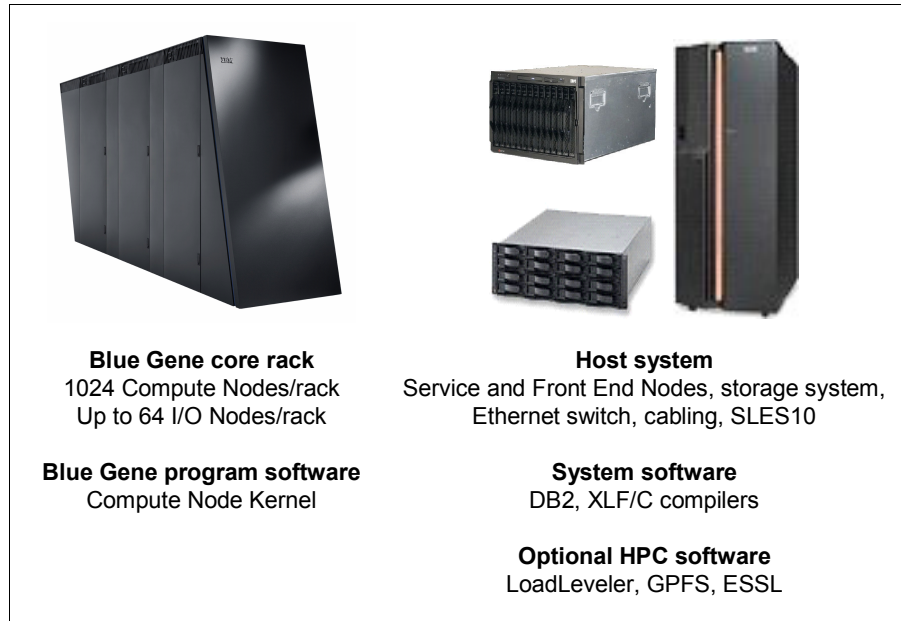


Figure 1-5 Blue Gene/P rack and host system

1.9.1 Service Node

The Service Node performs many functions for the operation of the Blue Gene/P system, including system boot, machine partitioning, system performance measurements, and monitoring system health. The Service Node uses DB2 as the data repository for system and state information. The Service Node can also be used as the Front End Node.

1.9.2 Front End Nodes

The Front End Node provide interfaces for users to login, compile their applications, and submit their jobs to run from these nodes. They have direct connections to both the Blue Gene/P internal VLAN and the public Ethernet networks.

1.9.3 Storage Nodes

The Storage Nodes provide mass storage for the Blue Gene/P system. We recommend that the Storage Nodes run GPFS locally in order to provide a single unified file system namespace to the Blue Gene/P system. However the I/O Nodes will access the GPFS file system over standard NFS mounts.

The storage rack generally contains the terminal server, Storage Nodes with RAM, Gigabit Ethernet adapters connected to the core Ethernet switch, and adapters connected to a hard disk drive (HDD).

1.10 Host system software

The operating system requires installation of SUSE Linux Enterprise Server 10 (SLES10, 64 bit) on the Service Node and Front End Node.

The following software applications for high-performance computing are optionally available for the Blue Gene/P system:

- ▶ Cluster System Management 1.5
Partition and job management provides allocation of nodes to job partitions; Service Node provides the core partition and job management services.
- ▶ File system: GPFS for Linux Server with NFS Client
- ▶ Job Scheduler: LoadLeveler for Blue Gene/P
- ▶ Engineering and Scientific Subroutine Library
- ▶ Application development tools for Blue Gene/P, which include debugging environments, application performance monitoring and tuning tools, and compilers



Software overview

In this chapter, we provide an overview of the software that runs on the Blue Gene/P system. As shown in Chapter 1, “Hardware overview” on page 3, the Blue Gene/P environment consists of Compute and I/O Nodes. It also has an external set of systems where users can perform system administration and management, partition and job management, application development, and debugging. In this heterogeneous environment, software must be able to interact.

Specifically, we cover the following topics:

- ▶ “Blue Gene/P software at a glance” on page 16
- ▶ “Compute Node Kernel” on page 17
- ▶ “Message Passing Interface on Blue Gene/P” on page 18
- ▶ “Memory considerations” on page 18
- ▶ “Other considerations” on page 22
- ▶ “Compilers overview” on page 23
- ▶ “I/O Node software” on page 24
- ▶ “Management software” on page 26

2.1 Blue Gene/P software at a glance

Blue Gene/P software includes the following key attributes among others:

- ▶ Full Linux kernel running on I/O Nodes
- ▶ Proprietary kernel dedicated for the Compute Nodes
- ▶ Message Passing Interface (MPI)⁹ support between nodes via MPI library support
- ▶ Open Multi-Processing (OpenMP)¹⁰ application programming interface (API)
- ▶ Scalable control system based on an external Service Node and Front End Node
- ▶ Standard IBM XL family of compilers¹¹ support with XLC/C++, XLF, and GNU Compiler Collection¹²
- ▶ Software support that includes LoadLeveler,¹³ GPFS,¹⁴ and Engineering and Scientific Subroutine Library (ESSL)¹⁵

From a software point of view, the Blue Gene/P system is comprised of the following components:

- ▶ A Compute Node
- ▶ I/O Node
- ▶ Front End Node where users compile and submit jobs
- ▶ The control management network
- ▶ The Service Node, which provides capabilities to manage jobs running in the racks
- ▶ Hardware in the racks

The Front End Node consists of the interactive resources on which users login to access the Blue Gene/P system. Users edit and compile applications, create job control files, launch jobs on the Blue Gene/P system, post process output, and perform other interactive activities. System administrators also use the Front End Node to control and configure the Blue Gene/P system.

An Ethernet switch is the main communication path for applications that run on the Compute Node to the external devices. This switch provides high-speed connectivity to the file system, which is the main disk storage for the Blue Gene/P system. This switch also gives other resources access to the files on the file system.

A Control and Management Network provides system administrators with a separate command and control path to the Blue Gene/P system. This private network is not available to unprivileged users.

The software for the Blue Gene/P system consists of the following integrated software subsystems:

- ▶ System administration and management
- ▶ Partition and job management
- ▶ Application development and debugging tools
- ▶ Compute Node Kernel and services
- ▶ I/O Node Kernel and services

The five software subsystems are required in three hardware subsystems:

- ▶ Host complex (including Front End Node and Service Node)
- ▶ I/O Node
- ▶ Compute Node

Figure 2-1 illustrates these components.

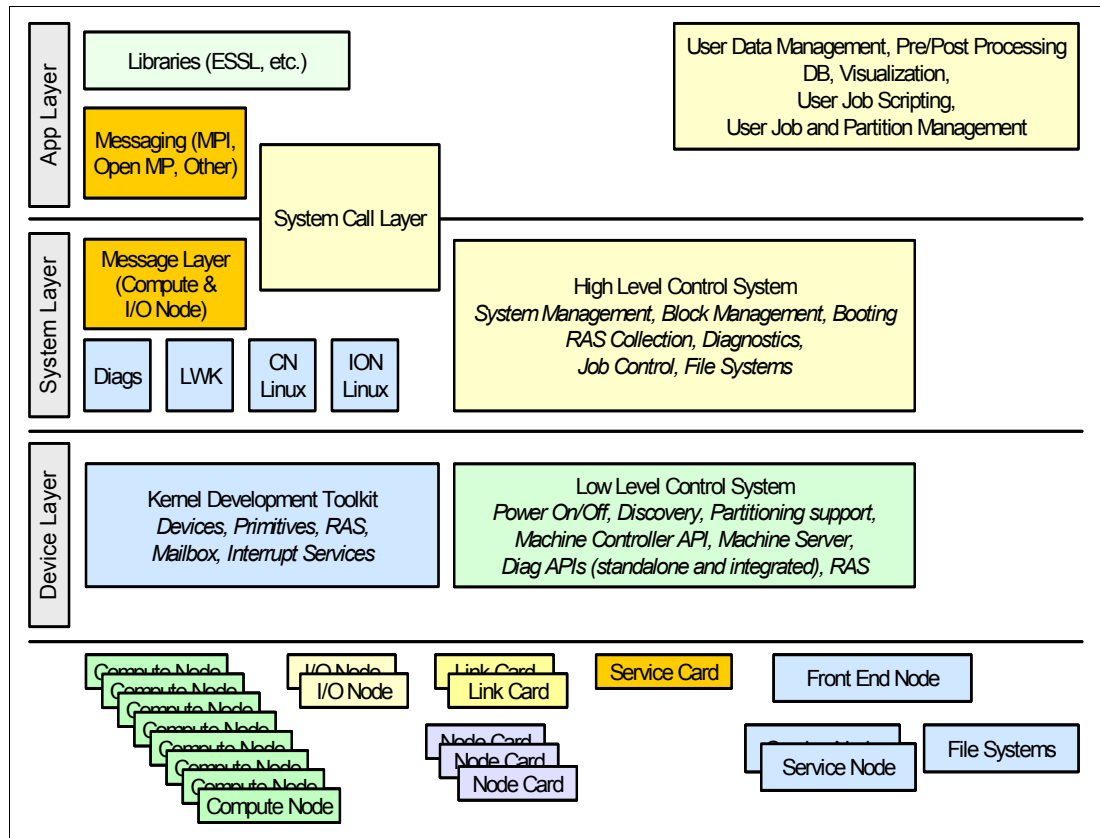


Figure 2-1 Software stack overview

The software environment illustrated in Figure 2-1 relies on a series of header files and libraries. A selected set is listed in Appendix B, “Header files and libraries” on page 271.

2.2 Compute Node Kernel

The Compute Node Kernel provides an environment for execution of user processes. Compute Node Kernel services include:

- ▶ Process creation and management.
- ▶ Memory management.
- ▶ Process debugging.
- ▶ Reliability Availability and Serviceability (RAS) management.
- ▶ File I/O.
- ▶ Network.

The Compute Nodes on Blue Gene/P are implemented as quad cores on a single chip with 2 GB of dedicated physical memory in which applications run.

There are three main modes in which a process is executed on Blue Gene/P nodes:

- ▶ Symmetrical Multiprocessing (SMP) Node Mode
- ▶ Virtual Node Mode (VN)
- ▶ Dual Node Mode (DUAL)

Application programmers see the Compute Node Kernel software as a Linux-like operating system. This type of operating system is accomplished on Blue Gene/P software stack by providing a standard set of runtime libraries for C, C++, and Fortran95. To the extent that is possible, the supported functions maintain open standard POSIX-compliant interfaces. We discuss the Compute Node Kernel further in Part 2, “Kernel overview” on page 27. Applications can access system calls that provide hardware or system features, as illustrated by the examples in Appendix C, “Files on architectural features” on page 275.

2.2.1 Threading support on Blue Gene/P

The threading implementation on the Blue Gene/P system supports OpenMP. The XL OpenMP implementation provides a futex compatible syscall interface, so that the NPTL pthreads implementation in glibc runs without modification. These syscalls allow only a total of four threads, limited support for `mmap`, and testing only with usage behavior of OpenMP. The Compute Node Kernel provides a special thread function for I/O handling in MPI.

Important: The Compute Node Kernel supports the execution of one quad-threaded process, where each of the four cores in the Blue Gene/P node is assigned hard affinity to each of a maximum of four threads. The Compute Node Kernel also supports the execution of four single-threaded processes per core on a node.

2.3 Message Passing Interface on Blue Gene/P

The implementation of MPI on the Blue Gene/P system is the MPICH2 standard that was developed by Argonne National Labs. For more information about MPICH2, see the Message Passing Interface (MPI) standard Web site at:

<http://www-unix.mcs.anl.gov/mpi/>

A function of the MPI-2 standard that is not supported by Blue Gene/P is Dynamic Process Management (creating new MPI processes).¹⁶ However, the various thread modes are supported.

2.4 Memory considerations

On the Blue Gene/P system, the entire physical memory of a Compute Node is 2 GB. Of that space, some is allocated for the Compute Node Kernel itself. In addition, shared memory space is also allocated to the user process at the time at which the process is created.

Important: In C, C++ and Fortran, the malloc routine returns a NULL pointer when users request more memory than the physical memory available. We recommend that you always check malloc return values for validity.

The hardware issues a segment violation (SEGV) interrupt and terminates the application on all nodes in the partition when referencing data using a NULL pointer.

The Compute Node Kernel keeps track of collisions of stack and heap as the heap is expanded via a `brk syscall`. On the Blue Gene/P system, there are stack guard pages.

The Compute Node Kernel and its private data are protected from read/write by the user process or threads. The code space of the process is protected from writing by the process or

threads. Code and read-only data are shared between the processes in Virtual Node Mode unlike in the Blue Gene/L system.

In general, give careful consideration to memory when writing applications for the Blue Gene/P system. Unlike the Blue Gene/L system, at the time at which this book was written, each node had 2 GB of physical memory.

As previously mentioned, memory addressing is an important topic in regard to the Blue Gene/P system. Here we update the section on memory addressing presented in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686, as it applies to Blue Gene/P. As mentioned in that section, when an application stores data in memory, it can be classified as follows:

- data** Initialized static and common variables
- bss** Uninitialized static and common variables
- heap** Controlled allocatable arrays
- stack** Controlled automatic arrays and variables

You can use the Linux **size** command to gain an idea of the memory size of the program. However, the **size** command does not provide any information about the runtime memory usage of the application nor on the classification of the types of data. Figure 2-2 illustrates memory addressing based on the different node modes that are available on the Blue Gene/P system.

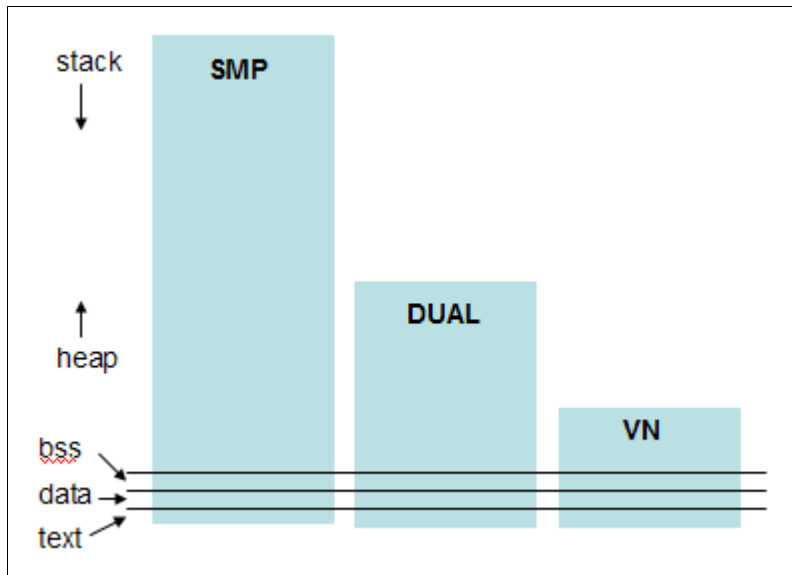


Figure 2-2 Memory addressing on the Blue Gene/P system as a function of the different node modes

Table 2-1 compares the three modes based on the memory addressing program introduced in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. This program is shown in Example 2-1 on page 20 with the memory parameters used here.

Table 2-1 Memory addressing as a function of the three Blue Gene/P node modes

Parameters	SMP	DUAL	VN
heapsize function address	10012c0	10012c0	10012c0
printf function address	1002434	1002434	1002434
end of code address	1067b38	1067b38	1067b38

Parameters	SMP	DUAL	VN
variable initialized address	1101c98	1101c98	1101c98
end of data address	11023a8	11023a8	11023a8
start of bss address	11023a8	11023a8	11023a8
variable uninitialized address	11023a8	11023a8	11023a8
end of bss address	1103cf0	1103cf0	1103cf0
start of heap address	1600010	1600010	1600010
end of heap_array0 address	7e6000c	3fc000c	202000c
start of heap_array address	7e601010	3fc01010	20201010
end of heap_array address	7e70100c	3fd0100c	2030100c
end of heap address	1127000	1127000	1127000
Heap size	144144 23310	144144 23310	144144 23310
start of stack address	7ffd31c	403fd31c	209fd31c
end of stack address	2ffd320	1dfd320	1dfd320

The text section starts at address 0. The heap section begins from the bottom, after the data and bss sections. The stack section starts from the top, at address 7ffd31c in SMP Node Mode (approximately 2 GB) and at address 403fd31c in Dual Node Mode (approximately 1 GB) and 209fd31c in Virtual Node Mode (approximately 512 MB).

Example 2-1 Program for memory addressing

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>           // for 'brk ()' and 'sbrk ()'

extern int  _etext;           // end of code area
extern int  _edata;           // end of data area
extern int  __bss_start;     // start of bss area
extern int  _end;            // end of bss area

#define SIZE 1024*256        // 1 MB of long
// #define SZ 2000           // SMP node mode
// #define SZ 998           // DUAL node mode
#define SZ 492              // VN node mode

unsigned long heapsize ()
{
    return (unsigned long) sbrk (0) - (unsigned long) & _end;
}

```

```

void * gotostack ()
{
    long st[SZ*SIZE];
    st[0]=123456;
    printf ("\nstart of stack address      %9lx\n", &st[SZ*SIZE-1]);
    printf ("end of stack address          %9lx\n\n", st);
}

int initialized = 123;          // goes to data area
int uninitialized;           // goes to bss
int main (int argc, char * argv [])
{
    int loop;
    long long_integer;
    long * heap_array;
    long * heap_array0;

    errno=0;
    if ((heap_array0 = (long *) malloc (SZ*SIZE*sizeof(long_integer))) == NULL)
        printf("error, could not allocate\n");
        if( errno !=0 ){
            printf ("malloc errno : %d\n",errno); errno=0; }
    if ((heap_array = (long *) malloc (SIZE*sizeof(long_integer))) == NULL)
        printf("error, could not allocate\n");
        if( errno !=0 ){
            printf ("malloc errno : %d\n",errno); errno=0;}
    printf ("Memory mapping\n\n");
    printf ("heapsize function address      %9lx\n", heapsize);
    printf ("printf function address           %9lx\n", printf);
    printf ("end of code address                 %9lx\n", &_etext);
    printf ("variable initialized address       %9lx\n", &initialized);
    printf ("end of data address                 %9lx\n", &_edata);
    printf ("start of bss address                %9lx\n", &_bss_start);
    printf ("variable uninitialized address     %9lx\n", &uninitialized);
    printf ("end of bss address                  %9lx\n", &_end);
    printf ("start of heap address               %9lx\n", heap_array0);
    printf ("end of heap_array0 address          %9lx\n", &heap_array0[SZ*SIZE-1]);
    printf ("start of heap_array address         %9lx\n", heap_array);
    printf ("end of heap_array address           %9lx\n", &heap_array[SIZE-1]);
    printf ("end of heap address                 %9lx\n", sbrk(0));
    long_integer=heapsize();
    printf ("\nHeap size %lu %9lx\n\n",long_integer,long_integer);
    gotostack();
}

```

2.4.1 Memory leaks

Given that there is no virtual paging on the Blue Gene/P system, any memory leaks in your application can quickly consume available memory. When writing applications for the Blue Gene/P system, you must be especially diligent that you release all memory that you allocate. This is true on any machine. Therefore, we recommend that an application is ported to multiple architectures.

2.4.2 Memory management

The Blue Gene/P computer implements a 32-bit memory model. It does not support a 64-bit memory model, but provides *large file support* and *64-bit integers*.

In the case of the Blue Gene/P system, if the memory requirement per MPI task is greater than 512 MB in Virtual Node Mode or greater than 1 GB in Dual Node Mode, then the application will not run on the Blue Gene/P system. However, in SMP Node Mode, 2 GB of memory are available. The application will only work if you take steps to reduce the memory footprint.

In some cases, you can reduce the memory requirement by distributing data that was replicated in the original code. In this case, additional communication might be needed. It might also be possible to reduce the memory footprint by being more careful about memory management in the application, such as by not defining arrays for the index that corresponds to the number of nodes.

2.4.3 Uninitialized pointers

Blue Gene/P applications run in the same address space as the Compute Node Kernel and the communications buffers. You can create a pointer that does not reference your own application's data, but rather that references the area used for communications. The Compute Node Kernel itself is well protected from rogue pointers.

2.5 Other considerations

It is important to understand that the operating system that is present on the Compute Node, the Compute Node Kernel, is not a full-fledged version of Linux. Because of this, there are areas in which you must use care, as explained in the following sections, when writing applications for the Blue Gene/P system. For a full list of supported system calls, see Part 2, "Kernel overview" on page 27.

2.5.1 Input/output

I/O is an area where you must pay special attention in your application. The Compute Node Kernel does not perform I/O. This is carried out by the I/O Node.

File I/O

A limited set of file I/O is supported. Do *not* attempt to use asynchronous file I/O, because it results in runtime errors.

Standard input

Standard input (stdin) is supported on the Blue Gene/P system. It is no longer necessary to pass input to your application using only file I/O.

Sockets calls

Sockets are supported on the Blue Gene/P system. For additional information, see Chapter 6, "System calls" on page 51.

2.5.2 Linking

Dynamic linking is not supported on the Blue Gene/L system. However, it is supported on the Blue Gene/P system. You can now statically link all code into your application or use dynamic linking.

2.6 Compilers overview

Read-only sections are supported in the Blue Gene/P system. However, this might not be true of read-only sections within dynamically located modules.

2.6.1 Programming environment overview

The diagram in Figure 2-3 provides a quick view into the software stack that supports the execution of Blue Gene/P applications.

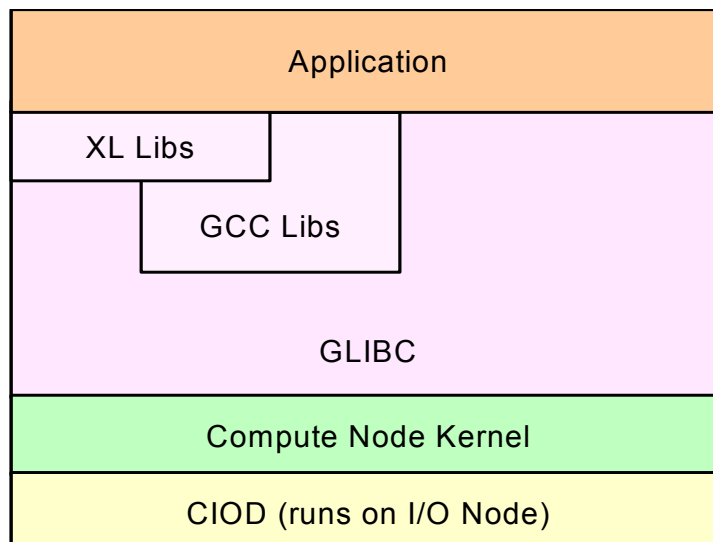


Figure 2-3 Software stack supporting the execution of Blue Gene/P applications

2.6.2 GNU Compiler Collection

The standard GNU Compiler Collection 4.1.1 for C, C++, and Fortran is supported on the Blue Gene/P system. The current versions are:

- ▶ gcc 4.1.1.
- ▶ binutils 2.17.
- ▶ glibc 2.4.

You can find the GNU Compiler Collection in the `/bgsys/drivers/ppcfloor/gnu-linux/bin` directory. For more information, see Chapter 8, “Developing applications with IBM XL compilers” on page 91.

2.6.3 IBM XL compilers

The following IBM XL compilers are supported for developing Blue Gene/P applications:

- ▶ XL C/C++ Advanced Edition V9.0 for Blue Gene/P
- ▶ XL Fortran Advanced Edition V11.1 for Blue Gene/P

See Chapter 8, “Developing applications with IBM XL compilers” on page 91, for more compiler-related information.

2.7 I/O Node software

The Blue Gene/P system is a massively parallel system with a large number of nodes. Compute Nodes are reserved for computations, and I/O is carried out via the I/O Nodes. These nodes serve as links between the Compute Nodes and external devices. For instance, applications running on Compute Nodes can access file servers and communicate with processes in other machines.

The I/O Node software and the Service Node software communicate to exchange various data relating to machine configuration and workload. Communications use a key-based authentication mechanism with keys using at least 256 bits.

The I/O Node Kernel is a standard Linux kernel and provides file-system access and sockets communication to applications that execute on the Compute Nodes.

2.7.1 I/O Node Kernel boot considerations

The I/O Node Kernel is designed to be booted as infrequently as possible due to the numerous possible failures of mounting remote file systems. The bootstrap process involves loading a ramdisk image and booting the Linux kernel. The ramdisk image is extracted to provide the initial file system, which contains minimal commands to mount the Service Node via the Network File System (NFS). The boot continues by running startup scripts from the NFS and running customer-supplied startup scripts to perform site-specific actions such as logging configuration and mounting high performance file systems.

The Blue Gene/P system has considerably more content over the Blue Gene/L system in the ramdisk image to reduce the load on the Service Node that was exported by the NFS as the I/O Node boot. Toolchain shared libraries and all of the basic Linux text and shell utilities are local to the ramdisk. Packages, such as GPFS, and customer provided scripts are NFS mounted for administrative convenience.

2.7.2 I/O Node file system services

The I/O Node Kernel supports an NFS client or GPFS client, which provides a file system service to application processes that execute on its associated Compute Node. The NFSv3 and GPFS file systems supported as part of the Blue Gene/L system continue with the Blue Gene/P system. As with the Blue Gene/L system, customers can still add their own parallel file systems by modifying Linux on the I/O Node as needed.

2.7.3 Socket services for the Compute Node Kernel

The I/O Node include a complete Internet Protocol (IP) stack, with TCP and UDP services. A subset of these services is available to user processes running on the Compute Node that is associated with an I/O Node. Application processes communicate with processes that are running on other systems using client side sockets via standard socket permissions and network connectivity. In addition, server-side sockets are available.

Note that the I/O Node implements the sockets so that all the Compute Nodes within a processor set (pset) behave as though the compute tasks are executing on the I/O Node. In particular, this means that the socket port number is a single address space within the pset and they share the IP address of the I/O Node.

2.7.4 I/O Node daemons

The I/O Node include the following daemons:

- ▶ Control and I/O daemon
- ▶ File-system client daemons
- ▶ Syslog
- ▶ sshd
- ▶ ntpd on at least one I/O Node

2.7.5 Control system

The control system retains the high-level components from the Blue Gene/L system with a considerable change in low-level components to accommodate the updated control hardware in the Blue Gene/P system as well as to increase performance for the monitoring system. The MMCS server and mcserver are now the processes that make up the control system on the Blue Gene/P system.

- ▶ The Midplane Management Control System (MMCS; console and server) is similar to the Blue Gene/L system in the way it handles commands, interacts with DB2, boots blocks, and runs jobs.
- ▶ mcServer is the process through which MMCS makes contact with the hardware (replacing idoproxy of the Blue Gene/L system). mcServer handles all direct interaction with the hardware. Low-level boot operations are now part of this process and not part of MMCS.
- ▶ The standard `mpirun` command to launch jobs can be used from any Front End Node or the Service Node. This command is often invoked automatically from a higher level scheduler.

The components that reside on the Service Node contain the following functions:

- ▶ Bridge APIs

A scheduler that dynamically creates and controls blocks typically uses Bridge APIs. A range of scheduler options includes ignoring these APIs and using `mpirun` on statically created blocks to full dynamic creation of blocks with pass-through midplanes. For the Blue Gene/P system, there is a new set of APIs that notifies the caller of any changes, in real time. Callers can register for various entities (*entities* are jobs, blocks, node cards, midplanes, and switches) and only see the changes. Callers can also set filters so that notifications occur for only specific jobs or blocks.

- ▶ **ciodb**

ciodb is now integrated as part of the MMCS server for the Blue Gene/P system. This is different from the Blue Gene/L system. ciodb is responsible for launching jobs on already booted blocks. Communication to ciodb occurs via the database and can be initiated by either `mpirun` or the Bridge APIs.
- ▶ **MMCS**

The MMCS daemon is responsible for configuring and booting blocks. It can be controlled either via a special console interface (similar to the Blue Gene/L system) or via the Bridge APIs. The mmcs daemon also is responsible for relaying RAS information into the RAS database.
- ▶ **mcServer**

The mcServer daemon has low-level control of the system, which includes a parallel efficient environmental monitoring capability as well as a parallel efficient reset and code load capability for configuring and booting blocks on the system. The diagnostics for the Blue Gene/P system directly leverage this daemon for greatly improved diagnostic performance over that of the Blue Gene/L system.
- ▶ **bgpmaster**

The bgpmaster daemon monitors the other daemons and restarts any failed components automatically.
- ▶ **Service actions**

Service actions are a suite of administrative shell commands that are used to service hardware. They are divided into device-specific actions with a “begin” and “end” action. Typically the “begin” action powers down hardware so it can be removed from the system, and the “end” action powers up the replacement hardware. The databases are updated with these operations, and they coordinate automatically with the scheduling system as well as the diagnostic system.

2.8 Management software

The Blue Gene/P management software is based on a set of databases that run on the Service Node. The database software is DB2.

2.8.1 Midplane Management Control System

Both Blue Gene/P hardware and software are controlled and managed by the MMCS. The Service Node, Front End Node, and the file servers are not under the control of the MMCS. The MMCS currently consists of several functions that interact with a DB2 database running on the Service Node.



Part 2

Kernel overview

The kernel provides the glue that makes all components in Blue Gene/P work together. In this part, we provide only an overview of the kernel functionality for applications developers. This part is for those who require information about system-related calls and interaction with the kernel.

This part contains the following chapters:

- ▶ Chapter 3, “Kernel functionality” on page 29
- ▶ Chapter 4, “Execution process modes” on page 37
- ▶ Chapter 5, “Memory” on page 43
- ▶ Chapter 6, “System calls” on page 51



Kernel functionality

In this chapter, we provide an overview of the functionality implemented as part of the Compute Node Kernel and I/O Node Kernel. We discuss the following topics:

- ▶ “System software overview” on page 30
- ▶ “Compute Node Kernel” on page 30
- ▶ “I/O Node Kernel” on page 32

3.1 System software overview

In general, the function of the kernel is to enable applications to run on a particular hardware system. This enablement consists of providing such services as applications execution, file I/O, memory allocation, and many others. In the case of the Blue Gene/P system, the system software provides two kernels:

- ▶ Compute Node Kernel
- ▶ I/O Node Kernel

3.2 Compute Node Kernel

The kernel that runs on the Compute Node is called the *Compute Node Kernel* and is IBM proprietary. It has a subset of the Linux system calls. The Compute Node Kernel is a flexible, lightweight kernel for Blue Gene/P Compute Nodes that are capable of supporting both diagnostic modes and user applications.

The Compute Node Kernel is intended to be a Linux-like operating system, from the application point-of-view, supporting a large subset of Linux compatible system calls. This subset is taken from the subset that is used successfully on the Blue Gene/L system, which demonstrates good compatibility and portability with Linux.

Now, as part of the Blue Gene/P system, the Compute Node Kernel supports threads and dynamic linking for further compatibility with Linux. The Compute Node Kernel has been tuned for the capabilities and performance of the Blue Gene/P System. In Figure 3-1, you see the interaction between the application space and the kernel space.

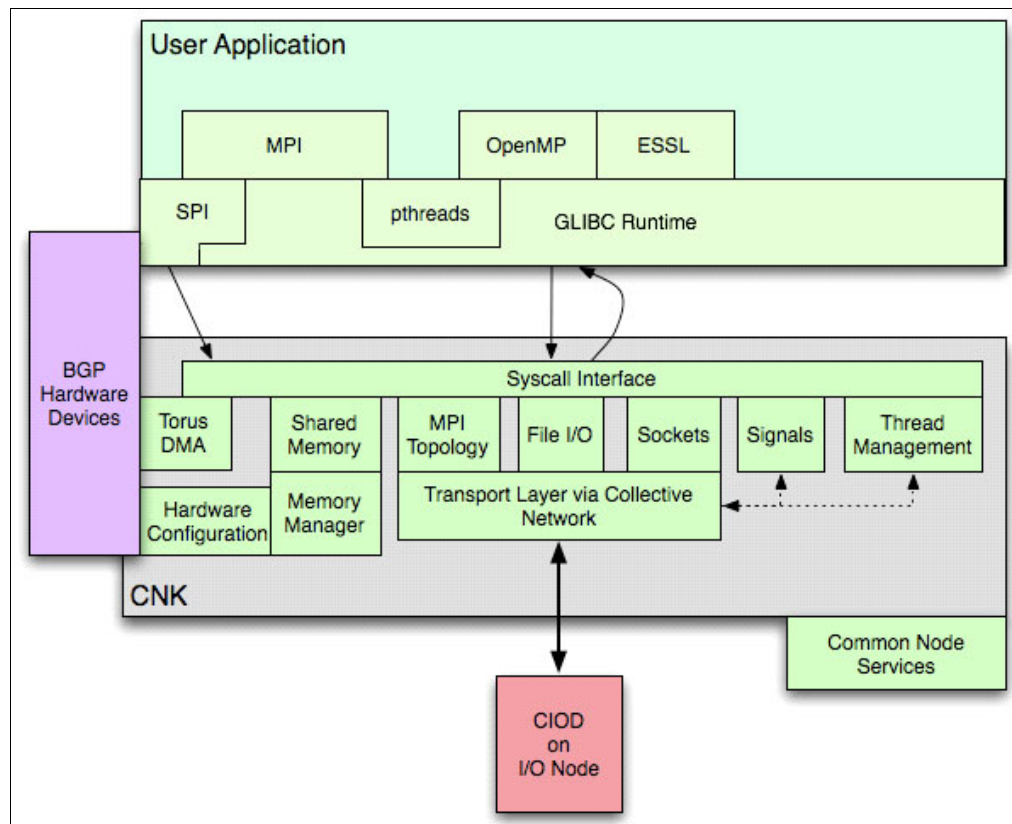


Figure 3-1 Compute Node Kernel overview

When running a user application, the Compute Node Kernel connects to the I/O Node via the collective network. This connection communicates to a process that is running on the Linux I/O Node called the *control and I/O daemon (CIOD)*. All function-shipped system calls are forwarded to the CIOD process and executed on the I/O Node.

At the user-application level, the Compute Node Kernel supports the following application programming interfaces (APIs) among others:

- ▶ Message Passing Interface (MPI)¹⁷ support between nodes via MPI library support
- ▶ Open Multi-Processing (OpenMP)¹⁸ API
- ▶ Standard IBM XL family of compilers⁻ support with XLC/C++, XLF, and GNU Compiler Collection¹⁹
- ▶ Highly optimized mathematical libraries such as IBM Engineering and Scientific Subroutine Library (ESSL)²⁰
- ▶ GNU Compiler Collection (GCC) C Library, or glibc, which is the C standard library and interface of GCC for a provider library plugging into an other library (system programming interfaces (SPIs))

The following services are some of those that are provided by the Compute Node Kernel:

- ▶ Torus direct memory access (DMA),²¹ which provides memory access for reading, writing, or doing both independently of the processing unit
- ▶ Shared-memory access on a local node
- ▶ Hardware configuration
- ▶ Memory management
- ▶ MPI topology
- ▶ File I/O
- ▶ Sockets connection
- ▶ Signals
- ▶ Thread management
- ▶ Transport layer via collective network

3.2.1 Boot sequence of a Compute Node

The Blue Gene/P hardware is a stateless system. When power is initially applied, the hardware must be externally initialized. Given the architectural and reliability improvements in the Blue Gene/P design, reset of the Compute Nodes should be an infrequent event.

The following procedure explains how to boot a Compute Node as part of the main partition. Independent reset of a single Compute Node and independent reset of a single I/O Node are different procedures.

The Compute Node Kernel must be loaded into memory after every reset of the Compute Node. In order to accomplish this task, several steps must occur to prepare a Compute Node Kernel for running an application:

1. The control system loads a small bootloader into SRAM.
2. The control system loads the *personality* into SRAM. The personality is a data structure that contains node-specific information, such as the X, Y, Z coordinates of the node.

Note: See Appendix C, “Files on architectural features” on page 275, for an example of how to use a personality.

3. The control system releases the Compute Node from reset.
4. The bootloader starts executing and initializes the hardware.
5. The bootloader communicates with the control system over the mailbox to load Common Node Services and Compute Node Kernel images.
6. The bootloader then transfers control to the Common Node Services.
7. The Common Node Services perform its setup and then transfer control to the Compute Node Kernel.
8. The Compute Node Kernel performs its setup and communicates to the CIOD.

At this point, the Compute Node Kernel will submit the job to the CIOD.

3.2.2 Common Node Services

Common Node Services provide low-level services that are both specific to the Blue Gene/P system and common to the Linux and the Compute Node Kernel. As such, these services provide a consistent implementation across node types while insulating the kernels from the details of the control system.

The Common Node Services provide the same low-level hardware initialization and setup interfaces to both Linux and the Compute Node Kernel. The interface to Common Node Services is described via doxygen and the interfaces are published.

The Common Node Services provide the following services:

- ▶ Access to the SRAM mailbox for performing I/O operations over the service network to the console
- ▶ Initialization of hardware for various networks
- ▶ Access to the personality
- ▶ Low-level services for RAS, including both event reporting and recovery handling
- ▶ Access to the Blue Gene interrupt controller

3.3 I/O Node Kernel

The kernel of the I/O Node (Figure 3-2) is referred as the *Mini-Control Program* (MCP). It is a port of the Linux Kernel, which means it is GPL/LGPL licensed. It is similar to the Blue Gene/L I/O Node. The I/O Node Kernel on the Blue Gene/P system has the following characteristics:

- ▶ Embedded Linux Kernel
 - Linux version 2.6.16
 - 4-way symmetrical multiprocessing (SMP)
 - Paging disabled (no swapping available)
- ▶ Ethernet
 - New 10 Gigabit Ethernet driver
 - Large Maximum Transmission Unit (MTU) support, which allows for Ethernet frames to be increased from the default value of 1500 bytes to 9000 bytes

- TCP checksum offload engine support
- Availability of /proc files for configuring and gathering status
- ▶ File systems supported
 - Network File System (NFS)
 - Parallel Virtual File System (PVFS)
 - General Parallel File System (GPFS)
 - Lustre File System
- ▶ CIOD
 - Lightweight proxy between Compute Nodes and the outside world
 - Debugger access into the Compute Nodes
 - SMP support

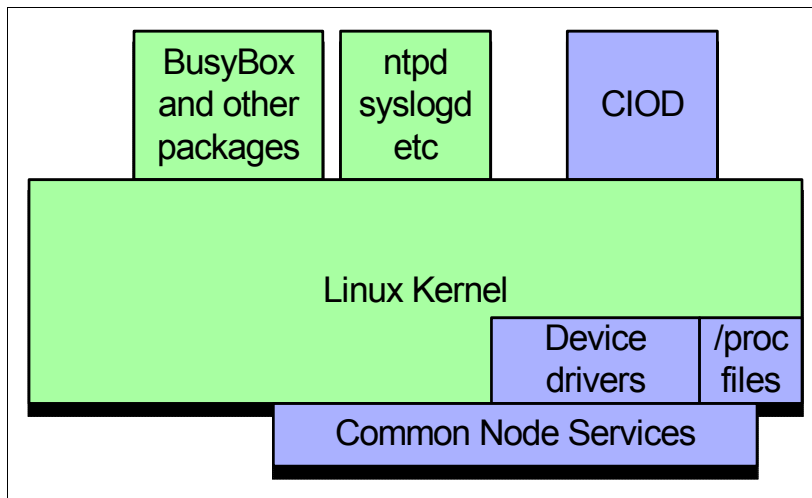


Figure 3-2 I/O Node Kernel overview

The I/O service is provided to the Compute Nodes from the Compute Node I/O proxy (CIOD), which is started by the initialization script during the boot procedure of the MCP. CIOD is a user-level process that controls and services applications in the Compute Node and interacts with the Midplane Management and Control System (MMCS).

3.3.1 Control and I/O daemon

The CIOD serves the following roles:

- ▶ Interface to and from the control system
- ▶ Proxy for the Compute Node
- ▶ Proxy for the debug server

Note: To access this functionality, use Telnet to connect to the I/O Node on port 9000 and type `help` for a list of commands.

Figure 3-3 shows a high-level overview of the CIOD.

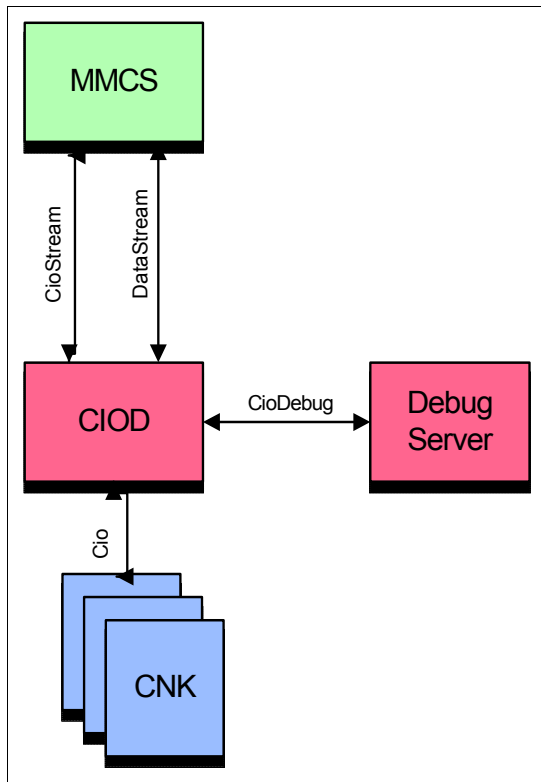


Figure 3-3 CIOD overview

The CIOD for the Blue Gene/P system includes the following major changes:

- ▶ Single process to many processes:
 - Takes advantage of 4-way SMP
 - Cleans up tracking of Compute Node
- ▶ CIOStream (control messages from the control system)
- ▶ DataStream (stdout/stderr/stdin messages)
- ▶ Cio protocol (interface to Compute Node Kernel)
- ▶ Support for tool daemons

CIOD threading architecture

CIOD reads from the collective network and places the message into the shared memory that is dedicated to the sending node I/O proxy. Figure 3-4 shows the threading architecture of the CIOD.

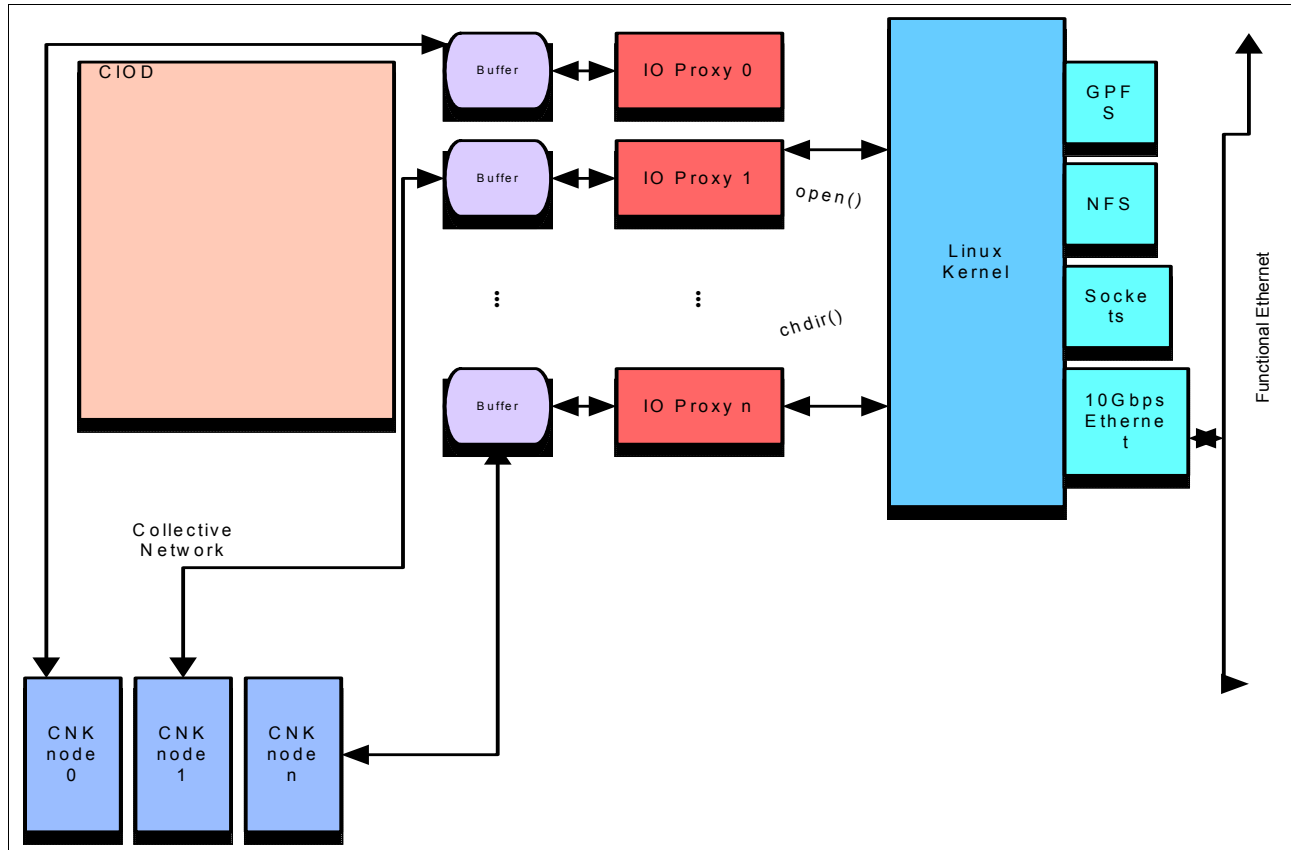


Figure 3-4 CIOD threading architecture



Execution process modes

The Compute Nodes on Blue Gene/P are implemented as quad-cores on a single chip with 2 GB of dedicated physical memory in which applications run. There are three main modes in which a process is executed on Blue Gene/P nodes:

- ▶ Symmetrical Multiprocessing (SMP) Node Mode
- ▶ Virtual Node Mode (VN)
- ▶ Dual Node Mode (DUAL)

In this chapter, we explore these modes in detail. In the Blue Gene/L system, we have only two node modes, which are the Coprocessor Node Mode and Virtual Node Mode.

4.1 Symmetrical Multiprocessing Node Mode

In the default mode of operation of the Blue Gene/P system, the SMP Node Mode, each physical Compute Node executes a single task (Message Passing Interface (MPI) task) per node with a maximum of four threads. The Blue Gene/P system software treats those four cores in a Compute Node symmetrically. This mode is referred as *SMP mode (1X4)*, where 1 corresponds to one task and 4 corresponds to four threads.

In Figure 4-1, you see the interaction of this mode between the application space and the kernel space. The task or process can have up to four threads. Pthreads and OpenMP are supported in this mode. In this mode, each thread is pinned to a processor.

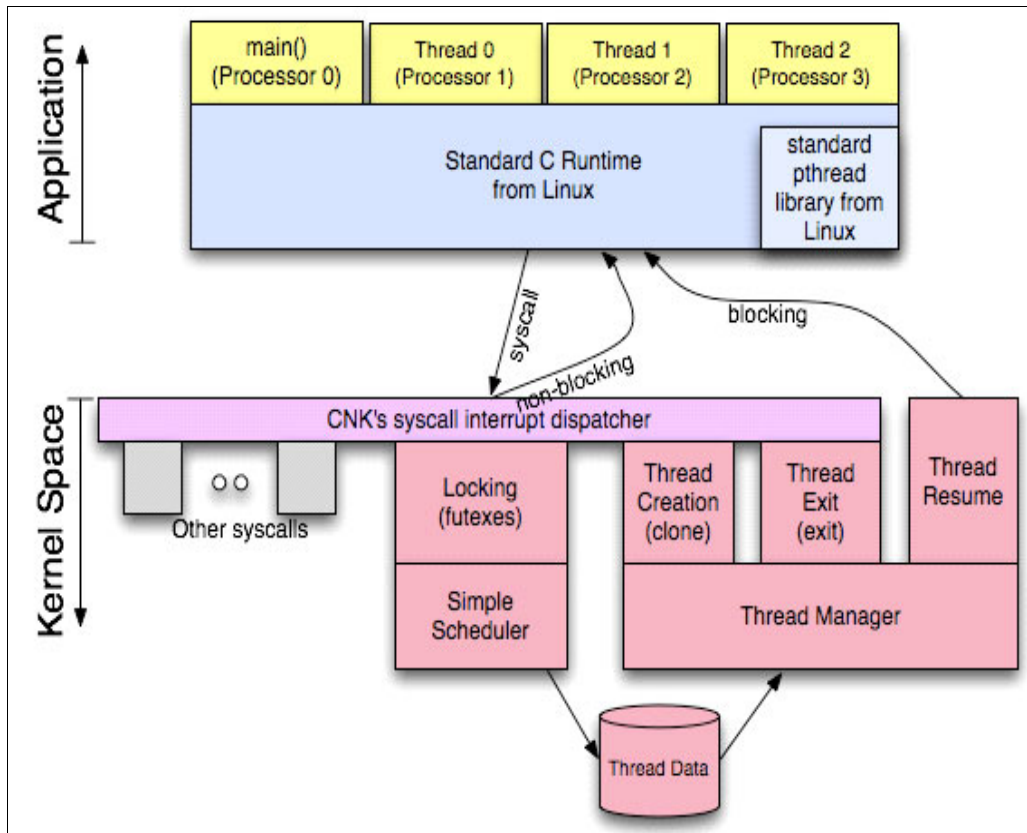


Figure 4-1 SMP Node Mode

4.2 Virtual Node Mode

The Compute Node Kernel in the Compute Nodes also supports a Virtual Node Mode of operation for the machine. In this mode, the kernel runs four separate processes on each Compute Node. Node resources (primarily the memory and the torus network) are shared by all processes. This mode is referred as *VN mode (4X1)*, where 4 corresponds to four tasks and 1 corresponds to one thread. In Figure 4-2, Virtual Node Mode is illustrated with four tasks per node and one thread per process. Shared memory is available between processes.

In Virtual Node Mode, an application can use any of the cores in a node simply by quadrupling its number of MPI tasks. The now distinct MPI tasks running on four cores of a Compute Node have to communicate to each other. This is done transparently via direct memory access (DMA) on the node. DMA puts data destined for a physically different node on the torus, while it locally copies data when it is destined for the same physical node.

In Virtual Node Mode, the four cores of a Compute Node act as different processes. Each has its own rank in the message layer. The message layer supports Virtual Node Mode by providing a correct torus to rank mapping and first in, first out (FIFO) pinning in this mode. The hardware FIFOs are shared equally between the processes. Torus coordinates are expressed by quadruplets instead of triplets. In Virtual Node Mode, communication between the four threads in a Compute Node is done via DMA local copies.

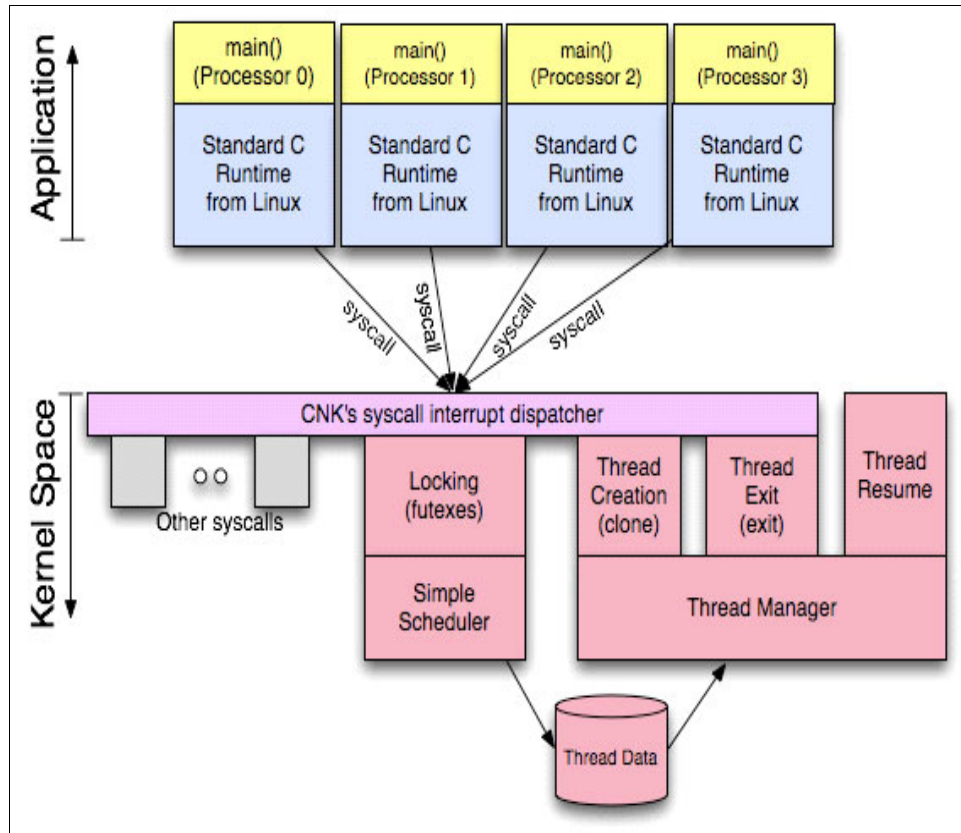


Figure 4-2 Virtual Node Mode

Each virtual node executes one compute process. Processes allocated in the same Compute Node share memory, which can be reserved at job launch. An application that wants to run with four tasks per node can dedicate a large portion for shared memory if the tasks need to share global data. This data can be read/write, and data coherency is handled in hardware.

The Blue Gene/P MPI implementation supports Virtual Node Mode operations by sharing the systems communications resources of a physical Compute Node between the four compute processes that execute on that physical node. The low-level communications library of the Blue Gene/P system, that is the message layer, virtualizes these communications resources into logical units that each process can use independently.

4.3 Dual Node Mode

A new mode in the Blue Gene/P system is the Dual Node Mode (DUAL). In this mode, each physical Compute Node executes two tasks (MPI tasks) per node with a maximum of four threads (two per task). Each task in Dual Node Mode gets half the memory and cores, so that it can run two threads per task. This mode is referred as *DUAL mode (2X2)*, where the first 2 corresponds to two tasks and the second 2 corresponds to two threads.

In Figure 4-3, you see two processes per node. Each process can have up to two threads. OpenMP and Pthreads are supported. Shared memory is available between processes. Threads are pinned to a processor.

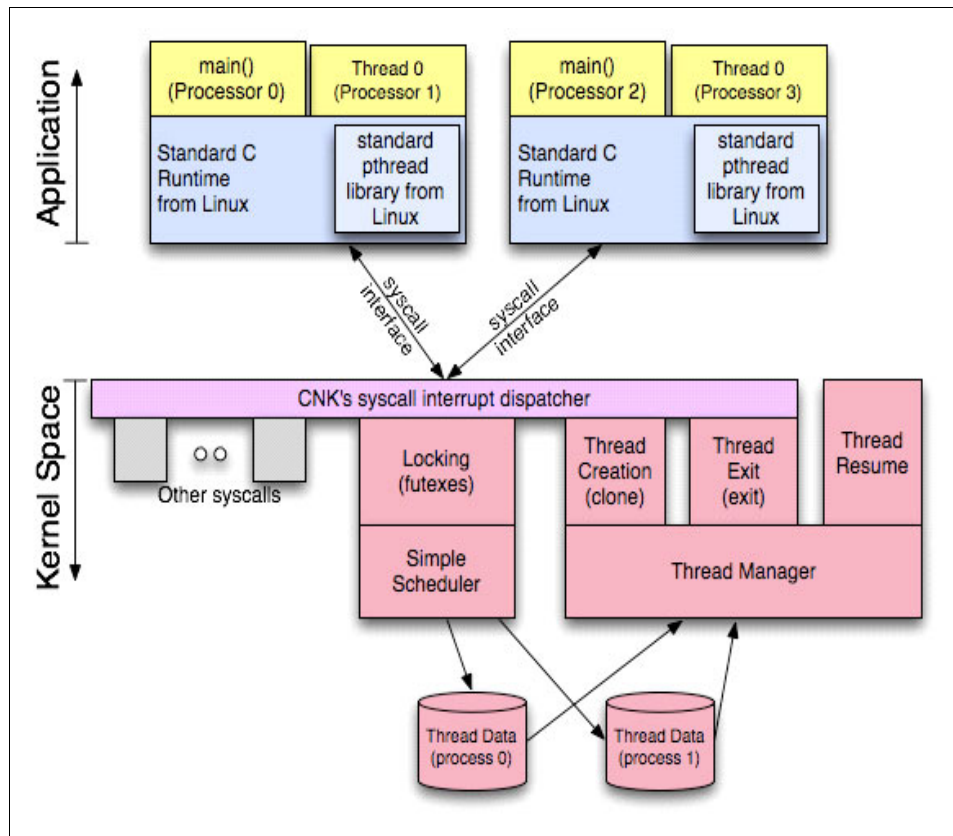


Figure 4-3 Dual Node Mode

4.4 Shared memory support

Shared memory is supported in Dual Node Mode and Virtual Node Mode process models. Shared-memory usage in the SMP Node Mode is excluded since each processor already has access to all of the node's memory.

Shared memory is allocated via standard Linux methods (**shm_open** and **mmap**). However, since the Compute Node Kernel does not have virtual pages, the physical memory that backs the shared memory must come out of a memory region that is dedicated for shared memory. This memory region has its size fixed at job launch.

BG_SHAREDMEMPOOLSIZ: The **BG_SHAREDMEMPOOLSIZ** environmental variable specifies in MB the amount of memory to be allocated. This can be done via the **mpirun -env** flag, for example, **BG_SHAREDMEMPOOLSIZ=8**. This allocates 8 MB of shared memory storage.

The user can change the amount of memory to be set aside for this memory region at job launch. Figure 4-4 illustrates shared-memory *allocation*.

```
fd = shm_open( SHM_FILE, O_RDWR, 0600 );
ftruncate( fds[0], MAX_SHARED_SIZE );
shmptr1 = mmap( NULL, MAX_SHARED_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Figure 4-4 Shared-memory allocation

Figure 4-5 illustrates shared-memory *deallocation*.

```
munmap(shmptr1, MAX_SHARED_SIZE);
close(fd);
shm_unlink(SHM_FILE);
```

Figure 4-5 shared-memory deallocation

The `shm_open()` and `shm_unlink()` routines access a pseudo-device, `/dev/shm/filename`, which the kernel interprets. Since multiple processes can access or close the shared-memory file, allocation and deallocation are tracked by a simple reference count. As such, the processes do not need to coordinate deallocation of the shared memory region.

4.5 Deciding which mode to use

The choice of the node mode largely depends on the type of application and the parallel paradigm that has been implemented for a particular application. The obvious case involves applications where a hybrid paradigm between MPI and OpenMP has been implemented. In this case, it is beneficial to use the SMP Node Mode. Single threaded applications should consider Virtual Node Mode.

I/O-intensive tasks that require a relatively large amount of data interchange between Compute Nodes benefit more by using Virtual Node Mode. Those applications that are primarily CPU bound, and do not have large working memory requirements (the application gets only half of the node memory), run more quickly in Virtual Node Mode.

4.6 Specifying a mode

The default mode for `mpirun` is the Virtual Node Mode. To specify SMP Node Mode and Dual Node Mode, you use the following commands:

```
mpirun ... -mode smp ...
mpirun ... -mode DUAL ...
```

See Chapter 13, “`mpirun`” on page 217, for more information about the `mpirun` command.



Memory

In this chapter, we provide an overview of the memory subsystem and explain how it relates to the Compute Node Kernel. This chapter includes the following topics:

- ▶ “Memory overview” on page 44
- ▶ “Memory management” on page 45
- ▶ “Memory protection” on page 47

5.1 Memory overview

Similar to the Blue Gene/L system, there is support for virtual memory on Blue Gene/P nodes. Memory is laid out as a single, flat, fixed-size virtual address space shared between the operating system kernel and the application program.

The Blue Gene/P system is a distributed-memory supercomputer, which includes an on-chip cache hierarchy, and memory is off-chip. It contains optimized on-chip symmetrical multiprocessing (SMP) support for locking and communication between the four ASIC processors.

The aggregate memory of the total machine is distributed in the style of a multi-computer, with no hardware sharing between nodes. The total physical memory amount supported is 2 GB per Compute Node.

The first level (L1) cache is contained within the PowerPC 450 core (see Figure 1-3 on page 9). The PowerPC 450 L1 cache is 64-way set associative.

The second level (L2R and L2W) caches, one dedicated per core, are 2 KB in size. They are fully associative and coherent. They act as prefetch and write-back buffers for L1 data. The L2 cache line is 128 bytes in size. Each L2 cache has one connection toward the L1 instruction cache running at full processor frequency. Each L2 cache also has two connections toward the L1 data cache, one for the writes and one for the loads, each running at full processor frequency. Read and write are 16 bytes wide.

The third level (L3) cache is 8-way set associative, 8 MB in size, with 128-byte lines. Both banks can be accessed by all processor cores. The L3 cache has three write queues and three read queues: one for each processor core and one for the 10 Gigabit network. Ethernet and direct memory access (DMA) share the L3 ports. Only one unit can use the port at a time. The Compute Nodes use DMA, and the I/O Nodes use the Ethernet. The last one is used on the I/O Node and for torus network DMA on the Compute Networks. All the write queues go across a four-line write buffer to access the eDRAM bank. Each of the two L3 banks implements thirty 128-byte-wide write combining buffers, for a total of sixty 128-byte-wide write combining buffers per chip.

Table 5-1 provides an overview of some of the features of different memory components.

Table 5-1 Memory system overview

Cache	Total per node	Size	Replacement policy	Associativity
L1 instruction	4	32 KB	Round-Robin	<ul style="list-style-type: none"> ▶ 64-way set-associative ▶ 16 sets ▶ 32-byte line size
L1 data	4	32 KB	Round-Robin	<ul style="list-style-type: none"> ▶ 64-way set-associative ▶ 16 sets ▶ 32-byte line size
L2 prefetch	4	14 x 256 bytes	Round-Robin	<ul style="list-style-type: none"> ▶ Fully associative (15-way) ▶ 128-byte line size
L3	2	2 x 4 MB	Least recently used	<ul style="list-style-type: none"> ▶ 8-way associative ▶ 2 bank interleaved ▶ 128-byte line size
Double data RAM (DDR)	2	<ul style="list-style-type: none"> ▶ Minimum 2 x 512 MB ▶ Maximum 4 GB 	N/A	<ul style="list-style-type: none"> ▶ 128-byte line size

5.2 Memory management

You must give careful consideration to managing memory on the Blue Gene/P system. This is particularly true in order to achieve optimal performance. The memory subsystem of Blue Gene/P nodes has specific characteristics and limitations that the programmer should know about.

5.2.1 L1 cache

On the Blue Gene/P system, the PowerPC 450 internal L1 cache does not have automatic prefetching. Explicit cache touch instructions are supported. Although the L1 instruction cache was designed with support for prefetches, it was disabled for efficiency reasons.

Figure 1-3 on page 9 shows the L1 caches in the PowerPC 450 architecture. The size of the L1 cache line is 32 bytes. The L1 cache has two buses toward the L2 cache: one for the stores and one for the loads. The buses are 128 bits in width and run at full processor frequency. The theoretical limit is 16 bytes/cycle. However, 4.6 bytes is achieved on L1 load misses and 5.6 bytes is achieved on all stores (write through). This value of 5.6 bytes is achieved for the stores but not for the loads. The L1 cache has only a three-line fetch buffer. Therefore, there are only three outstanding L1 cache line requests. The fourth one waits for the first one to complete before it can be sent.

The L1 hit latency is four cycles for floating-point and three cycles for integer. The L2 hit latency is at about 12 cycles for floating-point and 11 cycles for integer. The 4.6-byte throughput limitation is a result of the limited number of line fill buffers, L2 hit latency, the policy when a line fill buffer commits its data to L1, and the penalty of delayed load confirmation when running fully recoverable.

Since there are only three outstanding L1 cache line load requests at the same time, at most three cache lines can be obtained every 18 cycles. The maximum memory bandwidth is three times 32 bytes divided by 18 cycles, which yields 5.3 bytes per cycle, which written as an equation looks like this:

$$(3 \times 32 \text{ bytes}) / 18 \text{ cycles} = 5.3 \text{ bytes per cycle}$$

Important:

- ▶ Avoid instructions when prefetching data in L1 cache on the Blue Gene/P system. Using the processor, you can concurrently fill in three L1 cache lines. Therefore, it is mandatory to reduce the number of prefetching streams to three or less.

To optimize the floating point units (FPUs) and feed the floating point registers, a programmer can use the XL compiler directives or assembler instructions (dcbt) to prefetch data in the L1 data cache. The applications that are specially tuned for IBM POWER4™ or POWER5™ processors that take advantage of four or eight prefetching engines will choke the memory subsystem of the Blue Gene/P processor.
- ▶ To take advantage of the single-instruction, multiple-data (SIMD) instructions, it is essential to keep the data in the L1 cache as much as possible. Without an intensive reuse of data from the L1 cache and the registers, because of the number of registers, the memory subsystem is unable to feed the double FPU and provide two multiply-addition operations per cycle.

In the worst case, SIMD instructions can hurt the global performance of the application. For that reason, we advise that you disable the SIMD instructions in the porting phase by compiling with `-qarch=450`. Then recompile the code with `-qarch=450d` and analyze the performance impact of the SIMD instructions. Perform the analysis with a data set and a number of processors that is realistic in terms of memory usage.

Optimization tips:

- ▶ The optimization of the applications must be based on the 32 KB of the L1 cache.
- ▶ The benefits of the SIMD instructions might be cancelled out if data does not fit in the L1 cache.

5.2.2 L2 cache

The L2 cache is the hardware layer that provides the link between the embedded cores and the Blue Gene/P devices, such as the 8 MB L3-eDRAM and the 32 KB SRAM. The 2 KB L2 cache line is 128 bytes in size. Each L2 cache is connected to one processor core.

The L2 design and architecture were created to provide optimal support for the PowerPC 450 cores for scientific applications. Thus, a logic for automatic sequential stream detection and prefetching to the L2 added on the PowerPC 440 is still available on PowerPC 450. The logic is optimized to perform best on sequential streams with increasing addresses. The L2 boosts the overall performance for almost any application and does not require any special software provisions. It autonomously detects streams, issues the prefetch requests, and keeps the prefetched data coherent.

You can achieve latency and /bandwidth results close to the theoretical limits (4.6 bytes per cycle) dictated by the PowerPC 450 core by doing careful programming. The L2 accelerates memory accesses for one to seven sequential streams.

5.2.3 L3 cache

The L3 cache is 8 MB is size. The line size is 128 bytes. Both banks are directly accessed by all processor cores and the 10 Gb network, only on the I/O Node, and are used in Compute Nodes for torus DMA. There are three write queues and three read queues. The read queues directly access both banks.

Each L3 cache implements two sets of write buffer entries. Into each of the two sets, one 32-byte data line can deposit a set per cycle from any queue. In addition, one entry can be allocated for every cycle in each set. The write rate for random data is much higher in the Blue Gene/P system than in the Blue Gene/L system. The L3 cache can theoretically complete an aggregate of four write hits per chip every two cycles. However, banking conflicts reduce this number in most cases.

Optimization tip: Random access can divide the write sustained bandwidth of the L3 cache by a factor of three on Compute Nodes and more on I/O Nodes.

5.2.4 Double data RAM

The theoretical memory bandwidth on a Blue Gene/P node to transfer a 128-byte line from the external DDR to the L3 cache is 16 cycles. Nevertheless, this bandwidth can only be sustained with sequential access. Random access can reduce bandwidth significantly.

Table 5-2 illustrates latency and bandwidth estimates for the Blue Gene/P system.

Table 5-2 Latency and bandwidths estimates

	Latency ^a	Sustained bandwidth (bytes/cycle) ^{b, c}
		Sequential access
L1	3	8
L2	11	4.6
L3	50	4.6
External DDR (single processor)	104	40
External DDR (dual processor)		
External DDR (triple processor)		
External DDR (quad processor)		3.7

- a. This corresponds to integer load latency. Floating-point latency is one cycle higher.
- b. This is the maximum sustainable bandwidth for linear sequential access.
- c. Random access bandwidth is dependent on the access width and overlap access, respectively.

5.3 Memory protection

The PowerPC 450 processor has limited flexibility with regard to supported translation look-aside buffer (TLB) sizes and alignments. There is also a small number of TLB slots per processor. These limitations create situations where the dual goal of both static TLBs and memory protection is difficult to achieve with access to the entire memory space. This depends on the node's memory configuration, process model, and size of the applications sections.

On the Blue Gene/P system, the Compute Node Kernel reads only sections from the application. This prevents an application from accidentally corrupting its text (that is, its code) section due to an errant memory write. Additionally, the Compute Node Kernel prevents an application from corrupting the Compute Node Kernel text segments or any kernel data structures.

When a debugger is not attached to the running application, Compute Node Kernel can protect the active thread's stack using the data-address-compare debug registers in the PowerPC 450 processor. You can use this mechanism for stack protection without incurring TLB miss penalties. For the main thread, this protection is just above the maximum `mmap()` address. For a spawned thread, this protection is at the lower bound of the thread's stack. This protection is not available when the debugger is being used, because the debugger is managing those register settings.

The Compute Node Kernel is strict in terms of TLB setup. For example, the Compute Node Kernel does not create a 256 MB TLB that covers only 128 MB of real memory. By precisely creating the TLB map, any user-level page faults (also known as *segfaults*) are immediately caught.

In the default mode of operation of the Blue Gene/P system, which is SMP Node Mode, each physical Compute Node executes a single task (MPI task) per node with a maximum of four

threads. The Blue Gene/P system software treats those four core threads in a Compute Node symmetrically. Figure 5-1 illustrates how memory is accessed in SMP Node Mode. The user space is divided into user space “read, execute” and user-space “read/write, execute”. The latter corresponds to global variables, stack and heap. In this mode, the four threads have access to the global variables, stack, and heap.

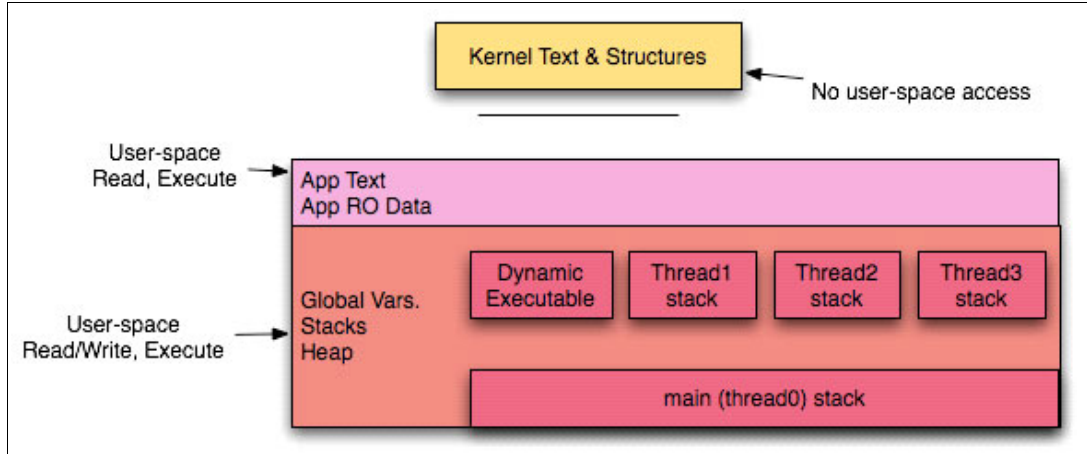


Figure 5-1 Memory access protection in SMP Node Mode

Figure 5-2 shows how memory is accessed in Virtual Node Mode. In this mode, the four core threads of a Compute Node act as different processes. The Compute Node Kernel reads only sections of an application from local memory. There is no user access between processes in the same node. User space is divided into user-space “read, execute” and user-space “read/write, execute”. The latter corresponds to global variables, stack, and heap. These two sections are designed to avoid data corruption.

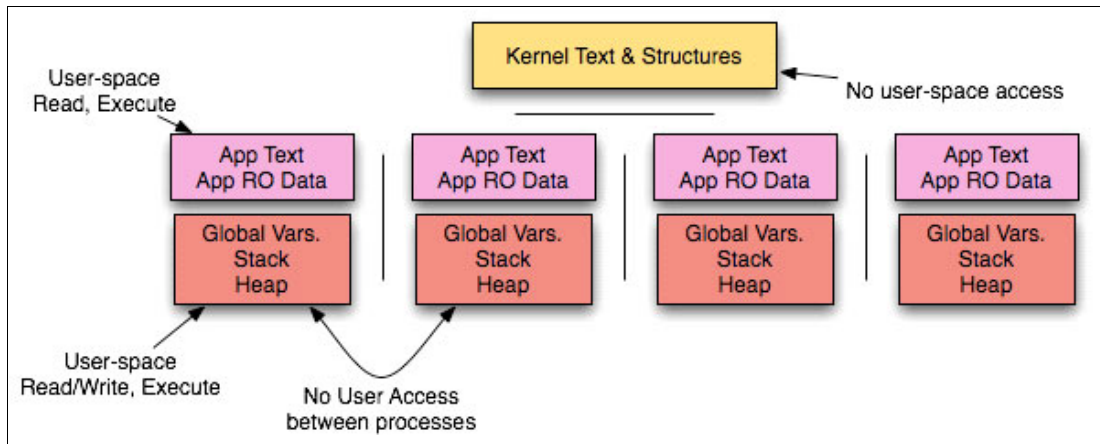


Figure 5-2 Memory access protections in Virtual Node Mode

In Virtual Node Mode, each physical Compute Node executes two tasks (MPI tasks) per node with a maximum of four threads. Each task in Dual Node Mode gets half the memory and cores so it can run two threads per task. Figure 5-3 shows that there is no user access between the two processes. Although there is a layer of shared-memory per node and the user-space “read, execute” is common to the two tasks, the two user-spaces “read/write, execute” are local to each process.

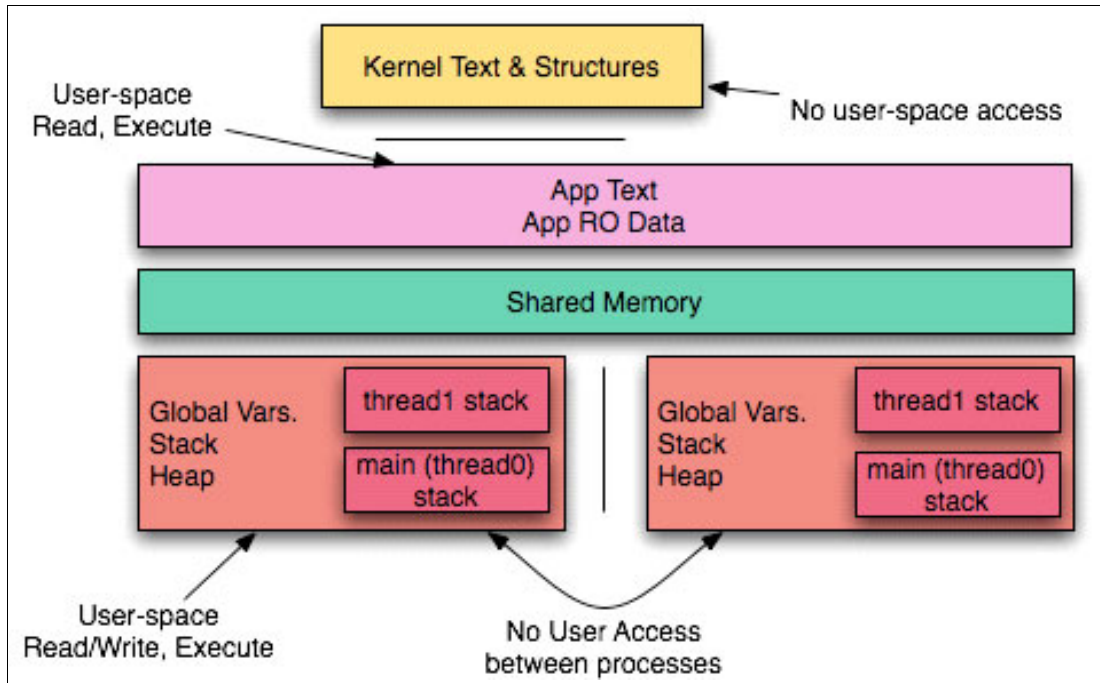


Figure 5-3 Memory access protections in Dual Node Mode



System calls

System calls provide an interface between an application and the kernel. In this chapter, we provide information about the service points through which applications running on the Compute Node request services from the Compute Node Kernel. This set of entry points into the Compute Node Kernel is referred as *system calls* (syscall). System calls on the Blue Gene/P system have substantially changed from system calls on the Blue Gene/L system. In this chapter, we describe system calls that are defined on the Blue Gene/P system.

In general, there are two types of system calls:

- ▶ Local system calls
- ▶ Function-shipped system calls

Local system calls are handled by the Compute Node Kernel only and provide Blue Gene/P-specific functionality. The following examples are of standard, local system calls:

- ▶ `brk()`
- ▶ `mmap()`
- ▶ `clone()`

Alternatively, function-shipped system calls are forwarded by the Compute Node Kernel over the collective network to the control and I/O daemon (CIOD). The CIOD then executes those system calls on the I/O Node and replies to the Compute Node Kernel with the resultant data. Examples of function-shipped system calls are functions that manipulate files and socket calls.

6.1 Introduction to the Compute Node Kernel

The role of the kernel on the Compute Node is to create an environment for the execution of a user process that is “Linux-like.” It is not a full Linux kernel implementation, but rather implements a subset of POSIX functionality.

The Compute Node Kernel is a single-process operating system. It is designed to provide the services that are needed by applications that are expected to run on the Blue Gene/P system, but not for all applications. The Compute Node Kernel is not intended to run system administration functions from the Compute Node.

To achieve the best reliability, a small and simple kernel is a design goal. This enables a simpler checkpoint function. See Chapter 10, “Checkpoint and restart support for applications” on page 151.

Compute Node application user: The Compute Node application never runs as the root user. In fact, it runs as the same user (uid) and group (gid) under which the job was submitted.

6.2 System calls

The Compute Node Kernel system calls are divided into the following categories:

- ▶ File I/O
- ▶ Directory operations
- ▶ Time
- ▶ Process information
- ▶ Signals
- ▶ Miscellaneous
- ▶ Sockets
- ▶ Compute Node Kernel

6.2.1 Return codes

As is true for return codes on a standard Linux system, a return code of zero from a system call indicates success. A value of negative one (-1) indicates a failure. In this case, `errno` contains further information about exactly what caused the problem.

6.2.2 Supported system calls

Table 6-1 lists all the function prototypes for system calls by category that are supported on the Blue Gene/P system.

Table 6-1 Supported system calls

Function prototype	Category	Header required	Description and type
<code>int access(const char *pathname, int mode);</code>	File I/O	<unistd.h>	Determines the accessibility of a file; function-shipped to CIOD; mode: R_OK, X_OK, F_OK; returns 0 if OK or -1 on error
<code>int chmod(const char *pathname, mode_t mode);</code>	File I/O	<sys/types.h> <sys>/<stat.h>	Changes the access permissions on an already open file; function-shipped to CIOD; mode: S_ISUID, S_ISGID, S_ISVTX, S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, and S_IXOTH; returns 0 if OK or -1 on error
<code>int chown(const char *pathname, uid_t owner, gid_t group);</code>	File I/O	<sys/types.h> <sys>/<stat.h>	Changes the owner and group of a file; function-shipped to CIOD
<code>int close(int filedes);</code>	File I/O	<unistd.h>	Closes a file descriptor; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int dup(int filedes);</code>	File I/O	<unistd.h>	Duplicates an open descriptor; function-shipped to CIOD; returns new file descriptor if OK or -1 on error
<code>int dup2(int filedes, int filedes2);</code>	File I/O	<unistd.h>	Duplicates an open descriptor; function-shipped to CIOD; returns new file descriptor if OK or -1 on error
<code>int fchmod(int filedes, mode_t mode);</code>	File I/O	<sys/types.h> <sys>/<stat.h>	Changes the mode of a file; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int fchown(int filedes, uid_t owner, gid_t group);</code>	File I/O	<sys/types.h> <unistd.h>	Changes the owner and group of a file; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int fcntl(int filedes, int cmd, int arg);</code>	File I/O	<sys/types.h> <unistd.h> <fcntl.h>	Performs the following operations on an open file, function-shipped to CIOD, mode: F_GETFL, F_DUPFD, F_GETLK, F_SETLK, F_SETLKW, F_GETLK64, F_SETLK64, and F_SETLKW64; what is returned depends on the command if OK or NULL on error
<code>int fstat(int filedes, struct stat *buf);</code>	File I/O	<sys/types.h> <sys>/<stat.h>	Gets the file status; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int stat64(const char *path, struct stat64 *buf);</code>	File I/O	<sys/types.h> <sys>/<stat.h>	Gets the file status
<code>int statfs(const char *path, struct statfs *buf);</code>	File I/O	<sys/vfs.h>	Gets file system statistics
<code>long fstatfs64 (unsigned int fd, size_t sz, struct statfs64 *buf);</code>	File I/O		Gets file system statistics

Function prototype	Category	Header required	Description and type
<code>int fsync(int <i>filedes</i>);</code>	File I/O	<unistd.h>	Synchronizes changes to a file; returns 0 if OK or -1 on error
<code>int ftruncate(int <i>filedes</i>, off_t <i>length</i>);</code>	File I/O	<sys/types.h> <unistd.h>	Truncates a file to a specified length; returns 0 if OK or -1 on error
<code>int ftruncate64(int <i>filedes</i>, off64_t <i>length</i>);</code>	File I/O	<unistd.h>	Truncates a file to a specified length for files larger than 2 GB; returns 0 if OK or -1 on error
<code>int lchown(const char *<i>pathname</i>, uid_t <i>owner</i>, gid_t <i>group</i>);</code>	File I/O	<sys/types.h> <unistd.h>	Changes the owner and group of a symbolic link; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int link(const char *<i>existingpath</i>, const char *<i>newpath</i>);</code>	File I/O	<unistd.h>	Links to a file; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>off_t lseek(int <i>filedes</i>, off_t <i>offset</i>, int <i>whence</i>);</code>	File I/O	<sys/types.h> <unistd.h>	Moves the read/write file offset; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int _llseek(unsigned int <i>fd</i>, unsigned long <i>offset_high</i>, unsigned long <i>offset_low</i>, loff_t *<i>result</i>, unsigned int <i>whence</i>);</code>	File I/O	<unistd.h> <sys/types.h> <linux/unistd.h> <errno.h>	Moves the read/write file offset
<code>int lstat(const char *<i>pathname</i>, struct stat *<i>buf</i>);</code>	File I/O	<sys/types.h> <sys/><stat.h>	Gets the symbolic link status; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int lstat64(const char *<i>pathname</i>, struct stat64 *<i>buf</i>);</code>	File I/O	<sys/types.h> <sys/stat.h>	Gets the symbolic link status; determines the size of a file that is larger than 2 GB
<code>int open(const char *<i>pathname</i>, int <i>oflag</i>, mode_t <i>mode</i>);</code>	File I/O	<sys/types.h> <sys/><stat.h> <fcntl.h>	Opens a file; function-shipped to CIOD; oflag: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_NOCTTY, O_NONBLOCK, O_SYNC, mode: S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, and S_IXOTH; returns file descriptor if OK or -1 on error
<code>ssize_t read(int <i>filedes</i>, void *<i>buf</i>, size_t <i>nbytes</i>);</code>	File I/O	<unistd.h>	Reads from a file; function-shipped to CIOD; returns number of bytes read if OK, 0 if end of file, or -1 on error
<code>int readlink(const char *<i>pathname</i>, char *<i>buf</i>, int <i>bufsize</i>);</code>	File I/O	<unistd.h>	Reads the contents of a symbolic link; function-shipped to CIOD; returns number of bytes read if OK or -1 on error
<code>ssize_t readv(int <i>filedes</i>, const struct iovec <i>iov</i>[], int <i>iovcnt</i>);</code>	File I/O	<sys/types.h> <sys/uo.h>	Reads a vector, function-shipped to CIOD; returns number of bytes read if OK or -1 on error
<code>int rename(const char *<i>oldname</i>, const char *<i>newname</i>);</code>	File I/O	<stdio.h>	Renames a file; function-shipped to CIOD; returns 0 if OK or -1 on error

Function prototype	Category	Header required	Description and type
<code>int stat(const char *pathname, struct stat *buf);</code>	File I/O	<sys/types.h> <sys/stat.h>	Gets the file status; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int stat64(const char *pathname, struct stat64 *buf);</code>	File I/O	<sys/types.h> <sys/stat.h>	Gets the file status
<code>int statfs (char *Path, struct statfs *StatusBuffer);</code>	File I/O	<sys/statfs.h>	Gets file system statistics
<code>long statfs64 (const char *path, size_t sz, struct statfs64 *buf);</code>	File I/O	<sys/statfs.h>	Gets file system statistics
<code>int symlink(const char *actualpath, const char *sympath);</code>	File I/O	<unistd.h>	Makes a symbolic link to a file; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int truncate(const char *pathname, off_t length);</code>	File I/O	<sys/types.h> <unistd.h>	Truncates a file to a specified length; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>truncate64</code>	File I/O		Truncates a file to a specified length
<code>mode_t umask(mode_t cmask);</code>	File I/O	<sys/types.h> <sys/stat.h>	Sets and gets the file mode creation mask; function-shipped to CIOD; returns the previous file mode creation mask
<code>int unlink(const char *pathname);</code>	File I/O	<unistd.h>	Removes a directory entry; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>int utime(const char *pathname, const struct utimbuf *times);</code>	File I/O	<sys/types.h> <utime.h>	Sets file access and modification times; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>ssize_t write(int filedes, const void *buff, size_t nbytes);</code>	File I/O	<unistd.h>	Writes to a file; function-shipped to CIOD; returns the number of bytes written if OK or -1 on error
<code>ssize_t writev(int filedes, const struct iovec iov[], int iovcntl);</code>	File I/O	<sys/types.h> <sys/uio.h>	Writes a vector; function-shipped to CIOD; returns the number of bytes written if OK or -1 on error
<code>int chdir(const char *pathname);</code>	Directory	<unistd.h>	Changes the working directory; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>char *getcwd(char *buf, size_t size);</code>	Directory	<unistd.h>	Gets the path name of the current working directory; function-shipped to CIOD; returns buf if OK or NULL on error
<code>int getdents(int filedes, char **buf, unsigned nbytes);</code>	Directory	<sys/types.h>	Gets the directory entries in a file system; function-shipped to CIOD; returns 0 if OK or -1 on error
<code>getdents64</code>	Directory	<sys/dirent.h>	Gets the directory entries in a file system
<code>int mkdir(const char *pathname, mode_t mode);</code>	Directory	<sys/types.h> <sys/stat.h>	Makes a directory; function-shipped to CIOD; mode S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, and S_IXOTH; returns 0 if OK or -1 on error

Function prototype	Category	Header required	Description and type
<code>int rmdir(const char *pathname);</code>	Directory	<unistd.h>	Removes a directory; returns 0 if OK or -1 on error
<code>int getitimer(int which, struct itimerval *value);</code>	Time	<sys/time.h>	Gets the value of the interval timer; local system call; returns 0 if OK or -1 on error
<code>int gettimeofday(struct timeval *restrict tp, void *restrict tzp);</code>	Time	<sys/time.h>	Gets the date and time; local system call; returns 0 if OK pr NULL on error
<code>int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);</code>	Time	<sys/time.h>	Sets the value of an interval timer; only the following operations are supported: <ul style="list-style-type: none"> ▶ ITIMER_PROF ▶ ITIMER_REAL Note: An application can only set one active timer at a time.
<code>time_t time(time_t *calptr);</code>	Time	<time.h>	Gets the time; local system call; returns the value of time if OK or -1 on error
<code>gid_t getgid(void);</code>	Process information	<unistd.h>	Gets the real group ID
<code>pid_t getpid(void);</code>	Process information	<unistd.h>	Gets the process ID. The value is the MPI rank of the node, meaning that 0 is a valid value.
<code>int getrlimit(int resource, struct rlimit *rlp)</code>	Process information	<sys/resource.h>	Gets information about resource limits
<code>int getrusage(int who, struct rusage *r_usage);</code>	Process information	<sys/resource.h>	Gets information about resource utilization. All time reported is attributed to the user application, so the reported system time is always zero.
<code>uid_t getuid(void);</code>	Process information	<unistd.h>	Gets the real user ID
<code>int setrlimit(int resource, const struct rlimit *rlp);</code>	Process information	<sys/resource.h>	Sets resource limits. Only RLIMIT_CORE can be set.
<code>clock_t times(struct tms *buf);</code>	Process information	<sys/times.h>	Gets the process times. All time reported is attributed to the user application, so the reported system time is always zero.
<code>int brk(void *end_data_segment);</code>	Miscellaneous	<unistd.h>	Changes the data segment size
<code>int execve(const char *filename, char *const argv[], char *const envp[]);</code>	Miscellaneous	<unistd.h>	Runs a new program in the process
<code>void exit(int status)</code>	Miscellaneous	<stdlib.h>	Terminates a process
<code>int uname(struct utsname *buf);</code>	Miscellaneous	<sys/utsname.h>	Gets the name of the current system, and other information, for example, version and release

6.2.3 Other system calls

Although there are many unsupported system calls, you must be aware of the following unsupported calls:

- ▶ The Blue Gene/P system does not support the use of the `system()` function. Therefore, for example, you cannot use something such as the `system('chmod -w file')` call. Although, `system()` is not a system call, it uses `fork()` and `exec()` via **glibc**. Both `fork()` and `exec()` are currently not implemented.
- ▶ The Blue Gene/P system does not provide the same support for `gethostname()` and `getlogin()` as Linux provides.
- ▶ Calls to `usleep()` are not supported.

See 6.6, “Unsupported system calls” on page 60, for a complete list of unsupported system calls.

6.3 System programming interfaces

Low-level access to Blue Gene/P specific interfaces, such as direct memory access (DMA), is provided by the system programming interfaces (SPIs). These interfaces provide a consistent interface for Linux and Compute Node Kernel-based applications to access the hardware.

The following Blue Gene/P-specific interfaces *are* included in the SPI:

- ▶ Collective network
- ▶ Torus network
- ▶ Direct memory access
- ▶ Global interrupts
- ▶ Performance counters
- ▶ Lockbox

The following items *are not* included in the SPI:

- ▶ L2
- ▶ Snoop
- ▶ L3
- ▶ DDR hardware initialization
- ▶ serdes
- ▶ Environmental monitor

This hardware is setup by either the bootloader or Common Node Services. The L1 interfaces, such as TLB miss handlers, are typically extremely operating system specific, and therefore an SPI is not defined. TOMAL and XEMAC are present in the Linux 10 Gb Ethernet device driver (and therefore open source), but there are no plans for an explicit SPI.

6.4 Socket support

The Compute Node Kernel provides socket support via the standard Linux `socketcall()` system call. The `socketcall()` is a kernel entry point for the socket system calls. It determines which socket function to call and points to a block that contains the actual parameters, which are passed through to the appropriate call. The Compute Node Kernel function-ships the `socketcall()` parameters to the CIOD, which then performs the requested operation. The CIOD is a user-level process that controls and services applications in the Compute Node and interacts with the Midplane Management Control System (MMCS).

This socket support allows the creation of both outbound and inbound socket connections using standard Linux application programming interfaces (APIs). For example, an outbound socket can be created by calling `socket()`, followed by `connect()`. An inbound socket can be created by calling `socket()` followed by `bind()`, `listen()`, and `accept()`.

Communication through the socket is provided via the `glibc` `send()`, `recv()`, and `select()` function calls. These function calls invoke the `socketcall()` system call with different parameters. Table 6-2 summarizes the list of Linux 2.4 socket system calls.

Table 6-2 Supported socket calls

Function prototype	Category	Header required	Description and type
<code>int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);</code>	Sockets	<sys/types.h> <sys/socket.h>	Extracts the connection request on the queue of pending connections; creates a new connected socket; returns a file descriptor if OK or -1 on error
<code>int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);</code>	Sockets	<sys/types.h> <sys/socket.h>	Assigns a local address; returns 0 if OK or -1 on error
<code>int connect(int socket, const struct sockaddr *address, socklen_t address_len);</code>	Sockets	<sys/types.h> <sys/socket.h>	Connects a socket; returns 0 if OK or -1 on error
<code>int getpeername(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);</code>	Sockets	<sys/socket.h>	Gets the name of the peer socket; returns 0 if OK or -1 on error
<code>int getsockname(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);</code>	Sockets	<sys/socket.h>	Retrieves the locally-bound socket name; stores the address in <code>sockaddr</code> ; and stores its length in the <code>address_len</code> argument; returns 0 if OK or -1 on error
<code>int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);</code>	Sockets	<sys/types.h> <sys/socket.h>	Manipulates options that are associated with a socket; returns 0 if OK or -1 on error
<code>int listen(int sockfd, int backlog);</code>	Sockets	<sys/socket.h>	Accepts connections; returns 0 if OK or -1 on error
<code>ssize_t recv(int s, void *buf, size_t len, int flags);</code>	Sockets	<sys/types.h> <sys/socket.h>	Receives a message only from a connected socket; returns 0 if OK or -1 on error
<code>ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);</code>	Sockets	<sys/types.h> <sys/socket.h>	Receives a message from a socket regardless of whether it is connected; returns 0 if OK or -1 on error
<code>ssize_t recvmsg(int s, struct msghdr *msg, int flags);</code>	Sockets	<sys/types.h> <sys/socket.h>	Receives a message from a socket regardless of whether it is connected; returns 0 if OK or -1 on error
<code>ssize_t send(int socket, const void *buffer, size_t length, int flags);</code>	Sockets	<sys/types.h> <sys/sockets.h>	Sends a message only to a connected socket; returns 0 if OK or -1 on error
<code>ssize_t sendto(int socket, const void *message, size_t length, int flags, const struct sockaddr *dest_addr, socklen_t dest_len);</code>	Sockets	<sys/types.h> <sys/socket.h>	Sends a message on a socket; returns 0 if OK or -1 on error

Function prototype	Category	Header required	Description and type
<code>ssize_t sendmsg(int s, const struct msghdr *msg, int flags);</code>	Sockets	<sys/types.h> <sys/socket.h>	Sends a message on a socket; returns 0 if OK or -1 on error
<code>int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);</code>	Sockets	<sys/types.h> <sys/socket.h>	Manipulates options that are associated with a socket; returns 0 if OK or -1 on error
<code>int shutdown(int s, int how);</code>	Sockets	<sys/socket.h>	Causes all or part of a connection on the socket to shut down; returns 0 if OK or -1 on error
<code>int socket(int domain, int type, int protocol);</code>	Sockets	<sys/types.h> <sys/socket.h>	Opens a socket; returns a file descriptor if OK or -1 on error
<code>int socketpair(int d, int type, int protocol, int sv[2]);</code>	Sockets	<sys/types.h> <sys/socket.h>	Creates an unnamed pair of connected sockets; returns 0 if OK or -1 on error

6.5 Signal support

The Compute Node Kernel provides ANSI-C signal support via the standard Linux system calls `signal()` and `kill()`. Additionally, signals can be delivered externally by using `mpirun`. Table 6-3 summarizes the supported signals.

Table 6-3 Supported signals

Function prototype	Category	Header required	Description and type
<code>int kill(pid_t pid, int sig);</code>	Signals	<sys/types.h> <signal.h>	Sends a signal. A signal can be sent only to the same process.
<code>int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);</code>	Signals	<signal.h>	Manages signals. The only flags supported are SA_RESETHAND and SA_NODEFER.
<code>typedef void (*sighandler_t)(int)</code> <code>sighandler_t signal(int signum, sighandler_t handler);</code>	Signals	<signal.h>	Manages signals
<code>typedef void (*sighandler_t)(int);</code> <code>sighandler_t signal(int signum, sighandler_t handler);</code>	Signals	<signal.h>	Returns from a signal handler

6.6 Unsupported system calls

The role of the kernel on the Compute Node is to create an environment for the execution of a user process that is “Linux-like.” It is not a full Linux kernel implementation, but rather implements a subset of the POSIX functionality. The following list indicates the system calls that are not supported:

- ▶ acct
- ▶ adjtimex
- ▶ afs_syscall
- ▶ bdflush
- ▶ break
- ▶ capget
- ▶ capset
- ▶ chroot
- ▶ clock_getres
- ▶ clock_gettime
- ▶ clock_nanosleep
- ▶ clock_settime
- ▶ create_module
- ▶ delete_module
- ▶ epoll_create
- ▶ epoll_ctl
- ▶ epoll_wait
- ▶ execve
- ▶ fadvise64
- ▶ fadvise64_64
- ▶ fchdir
- ▶ fdatsync
- ▶ fgetxattr
- ▶ flistxattr
- ▶ flock
- ▶ fork
- ▶ fremovexattr
- ▶ fsetxattr
- ▶ ftime
- ▶ get_kernel_syms
- ▶ iopl
- ▶ ipc
- ▶ kexec_load
- ▶ lgetxattr
- ▶ listxattr
- ▶ llistxattr
- ▶ lock
- ▶ lookup_dcookie
- ▶ lremovexattr
- ▶ lsetxattr
- ▶ mincore
- ▶ mknod
- ▶ modify_lft
- ▶ mount
- ▶ mpxmq_getsetattr
- ▶ mq_notify
- ▶ mq_open
- ▶ mq_timedreceive
- ▶ mq_timedsend
- ▶ mq_unlink
- ▶ multiplexer
- ▶ nfsservctl
- ▶ nice
- ▶ oldfstat
- ▶ oldlstat
- ▶ oldolduname
- ▶ olduname
- ▶ oldstat
- ▶ pciconfig_iobase
- ▶ pciconfig_read
- ▶ removexattr
- ▶ rtas
- ▶ rts_device_map
- ▶ rts_dma
- ▶ sched_get_priority_max
- ▶ sched_get_priority_min
- ▶ sched_getaffinity
- ▶ sched_getparam
- ▶ sched_getscheduler
- ▶ sched_rr_get_interval
- ▶ sched_setaffinity
- ▶ sched_setparam
- ▶ sched_setscheduler
- ▶ sched_yield
- ▶ select
- ▶ sendfile
- ▶ sendfile64
- ▶ setdomainname
- ▶ setgroups
- ▶ sethostname
- ▶ setpriority
- ▶ settimeofday
- ▶ setxattr
- ▶ stime
- ▶ stty
- ▶ swapcontext
- ▶ swapoff
- ▶ swapon
- ▶ sync
- ▶ sys_debug_setcontext

- ▶ getgroups
- ▶ getpgrp
- ▶ getpmsg
- ▶ getppid
- ▶ getpriority
- ▶ gettid
- ▶ getxattr
- ▶ gtty
- ▶ idle
- ▶ init_module
- ▶ io_cancel
- ▶ io_destroy
- ▶ io_getevents
- ▶ io_setup
- ▶ io_submit
- ▶ ioperm
- ▶ pciconfig_write
- ▶ personality
- ▶ pipe
- ▶ pivot_root
- ▶ pread64
- ▶ prof
- ▶ profil
- ▶ ptrace
- ▶ putpmsg
- ▶ pwrite64
- ▶ query_module
- ▶ quotactl
- ▶ readahead
- ▶ readdir
- ▶ reboot
- ▶ remap_file_pages
- ▶ sysfs
- ▶ syslog
- ▶ timer_create
- ▶ timer_delete
- ▶ timer_getoverrun
- ▶ timer_gettime
- ▶ timer_settime
- ▶ tuxcall
- ▶ umount
- ▶ umount2
- ▶ uselib
- ▶ ustat
- ▶ utimes
- ▶ vfork
- ▶ vhangup
- ▶ vm86

You can find additional information about these system calls on the `syscalls(2)` - Linux man page on the Web at:

<http://linux.die.net/man/2/syscalls>



Part 3

Applications environment

In this part, we provide an overview of some of the software that forms part of the applications environment. Throughout this book, we consider the applications environment as the collection of programs that are required to develop applications.

This part includes the following chapters:

- ▶ Chapter 7, “Parallel paradigms” on page 65
- ▶ Chapter 8, “Developing applications with IBM XL compilers” on page 91
- ▶ Chapter 9, “Running and debugging applications” on page 129
- ▶ Chapter 10, “Checkpoint and restart support for applications” on page 151
- ▶ Chapter 11, “Control system (Bridge) APIs” on page 159
- ▶ Chapter 12, “Real-time Notification APIs” on page 197
- ▶ Chapter 13, “mpirun” on page 217
- ▶ Chapter 14, “Dynamic Partition Allocator APIs” on page 237



Parallel paradigms

In this chapter, we discuss the parallel paradigms that are offered on the Blue Gene/P system. One such diagram is the Message Passing Interface (MPI),²² for a distributed-memory architecture, and OpenMP,²³ for shared-memory architectures.

In this chapter, we address the following topics:

- ▶ “Programming model” on page 66
- ▶ “Blue Gene/P MPI implementation” on page 66
- ▶ “MPI communications” on page 74
- ▶ “MPI functions” on page 76
- ▶ “Compiling MPI programs on Blue Gene/P” on page 77
- ▶ “MPI communications performance” on page 79
- ▶ “OpenMP” on page 83

7.1 Programming model

The Blue Gene/P system has a distributed memory system and uses explicit message passing to communicate between tasks that are running on different nodes. It also has shared memory on each node; OpenMP and thread parallelism are supported as well.

MPI is the supported message passing standard. It is the industry standard for message passing. For further information about MPI, refer to the Message Passing Interface Forum site on the Web at the following address:

<http://www.mpi-forum.org/>

If your code uses other message passing libraries, you must either change the message passing calls to MPI or use an intermediate layer that maps your library's calls to MPI.

7.2 Blue Gene/P MPI implementation

The current MPI implementation on the Blue Gene/P system supports the MPI-1.2 and MPI-2 standards of MPI Version 1.2. They are both extensions to the MPI-1.1 standard. The only exception is process management. For more information about process management, see the paper *Dynamic Process Management in an MPI Setting* by William Gropp and Ewing Lusk, on the Web at:

<http://www.cs.uiuc.edu/homes/wgropp/bib/papers/1995/sanantonio.pdf>

When starting applications on the Blue Gene/P system, you must consider the following additional details:

- ▶ The microkernel that is running on the Compute Nodes does not provide any mechanism for a command interpreter or shell. Only the executables can be started. Shell scripts are not supported. Therefore, if your application consists of a number of shell scripts that control its workflow, the workflow must be adapted. If you start your application with the `mpirun` command, you cannot start the main shell script with this command. Instead, you must run the scripts on the front-end node and only call `mpirun` at the innermost shell script level where the main application binary is called.
- ▶ Launching an application on the Blue Gene/P system is done in the single program, multiple data (SPMD) model. Within one run, you cannot load one executable onto a subset of the Compute Nodes and a different executable onto another subset of the Compute Nodes. If you need multiple program, multiple data (MPMD) functionality, you can build this functionality into your code by using a clause similar to the one shown in Example 7-1. This clause shifts the multiple program feature from the main program level into the subprogram level.

Example 7-1 Multiple program feature

```
IF (myrank==something) THEN
CALL some_subprogram(some_args)
ELSE
CALL another_subprogram(some_other_args)
END
```

Example 7-1 illustrates a way that you can load a single executable onto all nodes. The executable then branches into different subprograms depending on the local MPI rank.

The MPI implementation on the Blue Gene/P system is derived from the MPICH2 implementation of the Mathematics and Computer Science Division (MCS) at Argonne National Laboratory. For additional information, refer to the MPICH2 Web site at:

<http://www-unix.mcs.anl.gov/mpi/mpich/>

To support the Blue Gene/P hardware, the following additions and modifications have been made to the MPICH2 software architecture:

- ▶ A BGP driver has been added underneath the MPICH2 abstract device interface (ADI).
- ▶ Three types of glue code are provided for some MPI collectives. There is one for each of the three networks that can be used for MPI communication on the Blue Gene/P system:
 - Torus for the torus network
 - Tree for the collective network
 - Global Interrupt (GI) for the barrier network
- ▶ Optimized versions of the Cartesian functions exist (MPI_Dims_create, MPI_Cart_create, MPI_Cart_map).
- ▶ MPIX functions create hardware-specific MPI extensions.
- ▶ Other parallel paradigms have been included as shown in Figure 7-1.

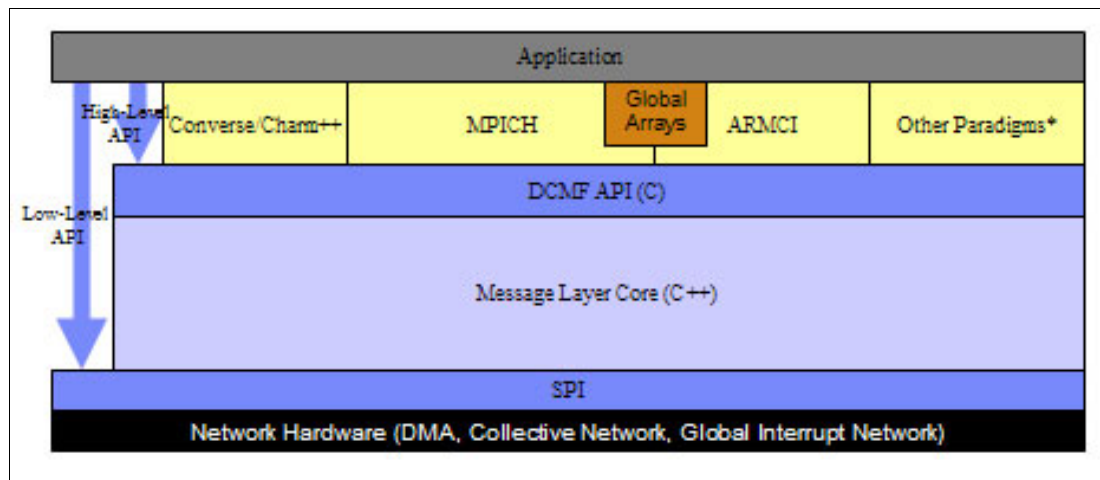


Figure 7-1 Software stack

From the application programmer's view, the most important aspect of these changes is the fact that the collective operations can use different networks under different circumstances. In 7.2.1, "High performance network for efficient parallel execution" on page 67, we briefly summarize the different networks on the Blue Gene/P system and network routing.

In 7.2.2, "Forcing MPI to allocate too much memory" on page 69, through 7.2.7, "Interlocking collectives with point-to-point calls" on page 73, we discuss several sample MPI codes to explain some of the implementation-dependent behaviors of the MPI library.

7.2.1 High performance network for efficient parallel execution

The Blue Gene/P system does not have a single type of network that is capable of transporting all protocols that are needed in such an environment. Therefore, the Blue Gene/P system has implemented separate networks for different types of communications.

Collective network

The three-dimensional (3D) torus is an efficient network for communicating with neighbors. However, such calls as all-to-one, one-to-all, and all-to-all more efficiently use the collective network. The collective network connects all the Compute Nodes in the shape of a tree. Any node can be the tree root. The MPI implementation uses the collective network, which is more efficient than the torus network for collective communication.

Point-to-point network

All MPI point-to-point communications are carried out via the torus network. The route from a sender to a receiver on a torus network has two possible paths:

- ▶ Deterministic routing

Packets from a sender to a receiver go along the same path. One advantage of this path is that the packet order is always maintained without additional logic. However, this technique also creates network hot spots if several point-to-point communications occur at the same time and their deterministic routes cross on some node.

- ▶ Adaptive routing

Different packets from the same sender to the same receiver can travel along different paths. The exact route is determined at run time depending on the current load. This technique generates a more balanced network load, but introduces a latency penalty.

Selecting deterministic or adaptive routing depends on the protocol that is used for the communication. The Blue Gene/P MPI implementation supports three different protocols:

- ▶ MPI short protocol

The MPI short protocol is used for short messages (less than 224 bytes), which consist of a single packet. These messages are always deterministically routed. The latency for eager messages is around 3.3 μ s.

- ▶ MPI eager protocol

The MPI eager protocol is used for medium-sized messages. It sends a message to the receiver without negotiating with the receiving side that the other end is ready to receive the message. This protocol also uses deterministic routes for its packets.

- ▶ MPI rendezvous protocol

Large (greater than 1200 bytes) messages are sent using the MPI rendezvous protocol. In this case, an initial connection between the two partners is established. Only after that will the receiver use direct memory access (DMA) to obtain the data from the sender. This protocol uses adaptive routing and is optimized for maximum bandwidth. Naturally, the initial rendezvous handshake increases the latency.

The Blue Gene/P MPI library supports a `DCMF_EAGER` variable (can be set via `mpirun`) to set the message size (in bytes) above which the rendezvous protocol should be used. Consider the following guidelines:

- ▶ Decrease the rendezvous threshold if any of the following situations are true:

- Many short messages are overloading the network.
- Eager messages are creating artificial hot spots.
- The program is not latency-sensitive.

- ▶ Increase the rendezvous threshold if any of the following situations are true:

- Most communication is a nearest-neighbor or at least close in Manhattan distance, where this distance is the shortest number of hops between a pair of nodes.
- You mainly use relatively long messages.
- You need better latency on medium-sized messages.

The following guidelines are also necessary for proper MPI usage:

- ▶ Overlap communication and computation using `MPI_Irecv` and `MPI_Isend`, which allow DMA to work in the background.

DMA and the tree and GI networks: The tree and GI networks do not use DMA. In this case, operations cannot be completed in the background.

- ▶ Avoid load imbalance.

This is important for all parallel systems. However, when scaling to the high numbers of tasks that are possible on the Blue Gene/P system, it is important to pay close attention to load balancing.

- ▶ Avoid buffered and synchronous sends; post receives in advance.

The MPI standard defines several specialized communication modes in addition to the standard send function, `MPI_Send()`. Avoid the buffered send function, `MPI_Bsend()`. If you use this function, forcing the MPI library to perform additional memory copies slows down the application, and you might run short of memory so additional buffering might not be possible at all. Using the synchronous send function `MPI_Ssend()` is discouraged because it is a non-local operation that incurs an increased latency compared to the standard send.

- ▶ Avoid vector data and non-contiguous data types.

While the MPI-derived data types can elegantly describe the layout of complex data structures, using these data types is generally detrimental to performance. Many MPI implementations pack (that is, memory-copy) such data objects before sending them. This packing of data objects is contrary to the original purpose of MPI-derived data types, namely to avoid such memory copies. In addition, the Blue Gene/P MPI implementation uses the chips' special quad-word load and quad-word store instructions, which require appropriately aligned and continuous data.

7.2.2 Forcing MPI to allocate too much memory

Forcing MPI to allocate too much memory is relatively easy to do with basic code. For example, the snippets of legal MPI code shown in Example 7-2 and Example 7-3 run the risk of forcing the MPI support to allocate too much memory, resulting in failure, because it forces excessive buffering of messages.

Example 7-2 CPU1 MPI code that can cause excessive memory allocation

```
MPI_Isend(cpu2, tag1);
MPI_Isend(cpu2, tag2);
...
MPI_Isend(cpu2, tagn);
```

Example 7-3 CPU2 MPI code that can cause excessive memory allocation

```
MPI_Recv(cpu1, tagn);
MPI_Recv(cpu1, tagn-1);
...
MPI_Recv(cpu1, tag1);
```

Example 7-3 illustrates a section of code that was particularly important on the Blue Gene/L system. However, on the Blue Gene/P system, we recommend this as good programming with practice.

Keep in mind the following points:

- ▶ The Blue Gene/P MPI rendezvous protocol does not allocate an unexpected buffer for the receive. This proper buffer allocation prevents most problems by drastically reducing the memory footprint of unexpected messages.
- ▶ The message queue is searched linearly to meet MPI matching requirements. If several messages are on the queue, the search can take longer.

You can accomplish the same goal and avoid memory allocation issues by recoding as shown in Example 7-4 and Example 7-5.

Example 7-4 CPU1 MPI code that can avoid excessive memory allocation

```
MPI_Isend(cpu2, tag1);  
MPI_Isend(cpu2, tag2);  
...  
MPI_Isend(cpu2, tagn);
```

Example 7-5 CPU2 MPI code that can avoid excessive memory allocation

```
MPI_Recv(cpu1, tag1);  
MPI_Recv(cpu1, tag2);  
...  
MPI_Recv(cpu1, tagn);
```

7.2.3 Not waiting for MPI_Test

According to the MPI standard, an application must either wait or continue testing until MPI_Test returns *true*. Not doing so causes small memory leaks, which can accumulate over time and cause a memory overrun. Example 7-6 shows the code and the problem.

Example 7-6 Potential memory overrun caused by not waiting for MPI_Test

```
req = MPI_Isend( ... );  
MPI_Test (req);  
... do something else; forget about req ...
```

Remember to use MPI_Wait or loop until MPI_Test returns *true*.

7.2.4 Flooding of messages

The code shown in Example 7-7, while legal, floods the network with messages. It can cause CPU 0 to run out of memory. Even though it can work, it is not scalable.

Example 7-7 Flood of messages resulting in a possible memory overrun

```
CPU 1 to n-1 code:  
MPI_Send(cpu0);  
  
CPU 0 code:  
for (i=1; i<n; i++)  
    MPI_Recv(cpu[i]);
```

7.2.5 Deadlock the system

The code shown in Example 7-8 is illegal according to the MPI standard. Each side does a blocking send to its communication partner before posting a receive for the message coming from the other partner.

Example 7-8 Deadlock code

```
TASK1 code:
MPI_Send(task2, tag1);
MPI_Recv(task2, tag2);
TASK2 code:
MPI_Send(task1, tag2);
MPI_Recv(task1, tag1);
```

In general, this code has a high probability of deadlocking the system. Obviously, you should not program this way. Make sure that your code conforms to the MPI specification. You can achieve this by either changing the order of sends and receives or by using non-blocking communication calls.

Important: The code presented in Example 7-8 will create a deadlock if the rendezvous protocol is used. The rendezvous protocol does not allocate buffers for these messages, which is not the case with the *eager protocol*.

While you should not rely on the runtime system to correctly handle non-conforming MPI code, it is easier to debug such situations when you receive a runtime error message than to try and detect a deadlock and trace it back to its root cause.

7.2.6 Violating MPI buffer ownership rules

A number of problems can arise when the send/receive buffers that participate in asynchronous message passing calls are accessed before it is legal to do so. All of the following examples are *illegal*, and therefore, you must avoid them.

The most obvious case is when you write to a send buffer before the MPI_Wait for that request has completed as shown in Example 7-9.

Example 7-9 Write to a send buffer before MPI_Wait has completed

```
req = MPI_Isend(buffer,&req);
buffer[0] = something;
MPI_Wait(req);
```

The code in Example 7-9 results in a race condition on any message passing machine. Depending on the runtime factors that are outside the application's control, sometimes the old buffer[0] is sent and sometimes the new value is sent.

A more subtle case is a read from the send buffer before the MPI_Wait for that request completes as shown in Example 7-10.

Example 7-10 Read from the send buffer before the MPI_Wait completes

```
req = MPI_Isend(buffer,&req);
z = buffer[0];
MPI_Wait(req);
```

Although not as obvious as the write case, the code in Example 7-10 is also prohibited by the MPI standard. The MPI runtime system has full control over the buffer until the MPI_Wait for the request completes. The application is not allowed to read it. In the current Blue Gene/P implementation, such code works as expected, but there is no guarantee that future versions of the MPI library will behave the same way.

In the last example in this thread, a receive buffer is read before the MPI_Wait for the asynchronous receive request has completed. See Example 7-11.

Example 7-11 Receive buffer before MPI_Wait has completed

```
req = MPI_Irecv(buffer);  
z = buffer[0];  
MPI_Wait (req);
```

The code shown in Example 7-11 is also illegal. The contents of the receive buffer are not guaranteed until after MPI_Wait is called.

Buffer alignment sensitivity

It is important to note that the MPI implementation on the Blue Gene/P system is sensitive to the alignment of the buffers that are being sent or received. Aligning buffers on 32-byte boundaries can improve performance. If the buffers are at least 16-bytes aligned, the messaging software can use internal math routines that are optimized for the double hummer architecture. Additionally, the L1 cache and DMA are optimized on 32-byte boundaries.

For buffers that are dynamically allocated (via malloc()), the following techniques can be used:

- ▶ Instead of using malloc(), use the following statement and specify 32 for the alignment parameter:

```
int posix_memalign(void **memptr, size_t alignment, size_t size)
```

This statement returns a 32-byte aligned pointer to the allocated memory. You can use free() to free the memory.

- ▶ Use malloc(), but request 32 bytes of more storage than required. Then round the returned address up to a 32-byte boundary as shown in Example 7-12.

Example 7-12 Request 32 bytes more storage than required

```
buffer_ptr_original = malloc(size + 32);  
buffer_ptr          = (char*)( ( (unsigned)buffer_ptr + 32 ) & 0xFFFFFE0 );  
.  
.  
.  
/* Use buffer_ptr on MPI operations */  
.  
.  
.  
free(buffer_ptr_original);
```

For buffers that are declared in static (global) storage, use `__attribute__((aligned(32)))` on the declaration as shown in Example 7-13.

Example 7-13 Buffers that are declared in static (global) storage

```
struct DataInfo
{
  unsigned int iarray[256];
  unsigned int count;
} data_info __attribute__((aligned ( 32)));
or
unsigned int data __attribute__((aligned ( 32)));
or
char data_array[512] __attribute__((aligned(32)));
```

For buffers that are declared in automatic (stack) storage, only up to a 16-byte alignment is possible. Therefore, use dynamically allocated aligned static (global) storage instead.

7.2.7 Interlocking collectives with point-to-point calls

Consider the code shown in Example 7-14, in which task 1 issues a barrier synchronization before the preceding asynchronous send is known to have completed.

Example 7-14 Barrier synchronization issued before the preceding asynchronous send completed

```
TASK1 code:
req = MPI_Isend(task2, &req);
MPI_Barrier();
MPI_Wait(req);
TASK2 code:
MPI_Recv(task1);
MPI_Barrier();
```

The receiver does not join the barrier before its (blocking) receive has completed. Therefore, this code will potentially deadlock if task 1 enters the barrier before the asynchronous send completes and if task 1 relies on the `MPI_Wait` to complete the send operation.

On the Blue Gene/P system, this kind of code works because the asynchronous send is handled by the torus network, where the barrier is handled by the barrier (global interrupt) network. Even though task 1 might have already entered the barrier, it is still possible to make progress on the point-to-point communications on the torus network, and the blocking receive on task 2 will eventually complete.

To avoid unexpected behavior, do not interlock collectives with point-to-point communications. For all collectives, except `MPI_Barrier`, the MPI standard clearly states that programmers should not rely on collective communications to synchronize the tasks, and at the same time, should structure their program in a way that allows for such synchronization to take place without causing a deadlock in the point-to-point communications. In general, we recommend that you do not mix collectives, which is not good programming practice.

7.3 MPI communications

In this section, we discuss Blue Gene/P-specific features that are related to MPI communications via the torus network.

7.3.1 Blue Gene/P MPI extensions

Three new application programming interfaces (APIs) make it easier to map nodes to specific hardware or processor set (pset) configurations. Application developers can use these functions, as explained in the following list, by including the mpix.h file:

- ▶ `int MPIX_Cart_comm_create (MPI_Comm *cart_comm);`

This function creates a four-dimensional (4D) Cartesian communicator that mimics the exact hardware on which it is run. The X, Y, and Z dimensions match those of the partition hardware, while the T dimension has cardinality 1 in symmetrical multiprocessing (SMP) Node Mode, cardinality 2 in Dual Node Mode, and cardinality 4 in Virtual Node Mode. The communicator wrap-around links match the true mesh or torus nature of the partition. In addition, the coordinates of a node in the communicator match exactly its coordinates in the partition.

It is important to understand that this is a collective operation and it must be run on all nodes. The function might be unable to complete successfully for several different reasons, mostly likely when it is run on fewer nodes than the entire partition. It is important to ensure that the return code is `MPI_SUCCESS` before continuing to use the returned communicator.

► `int MPIX_Pset_same_comm_create (MPI_Comm *pset_comm);`

This function is a collective operation that creates a set of communicators (each node seeing only one), where all nodes in a given communicator are part of the same pset (all share the same input/output (I/O) node). See Figure 7-2.

The most common use for this function is to coordinate access to the outside world to maximize the number of I/O Nodes. For example, node 0 in each of the communicators can be arbitrarily used as the “master node” for the communicator, collecting information from the other nodes for writing to disk.

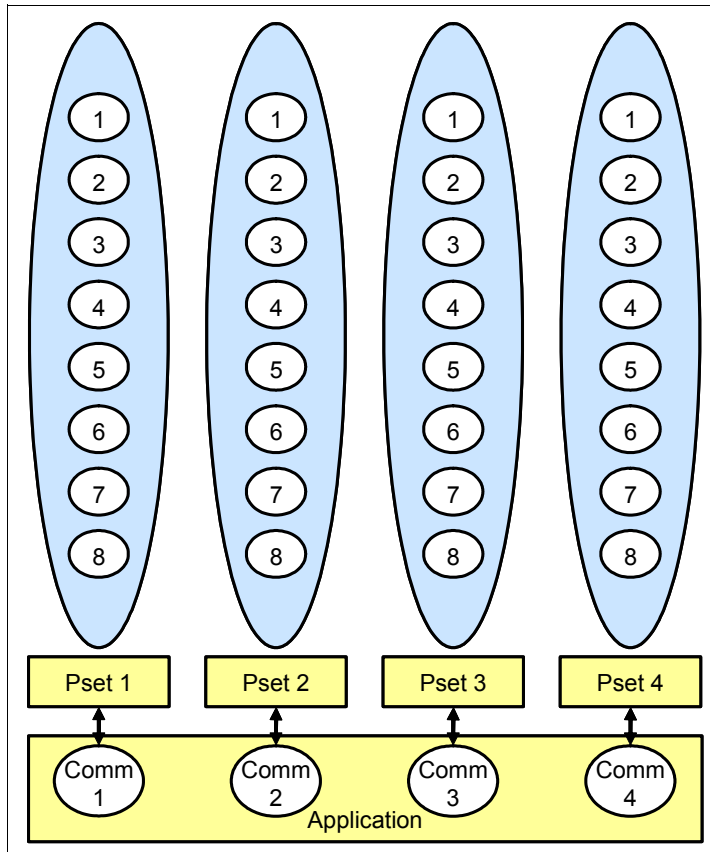


Figure 7-2 `MPIX_Pset_same_comm_create()` creating communicators

► `int MPIX_Pset_diff_comm_create (MPI_Comm *pset_comm);`

This function is a collective operation that creates a set of communicators (each node seeing only one), where no two nodes in a given communicator are part of the same pset (all have different I/O Nodes). See Figure 7-3. The most common use for this function is to coordinate access to the outside world to maximize the number of I/O Nodes. For example, an application that has an extremely high bandwidth per node requirement can run both `MPIX_Pset_same_comm_create()` and `MPIX_Pset_diff_comm_create()`.

Nodes without rank 0 in `MPIX_Pset_same_comm_create()` can sleep, leaving those with rank 0 independent and parallel access to the functional Ethernet. Those nodes all belong to the same communicator from `MPIX_Pset_diff_comm_create()`, allowing them to use that communicator instead of `MPI_COMM_WORLD` for group communication or coordination.

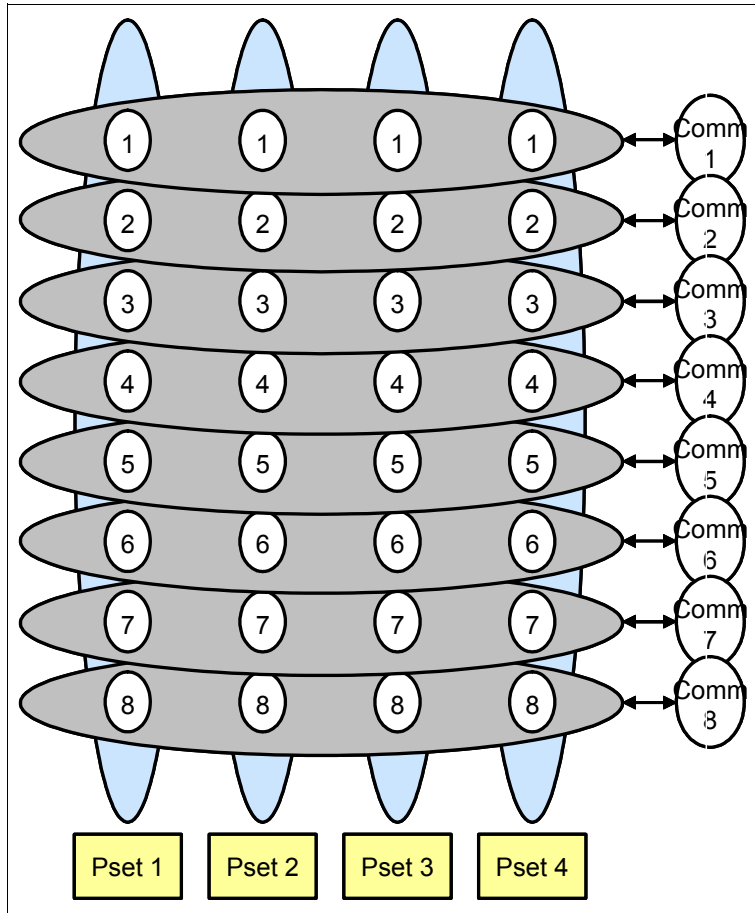


Figure 7-3 `PMI_Pset_diff_comm_create()` creating communicators

7.4 MPI functions

MPI functions have been extensively documented in the literature. In this section, we provide several useful references that provide a comprehensive description of the MPI functions.

Appendix A in *Parallel Programming in C with MPI and OpenMP*, by Michael J. Quinn,²⁴ describes all the MPI functions as defined in the MPI-1 standard. This reference also provides additional information and recommendations when to use each function.

In addition, you can find information about the MPI standard on the Message Passing Interface (MPI) standard Web site at:

<http://www-unix.mcs.anl.gov/mpi/>

A comprehensive list of the MPI functions is available on the MPI Routines page at:

<http://www-unix.mcs.anl.gov/mpi/www/www3/>

MPI Routines page includes MPI calls for C and Fortran. For more information, refer to the following books about MPI and MPI-2:

- ▶ *MPI: The Complete Reference, 2nd Edition, Volume 1*, by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra²⁵
- ▶ *MPI: The Complete Reference, Volume 2: The MPI-2 Extensions*, by William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir²⁶

For general information about MPICH2, refer to the MPICH2 Web page at:

<http://www-unix.mcs.anl.gov/mpi/mpich/>

Since teaching about MPI is beyond the scope of this book, refer to the following Web page for tutorials and extensive information about MPI:

<http://www-unix.mcs.anl.gov/mpi/learning.html>

7.5 Compiling MPI programs on Blue Gene/P

The XL C/C++ and Fortran95 family of compilers provide several different commands that you can use to invoke the compiler with the *bgxl* prefix, for example **bgx1c**. These are the most commonly used commands to invoke the different compilers. However, when using the standard commands, you must explicitly invoke all the libraries that are required.

Alternatively, simple scripts are available to compile or link MPI programs. These scripts include the default flags and libraries and can handle alternative compilers and the associated flags and libraries.

The following flags and environmental variables are provided with the scripts:

includedir, libdir	Directories that contain an installed mpich2
prefix, execprefix	Often used to define includedir and libdir
CC	C compiler
CXX	C++ compiler
FC	Fortran 77 compiler
MPI_CFLAGS	Any special flags needed to compile
MPI_LDFLAGS	Any special flags needed to link
MPILIBNAME	The name of the MPI library
MPI_OTHERLIBS	Other libraries that are needed in order to link

The compiler and the scripts also provide corresponding “_r” versions of most invocation commands to instruct the compiler to link and bind object files to thread-safe components and libraries, and produce threadsafe object code for compiler-created data and procedures. The following scripts are provided in the `/bgsys/drivers/ppcfloor/comm/bin` directory:

mpicc	GNU C compiler with MPI libraries and default compiler flags
mpich2version	Script to provide MPICH2 version information
mpicxx	GNU C++ compiler with MPI libraries and default compiler flags
mpif77	GNU Fortran 77 (gfortran) compiler with MPI libraries and default compiler flags

mpixlc	IBM XL C compiler with MPI libraries and no default compiler flags
mpixlc_r	Thread-safe version of mpixlc
mpixlcxx	IBM XL C++ compiler with MPI libraries and no default compiler flags
mpixlcxx_r	Thread-safe version of mpixlcxx
mpixlf2003	IBM XL Fortran 2003 compiler with MPI libraries and no default compiler flags
mpixlf2003_r	Thread-safe version of mpixlf2003
mpixlf77	IBM XL Fortran 77 compiler with MPI libraries and no default compiler flags
mpixlf77_r	Thread-safe version of mpixlf77
mpixlf90	IBM XL Fortran 90 compiler with MPI libraries and no default compiler flags
mpixlf90_r	Thread-safe version of mpixlf90
mpixlf95	IBM XL Fortran 95 compiler with MPI libraries and no default compiler flags
mpixlf95_r	Thread-safe version of mpixlf95

Note: When you invoke the previous scripts, if you do not specify `-O` (specify whether to optimize code during compilation), the default is set to `-O0`.

Example 7-15 shows a makefile that does not use the scripts and requires the programmer to identify all the libraries and include files.

Example 7-15 Makefile with explicit reference to libraries and include files

```

BGP_FLOOR = /bgsys/drivers/ppcfloor
BGP_IDIRS = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS  = -L$(BGP_FLOOR)/comm/lib -lmpich.cnk -L$(BGP_FLOOR)/comm/lib
-ldcmfcoll.cnk -ldcmf.cnk -lpthread -lrt -L$(BGP_FLOOR)/runtime/SPI -lSPI.cna

XL        = /opt/ibmcmp/xlf/bg/11.1/bin/bgxlf

EXE       = fhello
OBJ       = hello.o
SRC       = hello.f
FLAGS    = -O3 -qarch=450 -qtune=450 -I$(BGP_FLOOR)/comm/include
FLD      = -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
#       ${XL} $(FLAGS) $(BGP_LDIRS) -o $(EXE) $(OBJ) $(BGP_LIBS)
#       ${XL} $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC)
       ${XL} $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
       rm hello.o fhello

```

Alternatively, Example 7-16 uses MPI scripts, which create a simpler makefile. The MPI scripts also handle the proper order in which libraries should be called.

Example 7-16 Use of MPI script mpixlf77

```
XL          = /bgsys/drivers/ppcfloor/comm/bin/mpixlf77

EXE         = fhello
OBJ         = hello.o
SRC         = hello.f
FLAGS      = -O3 -qarch=450 -qtune=450
FLD        = -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
#        ${XL} $(FLAGS) $(BGP_LDIRS) -o $(EXE) $(OBJ) $(BGP_LIBS)
        ${XL} $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC)
        ${XL} $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
        rm hello.o fhello
```

7.6 MPI communications performance

Communications performance is an important aspect when running parallel applications, particularly, when running on a distributed-memory system such as the Blue Gene/P system. On both the Blue Gene/L and Blue Gene/P systems, instead of implementing a single type of network that is capable of transporting all protocols that are needed, these two systems have separate networks for different types of communications.

Usually the following measurements provide information about the network and can be used to look at parallel performance of applications:

Bandwidth	The number of MB of data that can be sent from one node to another node in one second
Latency	The amount of time it takes for the first byte that is sent from one node to reach its target node

The values for bandwidth and latency provide information about communication.

Here we illustrate two cases. The first case corresponds to a benchmark that involves a single transfer. The second case corresponds to a collective as defined in the *Intel® MPI Benchmarks*. Intel MPI Benchmarks was formerly known as “Pallas MPI Benchmarks (PMB-MPI1 for MPI1 standard functions only). Intel MPI Benchmarks - MPI1 provides a set of elementary MPI benchmark kernels.

For more details, see the product documentation included in the package that you can download from the following Intel Web page:

<http://www.intel.com/cd/software/products/asm-na/eng/219848.htm>

7.6.1 MPI point-to-point

In the Intel MPI Benchmarks, a single transfer corresponds to the PingPong and PingPing benchmarks. We illustrate a comparison between the Blue Gene/L and Blue Gene/P systems for the case of PingPong. This benchmark illustrates a single message that is transferred between two MPI tasks, in our case, on two different nodes.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/L system, the benchmark was run in co-processor mode. (See *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686.) On the Blue Gene/P system, we used the SMP Node Mode. `mpirun` was invoked as shown in Example 7-17 and Example 7-18 for the Blue Gene/L and Blue Gene/P systems respectively.

Example 7-17 mpirun on the Blue Gene/L system

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0
BGL_APP_L1_SWOA=0" -partition R000 -cwd /bglscratch/pallas -exe
/bglscratch/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512
PingPong" | tee IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Example 7-18 mpirun on the Blue Gene/P system

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP -partition R01-M1 -cwd
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB
.perf.rts -args "-msglen 4194304.txt -npmin 512 PingPong" | tee
IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Figure 7-4 shows the bandwidth on the torus network as a function of the message size, for one simultaneous pair of nearest neighbor communications. The protocol switch from short to eager is visible in both cases, where the eager to rendezvous switch is most pronounced on the Blue Gene/L system (see the asterisks (*)). This figure also shows the improved performance on the Blue Gene/P system (see the diamonds).

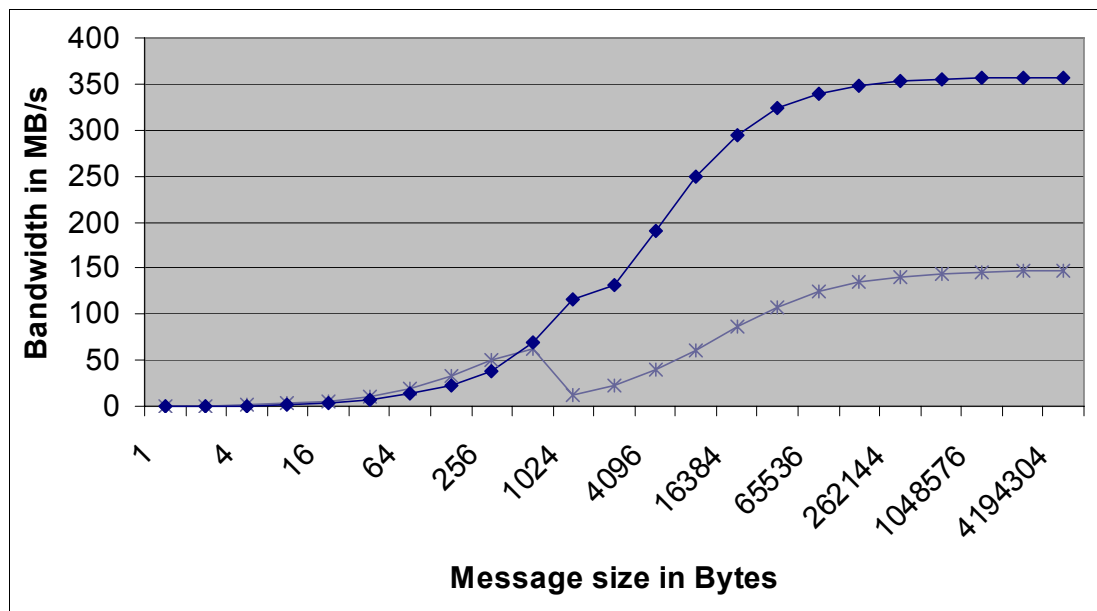


Figure 7-4 Bandwidth versus message size

7.6.2 MPI collective

In the Intel MPI Benchmarks, the collective benchmarks correspond to the Bcast, Allgather, Allgatherv, Alltoall, Alltoallv, Reduce, REduce_scatter, Allreduce, and Barrier benchmarks. We illustrate a comparison between the Blue Gene/L and Blue Gene/P systems for the case of Allreduce, which is a popular collective that is used in certain scientific applications. These benchmarks measure the message passing power of a system as well as the quality of the implementation.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/P system, the benchmark was run in co-processor mode. On the Blue Gene/L system, we used SMP node mode. `mpirun` was invoked as shown in Example 7-19 and Example 7-20 for the Blue Gene/L and Blue Gene/P systems respectively.

Example 7-19 mpirun on the Blue Gene/L system

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0
BGL_APP_L1_SWOA=0" -partition R000 -cwd
/bglscratch/BGTH/testsmall1512nodeBGL/pallas -exe
/bglscratch/BGTH/testsmall1512nodeBGL/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen
4194304.txt -npmin 512 Allreduce" | tee
IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Example 7-20 mpirun on the Blue Gene/P system

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP -partition R01-M1 -cwd
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB
.perf.rts -args "-msglen 4194304.txt -npmin 512 Allreduce" | tee
IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Collective operations are more efficient on the Blue Gene/P system. You should try to use collective operations instead of point-to-point communication wherever possible. The overhead for point-to-point communications is much larger than for collectives. Unless all of your point-to-point communication is purely to the nearest neighbor, it is also difficult to avoid network congestion on the torus network.

Alternatively, collective operations can use the barrier (global interrupt) network or the torus network. If they run over the torus network, they can still be optimized by using specially designed communication patterns that achieve optimum performance. Doing this manually with point-to-point operations is possible in theory, but in general, the implementation in the Blue Gene/P MPI library will offer superior performance.

With point-to-point communication, the goal of reducing the point-to-point Manhattan distances necessitates a good mapping of MPI tasks to the physical hardware. For collectives, mapping is equally important because most collective implementations prefer certain communicator shapes to achieve optimum performance. Refer to Appendix E, "Mapping" on page 281, which illustrates the technique of mapping.

Similar to point-to-point communications, collective communications also work best if you do not use complicated derived data types, and if your buffers are aligned to 16-byte boundaries. While the MPI standard explicitly allows for MPI collective communications to take place at the same time as point-to-point communications (on the same communicator), generally we do not recommend this for performance reasons.

Table 7-1 summarizes the MPI collectives that have been optimized on the Blue Gene/P system, together with their performance characteristics when executed on the various networks of the Blue Gene/P system.

Table 7-1 MPI collectives optimized on the Blue Gene/P system

MPI routine	Condition	Network	Performance
MPI_Barrier	MPI_COMM_WORLD	Barrier (global interrupt) network	1.2 μ s
MPI_Barrier	Any communicator	Torus network	30 μ s
MPI_Broadcast	MPI_COMM_WORLD	Collective network	817 MBps
MPI_Broadcast	Rectangular communicator	Torus network	934 MBps
MPI_Allreduce	MPI_COMM_WORLD fixed-point	collective network	778 MBps
MPI_Allreduce	MPI_COMM_WORLD floating point	Collective network	98 MBps
MPI_Alltoall[v]	Any communicator	Torus network	84-97% peak
MPI_Allgatherv	N/A	Torus network	same as broadcast

Figure 7-5 shows a comparison between the Blue Gene/L and Blue Gene/P systems for the MPI_Allreduce() type of communication.

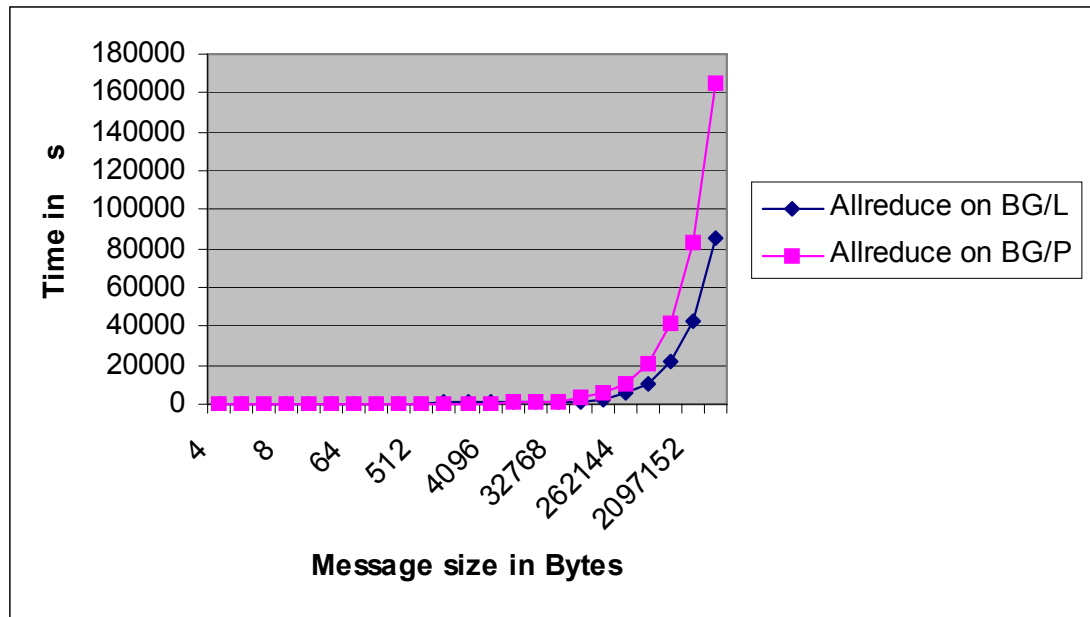


Figure 7-5 MPI_Allreduce() performance on 512 nodes

7.7 OpenMP

The OpenMP API is supported on the Blue Gene/P system for shared-memory parallel programming in C/C++ and Fortran. This API has been jointly defined by a group of hardware and software vendors and has evolved as a standard for shared-memory parallel programming.

OpenMP consists of a collection of compiler directives and a library of functions that can be invoked within an OpenMP program. This combination provides a simple interface for developing parallel programs on shared-memory architectures. In the case of the Blue Gene/P system, it allows the user to exploit the SMP mode on each Compute Node. Multi-threading is now enabled on the Blue Gene/P system. Using OpenMP, the user can have access to data parallelism as well as functional parallelism.

For additional information, refer to the official OpenMP Web site at:

<http://www.openmp.org/>

7.7.1 OpenMP implementation for Blue Gene/P

The Blue Gene/P system supports shared-memory parallelism on single nodes. The XL compilers support the following constructs:

- ▶ Full support for OpenMP 2.5 standard
- ▶ Support for the use of the same infrastructure as the OpenMP on AIX® and Linux
- ▶ Interoperability with MPI
 - MPI at outer level, across the Compute Nodes
 - OpenMP at the inner level, within a Compute Node
- ▶ Autoparallelization based on the same parallel execution framework
 - Enables autoparallelization as one of the loop optimizations
- ▶ Thread-safe version for each compiler
 - `bgx1f_r`
 - `bgx1c_r`
 - `bgcc_r`
- ▶ Use of the thread-safe compiler version with any threaded, OpenMP, or SMP application
 - `-qsmp` must be used on OpenMP or SMP applications.
 - `-qsmp` by itself automatically parallelizes loops.
 - `-qsmp=omp` parallelizes based on OpenMP directives in the code.
 - Shared-memory model is on the Blue Gene/P system.

7.7.2 Selected OpenMP compiler directives

The latest set of OpenMP compiler directives is documented in the OpenMP ARB release Version 2.5 specification. Version 2.5 combines Fortran and C/C++ specifications into a single specification. It also fixes inconsistencies. We summarize some of the directives as follows:

parallel	Directs the compiler for that section of the code to be executed in parallel by multiple threads
for	Directs the compiler to execute a <code>for</code> loop with independent iterations; iterations can be executed by different threads in parallel
parallel for	The syntax for parallel loops

sections	Directs the compiler of blocks of non-iterative code that can be executed in parallel
parallel sections	Syntax for parallel sections
critical	Restricts the following section of the code to be executed by a single thread at a time
single	Directs the compiler to execute a section of the code by a single thread

Parallel operations are often expressed in C/C++ and Fortran95 programs as for loops as shown in Example 7-21.

Example 7-21 for loops in Fortran and C

```
for (i = start; i < num; i += end)
{ array[i] = 1; m[i] = c;}
```

or

```
integer i, n, sum
sum = 0
do 5 i = 1, n
sum = sum + i
5 continue
```

The compiler can automatically locate and, where possible, parallelize all countable loops in your program code in the following situations:

- ▶ There is no branching into or out of the loop.
- ▶ An increment expression is not within a critical section.
- ▶ A countable loop is automatically parallelized only if all of the following conditions are met:
 - The order in which loop iterations start or end does not affect the results of the program.
 - The loop does not contain I/O operations.
 - Floating point reductions inside the loop are not affected by round-off error, unless the `-qnostrict` option is in effect.
 - The `-qnostrict_induction` compiler option is in effect.
 - The `-qsmp=auto` compiler option is in effect.
 - The compiler is invoked with a thread-safe compiler mode.

In the case of C/C++ programs, OpenMP is invoked via pragmas as shown in Example 7-22.

Pragma: The word “pragma” is short for *pragmatic information*.²⁷ Pragma is a way to communicate information to the compiler:

```
#pragma omp <rest of pragma>
```

Example 7-22 pragma usage

```
#pragma omp parallel for
for (i = start; i < num; i += end)
{ array[i] = 1; m[i] = c;}
```

The for loop must not contain statements, such as the following examples, that allow the loop to be exited prematurely:

- ▶ break
- ▶ return
- ▶ exit
- ▶ go to labels outside the loop

In a for loop, the master thread creates additional threads. The loop is executed by all threads, where every thread has its own address space that contains all of the variables that the thread can access. Such variables might be:

- ▶ Static variables.
- ▶ Dynamically allocated data structures in the heap.
- ▶ Variables on the run-time stack.

In addition, variables must be defined according to the type. Shared variables have the same address in the execution context of every thread. It is important to understand that all threads have access to shared variables. Alternatively, private variables have a different address in the execution memory of every thread. A thread can access its own private variables, but it *cannot* access the private variable of another thread.

Pragma parallel: In the case of the parallel for pragma, variables are shared by default, with exception of the loop index.

Example 7-23 shows a simple Fortran95 example that illustrates the difference between private and shared variables.

Example 7-23 Fortran example using the parallel do directive

```
program testmem
  integer n
  parameter (n=2)
  parameter (m=1)
  integer a(n), b(m)
!$OMP parallel do
  do i = 1, n
    a(i) = i
  enddo
  write(6,*)'Done: testmem'
end
```

In Example 7-23, no variables are explicitly defined as neither private nor shared. In this case, by default, the compiler assigns the variable that is used for the do-loop index as private. The rest of the variables are shared. Figure 7-6 illustrates both private and shared variables as shown in *Parallel Programming in C with MPI and OpenMP*²⁸. In this figure, the blue and yellow arrows indicate which variables are accessible by all the threads.

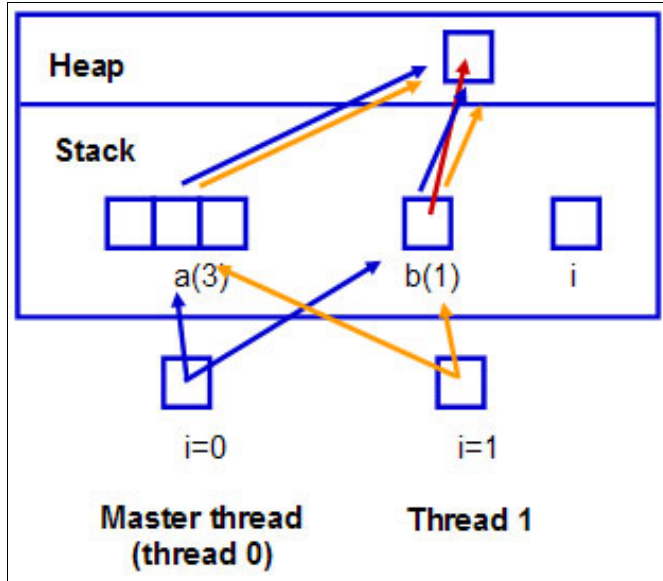


Figure 7-6 Memory layout for private and shared variables

7.7.3 Selected OpenMP compiler functions

The following functions are selected for the OpenMP compiler:

- omp_get_num_procs** Returns the number of processors
- omp_get_num_threads** Returns the number of threads in a particular parallel region
- omp_get_thread_num** Returns the thread identification number
- omp_set_num_threads** Allocates numbers of threads for a particular parallel region

7.7.4 Performance

To illustrate the effect of selected OpenMP compiler directives and the implications in terms of performance of a particular do loop, we chose the π programs presented in *Parallel Programming in C with MPI and OpenMP*²⁹ and apply them to the Blue Gene/P system. These simple examples illustrate how to use these directives and some of the implications in selecting a particular directive over another directive. Example 7-24 shows a simple program to compute π .

Example 7-24 Sequential version of the pi.c program

```
int main(argc, argv)
int argc;
char *argv[];
{
    long n, i;
    double area, pi, x;
    n = 1000000000;
    area = 0.0;
```

```

    for (i = 0; i < n; i++) {
        x = (i+0.5)/n;
        area += 4.0 / (1.0 + x*x);
    }
    pi = area / n;
    printf ("Estimate of pi: %7.5f\n", pi);
}

```

The first way to parallelize this code is to include an OpenMP directive to parallelize the for loop as shown in Example 7-25.

Example 7-25 Simple use of parallel for

```

#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
char *argv[];
{
    int num_threads;
    long n, i;
    double area, pi, x;
    long long time0, time1;
    double cycles, sec_per_cycle, factor;
    n = 1000000000;
    area = 0.0;
    time0 = timebase();
#pragma omp parallel for private(x)
    for (i = 0; i < n; i++) {
        x = (i+0.5)/n;
        area += 4.0 / (1.0 + x*x);
    }
    pi = area / n;
    printf ("Estimate of pi: %7.5f\n", pi);
    time1 = timebase();
    cycles = time1 - time0;
    factor = 1.0/850000000.0;
    sec_per_cycle = cycles * factor;
    printf("Total time %lf \n",sec_per_cycle, "Seconds \n");
}

```

Unfortunately this simple approach creates a race condition when computing the area. While different threads compute and update the value of the area, other threads might be computing and updating area as well, therefore producing the wrong results. There are two ways to solve this particular race condition. One way is to use a *critical pragma* to ensure mutual exclusion among the threads, and the other way is to use the *reduction clause*.

Example 7-26 illustrates use of the critical pragma.

Example 7-26 Usage of critical pragma

```
#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
char *argv[];
{
    int num_threads;
    long n, i;
    double area, pi, x;
    long long time0, time1;
    double cycles, sec_per_cycle, factor;
    n    = 1000000000;
    area = 0.0;
    time0 = timebase();
#pragma omp parallel for private(x)
    for (i = 0; i < n; i++) {
        x = (i+0.5)/n;
#pragma omp critical
        area += 4.0 / (1.0 + x*x);
    }
    pi = area / n;
    printf ("Estimate of pi: %7.5f\n", pi);
    time1 = timebase();
    cycles = time1 - time0;
    factor = 1.0/850000000.0;
    sec_per_cycle = cycles * factor;
    printf("Total time %lf \n",sec_per_cycle, "Seconds \n");
}
```

Example 7-27 corresponds to the reduction clause.

Example 7-27 Usage of the reduction clause

```
#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
char *argv[];
{
    int num_threads;
    long n, i;
    double area, pi, x;
    long long time0, time1;
    double cycles, sec_per_cycle, factor;
    n    = 1000000000;
    area = 0.0;
    time0 = timebase();
#pragma omp parallel for private(x) reduction(+: area)
```



```

for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0 / (1.0 + x*x);
}
pi = area / n;
printf ("Estimate of pi: %7.5f\n", pi);
time1 = timebase();
cycles = time1 - time0;
factor = 1.0/850000000.0;
sec_per_cycle = cycles * factor;
printf("Total time %lf \n",sec_per_cycle, "Seconds \n");
}

```

To compile these two programs on the Blue Gene/P system, the makefile for pi_critical.c shown in Example 7-28 can be used. A similar makefile can be used for the program illustrated in Example 7-27.

Example 7-28 Makefile for the pi_critical.c program

```

BGP_FLOOR = /bgsys/drivers/ppcfloor
BGP_IDIRS = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS = -L$(BGP_FLOOR)/comm/lib -L$(BGP_FLOOR)/runtime/SPI -lmpich.cnk
-lbcmfcoll.cnk -ldcmf.cnk -lrt -lSPI.cna -lpthread

XL = /opt/ibmcmp/vac/bg/9.0/bin/bgxlc_r

EXE = pi_critical_bgp
OBJ = pi_critical.o
SRC = pi_critical.c
FLAGS = -O3 -qsmp=omp:noauto -qthreaded -qarch=450 -qtune=450
-I$(BGP_FLOOR)/comm/include
FLD = -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
    ${XL} $(FLAGS) -o $(EXE) $(OBJ) timebase.o $(BGP_LIBS)
$(OBJ): $(SRC)
    ${XL} $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
    rm pi_critical.o pi_critical_bgp

```

Table 7-2 illustrates the performance improvement by using the reduction clause.

Table 7-2 Parallel performance using critical pragma versus reduction clause

Threads	Execution time in (seconds)	
	Using critical pragma	Using reduction clause
1		
POWER4 1.0 GHz	586.37	20.12
POWER5 1.9 GHz	145.03	5.22
POWER6™ 4.7 GHz	180.80	4.78
Blue Gene/P	560.08	12.80

	Execution time in (seconds)	
Threads	Using critical pragma	Using reduction clause
2		
POWER4 1.0 GHz	458.84	10.08
POWER5 1.9 GHz	374.10	2.70
POWER6 4.7 GHz	324.71	2.41
Blue Gene/P	602.62	6.42
4		
POWER4 1.0 GHz	552.54	5.09
POWER5 1.9 GHz	428.42	1.40
POWER6 4.7 GHz	374.51	1.28
Blue Gene/P	582.95	3.24

For a more in-depth information with additional examples, we recommend that you read the *Parallel Programming in C with MPI and OpenMP*.³⁰ In this section, we selected to illustrate only the π program.



Developing applications with IBM XL compilers

With the IBM XL family of optimizing compilers, you can develop C, C++, and Fortran applications for the Blue Gene/P system. This family comprises the following products, which we refer to in this chapter as *Blue Gene XL compilers*:

- ▶ XL C/C++ Advanced Edition V9.0 for Blue Gene
- ▶ XL Fortran Advanced Edition V11.1 for Blue Gene

The information that we present in this chapter is specific to the Blue Gene/P supercomputer. It does not include general XL compiler information. For complete documentation about these compilers, refer to the libraries at the following Web addresses:

- ▶ XL C/C++
<http://www.ibm.com/software/awdtools/xlcpp/library/>
- ▶ XL Fortran
<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/>

In this chapter, we discuss specific considerations for developing, compiling, and optimizing C/C++ and Fortran applications for the Blue Gene/P PowerPC 450 processor and a Single Instruction Multiple Data (SIMD), double precision floating point multiply add unit (double floating point multiply add (FMA)).

Several documents cover part of the material presented in this chapter. In addition to the XL family of compilers manuals that we reference throughout this chapter, we recommend that you read the following documents:

- ▶ *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686
- ▶ *IBM System Blue Gene Solution: Application Development*, SG24-7179

We also recommend that you read the article by Mark Mendell, “Exploiting the Dual Floating Point Units in Blue Gene/L,” which provides detailed information about the SIMD functionality in the XL family of compilers. You can find this article on the Web at:

<http://www-1.ibm.com/support/docview.wss?uid=swg27007511>

8.1 What is new

The Blue Gene/P system uses the same XL family of compilers as the Blue Gene/L system. The Blue Gene/P system supports cross-compilation, and the compilers run on the Front End Node. The compilers for the Blue Gene/P system have specific optimizations for its architecture. In particular, the XL family of compilers generate code appropriate for the double floating-point unit (FPU) of the Blue Gene/P system.

The Blue Gene/P system has compilers for the C, C++, and Fortran programming languages. The compilers on the Blue Gene/P system take advantage of the double FPU available on the Blue Gene/P system. They also incorporate code optimizations specific to the Blue Gene/P instruction scheduling and memory hierarchy characteristics.

In addition to the XL family of compilers, the Blue Gene/P system supports a version of the GNU compilers for C, C++, and Fortran. These compilers do not generate highly optimized code for the Blue Gene/P system. In particular, they do not automatically generate code for the double FPUs, and they do not support OpenMP.

Tools that are commonly associated with the GNU compilers (known as “binutils”) are supported in the Blue Gene/P system. The same set of compilers and tools is used for both Linux and the Blue Gene/P proprietary operating system. The Blue Gene/P system supports the execution of Python-based user applications.

The GNU compiler toolchain also provides the dynamic linker, which is used both by Linux and the Blue Gene/P proprietary operating system to support dynamic objects. The toolchain is tuned to support both environment. The GNU “aux vector” technique is employed to pass kernel specific information to the C library when tuning must be specific to one of the kernels.

8.2 Compiling and linking applications on Blue Gene/P

In this section, we provide information about compiling and linking applications that will run on the Blue Gene/P system. For complete information about compiler and linker options, see the following documents that are available on the Web:

- ▶ *XL C/C++ Compiler Reference*
<http://www-306.ibm.com/software/awdtools/xlcpp/library/>
- ▶ *XL Fortran User Guide*
<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/>

You can also find these documents in the following directories:

- ▶ /opt/ibmcmp/vacpp/bg/9.0/doc (C and C++)
- ▶ /opt/ibmcmp/xf/bg/11.1/doc (Fortran)

The compilers can be found in the following directories:

- ▶ /opt/ibmcmp/vac/bg/9.0/bin
- ▶ /opt/ibmcmp/vacpp/bg/9.0/bin
- ▶ /opt/ibmcmp/xf/bg/11.1/bin

In the Blue Gene/P release, you will notice the following differences for compiling and linking applications:

- ▶ Blue Gene/P compiler wrapper names have changed:
 - `blrts_` is replaced by `bg`.
 - `xl11.1`, `vacpp 9.0`, and `vac 9.0` on the Blue Gene/L system support both `blrts_` and `bg`.
- ▶ `-qarch=450d/450` is for the Blue Gene/P system, and `440d/440` is for the Blue Gene/L system.

8.3 Default compiler options

Compilations most commonly occur on the Front End Node. The resulting program can run on the Blue Gene/P system without manually copying the executable to the Service Node. See Chapter 9, “Running and debugging applications” on page 129, and Chapter 13, “`mpirun`” on page 217, to learn how to run programs on the Blue Gene/P system.

The script or makefile that you use to invoke the compilers should have certain compiler options. Specifically the architecture-specific options, which optimize processing for the Blue Gene/P 450d processor architecture, should be set to the following defaults:

- ▶ `-qarch=450d`

Generates parallel instructions for the PowerPC 450 processor and a SIMD, double precision floating point multiply add unit (double FMA). If you encounter problems with code generation, you can reset this option to `-qarch=450`. This option generates code for a single FPU only, but it can give correct results if invalid code is generated by `-qarch=450d`.
- ▶ `-qtune=450`

Optimizes object code for the 450 family of processors. Single FPU only.
- ▶ `-qcache=level=1:type=i:size=32:line=32:assoc=64:cost=8`

Specifies the L1 instruction cache configuration for the Blue Gene/P architecture to allow greater optimization with options `-O4` and `-O5`.
- ▶ `-qcache=level=1:type=d:size=32:line=32:assoc=64:cost=8`

Specifies the L1 data cache configuration for the Blue Gene/P architecture to allow greater optimization with options `-O4` and `-O5`.
- ▶ `-qcache=level=2:type=c:size=4096:line=128:assoc=8:cost=40`

Specifies the L2 (combined data and instruction) cache configuration for the Blue Gene/P architecture to allow greater optimization with options `-O4` and `-O5`.
- ▶ `-qnoautoconfig`

Allows code to be cross-compiled on other machines at optimization levels `-O4` or `-O5`, by preserving the Blue Gene/P architecture-specific options.

Scripts are already available that do much of this for you. They reside in the same `bin` directory as the compiler binary (`/opt/ibmcmp/xlf/bg/11.1/bin` or `/opt/ibmcmp/vacpp/bg/9.0/bin` or `/opt/ibmcmp/vac/bg/9.0/bin`). Table 8-1 on page 94 lists the names.

Table 8-1 Scripts available in the bin directory for compiling and linking

Language	Script name or names
C	bgc89, bgc99, bgcc, bgxlc bgc89_r, bgc99_r bgcc_r, bgxlc_r
C++	bgxlc++, bgxlc++_r, bgxlc, bgxlc_r
Fortran	bgf2003, bgf95, bgxlf2003, bgxlf90_r, bgxlf_r, bgf77, bgfort77, bgxlf2003_r, bgxlf95, bgf90, bgxlf, bgxlf90, bgxlf95_r

Important: The double FPU does not generate exceptions. Therefore, the `-qfltrap` option is invalid with the 450d processor. Instead the user should reset the 450d processor to `-qarch=450`.

8.4 Unsupported options

The following compiler options, although available for other IBM systems, are not supported by the Blue Gene/P hardware. Therefore, do not use them:

- ▶ `-q64`: The Blue Gene/P system uses a 32-bit architecture; you cannot compile in 64-bit mode.
- ▶ `-qaltivec`: The 450 processor does not support VMX instructions or vector data types.

8.5 Support for threads, OpenMP, and SMP

The Blue Gene/P system supports shared-memory parallelism on single nodes. The XL compilers support the following constructs:

- ▶ Full support for the OpenMP 2.5 standard³¹
- ▶ Use of the same infrastructure as the OpenMP that is supported on AIX and Linux
- ▶ Interoperability with MPI
 - MPI at outer level, across the Compute Nodes
 - OpenMP at the inner level, within a Compute Node
- ▶ Autoparallelization based on the same parallel execution framework
 - Enablement of autoparallelization as one of the loop optimizations
- ▶ Thread-safe version for each compiler
 - `bgxlf_r`
 - `bgxlc_r`
 - `bgxlc_r`
 - `bgcc_r`

The thread-safe compiler version should be used with any threaded, OpenMP, or SMP application.

Thread-safe libraries: Thread-safe libraries ensure that data access and updates are synchronized between threads.

- ▶ Usage of `-qsmp` and `-qthreaded` OpenMP and SMP applications
 - `-qsmp` by itself automatically parallelizes loops.
 - `-qsmp=omp` automatically parallelizes based on OpenMP directives in the code.
 - `-qsmp=omp:noauto -qthreaded` should be used when parallelizing codes manually. It prevents the compiler from trying to automatically parallelize loops.

Note: See the language reference for more details about the `-qsmp` suboptions at: <http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp>

8.6 XL runtime libraries

The libraries listed in Table 8-2 are linked into your application automatically by the XL linker when you create your application.

MASS libraries: The exception to this statement is for the `libmassv.a` file (the Mathematical Acceleration Subsystem (MASS) libraries). This file must be explicitly specified on the linker command. See 8.7, “Mathematical Acceleration Subsystem libraries” on page 96, for information about the MASS libraries.

Table 8-2 XL static and dynamic libraries

File name	Description
<code>libibmc++.a</code>	IBM C++ library
<code>libxlf90.a</code> , <code>libxlf90.so</code>	IBM XLF runtime library
<code>libxlfmath.a</code> , <code>libxlfmath.so</code>	IBM XLF stubs for math routines in system library <code>libm</code> , for example, <code>_sin()</code> for <code>sin()</code> , <code>_cos()</code> for <code>cos()</code> , and so on
<code>libxlfpm4.a</code> , <code>libxlfpm4.so</code>	IBM XLF to be used with <code>-qautobd1=db14</code> (promote floating-point objects that are single precision)
<code>libxlfpad.a</code> , <code>libxlfpad.so</code>	IBM XLF runtime routines to be used with <code>-qautobd1=db1pad</code> (promote floating-point objects and <code>pad</code> other types if they can share storage with promoted objects)
<code>libxlfpm8.a</code> , <code>libxlfpm8.so</code>	IBM XLF runtime routines to be used with <code>-qautobd1=db18</code> (promote floating-point objects that are double precision)
<code>libxl.a</code>	IBM low-level runtime library
<code>libxlopt.a</code>	IBM XL optimized intrinsic library <ul style="list-style-type: none"> ▶ Vector intrinsic functions ▶ BLASS routines
<code>libmass.a</code>	IBM XL MASS library: Scalar intrinsic functions
<code>libmassv.a</code>	IBM XL MASSV library: Vector intrinsic functions
<code>ibxlomp_ser.a</code>	IBM XL Open MP compatibility library

8.7 Mathematical Acceleration Subsystem libraries

The MASS consists of libraries of tuned mathematical intrinsic functions that are available in versions for the AIX and Linux machines, including the Blue Gene/P system. The MASS libraries provide improved performance over the standard mathematical library routines, are thread-safe, and support compilations in C, C++, and Fortran applications. For more information about MASS, refer to the Mathematical Acceleration Subsystem Web page at:

<http://www-306.ibm.com/software/awdtools/mass/index.html>

8.8 Engineering and Scientific Subroutine Library libraries

The Engineering and Scientific Subroutine Library (ESSL) for Linux on POWER supports the Blue Gene/P system. ESSL provides over 150 math subroutines that have been specifically tuned for performance on the Blue Gene/P system. For more information about ESSL, refer to the Engineering Scientific Subroutine Library and Parallel ESSL Web page at:

<http://www-03.ibm.com/systems/p/software/essl.html>

Important: When using IBM XL Fortran V11.1 for IBM System Blue Gene, customers must use ESSL 4.3.1. If an attempt is made to install a wrong mix of ESSL and XLF, the rpm installation fails with a dependency error message.

8.9 Tuning your code for Blue Gene/P

In the sections that follow, we describe strategies that you can use to best exploit the SIMD capabilities of the Blue Gene/P 450 processor and the XL compilers' advanced instruction scheduling, as well as to register the allocation algorithms.

8.9.1 Using the compiler optimization options

The -03 compiler option provides a high level of optimization and automatically sets other options that are especially useful on the Blue Gene/P system. The -qhot=simd option enables SIMD vectorization of loops. It is enabled by default if you use -04, -05, or -qhot.

For more information about optimization options, see the following references:

- ▶ “Optimizing your applications” in the *XL C/C++ Programming Guide*, under Product Documentation on the following Web page

<http://www-306.ibm.com/software/awdtools/xlcpp/library/>

- ▶ “Optimizing XL Fortran programs” in the *XL Fortran User Guide*, under Product Documentation on the following Web page

<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/>

8.9.2 PowerPC 450 processor parallel double-precision floating point multiply add unit

Similar to the Blue Gene/L system, floating point instructions can operate simultaneously on the primary and secondary registers. Figure 8-1 illustrates these registers.

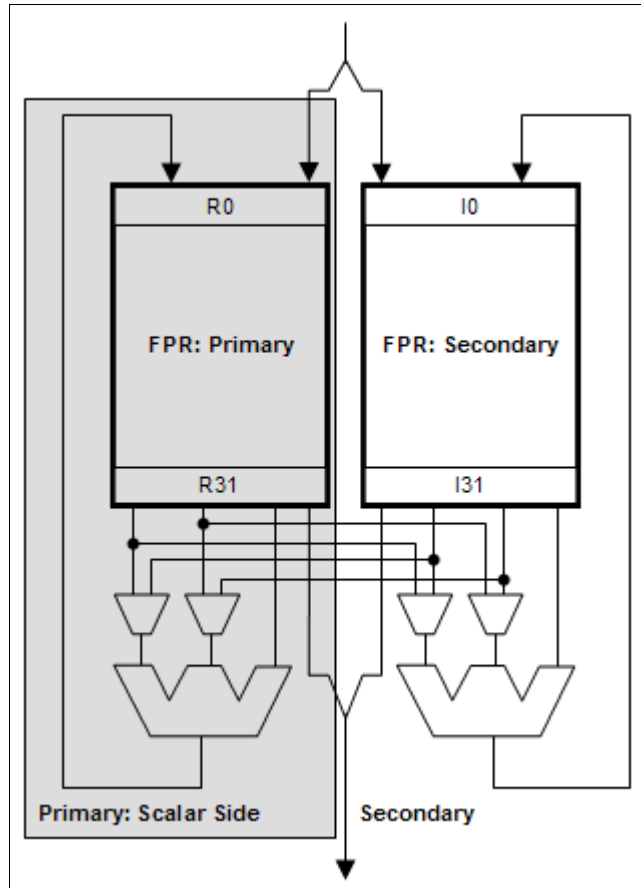


Figure 8-1 Blue Gene/P dual floating point unit

The registers allow the PowerPC 450 processor to operate certain identical operations in parallel. Load/store instructions can also be issued with a single instruction. For more detailed information, see the white paper *Exploiting the Dual Floating Point Units in Blue Gene/L* on the Web at:

<http://www-1.ibm.com/support/docview.wss?uid=swg27007511>

The IBM XL compilers leverage this functionality under the following conditions:

- ▶ Parallel instructions are issued for load/store instructions if the alignment and size are aligned with *natural alignment*. This is 16 bytes for a pair of doubles, but only 8 bytes for a pair of floats.
- ▶ The compiler can issue parallel instructions when the application vectors have stride-one memory accesses. However, the compiler via IPA issues parallel instructions with non-stride-one data in certain loops, if it can be shown to improve performance.
- ▶ `-qhot=simd` is the default with `-qarch=450d`.

- ▶ -04 provides analysis at compile time with limited scope analysis and issuing parallel instructions (SIMD).
- ▶ -05 provides analysis for the entire program at link time to propagate alignment information. You must compile and link with -05 to obtain the full benefit.

8.9.3 Using Single Instruction Multiple Data instructions in applications

On the Blue Gene/P system, normal PowerPC assembler instructions use the primary floating point pipe. To enable instructions in parallel, special assembly instructions must be generated using the following compiler options:

- qarch=450d** This flag in the compiler enables parallel instructions to use the primary and secondary registers (SIMD instructions). See Table 8-1.
- qtune=450** This flag optimizes code for the IBM 450 microprocessors, as previously mentioned.
- O2 and up** This option in the compiler enables parallel instructions.

The XL compiler optimizer consists of two major parts:

- ▶ Toronto Portable Optimizer (TPO) for high-level inter-procedural optimization
- ▶ Toronto Optimizing Back End with Yorktown (TOBEY) for low-level back-end optimization

SIMD instructions occur in both optimizers. SIMD instruction generation in TOBEY is activated by default for -02 and up. SIMD generation in TPO is added when using -qhot, -04, or -05. Specifically, the -qhot option adds SIMD generation, but options -04 and -05 automatically call -qhot. For more details, see the C, C++, and Fortran manuals on the Web at the following addresses:

- ▶ XL C/C++
<http://www.ibm.com/software/awdtools/xlcpp/library/>
- ▶ XL Fortran
<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/>

For some applications, the compiler generates more efficient code without the TPO SIMD level. If you have statically allocated array, and a loop in the same routine, call TOBEY with -qhot or -04. Nevertheless, on top of SIMD generation from TOBEY, with -qhot, optimizations are enabled that can alter the semantic of the code and on rare occasions can generate less efficient code. Also, with -qhot=nosimd, you can suppress some of these optimizations.

To use the SIMD capabilities of the XL compilers:

1. Start to compile:
 - 03 -qarch=450d -qtune=450

We recommend that you use -qarch=450d -qtune=450, in this order. The compiler only generates SIMD instructions from -02 and up.
2. Increase the optimization level, and call the high level inter-procedural optimizer:
 - -05 (link time, whole-program analysis, and SIMD instruction)
 - -04 (compile time, limited scope analysis, and SIMD instructions)
 - -03 -qhot=simd (compile time, less optimization, and SIMD instructions)
3. Tune your program:
 - a. Check the SIMD instruction generation in the object code listing (-qsource -qlist).
 - b. Use compiler feedback (-qreport -qhot) to guide you.
 - c. Help the compiler with extra information (directives and pragmas).

- Enter the alignment information with directives and pragmas. In C, enter:

```
__alignx
```

In Fortran, enter:

```
ALIGNX
```

- Tell the compiler that data accessed through pointers is disjoint. In C, enter:
- Use constant loop bound, #define, when possible.
- Use data flow instead of control flow.
- Use select instead of if/then/else. Use macros instead of calls.
- Tell the compiler not to generate SIMD instructions if it is not profitable (trip count low). In C, enter:

```
#pragma nosimd
```

In Fortran, enter the following line just before the loop:

```
!IBM* NOSIMD
```

- Many applications can require modifying algorithms. The previous bullet, which explains how not to generate SIMD instructions, gives constructs that might help to modify the code. Here are hints to use when modifying your code:
 - Loops must be stride one accesses.
 - For function calls in loop:
 - Try to inline the calls.
 - Loop with if statement.
 - Use pointer and aliasing.
 - Use integer operations.
 - Assumed shape arrays in Fortran 90 can hurt enabling SIMD instructions.
- Generate compiler diagnostics to help you modify and understand how the compiler is optimizing sections of your applications:

The `-qreport` compiler option generates a diagnostic report about SIMD instruction generation. To analyze the generated code and the use of quadword loads and stores, you must look at the pseudo assembler code within the object file listing. The diagnostic report provides two types of information about SIMD generation:

- Information on success

```
(simdizable) [feature] [version]
```

[feature] further characterizes the simdizable loop:

misalign (compile time store)

Refers to a simdizable loop with misaligned accesses.

shift (4 compile time)

Refers to a simdizable loop with 4 stream shift inserted. `shift` refers to the number of misaligned data references that were found. It has a performance impact since these loops must be loaded across, and then an extra select instruction must be inserted.

priv

Indicates that the compiler has generated a private variable. `priv` means a private variable was found. In general, it should have no performance impact, but in practice it sometimes does.

reduct Indicates that a simdizable loop has a reduction construct. reduct means that a reduction was found. It is simdized using partial sums, which must be added at the end of the loop.

[version] further characterizes if and why versioned loops were created:

relative align Indicates the version for relative alignment. The compiler has generated a test and two versions.

trip count Versioned for a short runtime trip count.

- Information on failure
 - In case of misalignment: misalign(...)
 - * Non-natural: Non-naturally aligned accesses
 - * Run time: Runtime alignment
 - About the structure of the loop
 - * Irregular loop structure (while-loop)
 - * Contains control flow: if/then/else
 - * Contains function call: Function call bans SIMD instructions
 - * Trip count too small
 - About dependences: dependence due to aliasing
 - About array references
 - * Access not stride one
 - * Memory accesses with unsupported alignment
 - * Contains runtime shift
 - About pointer references: Non-normalized pointer accesses

8.10 Tips for optimizing constructs

The following sections are an excerpt from the *IBM System Blue Gene Solution: Application Development*, SG24-7179, but tailored to the Blue Gene/P system since they apply here as well. They provide useful tips on how to optimize certain constructs in your code.

8.10.1 Structuring data in adjacent pairs

The Blue Gene/P 450d processor's dual FPU includes special instructions for parallel computations. The compiler tries to pair adjacent single-precision or double-precision floating point values to operate on them in parallel. Therefore, you can accelerate computations by defining data objects that occupy adjacent memory blocks and are naturally aligned. These include arrays or structures of floating-point values and complex data types.

Whether you use an array, a structure, or a complex scalar, the compiler searches for sequential pairs of data for which it can generate parallel instructions. For example, using the C code in Example 8-1, each pair of elements in a structure can be operated on in parallel.

Example 8-1 Adjacent paired data

```
struct quad {
    double a, b, c, d;
};

struct quad x, y, z;

void foo()
```

```

{
  z.a = x.a + y.a;
  z.b = x.b + y.b; /* can load parallel (x.a,x.b), and (y.a, y.b), do parallel add, and
store parallel (z.a, z.b) */

  z.c = x.c + y.c;
  z.d = x.d + y.d; /* can load parallel (x.c,x.d), and (y.c, y.d), do parallel add, and
store parallel (z.c, z.d) */
}

```

The advantage of using complex types in arithmetic operations is that the compiler automatically uses parallel add, subtract, and multiply instructions when complex types appear as operands to addition, subtraction, and multiplication operators. Furthermore, the data that you provide does not need to represent complex numbers. In fact, both elements are represented internally as two real values. See 8.10.8, “Complex type manipulation functions” on page 109, for a description of the set of built-in functions that are available for the Blue Gene/P system. These functions are especially designed to efficiently manipulate complex-type data and include a function to convert non-complex data to complex types.

8.10.2 Using vectorizable basic blocks

The compiler schedules instructions most efficiently within *extended basic blocks*. Extended basic blocks are code sequences that can contain conditional branches but have no entry points other than the first instruction. Specifically, minimize the use of branching instructions for:

- ▶ Handling special cases, such as the generation of not-a-number (NaN) values.
- ▶ C/C++ error handling that sets a value for `errno`.
To explicitly inform the compiler that none of your code will set `errno`, you can compile with the `-qignerrno` compiler option (automatically set with `-O3`).
- ▶ C++ exception handlers.
To explicitly inform the compiler that none of your code will throw any exceptions, and therefore, that no exception-handling code must be generated, you can compile with the `-qnoeh` compiler option.

In addition, the optimal basic blocks remove dependencies between computations, so that the compiler views each statement as entirely independent. You can construct a basic block as a series of independent statements or as a loop that repeatedly computes the same basic block with different arguments.

If you specify the `-qhot=simd` compilation option, along with a minimum optimization level of `-O2`, the compiler can then vectorize these loops by applying various transformations, such as unrolling and software pipelining. See 8.10.4, “Removing possibilities for aliasing (C/C++)” on page 102, for additional strategies for removing data dependencies.

8.10.3 Using inline functions

An inline function is expanded in any context in which it is called. This expansion avoids the normal performance overhead associated with the branching for a function call, and it allows functions to be included in basic blocks. The XL C/C++ and Fortran compilers provide several options for inlining.

The following options instruct the compiler to automatically inline all functions it deems appropriate:

- ▶ XL C/C++
 - -O through -O5
 - -qipa
- ▶ XL Fortran
 - -O4 or -O5
 - -qipa

With the following options, you can select or name functions to be inlined:

- ▶ XL C/C++
 - -qinline
 - -Q
- ▶ XL Fortran
 - -Q

In C/C++, you can also use the standard `inline` function specifier or the `__attribute__((always_inline))` extension in your code to mark a function for inlining.

Usage of inlining: Do not overuse inlining, because there are limits on how much inlining will be done. Mark the most important functions.

For more information about the various compiler options for controlling function inlining, see the following publications:

- ▶ *XL C/C++ Compiler Reference*
<http://www-306.ibm.com/software/awdtools/xlcpp/library/>
- ▶ *XL Fortran User Guide*
<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/>

Also available from this Web address, refer to the *XL C/C++ Language Reference* for information about the different variations of the `inline` keyword supported by XL C and C++, as well as the inlining function attribute extensions.

8.10.4 Removing possibilities for aliasing (C/C++)

When you use pointers to access array data in C/C++, the compiler cannot assume that the memory accessed by pointers will not be altered by other pointers that refer to the same address. For example, if two pointer input parameters share memory, the instruction to store the second parameter can overwrite the memory read from the first load instruction. This means that, after a store for a pointer variable, any load from a pointer must be reloaded. Consider the code in Example 8-2.

Example 8-2 Sample code

```
int i = *p;  
*q = 0;  
j = *p;
```

If `*q` aliases `*p`, then the value must be reloaded from memory. If `*q` does not alias `*p`, the old value that is already loaded into `i` can be used.

To avoid the overhead of reloading values from memory every time they are referenced in the code, and to allow the compiler to simply manipulate values that are already resident in registers, there are several strategies you can use. One approach is to assign input array element values to local variables and perform computations only on the local variables, as shown in Example 8-3.

Example 8-3 Array parameters assigned to local variables

```
#include <math.h>
void reciprocal_roots (const double* x, double* f)
{
    double x0 = x[0] ;
    double x1 = x[1] ;
    double r0 = 1.0/sqrt(x0) ;
    double r1 = 1.0/sqrt(x1) ;
    f[0] = r0 ;
    f[1] = r1 ;
}
```

If you are certain that two references do not share the same memory address, another approach is to use the `#pragma disjoint` directive. This directive asserts that two identifiers do not share the same storage, within the scope of their use. Specifically, you can use `pragma` to inform the compiler that two pointer variables do not point to the same memory address. The directive in Example 8-4 indicates to the compiler that the pointers-to-arrays of double `x` and `f` do not share memory.

Example 8-4 The #pragma disjoint directive

```
__inline void ten_reciprocal_roots (double* x, double* f)
{
    #pragma disjoint (*x, *f)
    int i;
    for (i=0; i < 10; i++)
        f[i]= 1.0 / sqrt (x[i]);
}
```

Important: The correct functioning of this directive requires that the two pointers be disjoint. If they are not, the compiled program will not run correctly.

8.10.5 Structure computations in batches

Floating-point operations are pipelined in the 450 processor, so that one floating-point calculation is performed per cycle, with a latency of approximately five cycles. Therefore, to keep the 450 processor's floating-point units busy, organize floating-point computations to perform step-wise operations in batches, for example, arrays of five elements and loops of five iterations. For the 450d, which has two FPUs, use batches of ten.

For example, with the 450d, at high optimization, the function in Example 8-5 on page 104 should perform ten parallel reciprocal roots in about five cycles more than a single reciprocal root. This is because the compiler will perform two reciprocal roots in parallel and then use the "empty" cycles to run four more parallel reciprocal roots.

Example 8-5 Function to calculate reciprocal roots for arrays of ten elements

```
__inline void ten_reciprocal_roots (double* x, double* f)
{
#pragma disjoint (*x, *f)

    int i;
    for (i=0; i < 10; i++)
        f[i]= 1.0 / sqrt (x[i]);
}
```

The definition in Example 8-6 shows “wrapping” the inlined, optimized `ten_reciprocal_roots` function, in Example 8-5, inside a function that allows you to pass in arrays of any number of elements. This function then passes the values in batches of ten to the `ten_reciprocal_roots` function and calculates the remaining operations individually.

Example 8-6 Function to pass values in batches of ten

```
static void unaligned_reciprocal_roots (double* x, double* f, int n)
{
#pragma disjoint (*x, *f)
    while (n >= 10) {
        ten_reciprocal_roots (x, f);
        x += 10;
        f += 10;
    }
    /* remainder */
    while (n > 0) {
        *f = 1.0 / sqrt (*x);
        f++, x++;
    }
}
```

8.10.6 Checking for data alignment

The Blue Gene/P architecture allows for two double-precision values to be loaded in parallel in a single cycle, provided that the load address is aligned so that the values that are loaded do not cross a cache-line boundary. If they cross this boundary, the hardware generates an alignment trap. This trap can cause the program to crash or result in a severe performance penalty if fixed at run-time by the kernel.

The compiler does not generate these parallel load and store instructions unless it is sure that it is safe to do so. For non-pointer local and global variables, the compiler knows when this is safe. To allow the compiler to generate these parallel loads and stores for accesses through pointers, include code that tests for correct alignment and that gives the compiler hints.

To test for alignment, first create one version of a function which asserts the alignment of an input variable at that point in the program flow. You can use the C/C++ `__alignx` built-in function or the Fortran `ALIGNX` function to inform the compiler that the incoming data is correctly aligned according to a specific byte boundary, so it can efficiently generate loads and stores.

The function takes two arguments. The first argument is an integer constant expressing the number of alignment bytes (must be a positive power of two). The second argument is the variable name, typically a pointer to a memory address.

Example 8-7 shows the C/C++ prototype for the function.

Example 8-7 C/C++ prototype

```
extern
#ifdef __cplusplus
"builtin"
#endif
void __alignx (int n, const void *addr)
```

Here n is the number of bytes. For example, `__align(16, y)` specifies that the address y is 16-byte aligned.

In Fortran95, the built-in subroutine is `ALIGNX(K,M)`, where K is of type `INTEGER(4)`, and M is a variable of any type. When M is an integer pointer, the argument refers to the address of the pointee.

Example 8-8 asserts that the variables x and f are aligned along 16-byte boundaries.

Example 8-8 Using the __alignx built-in function

```
#include <math.h>
#include <builtins.h>
__inline void aligned_ten_reciprocal_roots (double* x, double* f)
{
#pragma disjoint (*x, *f)
int i;
    __alignx (16, x);
    __alignx (16, f);
    for (i=0; i < 10; i++)
        f[i]= 1.0 / sqrt (x[i]);
}
```

The __alignx function: The `__alignx` function does not perform any alignment. It merely informs the compiler that the variables are aligned as specified. If the variables are not aligned correctly, the program does not run properly.

After you create a function to handle input variables that are correctly aligned, you can then create a function that tests for alignment and then calls the appropriate function to perform the calculations. The function in Example 8-9 checks to see whether the incoming values are correctly aligned. Then it calls the “aligned” (Example 8-8) or “unaligned” (Example 8-5) version of the function according to the result.

Example 8-9 Function to test for alignment

```
void reciprocal_roots (double *x, double *f, int n)
{
    /* are both x & f 16 byte aligned? */
    if ( (((int) x) | ((int) f)) & 0xf) == 0) /* This could also be done as:
                                                if (((int) x % 16 == 0) && ((int) f % 16) == 0) */
        aligned_ten_reciprocal_roots (x, f, n);
    else
        ten_reciprocal_roots (x, f, n);
}
```

The alignment test in Example 8-9 provides an optimized method of testing for 16-byte alignment by performing a bit-wise OR on the two incoming addresses and testing whether the lowest four bits are 0 (that is, 16-byte aligned).

8.10.7 Using XL built-in floating-point functions for Blue Gene/P

The XL C/C++ and Fortran95 compilers include a large set of built-in functions that are optimized for the PowerPC architecture. For a full description of them, refer to the following documents:

- ▶ Appendix B: “Built-In Functions” in *XL C/C++ Compiler Reference*
<http://www-306.ibm.com/software/awdtools/xlcpp/library/>
- ▶ “Intrinsic Procedures” in *XL Fortran Language Reference*
<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/>

In addition, on the Blue Gene/P system, the XL compilers provide a set of built-in functions that are specifically optimized for the PowerPC 450d dual FPU. These built-in functions provide an almost one-to-one correspondence with the dual floating-point instruction set.

All of the C/C++ and Fortran built-in functions operate on complex data types, which have an underlying representation of a two-element array, in which the real part represents the *primary* element and the imaginary part represents the *secondary* element. The input data that you provide does not need to represent complex numbers. In fact, both elements are represented internally as two real values. None of the built-in functions performs complex arithmetic. A set of built-in functions designed to efficiently manipulate complex-type variables is also available.

The Blue Gene/P built-in functions perform several types of operations as explained in the following paragraphs.

Parallel operations perform SIMD computations on the primary and secondary elements of one or more input operands. They store the results in the corresponding elements of the output. As an example, Figure 8-2 illustrates how a parallel-multiply operation is performed.

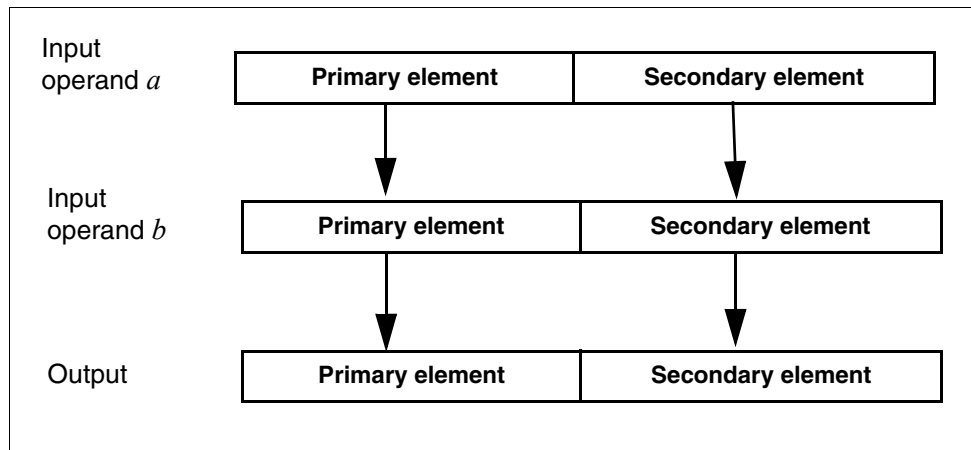


Figure 8-2 Parallel operations

Cross operations perform SIMD computations on the opposite primary and secondary elements of one or more input operands. They store the results in the corresponding elements in the output. As an example, Figure 8-3 illustrates how a cross-multiply operation is performed.

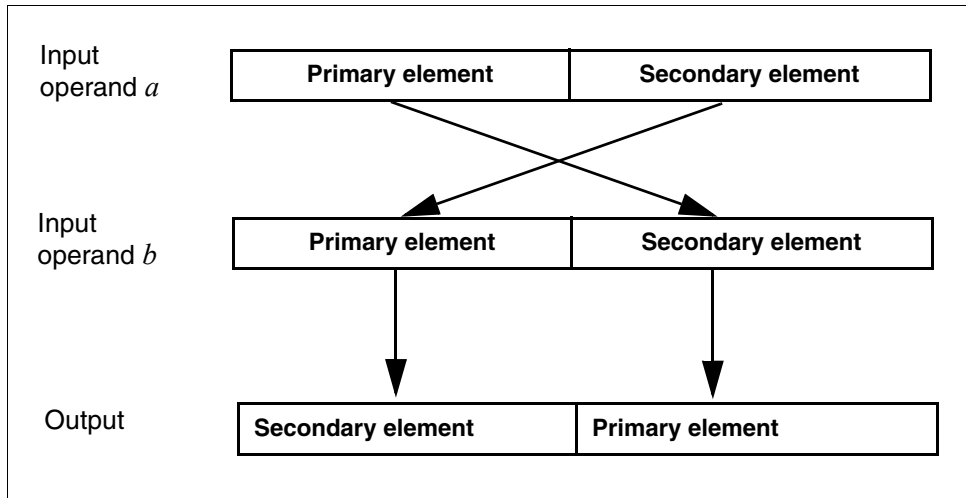


Figure 8-3 Cross-multiply operations

Copy-primary operations perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the primary element of the first operand is replicated to the secondary element. As an example, Figure 8-4 illustrates how a cross-primary-multiply operation is performed.

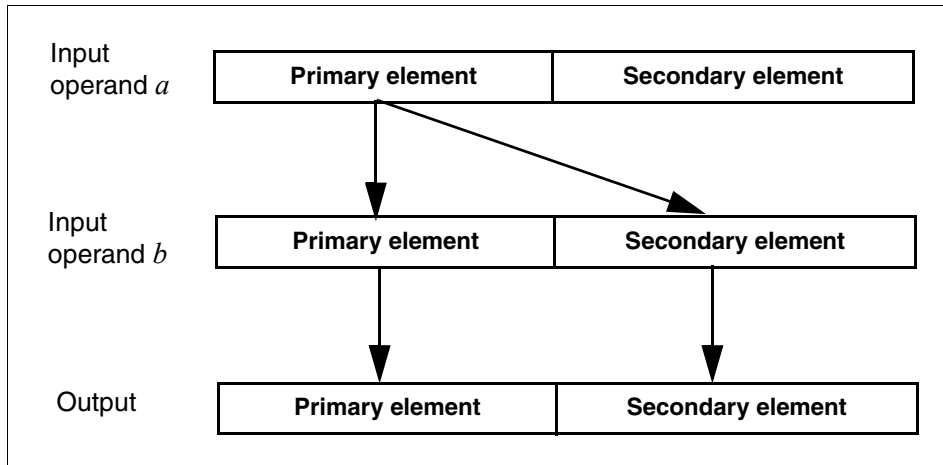


Figure 8-4 Copy-primary multiply operations

Copy-secondary operations perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the secondary element of the first operand is replicated to the primary element. As an example, Figure 8-5 illustrates how a cross-secondary multiply operation is performed.

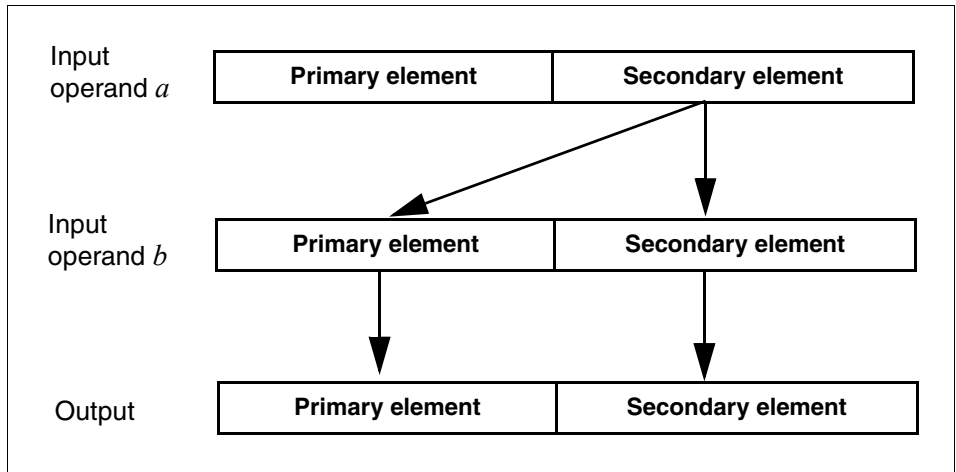


Figure 8-5 Copy-secondary multiply operations

In *cross-copy operations*, the compiler crosses either the primary or secondary element of the first operand, so that copy-primary and copy-secondary operations can be used interchangeably to achieve the same result. The operation is performed on the total value of the first operand. As an example, Figure 8-6 illustrates the result of a cross-copy multiply operation.

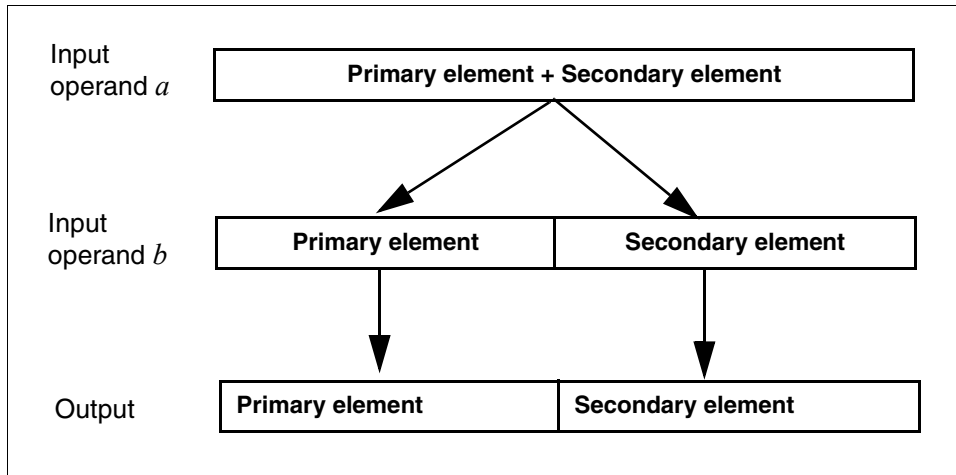


Figure 8-6 Cross-copy multiply operations

In the following paragraphs, we describe the available built-in functions by category. For each function, the C/C++ prototype is provided. In C, you do not need to include a header file to obtain the prototypes. The compiler includes them automatically. In C++, you must include the header file `builtins.h`.

Fortran does not use prototypes for built-in functions. Therefore, the interfaces for the Fortran95 functions are provided in textual form. The function names *omit* the double underscore (`__`) in Fortran95.

All of the built-in functions, with the exception of the complex type manipulation functions, require compilation under `-qarch=450d`. This is the default setting on the Blue Gene/P system.

To help clarify the English description of each function, the following notation is used:

element(variable)

Here *element* represents one of *primary* or *secondary*, and *variable* represents input variable *a*, *b*, or *c*, and the output variable *result*. For example, consider the following formula:

primary(result) = primary(a) + primary(b)

This formula indicates that the primary element of input variable *a* is added to the primary element of input variable *b* and stored in the primary element of the result.

To optimize your calls to the Blue Gene/P built-in functions, follow the guidelines provided in 8.9, “Tuning your code for Blue Gene/P” on page 96. Using the `alignx` built-in function (described in 8.10.6, “Checking for data alignment” on page 104), and specifying the `disjoint` pragma (described in 8.10.4, “Removing possibilities for aliasing (C/C++)” on page 102), are recommended for code that calls any of the built-in functions.

8.10.8 Complex type manipulation functions

Complex type manipulation functions, listed in Table 8-3, are useful for efficiently manipulating complex data types. Using these functions, you can automatically convert real floating-point data to complex types and extract the real (primary) and imaginary (secondary) parts of complex values.

Table 8-3 Complex type manipulation functions

Function	Convert dual reals to complex (single-precision): <code>__cmlplx</code>
Purpose	Converts two single-precision real values to a single complex value. The real <i>a</i> is converted to the primary element of the return value, and the real <i>b</i> is converted to the secondary element of the return value.
Formula	primary(result) = a secondary(result) = b
C/C++ prototype	float _Complex __cmlplx (float a, float b);
Fortran descriptions	CMPLXF(A,B) where A is of type REAL(4) where B is of type REAL(4) result is of type COMPLEX(4)
Function	Convert dual reals to complex (double-precision): <code>__cmlpx</code>
Purpose	Converts two double-precision real values to a single complex value. The real <i>a</i> is converted to the primary element of the return value, and the real <i>b</i> is converted to the secondary element of the return value.
Formula	primary(result) = a secondary(result) = b
C/C++ prototype	double _Complex __cmlpx (double a, double b); long double _Complex __cmlpxl (long double a, long double b); ^a
Fortran descriptions	CMPLX(A,B) where A is of type REAL(8) where B is of type REAL(8) result is of type COMPLEX(8)

Function	Extract real part of complex (single-precision): <code>__crealf</code>
Purpose	Extracts the primary part of a single-precision complex value <i>a</i> , and returns the result as a single real value.
Formula	result =primary(<i>a</i>)
C/C++ prototype	float <code>__crealf</code> (float <code>_Complex a</code>);
Fortran descriptions	CREALF(A) where A is of type COMPLEX(4) result is of type REAL(4)
Function	Extract real part of complex (double-precision): <code>__creal</code>, <code>__creall</code>
Purpose	Extracts the primary part of a double-precision complex value <i>a</i> , and returns the result as a single real value.
Formula	result =primary(<i>a</i>)
C/C++ prototype	double <code>__creal</code> (double <code>_Complex a</code>); long double <code>__creall</code> (long double <code>_Complex a</code>); ^a
Fortran descriptions	CREAL(A) where A is of type COMPLEX(8) result is of type REAL(8) CREALL(A) where A is of type COMPLEX(16) result is of type REAL(16)
Function	Extract imaginary part of complex (single-precision): <code>__cimagf</code>
Purpose	Extracts the secondary part of a single-precision complex value <i>a</i> , and returns the result as a single real value.
Formula	result =secondary(<i>a</i>)
C/C++ prototype	float <code>__cimagf</code> (float <code>_Complex a</code>);
Fortran descriptions	CIMAGF(A) where A is of type COMPLEX(4) result is of type REAL(4)
Function	Extract imaginary part of complex (double-precision): <code>__cimag</code>, <code>__cimagl</code>
Purpose	Extracts the imaginary part of a double-precision complex value <i>a</i> , and returns the result as a single real value.
Formula	result =secondary(<i>a</i>)
C/C++ prototype	double <code>__cimag</code> (double <code>_Complex a</code>); long double <code>__cimagl</code> (long double <code>_Complex a</code>); ^a
Fortran descriptions	CIMAG(A) where A is of type COMPLEX(8) result is of type REAL(8) CIMAGL(A) where A is of type COMPLEX(16) result is of type REAL(16)

a. 128-bit C/C++ long double types are not supported on Blue Gene/L. Long doubles are treated as regular double-precision longs.

8.10.9 Load and store functions

Table 8-4 lists and explains the various parallel load and store functions that are available.

Table 8-4 Load and store functions

Function	Parallel load (single-precision): <code>__lfps</code>
Purpose	Loads parallel single-precision values from the address of <i>a</i> , and converts the results to double-precision. The first word in <i>address(a)</i> is loaded into the primary element of the return value. The next word, at location <i>address(a)+4</i> , is loaded into the secondary element of the return value.
Formula	primary(result) = a[0] secondary(result) = a[1]
C/C++ prototype	double _Complex __lfps (float * a);
Fortran description	LOADFP(A) where A is of type REAL(4) result is of type COMPLEX(8)
Function	Cross load (single-precision): <code>__lfxs</code>
Purpose	Loads single-precision values that have been converted to double-precision, from the address of <i>a</i> . The first word in <i>address(a)</i> is loaded into the secondary element of the return value. The next word, at location <i>address(a)+4</i> , is loaded into the primary element of the return value.
Formula	primary(result) = a[1] secondary(result) = a[0]
C/C++ prototype	double _Complex __lfxs (float * a);
Fortran description	LOADFX(A) where A is of type REAL(4) result is of type COMPLEX(8)
Function	Parallel load: <code>__lfpd</code>
Purpose	Loads parallel values from the address of <i>a</i> . The first word in <i>address(a)</i> is loaded into the primary element of the return value. The next word, at location <i>address(a)+8</i> , is loaded into the secondary element of the return value.
Formula	primary(result) = a[0] secondary(result) = a[1]
C/C++ prototype	double _Complex __lfpd(double* a);
Fortran description	LOADFP(A) where A is of type REAL(8) result is of type COMPLEX(8)
Function	Cross load: <code>__lfxd</code>
Purpose	Loads values from the address of <i>a</i> . The first word in <i>address(a)</i> is loaded into the secondary element of the return value. The next word, at location <i>address(a)+8</i> , is loaded into the primary element of the return value.
Formula	primary(result) = a[1] secondary(result) = a[0]

C/C++ prototype	double _Complex __lfxd (double * a);
Fortran description	LOADFX(A) where A is of type REAL(8) result is of type COMPLEX(8)
Function	Parallel store (single-precision): __stfps
Purpose	Stores in parallel double-precision values that have been converted to single-precision, into <i>address(b)</i> . The primary element of <i>a</i> is converted to single-precision and stored as the first word in <i>address(b)</i> . The secondary element of <i>a</i> is converted to single-precision and stored as the next word at location <i>address(b)+4</i> .
Formula	b[0] = primary(a) b[1]= secondary(a)
C/C++ prototype	void __stfps (float * b, double _Complex a);
Fortran description	STOREFP(B, A) where B is of type REAL(4) A is of type COMPLEX(8) result is none
Function	Cross store (single-precision): __stfxs
Purpose	Stores double-precision values that have been converted to single-precision, into <i>address(b)</i> . The secondary element of <i>a</i> is converted to single-precision and stored as the first word in <i>address(b)</i> . The primary element of <i>a</i> is converted to single-precision and stored as the next word at location <i>address(b)+4</i> .
Formula	b[0] = secondary(a) b[1] = primary(a)
C/C++ prototype	void __stfxs (float * b, double _Complex a);
Fortran description	STOREFX(B, A) where B is of type REAL(4) A is of type COMPLEX(8) result is none
Function	Parallel store: __stfpd
Purpose	Stores in parallel values into <i>address(b)</i> . The primary element of <i>a</i> is stored as the first double word in <i>address(b)</i> . The secondary element of <i>a</i> is stored as the next double word at location <i>address(b)+8</i> .
Formula	b[0] = primary(a) b[1] = secondary(a)
C/C++ prototype	void __stfpd (double * b, double _Complex a);
Fortran description	STOREFP(B, A) where B is of type REAL(8) A is of type COMPLEX(8) result is none

Function	Cross store: <code>__stfxd</code>
Purpose	Stores values into <i>address(b)</i> . The secondary element of <i>a</i> is stored as the first double word in <i>address(b)</i> . The primary element of <i>a</i> is stored as the next double word at location <i>address(b)+8</i> .
Formula	b[0] = secondary(a) b[1] = primary(a)
C/C++ prototype	void <code>__stfxd</code> (double * b, double <code>_Complex</code> a);
Fortran description	STOREFP(B, A) where B is of type REAL(8) A is of type COMPLEX(8) result is none
Function	Parallel store as integer: <code>__stfpiw</code>
Purpose	Stores in parallel floating-point double-precision values into <i>b</i> as integer words. The lower-order 32 bits of the primary element of <i>a</i> are stored as the first integer word in <i>address(b)</i> . The lower-order 32 bits of the secondary element of <i>a</i> are stored as the next integer word at location <i>address(b)+4</i> . This function is typically preceded by a call to the <code>__fpctiw</code> or <code>__fpctiwz</code> built-in functions, described in , “Unary functions” on page 114, which perform parallel conversion of dual floating-point values to integers.
Formula	b[0] = primary(a) b[1] = secondary(a)
C/C++ prototype	void <code>__stfpiw</code> (int * b, double <code>_Complex</code> a);
Fortran description	STOREFP(B, A) where B is of type INTEGER(4) A is of type COMPLEX(8) result is none

8.10.10 Move functions

Table 8-5 lists and explains the parallel move functions that are available.

Table 8-5 Move functions

Function	Cross move: <code>__fxmr</code>
Purpose	Swaps the values of the primary and secondary elements of operand <i>a</i> .
Formula	primary(result) = secondary(a) secondary(result) = primary(a)
C/C++ prototype	double <code>_Complex</code> <code>__fxmr</code> (double <code>_Complex</code> a);
Fortran description	FXMR(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

8.10.11 Arithmetic functions

In the following sections, we describe all the arithmetic built-in functions, categorized by their number of operands.

Unary functions

Unary functions operate on a single input operand. These functions are listed in Table 8-6.

Table 8-6 *Unary functions*

Function	Parallel convert to integer: <code>__fpctiw</code>
Purpose	Converts in parallel the primary and secondary elements of operand <i>a</i> to 32-bit integers using the current rounding mode. After a call to this function, use the <code>__stfpw</code> function to store the converted integers in parallel, as explained in 8.10.9, "Load and store functions" on page 111.
Formula	primary(result) = primary(<i>a</i>) secondary(result) = secondary(<i>a</i>)
C/C++ prototype	<code>double _Complex __fpctiw (double _Complex a);</code>
Fortran purpose	FPCTIW(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel convert to integer and round to zero: <code>__fpctiwz</code>
Purpose	Converts in parallel the primary and secondary elements of operand <i>a</i> to 32 bit integers and rounds the results to zero. After a call to this function, use the <code>__stfpw</code> function to store the converted integers in parallel, as explained in 8.10.9, "Load and store functions" on page 111.
Formula	primary(result) = primary(<i>a</i>) secondary(result) = secondary(<i>a</i>)
C/C++ prototype	<code>double _Complex __fpctiwz(double _Complex a);</code>
Fortran description	FPCTIWZ(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel round double-precision to single-precision: <code>__fprsp</code>
Purpose	Rounds in parallel the primary and secondary elements of double-precision operand <i>a</i> to single precision.
Formula	primary(result) = primary(<i>a</i>) secondary(result) = secondary(<i>a</i>)
C/C++ prototype	<code>double _Complex __fprsp (double _Complex a);</code>
Fortran description	FPRSP(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Function	Parallel reciprocal estimate: __fpre
Purpose	Calculates in parallel double-precision estimates of the reciprocal of the primary and secondary elements of operand <i>a</i> .
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpre(double _Complex a);
Fortran description	FPRE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel reciprocal square root: __fprsqrte
Purpose	Calculates in parallel double-precision estimates of the reciprocals of the square roots of the primary and secondary elements of operand <i>a</i> .
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fprsqrte (double _Complex a);
Fortran description	FPRSQRTE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel negate: __fpneg
Purpose	Calculates in parallel the negative values of the primary and secondary elements of operand <i>a</i> .
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpneg (double _Complex a);
Fortran description	FPNEG(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel absolute: __fpabs
Purpose	Calculates in parallel the absolute values of the primary and secondary elements of operand <i>a</i> .
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpabs (double _Complex a);
Fortran description	FPABS(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Function	Parallel negate absolute: <code>__fpnabs</code>
Purpose	Calculates in parallel the negative absolute values of the primary and secondary elements of operand <i>a</i> .
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpnabs (double _Complex a);
Fortran description	FPNABS(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Binary functions

Binary functions operate on two input operands. The functions are listed in Table 8-7.

Table 8-7 Binary functions

Function	Parallel add: <code>__fpadd</code>
Purpose	Adds in parallel the primary and secondary elements of operands <i>a</i> and <i>b</i> .
Formula	primary(result) = primary(a) + primary(b) secondary(result) = secondary(a) + secondary(b)
C/C++ prototype	double _Complex __fpadd (double _Complex a, double _Complex b);
Fortran description	FPADD(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel subtract: <code>__fpsub</code>
Purpose	Subtracts in parallel the primary and secondary elements of operand <i>b</i> from the corresponding primary and secondary elements of operand <i>a</i> .
Formula	primary(result) = primary(a) - primary(b) secondary(result) = secondary(a) - secondary(b)
C/C++ prototype	double _Complex __fpsub (double _Complex a, double _Complex b);
Fortran description	FPSUB(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel multiply: <code>__fpmul</code>
Purpose	Multiplies in parallel the values of primary and secondary elements of operands <i>a</i> and <i>b</i> .
Formula	primary(result) = primary(a) × primary(b) secondary(result) = secondary(a) × secondary(b)
C/C++ prototype	double _Complex __fpmul (double _Complex a, double _Complex b);

Fortran description	FPMUL(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Cross multiply: __fxmul
Purpose	The product of the secondary element of <i>a</i> and the primary element of <i>b</i> is stored as the primary element of the return value. The product of the primary element of <i>a</i> and the secondary element of <i>b</i> is stored as the secondary element of the return value.
Formula	primary(result) = secondary(a) x primary(b) secondary(result) = primary(a) x secondary(b)
C/C++ prototype	double _Complex __fxmul (double _Complex a, double _Complex b);
Fortran description	FXMUL(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Cross copy multiply: _fxpmul, __fxsmul
Purpose	Both of these functions can be used to achieve the same result. The product of <i>a</i> and the primary element of <i>b</i> is stored as the primary element of the return value. The product of <i>a</i> and the secondary element of <i>b</i> is stored as the secondary element of the return value.
Formula	primary(result) = a x primary(b) secondary(result) = a x secondary(b)
C/C++ prototype	double _Complex __fxpmul (double _Complex b, double a); double _Complex __fxsmul (double _Complex b, double a);
Fortran description	FXPMUL(B,A) or FXSMUL(B,A) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Multiply-add functions

Multiply-add functions take three input operands, multiply the first two, and add or subtract the third. Table 8-8 lists these functions.

Table 8-8 Multiply-add functions

Function	Parallel multiply-add: __fpmadd
Purpose	The sum of the product of the primary elements of <i>a</i> and <i>b</i> , added to the primary element of <i>c</i> , is stored as the primary element of the return value. The sum of the product of the secondary elements of <i>a</i> and <i>b</i> , added to the secondary element of <i>c</i> , is stored as the secondary element of the return value.
Formula	primary(result) = primary(a) x primary(b) + primary(c) secondary(result) = secondary(a) x secondary(b) + secondary(c)
C/C++ prototype	double _Complex __fpmadd (double _Complex c, double _Complex b, double _Complex a);

Fortran description	FPMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel negative multiply-add: __fpmadd
Purpose	The sum of the product of the primary elements of <i>a</i> and <i>b</i> , added to the primary element of <i>c</i> , is negated and stored as the primary element of the return value. The sum of the product of the secondary elements of <i>a</i> and <i>b</i> , added to the secondary element of <i>c</i> , is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(primary(a) × primary(b) + primary(c)) secondary(result) = -(secondary(a) × secondary(b) + secondary(c))
C/C++ prototype	double _Complex __fpmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel multiply-subtract: __fpmsub
Purpose	The difference of the primary element of <i>c</i> , subtracted from the product of the primary elements of <i>a</i> and <i>b</i> , is stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of the secondary elements of <i>a</i> and <i>b</i> , is stored as the secondary element of the return value.
Formula	primary(result) = primary(a) × primary(b) - primary(c) secondary(result) = secondary(a) × secondary(b) - secondary(c)
C/C++ prototype	double _Complex __fpmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Parallel negative multiply-subtract: __fpmnsb
Purpose	The difference of the primary element of <i>c</i> , subtracted from the product of the primary elements of <i>a</i> and <i>b</i> , is negated and stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of the secondary elements of <i>a</i> and <i>b</i> , is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(primary(a) × primary(b) - primary(c)) secondary(result) = -(secondary(a) × secondary(b) - secondary(c))
C/C++ prototype	double _Complex __fpmnsb (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Function	Cross multiply-add: <code>__fxmadd</code>
Purpose	The sum of the product of the primary element of <i>a</i> and the secondary element of <i>b</i> , added to the primary element of <i>c</i> , is stored as the primary element of the return value. The sum of the product of the secondary element of <i>a</i> and the primary <i>b</i> , added to the secondary element of <i>c</i> , is stored as the secondary element of the return value.
Formula	primary(result) = primary(a) × secondary(b) + primary(c) secondary(result) = secondary(a) × primary(b) + secondary(c)
C/C++ prototype	double _Complex __fxmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Cross negative multiply-add: <code>__fxnmadd</code>
Purpose	The sum of the product of the primary element of <i>a</i> and the secondary element of <i>b</i> , added to the primary element of <i>c</i> , is negated and stored as the primary element of the return value. The sum of the product of the secondary element of <i>a</i> and the primary element of <i>b</i> , added to the secondary element of <i>c</i> , is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(primary(a) × secondary(b) + primary(c)) secondary(result) = -(secondary(a) × primary(b) + secondary(c))
C/C++ prototype	double _Complex __fxnmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Cross multiply-subtract: <code>__fxmsub</code>
Purpose	The difference of the primary element of <i>c</i> , subtracted from the product of the primary element of <i>a</i> and the secondary element of <i>b</i> , is stored as the primary element of the return secondary element of <i>a</i> and the primary element of <i>b</i> is stored as the secondary element of the return value.
Formula	primary(result) = primary(a) × secondary(b) - primary(c) secondary(result) = secondary(a) × primary(b) - secondary(c)
C/C++ prototype	double _Complex __fxmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Function	Cross negative multiply-subtract: <code>__fxnmsub</code>
Purpose	The difference of the primary element of <i>c</i> , subtracted from the product of the primary element of <i>a</i> and the secondary element of <i>b</i> , is negated and stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of the secondary element of <i>a</i> and the primary element of <i>b</i> , is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(primary(a) × secondary(b) - primary(c)) secondary(result) = -(secondary(a) × primary(b) - secondary(c))
C/C++ prototype	double _Complex __fxnmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Cross copy multiply-add: <code>__fxcpmadd</code> , <code>__fxcsmadd</code>
Purpose	Both of these functions can be used to achieve the same result. The sum of the product of <i>a</i> and the primary element of <i>b</i> , added to the primary element of <i>c</i> , is stored as the primary element of the return value. The sum of the product of <i>a</i> and the secondary element of <i>b</i> , added to the secondary element of <i>c</i> , is stored as the secondary element of the return value.
Formula	primary(result) = a × primary(b) + primary(c) secondary(result) = a × secondary(b) + secondary(c)
C/C++ prototype	double _Complex __fxcpmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsmadd (double _Complex c, double _Complex b, double a);
Fortran description	FXCPMADD(C,B,A) or FXCSMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	Cross copy negative multiply-add: <code>__fxcpnmadd</code> , <code>__fxcsnmadd</code>
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of <i>c</i> , subtracted from the product of <i>a</i> and the primary element of <i>b</i> , is negated and stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of <i>a</i> and the secondary element of <i>b</i> , is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(a × primary(b) + primary(c)) secondary(result) = -(a × secondary(b) + secondary(c))
C/C++ prototype	double _Complex __fxcpnmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmadd (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNMADD(C,B,A) or FXCSNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)

Function	Cross copy multiply-subtract: <code>__fxcpmsub</code> , <code>__fxcsmsub</code>
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of <i>c</i> , subtracted from the product of <i>a</i> and the primary element of <i>b</i> , is stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of <i>a</i> and the secondary element of <i>b</i> , is stored as the secondary element of the return value.
Formula	primary(result) = $a \times \text{primary}(b) - \text{primary}(c)$ secondary(result) = $a \times \text{secondary}(b) - \text{secondary}(c)$
C/C++ prototype	double _Complex __fxcpmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsmsub (double _Complex c, double _Complex b, double a);
Fortran description	FXCPMSUB(C,B,A) or FXCSMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	Cross copy negative multiply-subtract: <code>__fxcpnmsub</code> , <code>__fxcsnmsub</code>
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of <i>c</i> , subtracted from the product of <i>a</i> and the primary element of <i>b</i> , is negated and stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of <i>a</i> and the secondary element of <i>b</i> , is negated and stored as the secondary element of the return value.
Formula	primary(result) = $-(a \times \text{primary}(b) - \text{primary}(c))$ secondary(result) = $-(a \times \text{secondary}(b) - \text{secondary}(c))$
C/C++ prototype	double _Complex __fxcpnmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmsub (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNMSUB(C,B,A) or FXCSNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	Cross copy sub-primary multiply-add: <code>__fxcpnpma</code> , <code>__fxcsnpma</code>
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of <i>c</i> , subtracted from the product of <i>a</i> and the primary element of <i>b</i> , is negated and stored as the primary element of the return value. The sum of the product of <i>a</i> and the secondary element of <i>b</i> , added to the secondary element of <i>c</i> , is stored as the secondary element of the return value.
Formula	primary(result) = $-(a \times \text{primary}(b) - \text{primary}(c))$ secondary(result) = $a \times \text{secondary}(b) + \text{secondary}(c)$
C/C++ prototype	double _Complex __fxcpnpma (double _Complex c, double _Complex b, double a); double _Complex __fxcsnpma (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNPMA(C,B,A) or FXCSNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)

Function	Cross copy sub-secondary multiply-add: <code>__fxcpnsma</code> , <code>__fxcsnsma</code>
Purpose	Both of these functions can be used to achieve the same result. The sum of the product of <i>a</i> and the primary element of <i>b</i> , added to the primary element of <i>c</i> , is stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of <i>a</i> and the secondary element of <i>b</i> , is negated and stored as the secondary element of the return value.
Formula	primary(result) = $a \times \text{primary}(b) + \text{primary}(c)$ secondary(result) = $-(a \times \text{secondary}(b) - \text{secondary}(c))$
C/C++ prototype	double _Complex __fxcpnsma (double _Complex c, double _Complex b, double a); double _Complex __fxcsnsma (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNSMA(C,B,A) or FXCSNSMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	Cross mixed multiply-add: <code>__fxcxma</code>
Purpose	The sum of the product of <i>a</i> and the secondary element of <i>b</i> , added to the primary element of <i>c</i> , is stored as the primary element of the return value. The sum of the product of <i>a</i> and the primary element of <i>b</i> , added to the secondary element of <i>c</i> , is stored as the secondary element of the return value.
Formula	primary(result) = $a \times \text{secondary}(b) + \text{primary}(c)$ secondary(result) = $a \times \text{primary}(b) + \text{secondary}(c)$
C/C++ prototype	double _Complex __fxcxma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	Cross mixed negative multiply-subtract: <code>__fxcxnms</code>
Purpose	The difference of the primary element of <i>c</i> , subtracted from the product of <i>a</i> and the secondary element of <i>b</i> , is negated and stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of <i>a</i> and the primary element of <i>b</i> , is negated and stored as the primary secondary of the return value.
Formula	primary(result) = $-(a \times \text{secondary}(b) - \text{primary}(c))$ secondary(result) = $-(a \times \text{primary}(b) - \text{secondary}(c))$
C/C++ prototype	double _Complex __fxcxnms (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNMS(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)

Function	Cross mixed sub-primary multiply-add: <code>__fxcxnpma</code>
Purpose	The difference of the primary element of <i>c</i> , subtracted from the product of <i>a</i> and the secondary element of <i>b</i> , is stored as the primary element of the return value. The sum of the product of <i>a</i> and the primary element of <i>b</i> , added to the secondary element of <i>c</i> , is stored as the secondary element of the return value.
Formula	primary(result) = -(a × secondary(b) - primary(c)) secondary(result) = a × primary(b) + secondary(c)
C/C++ prototype	double _Complex __fxcxnpma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	Cross mixed sub-secondary multiply-add: <code>__fxcxnsma</code>
Purpose	The sum of the product of <i>a</i> and the secondary element of <i>b</i> , added to the primary element of <i>c</i> , is stored as the primary element of the return value. The difference of the secondary element of <i>c</i> , subtracted from the product of <i>a</i> and the primary element of <i>b</i> , is stored as the secondary element of the return value.
Formula	primary(result) = a × secondary(b) + primary(c) secondary(result) = -(a × primary(b) - secondary(c))
C/C++ prototype	double _Complex __fxcxnsma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNSMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)

8.10.12 Select functions

Table 8-9 lists and explains the parallel select functions that are available.

Table 8-9 Select functions

Function	Parallel select: <code>__fpsel</code>
Purpose	The value of the primary element of <i>a</i> is compared to zero. If its value is equal to or greater than zero, the primary element of <i>c</i> is stored in the primary element of the return value. Otherwise the primary element of <i>b</i> is stored in the primary element of the return value. The value of the secondary element of <i>a</i> is compared to zero. If its value is equal to or greater than zero, the secondary element of <i>c</i> is stored in the secondary element of the return value. Otherwise, the secondary element of <i>b</i> is stored in the secondary element of the return value.
Formula	primary(result) = if primary(a) ≥ 0 then primary(c); else primary(b) secondary(result) = if secondary(a) ≥ 0 then primary(c); else secondary(b)
C/C++ prototype	double _Complex __fpsel (double _Complex a, double _Complex b, double _Complex c);

Function	Parallel select: <code>__fpisel</code>
Fortran description	FPSEL(A,B,C) where A is of type COMPLEX(8) where B is of type COMPLEX(8) where C is of type COMPLEX(8) result is of type COMPLEX(8)

8.10.13 Examples of built-in functions usage

Using the following definitions, you can create a custom parallel add function that uses the parallel load and add built-in functions to add two double floating-point values in parallel and return the result as a complex number. See Example 8-10 for C/C++ and Example 8-11 for Fortran.

Example 8-10 Using built-in functions in C/C++

```
double _Complex padd(double *x, double *y)
{
double _Complex a,b,c;
/* note possibility of alignment trap if (((unsigned int) x) % 32) >= 17) */

a = __lfpd(x); //load x[0] to the primary part of a, x[1] to the secondary part of a
b = __lfpd(y); //load y[0] to primary part of b, y[1] to the secondary part of b
c = __fpadd(a,b); // the primary part of c = x[0] + y[0]
/* the secondary part of c = x[1] + y[1] */
return c;

/* alternately: */
return __fpadd(__lfpd(x), __lfpd(y)); /* same code generated with optimization
enabled */
}
```

Example 8-11 Using built-in functions in Fortran

```
FUNCTION PADD (X, Y)
COMPLEX(8) PADD
REAL(8) X, Y
COMPLEX(8) A, B, C

A = LOADFP(X)
B = LOADFP(Y)
PADD = FPADD(A,B)

RETURN
END
```

Example 8-12 Double precision square matrix multiply example

```
subroutine dsqmm(a, b, c, n)
!
!# (C) Copyright IBM Corp. 2006 All Rights Reserved.
!# Rochester, MN
!
  implicit none
  integer i, j, k, n
  integer ii, jj, kk
  integer istop, jstop, kstop
  integer, parameter :: nb = 36  ! blocking factor
  complex(8) zero
  complex(8) a00, a01
  complex(8) a20, a21
  complex(8) b0, b1, b2, b3, b4, b5
  complex(8) c00, c01, c02, c03, c04, c05
  complex(8) c20, c21, c22, c23, c24, c25
  real(8) a(n,n), b(n,n), c(n,n)

  zero = (0.0d0, 0.0d0)

  !-----
  ! Double-precision square matrix-matrix multiplication.
  !-----
  ! This version uses 6x4 outer loop unrolling.
  ! The cleanup loops have been left out, so the results
  ! are correct for dimensions that are multiples of the
  ! two unrolling factors: 6 and 4.
  !-----

  do jj = 1, n, nb

    if ((jj + nb - 1) .lt. n) then
      jstop = (jj + nb - 1)
    else
      jstop = n
    endif

    do ii = 1, n, nb

      if ((ii + nb - 1) .lt. n) then
        istop = (ii + nb - 1)
      else
        istop = n
      endif

      !-----
      ! initialize a block of c to zero
      !-----
      do j = jj, jstop - 5, 6
        do i = ii, istop - 1, 2
          call storefp(c(i,j) , zero)
          call storefp(c(i,j+1), zero)
          call storefp(c(i,j+2), zero)
          call storefp(c(i,j+3), zero)
          call storefp(c(i,j+4), zero)
          call storefp(c(i,j+5), zero)
        end do
      end do
    end do
  end do
```

```

!-----
! multiply block by block with 6x4 outer loop un-rolling
!-----
do kk = 1, n, nb
  if ((kk + nb - 1) .lt. n) then
    kstop = (kk + nb - 1)
  else
    kstop = n
  endif

  do j = jj, jstop - 5, 6
    do i = ii, istop - 3, 4

      c00 = loadfp(c(i,j ))
      c01 = loadfp(c(i,j+1))
      c02 = loadfp(c(i,j+2))
      c03 = loadfp(c(i,j+3))
      c04 = loadfp(c(i,j+4))
      c05 = loadfp(c(i,j+5))

      c20 = loadfp(c(i+2,j ))
      c21 = loadfp(c(i+2,j+1))
      c22 = loadfp(c(i+2,j+2))
      c23 = loadfp(c(i+2,j+3))
      c24 = loadfp(c(i+2,j+4))
      c25 = loadfp(c(i+2,j+5))

      a00 = loadfp(a(i,kk ))
      a20 = loadfp(a(i+2,kk ))
      a01 = loadfp(a(i,kk+1))
      a21 = loadfp(a(i+2,kk+1))

      do k = kk, kstop - 1, 2
        b0 = loadfp(b(k,j ))
        b1 = loadfp(b(k,j+1))
        b2 = loadfp(b(k,j+2))
        b3 = loadfp(b(k,j+3))
        b4 = loadfp(b(k,j+4))
        b5 = loadfp(b(k,j+5))
        c00 = fxcpmadd(c00, a00, real(b0))
        c01 = fxcpmadd(c01, a00, real(b1))
        c02 = fxcpmadd(c02, a00, real(b2))
        c03 = fxcpmadd(c03, a00, real(b3))
        c04 = fxcpmadd(c04, a00, real(b4))
        c05 = fxcpmadd(c05, a00, real(b5))
        c20 = fxcpmadd(c20, a20, real(b0))
        c21 = fxcpmadd(c21, a20, real(b1))
        c22 = fxcpmadd(c22, a20, real(b2))
        c23 = fxcpmadd(c23, a20, real(b3))
        c24 = fxcpmadd(c24, a20, real(b4))
        c25 = fxcpmadd(c25, a20, real(b5))
        a00 = loadfp(a(i,k+2 ))
        a20 = loadfp(a(i+2,k+2 ))
        c00 = fxcpmadd(c00, a01, imag(b0))
        c01 = fxcpmadd(c01, a01, imag(b1))
        c02 = fxcpmadd(c02, a01, imag(b2))
        c03 = fxcpmadd(c03, a01, imag(b3))
        c04 = fxcpmadd(c04, a01, imag(b4))
        c05 = fxcpmadd(c05, a01, imag(b5))
        c20 = fxcpmadd(c20, a21, imag(b0))
      enddo
    enddo
  enddo

```

```

        c21 = fxcpmadd(c21, a21, imag(b1))
        c22 = fxcpmadd(c22, a21, imag(b2))
        c23 = fxcpmadd(c23, a21, imag(b3))
        c24 = fxcpmadd(c24, a21, imag(b4))
        c25 = fxcpmadd(c25, a21, imag(b5))
        a01 = loadfp(a(i,k+3))
        a21 = loadfp(a(i+2,k+3))
    end do

    call storefp(c(i ,j ), c00)
    call storefp(c(i ,j+1), c01)
    call storefp(c(i ,j+2), c02)
    call storefp(c(i ,j+3), c03)
    call storefp(c(i ,j+4), c04)
    call storefp(c(i ,j+5), c05)

    call storefp(c(i+2,j ), c20)
    call storefp(c(i+2,j+1), c21)
    call storefp(c(i+2,j+2), c22)
    call storefp(c(i+2,j+3), c23)
    call storefp(c(i+2,j+4), c24)
    call storefp(c(i+2,j+5), c25)

    end do
    end do

    end do !kk

    end do !ii

    end do !jj

end

```



Running and debugging applications

In this chapter, we explain how to run and debug applications on the Blue Gene/P system. These types of tools are essential for applications developers. Although, we do not cover all of the existing tools, we provide an overview of some of the currently available tools.

We cover the following topics:

- ▶ “Running applications” on page 130
- ▶ “Debugging applications” on page 132

9.1 Running applications

There are several ways to run Blue Gene/P applications. We briefly discuss each method and provide references for more detailed documentation.

9.1.1 MMCS console

It is possible to run applications directly from the MMCS console. The main drawback to using this approach is that it requires users to have direct access to the Service Node, which is undesirable from a security perspective.

When using the MMCS console, it is necessary to first manually select and allocate a block. A *block* in this case refers to a partition or set of nodes to run the job. (See Appendix A, “Blue Gene/P hardware naming convention” on page 265, for more information.) At this point, it is possible to run Blue Gene/P applications. The set of commands in Example 9-1 from the MMCS console window show how to accomplish this. The names can be site specific, but it illustrates the procedure.

To start the console session, use the sequence of commands shown in Example 9-1 on the Service Node.

Example 9-1 Starting the console session

```
cd /bgsys/drivers/ppcfloor/bin
source ~/bgpsysdb/sqllib/db2profile
mmcs_db_console --bgpadminingroup p/bluegene/bgpa1
connecting to mmcs_server
connected to mmcs_server
connected to DB2
mmcs$list_blocks
OK
N00_64_1      B manojd (1)  connected
N02_32_1      I walkup (0)  connected
N04_32_1      B manojd (1)  connected
N05_32_1      B manojd (1)  connected
N06_32_1      I sameer77(1) connected
N07_32_1      I gdozsa (1)  connected
N08_64_1      I vezolle (1)  connected
N12_32_1      I vezolle (0)  connected
mmcs$ allocate N14_32_1
OK
mmcs$ list_blocks
OK
N00_64_1      B manojd (1)  connected
N02_32_1      I walkup (0)  connected
N04_32_1      B manojd (1)  connected
N05_32_1      B manojd (1)  connected
N06_32_1      I sameer77(1) connected
N07_32_1      I gdozsa (1)  connected
N08_64_1      I vezolle (1)  connected
N12_32_1      I vezolle (0)  connected
N14_32_1      I cpsosa (1)  connected
mmcs$ submitjob N14_32_1 /bgusr/cpsosa/hello/c/omp_hello_bgp /bgusr/cpsosa/hello/c
OK
jobId=14008
mmcs$ free N14_32_1
```

```
OK
mmcs$ quit
OK
mmcs_db_console is terminating, please wait...
mmcs_db_console: closing database connection
mmcs_db_console: closed database connection
mmcs_db_console: closing console port
mmcs_db_console: closed console port
```

For more information about using the MMCS console, see *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417.

9.1.2 mpirun

In the absence of a scheduling application, we recommend that you use **mpirun** to run Blue Gene/P applications. Users can access this application from the Front End Node, which provides better security protection than using the MMCS console. For more complete information about using **mpirun**, see Chapter 13, “mpirun” on page 217.

With **mpirun**, you can select and allocate a block and run a Message Passing Interface (MPI) application, all in one step as shown in Example 9-2.

Example 9-2 Using mpirun

```
cpsosa@cartes:/bgusr/cpsosa/red/pi/c> csh
cartes pi/c> set MPIRUN="/bgusr/drivers/ppcfloor/bin/mpirun"
cartes pi/c> set MPIOPT="-np 1"
cartes pi/c> set MODE="-mode SMP"
cartes pi/c> set PARTITION="-partition N14_32_1"
cartes pi/c> set WDIR="-cwd /bgusr/cpsosa/red/pi/c"
cartes pi/c> set EXE="-exe /bgusr/cpsosa/red/pi/c/pi_critical_bgp"
cartes pi/c> $MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=1"
Estimate of pi: 3.14159
Total time 560.055988
```

All output in this example is sent to the screen. In order for this information to be sent to a file, you must add the following line, for example, to the end of the **mpirun** command:

```
>/bgusr/cpsosa/red/pi/c/pi_critical.stdout 2>/bgusr/cpsosa/red/pi/c/pi_critical.stderr
```

This line sends standard output to a file called *pi_critical.stdout* and standard error to a file called *pi_critical.stderr*. Both files are in the */bgusr/cpsosa/red/pi/c* directory.

9.1.3 LoadLeveler

At present, LoadLeveler support for the Blue Gene/P system is provided via a PRPQ. The IBM Tivoli® Workload Scheduler LoadLeveler product is intended to manage both serial and parallel jobs over a cluster of servers. This distributed environment consists of a pool of machines or servers, often referred to as a *LoadLeveler cluster*. Machines in the pool can be of several types: desktop workstations available for batch jobs (usually when not in use by their owner), dedicated servers, and parallel machines.

LoadLeveler allocates machine resources in the cluster to run jobs. The scheduling of jobs depends on the availability of resources within the cluster and various rules, which can be defined by the LoadLeveler administrator. A user submits a job using a job command file. The LoadLeveler scheduler attempts to find resources within the cluster to satisfy the requirements of the job. LoadLeveler maximizes the efficiency of the cluster by maximizing

the utilization of resources, while at the same time minimizing the job turnaround time experienced by users.

LoadLeveler provides a rich set of functions for job scheduling and cluster resource management. Some of the tasks that LoadLeveler can perform include:

- ▶ Choosing the next job to run.
- ▶ Examining the job requirements.
- ▶ Collecting available resources in the cluster.
- ▶ Choosing the “best” machines for the job.
- ▶ Dispatching the job to the selected machine.
- ▶ Controlling running jobs.
- ▶ Creating reservations and scheduling jobs to run in the reservations.
- ▶ Job preemption to enable high priority jobs to run immediately.
- ▶ Fair share scheduling to automatically balance resources among users or groups of users.
- ▶ Co-scheduling to enable several jobs to be scheduled to run at the same time.
- ▶ Multi-cluster support to allow several LoadLeveler clusters to work together to run user jobs.

For more information about LoadLeveler support, see Chapter 10 of *IBM System Blue Gene Solution: Configuring and Maintaining Your Environment*, SG24-7352, which describes step-by-step how to use LoadLeveler on the Blue Gene/L system. Almost all of the contents are still applicable to a Blue Gene/P system.

The LoadLeveler installation procedure is slightly different for the Blue Gene/P system. New functions are provided for both the Blue Gene/P and Blue Gene/L systems at the same time that basic LoadLeveler support for Blue Gene/P is provided.

9.1.4 Other scheduler products

You can use custom scheduling applications to run applications on the Blue Gene/P system. You write custom “glue” code between the scheduler and the Blue Gene/P system by using the Bridge application programming interfaces (APIs), which are described in Chapter 11, “Control system (Bridge) APIs” on page 159, and Chapter 12, “Real-time Notification APIs” on page 197.

9.2 Debugging applications

In this section, we discuss the debuggers that are supported by the Blue Gene/P system.

9.2.1 General debugging architecture

Four pieces of code are involved when debugging applications on the Blue Gene/P system:

- ▶ The Compute Node Kernel, which provides the low-level primitives that are necessary to debug an application
- ▶ The control and I/O daemon (CIOD) running on the I/O Nodes, which provides control and communications to Compute Nodes

- ▶ A “debug server” running on the I/O Nodes, which is vendor-supplied code that interfaces with the CIOD
- ▶ A debug client running on a Front End Node, which is where the user does their work interactively

A debugger must interface to the Compute Node through an API implemented in CIOD in order to debug an application running on a Compute Node. This debug code is started on the I/O Nodes by the control system and can interface with other software, such as a GUI or command line utility on a front-end system. The code running on the I/O Nodes using the API in CIOD is referred to as a *debug server*. It is provided by the debugger vendor for use with the Blue Gene/P system. Many possible debug servers are possible.

A *debug client* is a piece of code that runs on a Front End Node that the user interacts with directly. It makes remote requests to the debug server running on the I/O Nodes, which in turn passes the request through CIOD and eventually to the Compute Node. The debug client and debug server usually communicate using TCP/IP.

9.2.2 GNU Project debugger

The GNU Project debugger (GDB) is the primary debugger of the GNU project. You can learn more about GDB on the Web at the following address:

<http://www.gnu.org/software/gdb/gdb.html>

A great amount of documentation is available about the GDB. Since we do not discuss how to use it in this book, refer to the following Web site for details:

<http://www.gnu.org/software/gdb/documentation/>

Support has been added to the Blue Gene/P system for which the GDB can work with applications that run on Compute Nodes. IBM provides a simple debug server called *gdbserver*. Each running instance of GDB is associated with one, and only one, Compute Node. If you must debug an MPI application that runs on multiple Compute Nodes, and you must, for example, view variables that are associated with more than one instance of the application, you run multiple instances of GDB.

Most people use GDB to debug local processes that run on the same machine on which they are running GDB. With GDB, you also have the ability to do remote debug via a GDB server on the remote machine. GDB on the Blue Gene/L system is used in this mode. We refer to GDB as the “GDB client,” although most users recognize it as GDB used in a slightly different manner.

Limitations

Gdbserver implements the minimum number of primitives required by the GDB remote protocol specification. As such, advanced features that might be available in other implementations are not available in this implementation. However, enough is implemented to make it a useful tool. Here are some of the limitations:

- ▶ Each instance of a GDB client can connect to and debug one Compute Node. To debug multiple Compute Nodes at the same time, you must run multiple GDB clients at the same time. Although you might need multiple GDB clients for multiple Compute Nodes, one *gdbserver* on each I/O Node is all that is required. The Blue Gene/P control system manages that part.
- ▶ IBM does not ship a GDB client with the Blue Gene/P system. The user can use an existing GDB client to connect to the IBM-supplied *gdbserver*. Most functions will work, but standard GDB clients are not aware of the full “double hummer” floating point register set

that Blue Gene/L provides. The GDB clients that come with SUSE Linux Enterprise Server (SLES) 10 for PowerPC are known to work.

- ▶ To debug an application, the debug server must be started and running before you attempt to debug. Using an option on the `mpirun` command, you can get the debug server running before your application does. If you do not use this option and you must debug your application, you do not have a mechanism to start the debug server and thus have no way to debug your application.
- ▶ Gdbserver is not aware of user-specified MPI topologies. You can still debug your application, but the connection information given to you by `mpirun` for each MPI rank can be incorrect.

Prerequisite software

The GDB should have been installed during the installation procedure. You can verify the installation by seeing if the `/bgsys/drivers/ppcfloor/gnu-linux/bin/gdb` file exists on your Front End Node.

The rest of the software support required for GDB should be installed as part of the control programs.

Preparing your program

The MPI, OpenMP, or MPI-OpenMP program that you want to debug must be compiled in a manner that allows for debugging information (symbol tables, ties to source, and so on) to be included in the executable. In addition, do *not* use compiler optimization because it makes it difficult, if not impossible, to tie object code back to source. For example, when compiling a program written in Fortran that you want to debug, compile the application using an invocation similar to one shown in Example 9-3.

Example 9-3 Makefile used for building the program with debugging flags

```
BGP_FLOOR = /bgsys/drivers/ppcfloor
BGP_IDIRS = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS  = -L$(BGP_FLOOR)/comm/lib -lmpich.cnk -L$(BGP_FLOOR)/comm/lib -ldcmfcoll.cnk
-lldcmf.cnk -lpthread -lrt -L$(BGP_FLOOR)/runtime/SPI -lSPI.cna

XL        = /opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90

EXE       = example_9_4_bgp
OBJ       = example_9_4.o
SRC       = example_9_4.f
FLAGS    = -g -O0 -qarch=450 -qtune=450 -I$(BGP_FLOOR)/comm/include
FLD      = -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
    ${XL} $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC)
    ${XL} $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
    rm *.o example_9_4_bgp

cpsosa@descartes:/bgusr/cpsosa/red/debug> make
```

```
/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90 -g -O0 -qarch=450 -qtune=450
-I/bgsys/drivers/ppcfloor/comm/include -I/bgsys/drivers/ppcfloor/arch/include
-I/bgsys/drivers/ppcfloor/comm/include -c example_9_4.f
** nooffset === End of Compilation 1 ===
1501-510 Compilation successful for file example_9_4.f.
/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90 -g -O0 -qarch=450 -qtune=450
-I/bgsys/drivers/ppcfloor/comm/include -o example_9_4_bgp example_9_4.o
-L/bgsys/drivers/ppcfloor/comm/lib -lmpich.cnk -L/bgsys/drivers/ppcfloor/comm/lib -ldcmfcoll.cnk
-ldcmf.cnk -lpthread -lrt -L/bgsys/drivers/ppcfloor/runtime/SPI -lSPI.cna
```

The `-g` switch tells the compiler to include debug information. The `-O0` (the letter capital “O” followed by a zero) switch tells it to disable optimization.

For more information about the IBM XL compilers for the Blue Gene/P system, see Chapter 8, “Developing applications with IBM XL compilers” on page 91.

Important: Make sure that the text file that contains the source for your program is located in the same directory as the program itself and has the same file name (different extension).

Debugging

Follow the steps in this section to start debugging your application. For the sake of this example, let us say that the program’s name is `example_9_4_bgp` as illustrated in Example 9-4 on page 136 (source code not shown), and the source code file is `example_9_4.f`. We use a partition (block) called `N14_32_1`.

An extra parameter (`-start_gdbserver...`) is passed in on the `mpirun` command. The extra option changes the way `mpirun` loads and executes your code. Here is a brief summary of the changes:

1. The code is loaded onto the Compute Nodes (in our example, the executable is `example_9_4_bgp`), but it does not start running immediately.
2. The control system starts the specified debug server (`gdbserver`) on all of the I/O Nodes in the partition that is running your job, which in our example is `N14_32_1`.
3. The `mpirun` command pauses, so that you a chance to connect GDB clients to the Compute Nodes that you are going to debug.
4. When you are done connecting GDB clients to Compute Nodes, you press Enter to signal the `mpirun` command, and then the application starts running on the Compute Nodes.

During the pause in step 3, you have an opportunity to connect the GDB clients to the Compute Nodes before the application runs, which is desirable if you must start the application under debugger control. This step is optional. If you do not connect before the application starts running on the Compute Nodes, you can still connect later because the debugger server was started on the I/O Nodes.

To start debugging your application:

1. Open two separate console shells.
2. Go to the first shell window.
 - a. Change to the directory (**cd**) that contains your program executable. In our example, the directory is `/bgusr/cpsosa/red/debug`.
 - b. Start your application using **mpirun** with a command similar to the one shown in Example 9-4. You should see messages in the console, like those shown in Example 9-4.

Example 9-4 Messages in the console

```
set MPIRUN="/bgsys/drivers/ppcfloor/bin/mpirun"
set MPIOPT="-np 1"
set MODE="-mode SMP"
set PARTITION="-partition N14_32_1"
set WDIR="-cwd /bgusr/cpsosa/red/debug"
set EXE="-exe /bgusr/cpsosa/red/debug/example_9_4_bgp"
#
$MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=4" -start_gdbserver
/bgsys/drivers/ppcfloor/ramdisk/sbin/gdbserver -verbose 1
#
echo "That's all folks!!"

descartes red/debug> set EXE="-exe /bgusr/cpsosa/red/debug/example_9_4_bgp"
descartes red/debug> $MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=4"
-start_gdbserver /bgsys/drivers/ppcfloor/ramdisk/sbin/gdbserver -verbose 1
<Sep 15 10:14:58.642369> FE_MPI (Info) : Invoking mpirun backend
<Sep 15 10:14:05.741121> BRIDGE (Info) : rm_set_serial() - The machine serial number (alias) is
BGP
<Sep 15 10:15:00.461655> FE_MPI (Info) : Preparing partition
<Sep 15 10:14:05.821585> BE_MPI (Info) : Examining specified partition
<Sep 15 10:14:10.085997> BE_MPI (Info) : Checking partition N14_32_1 initial state ...
<Sep 15 10:14:10.086041> BE_MPI (Info) : Partition N14_32_1 initial state = READY ('I')
<Sep 15 10:14:10.086059> BE_MPI (Info) : Checking partition owner...
<Sep 15 10:14:10.086087> BE_MPI (Info) : partition N14_32_1 owner is 'cpsosa'
<Sep 15 10:14:10.088375> BE_MPI (Info) : Partition owner matches the current user
<Sep 15 10:14:10.088470> BE_MPI (Info) : Done preparing partition
<Sep 15 10:15:04.804078> FE_MPI (Info) : Adding job
<Sep 15 10:14:10.127380> BE_MPI (Info) : Adding job to database...
<Sep 15 10:15:06.104035> FE_MPI (Info) : Job added with the following id: 14035
<Sep 15 10:15:06.104096> FE_MPI (Info) : Loading Blue Gene job
<Sep 15 10:14:11.426987> BE_MPI (Info) : Loading job 14035 ...
<Sep 15 10:14:11.450495> BE_MPI (Info) : Job load command successful
<Sep 15 10:14:11.450525> BE_MPI (Info) : Waiting for job 14035 to get to Loaded/Running state
...
<Sep 15 10:14:16.458474> BE_MPI (Info) : Job 14035 switched to state LOADED
<Sep 15 10:14:21.467401> BE_MPI (Info) : Job loaded successfully
<Sep 15 10:15:16.179023> FE_MPI (Info) : Starting debugger setup for job 14035
<Sep 15 10:15:16.179090> FE_MPI (Info) : Setting debug info in the block record
<Sep 15 10:14:21.502593> BE_MPI (Info) : Setting debugger executable and arguments in block
description
<Sep 15 10:14:21.523480> BE_MPI (Info) : Debug info set successfully
<Sep 15 10:15:16.246415> FE_MPI (Info) : Query job 14035 to find MPI ranks for compute nodes
<Sep 15 10:15:16.246445> FE_MPI (Info) : Getting process table information for the debugger
```



```

<Sep 15 10:14:22.661841> BE_MPI (Info) : Query job completed - proctable is filled in
<Sep 15 10:15:17.386617> FE_MPI (Info) : Starting debugger servers on I/O nodes for job 14035
<Sep 15 10:15:17.386663> FE_MPI (Info) : Attaching debugger to a new job.
<Sep 15 10:14:22.721982> BE_MPI (Info) : Debugger servers are now spawning
<Sep 15 10:15:17.446486> FE_MPI (Info) : Notifying debugger that servers have been spawned.

```

Make your connections to the compute nodes now - press [Enter] when you are ready to run the app. To see the IP connection information for a specific compute node, enter its MPI rank and press [Enter]. To see all of the compute nodes, type 'dump_proctable'.

```

>
<Sep 15 10:17:20.754179> FE_MPI (Info) : Debug setup is complete
<Sep 15 10:17:20.754291> FE_MPI (Info) : Waiting for Blue Gene job to get to Loaded state
<Sep 15 10:16:26.118529> BE_MPI (Info) : Waiting for job 14035 to get to Loaded/Running state
...
<Sep 15 10:16:31.128079> BE_MPI (Info) : Job loaded successfully
<Sep 15 10:17:25.806882> FE_MPI (Info) : Beginning job 14035
<Sep 15 10:16:31.129878> BE_MPI (Info) : Beginning job 14035 ...
<Sep 15 10:16:31.152525> BE_MPI (Info) : Job begin command successful
<Sep 15 10:17:25.871476> FE_MPI (Info) : Waiting for job to terminate
<Sep 15 10:16:31.231304> BE_MPI (Info) : IO - Threads initialized
<Sep 15 10:27:31.301600> BE_MPI (Info) : I/O output runner thread terminated
<Sep 15 10:27:31.301639> BE_MPI (Info) : I/O input runner thread terminated
<Sep 15 10:27:31.355816> BE_MPI (Info) : Job 14035 switched to state TERMINATED ('T')
<Sep 15 10:27:31.355848> BE_MPI (Info) : Job successfully terminated - TERMINATED ('T')
<Sep 15 10:28:26.113983> FE_MPI (Info) : Job terminated normally
<Sep 15 10:28:26.114057> FE_MPI (Info) : exit status = (0)
<Sep 15 10:27:31.435578> BE_MPI (Info) : Starting cleanup sequence
<Sep 15 10:27:31.435615> BE_MPI (Info) : cleanupDatabase() - job already terminated / hasn't
been added
<Sep 15 10:27:31.469474> BE_MPI (Info) : cleanupDatabase() - Partition was supplied with READY
('I') initial state
<Sep 15 10:27:31.469504> BE_MPI (Info) : cleanupDatabase() - No need to destroy the partition
<Sep 15 10:28:26.483855> FE_MPI (Info) : == FE completed ==
<Sep 15 10:28:26.483921> FE_MPI (Info) : == Exit status: 0 ==

```

c. Find the IP address and port of the Compute Node that you want to debug. You can do this using either of the following ways:

- Enter the rank of the program instance that you want to debug and press Enter.
- Dump the address or port of each node by typing `dump_proctable` and press Enter.

See Example 9-5.

Example 9-5 Finding the IP address and port of the Compute Node for debugging

```

> 2
MPI Rank 2: Connect to 172.30.255.85:7302
> 4
MPI Rank 4: Connect to 172.30.255.85:7304
>
or
> dump_proctable
MPI Rank 0: Connect to 172.24.101.128:7310
>

```

3. From the second shell, follow these steps:
 - a. Change to the directory (`cd`) that contains your program executable.
 - b. Type the following command, using the name of your own executable instead of `example_9_4_bgp`:

```
/bgsys/drivers/ppcfloor/gnu-linux/bin/gdb example_9_4_bgp
```
 - c. Enter the following command, using the address of the Compute Node that you want to debug and determined in step 2c:

```
target remote ipaddr:port
```

You are now debugging the specified application on the configured Compute Node.
4. Set one or more breakpoints (using the GDB `break` command). Press Enter from the first shell to continue that application.

If successful, your breakpoint should eventually be reached in the second shell and you can use standard GDB commands to continue.

9.2.3 Core Processor debugger

Core Processor is a basic tool that can help you debug your application. This tool is discussed in detail in *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417. In the following sections, we briefly describe how to use it to debug applications.

9.2.4 Starting the Core Processor tool

To start the Core Processor tool:

1. Export `DISPLAY` and make sure it works.
2. Type `coreprocessor.pl` to specify the Core Processor tool. You might need to specify the full path.
3. From the GUI window that opens, click **OK**. The Perl script is invoked automatically.

Figure 9-1 shows how the Core Processor tool GUI looks after the Perl script is invoked. The Core Processor windows do not provide any initial information. You must explicitly select a task that is provided via the GUI.

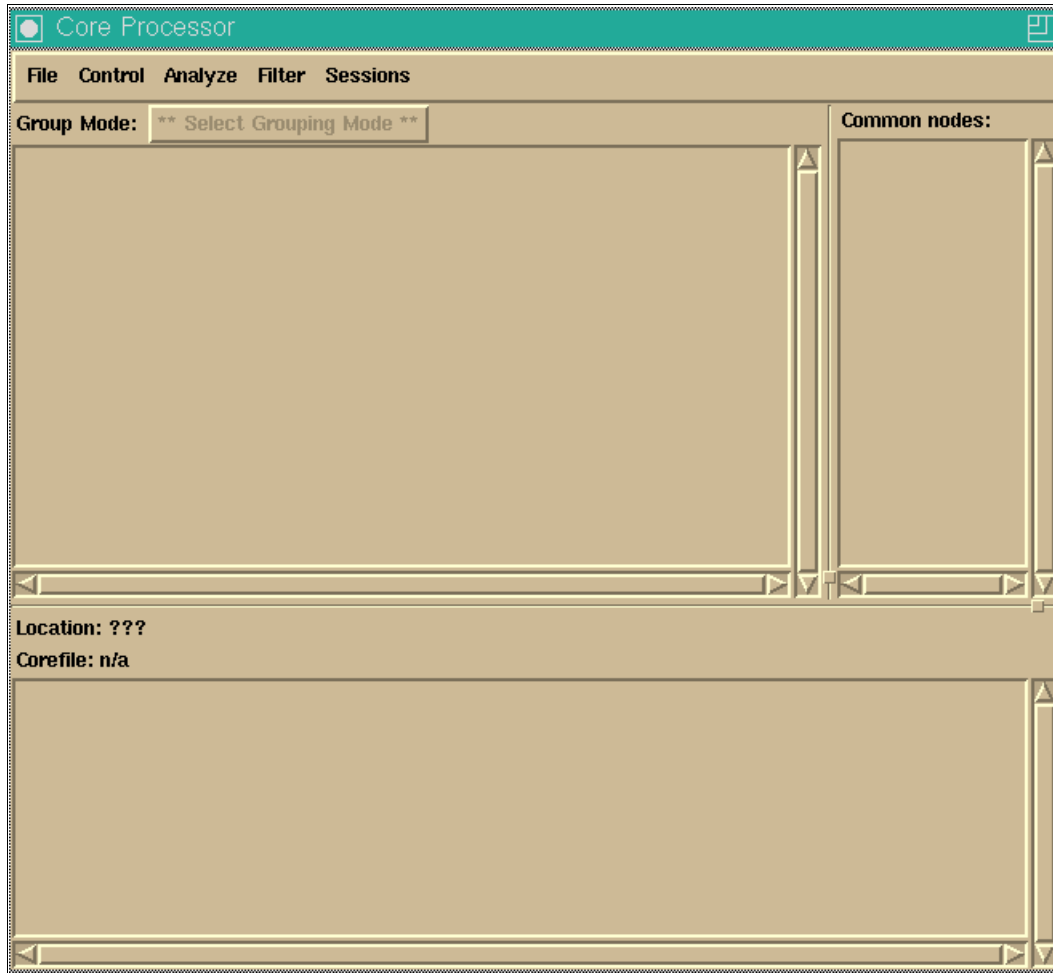


Figure 9-1 Core Processor initial window

9.2.5 Attaching running applications

To do a live debug on Compute Nodes:

1. Start the Core Processor GUI as explained in the previous section.
2. Select **File** → **Attach To Block**.

3. In the Attach Coreprocessor window (Figure 9-2 on page 140), supply the following information:
 - Session Name: You can run more than one session at a time, so use this option to distinguish between multiple sessions.
 - Block name
 - CNK binary (with path): To see both your application and the Compute Node Kernel in the stack, specify your application binary and the Compute Node Kernel image separated by a colon (:) as shown in the following example:


```
/bgsys/drivers/ppcfloor/cnk/bgp_kernel.cn:/bguser/bguser/hello_mpi_loop.rts
```
 - User name or owner of the Midplane Management Control System (MMCS) block
 - Port: TCP port on which the MMCS server is listening for console connections, which is probably 32031.
 - Host name or TCP/IP address for the MMCS server: Typically this is localhost or the Service Node's TCP/IP address.

Click the **Attach** button.

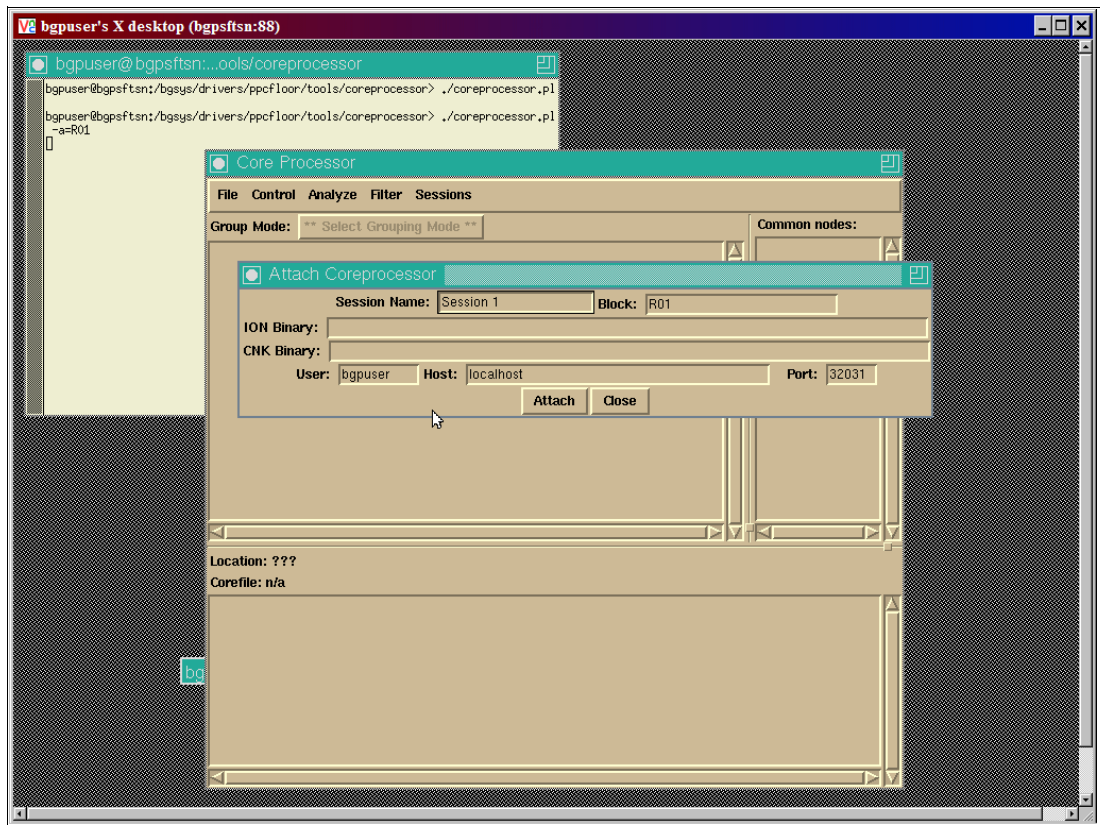


Figure 9-2 Core Processor Attach window

4. At this point, you have not yet affected the state of the processors. Choose **Select Grouping Mode** → **Processor Status**.

Notice the text in the upper left pane (Figure 9-3). The Core Processor tool posts the status ?RUN? because it does not yet know the state of the processors. (2048) is the number of nodes in the block that are in that state. The number in parentheses always indicates the number of nodes that share the attribute that is displayed on the line, which is the processor state in this case.

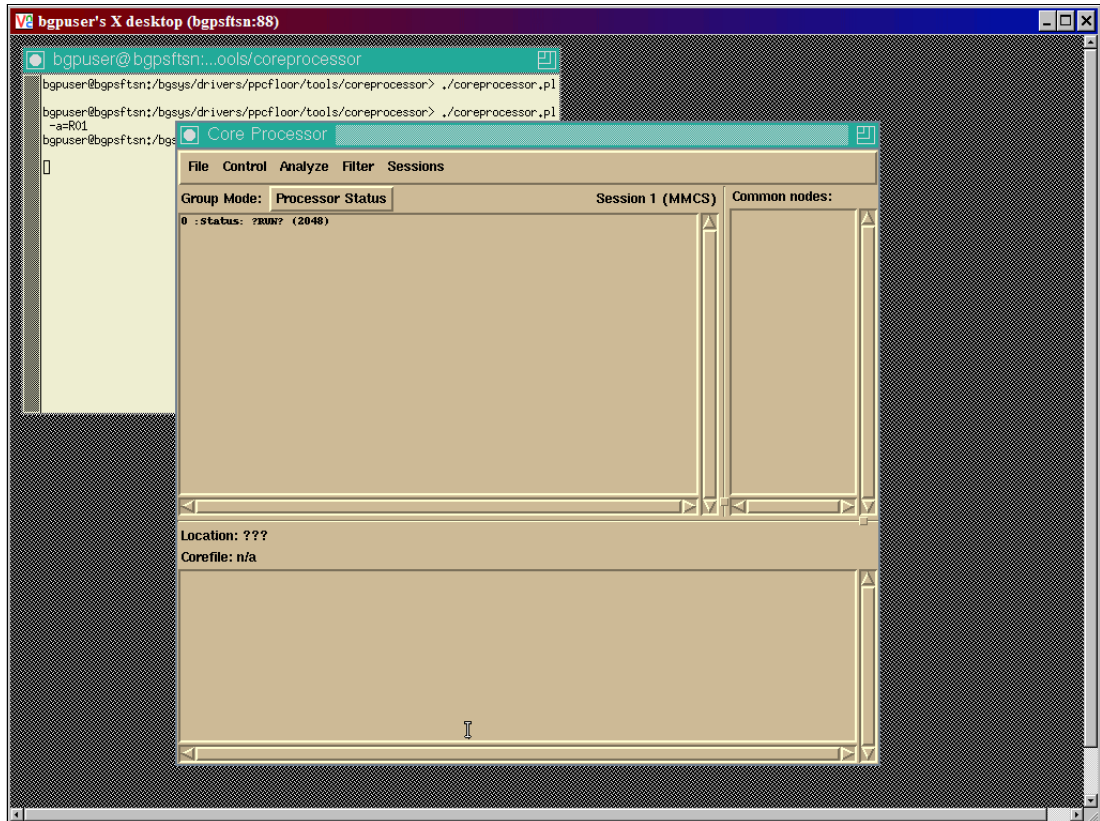


Figure 9-3 Processor status

5. Back at the main screen (Figure 9-1 on page 139), click the **Select Grouping Mode** button.
6. Choose one of the **Stack Traceback** options. The Core Processor tool halts all the Compute Node processor cores and displays the requested information. Choose each of the options on that menu in turn so that you can see the variety of data formats that are available.

Stack Traceback (condensed)

In the condensed version of Stack Traceback, data from all nodes is captured. The unique instruction addresses per stack frame are grouped and displayed. However, the last stack frame is grouped based on the function name, not the IAR. This is normally the most useful mode for debug (Figure 9-4).

File Control Analyze Filter Sessions	
Group Mode:	Stack Traceback (condensed) Session 1 (MMC)
0 :	Compute Node (128)
1 :	0xffffffffc (128)
2 :	__libc_start_main (32)
3 :	generic_start_main (32)
4 :	main (16)
5 :	Allgather (16)
6 :	PMPI_Allgather (16)
7 :	MPIDO_Allgather (8)
8 :	MPIDO_Allreduce (8)
9 :	MPID_Progress_wait (1)
10:	DCMF_CriticalSection_cycle (1)
9 :	MPID_Progress_wait (7)
10:	DCMF_Messenger_advance (1)
11:	DCMF::Queueing::Lockbox::Device::advance() (1)
10:	DCMF_Messenger_advance (1)
11:	DCMF::Queueing::Tree::Device::advance() (1)
10:	DCMF_Messenger_advance (5)
11:	DCMF::DMA::Device::advance() (2)
12:	DCMF::DMA::RecFifoGroup::advance() (2)
13:	DMA_RecFifoSimplePollNormalFifoById (2)
11:	DCMF::DMA::Device::advance() (3)
7 :	MPIDO_Allgather (8)
8 :	MPIDO_Allreduce (8)
9 :	MPIR_Allreduce (8)
10:	MPIC_Sendrecv (8)
11:	MPID_Progress_wait (8)
12:	DCMF_Messenger_advance (8)
13:	DCMF::Queueing::GI::Device::advance() (1)
13:	DCMF::DMA::Device::advance() (3)
14:	DCMF::DMA::RecFifoGroup::advance() (3)
15:	DMA_RecFifoSimplePollNormalFifoById (3)

Figure 9-4 Stack Traceback (condensed)

Stack Traceback (detailed)

In Stack Traceback (detailed), data from all nodes is captured (Figure 9-5). The unique instruction addresses per stack frame are grouped and displayed. The IAR at each stack frame is also displayed.

```

Core Processor
File Control Analyze Filter Sessions
Group Mode: Stack Traceback (detailed) Session 1 (MMC)
0 : compute Node (128)
1 : (IAR=0xfffffff0) 0xfffffff0 (128)
2 : (IAR=0x010a873c) __libc_start_main (32)
3 : (IAR=0x010a84dc) generic_start_main (32)
4 : (IAR=0x01001674) main (16)
5 : (IAR=0x01006b1c) Allgather (16)
6 : (IAR=0x01012784) PMPI_Allgather (16)
7 : (IAR=0x01035ab0) MPIDO_Allgather (16)
8 : (IAR=0x01034b5c) MPIDO_Allreduce (16)
9 : (IAR=0x01008e58) MPID_Allreduce (4)
10 : (IAR=0x010176b4) MPIC_Sendrecv (4)
11 : (IAR=0x0102d7ac) MPID_Progress_wait (4)
12 : (IAR=0x01072e50) DCMF_Messenger_advance (1)
13 : (IAR=0x0109628c) DCMF::Queueing::Tree::Device::advance() (1)
12 : (IAR=0x01072e5c) DCMF_Messenger_advance (3)
13 : (IAR=0x0109255c) DCMF::DMA::Device::advance() (1)
14 : (IAR=0x010902e0) DCMF::DMA::RecFifoGroup::advance() (1)
13 : (IAR=0x010925b4) DCMF::DMA::Device::advance() (1)
13 : (IAR=0x01092688) DCMF::DMA::Device::advance() (1)
9 : (IAR=0x010090fc) MPID_Allreduce (12)
10 : (IAR=0x010176b4) MPIC_Sendrecv (12)
11 : (IAR=0x0102d7b4) MPID_Progress_wait (1)
12 : (IAR=0x01074bb4) DCMF_CriticalSection_cycle (1)
11 : (IAR=0x0102d7ac) MPID_Progress_wait (11)
12 : (IAR=0x01072e38) DCMF_Messenger_advance (1)
13 : (IAR=0x01096718) DCMF::Queueing::GI::Device::advance() (1)
12 : (IAR=0x01072e50) DCMF_Messenger_advance (1)
12 : (IAR=0x01072e5c) DCMF_Messenger_advance (9)
13 : (IAR=0x01092560) DCMF::DMA::Device::advance() (1)
13 : (IAR=0x010925b4) DCMF::DMA::Device::advance() (1)
13 : (IAR=0x010925f0) DCMF::DMA::Device::advance() (1)
13 : (IAR=0x01092680) DCMF::DMA::Device::advance() (1)
13 : (IAR=0x01092718) DCMF::DMA::Device::advance() (1)
13 : (IAR=0x0109255c) DCMF::DMA::Device::advance() (2)

```

Figure 9-5 Stack Traceback (detailed)

Stack Traceback (survey)

Stack Traceback (survey) is a quick but potentially inaccurate mode. IARs are initially captured and stack data is collected for each node from a group of nodes that contain the same IAR. The stack data fetched for that one node is then applied to all nodes with the same IAR. Figure 9-6 shows an example of the survey mode.

```

File Control Analyze Filter Sessions
Group Mode: Stack Traceback (survey) Session 1 (MMCS)
0 : Compute Node (128)
1 :   0xffffffff (128)
2 :   __libc_start_main (32)
3 :     generic_start_main (32)
4 :       main (14)
5 :         MPI_Barrier (14)
6 :           MPIDO_Barrier (14)
7 :             MPID_Progress_wait (14)
8 :               DCMF_Messenger_advance (1)
8 :               DCMF_Messenger_advance (3)
9 :                 DCMF::DMA::Device::advance() (1)
9 :                 DCMF::Queueing::Tree::Device::advance() (2)
8 :               DCMF_Messenger_advance (4)
9 :                 DCMF::Queueing::GI::Device::advance() (4)
8 :               DCMF_Messenger_advance (6)
9 :                 DCMF::DMA::Device::advance() (3)
10:                 DCMF::DMA::RecFifoGroup::advance() (1)
10:                 DCMF::DMA::RecFifoGroup::advance() (2)
11:                   DMA_RecFifoSimplePollNormalFifoById (2)
9 :                   DCMF::DMA::Device::advance() (3)
4 :             main (18)
5 :               Allgather (18)
6 :                 PMPI_Allgather (18)
7 :                   MPIDO_Allgather (18)
8 :                     MPIDO_Allreduce (18)
9 :                       MPID_Allreduce (2)
10:                        MPIC_Sendrecv (2)
11:                          MPID_Progress_wait (2)
12:                            DCMF_Messenger_advance (1)
13:                              DCMF::Queueing::Tree::Device::advance() (1)
12:                            DCMF_Messenger_advance (1)
13:                              DCMF::DMA::Device::advance() (1)
14:                                DCMF::DMA::RecFifoGroup::advance() (1)
9 :                                  MPID_Allreduce (16)

```

Figure 9-6 Stack Traceback (survey)

Refer to the following points to help you use the tool more effectively:

- ▶ The number at the far left, before the colon, indicates the depth within the stack.
- ▶ The number in parentheses at the end of each line indicates the number of nodes that share the same stack frame.
- ▶ If you click any line in the stack dump, the pane on the right (labeled Common nodes) shows the list of nodes that share that stack frame. See Figure 9-7 on page 145.
- ▶ When you click one of the stack frames and then select **Control** → **Run**, the action is performed for all nodes that share that stack frame. A new Processor Status summary is displayed. If you again chose a Stack Traceback option, the running processors are halted and the stacks are refetched.
- ▶ You can hold down the Shift key and click several stack frames if you want to control all procedures that are at a range of stack frames.
- ▶ From the **Filter** menu option, you can select **Group Selection** → **Create Filter** to add a filter with the name that you specify in the Filter pull-down. When the box for your filter is highlighted, only the data for those processors is displayed in the upper left window. You can create several filters if you want.
- ▶ Set Group Mode to *Ungrouped* or *Ungrouped with Traceback* to control one processor at a time.

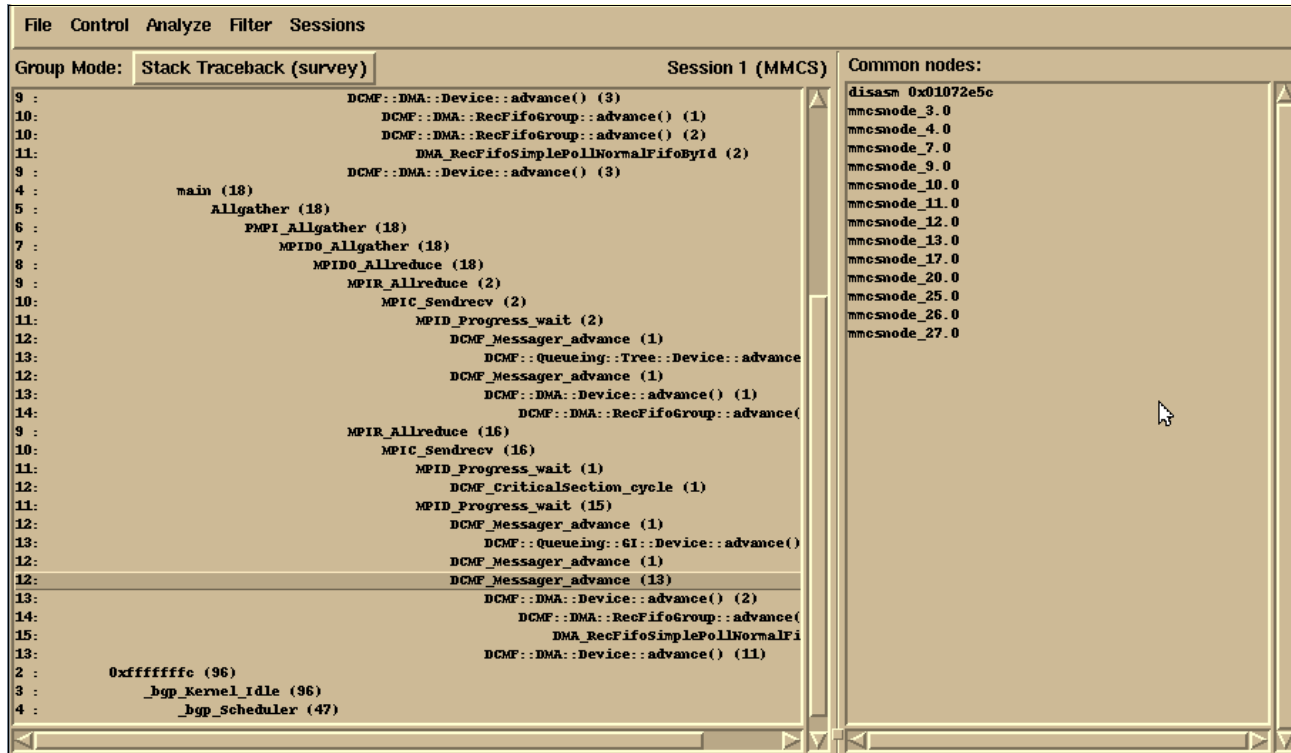


Figure 9-7 Stack Traceback common nodes

9.2.6 Saving your information

To save the current contents of Traceback information of the upper left pane, select **File** → **Save Traceback** to a file of your choice.

To gain more complete data, select **File** → **Take Snapshot**. Notice that you then have two sessions to choose from on the Sessions menu. The original session is (MMCS) and the second one is (SNAP). The snapshot is exactly what the name implies, a picture of the debug session at a particular point. Notice that you cannot start or stop the processors from the snapshot session. You can choose **File** → **Save Snapshot** to save the snapshot to a file. If you are sending data to IBM for debug, Save Snapshot is a better choice than Save Traceback because the snapshot includes objdump data.

If you choose **File** → **Quit** and processors are halted, you are given an option to restart them before quitting.

9.2.7 Debugging live I/O Node problems

It is possible to debug I/O Node as well as Compute Nodes, but you normally want to avoid doing so. Collecting data causes the processor to be stopped, and stopping the I/O Node processors can cause problems with your file system. In addition, the Compute Nodes will not be able to communicate with the I/O Nodes. If you want to debug an I/O Node, you must specify the I/O Node binary when you select **File** → **Attach** to block the window and choose **Filter** → **Debug I/O Nodes**.

9.2.8 Debugging core files

To work with core files, select **File** → **Load Core**. In the window, specify the following information:

- ▶ The location of the Compute Node Kernel binary or binaries
- ▶ The core files location
- ▶ The lowest and highest-numbered core files that you want to work with (The default is all available core files.)

Click the **Load Cores** button when you have specified the information.

The same Grouping Modes are available for core file debug as for live debug. Figure 9-8 shows an output example of the Condensed Stack Traceback options from a core file. Condensed mode is the easiest format to work with.

```
File Control Analyze Filter Sessions
Group Mode: Stack Traceback (condensed) Session 1 (CORE) Common nodes:
0 : Compute Node (128)
1 : 0xffffffff (128)
2 : 0x010b350e (32)
3 : 0x010b31cc (32)
4 : 0x010a74a0 (32)
5 : 0x010b37c4 (32)
6 : 0x010dce08 (32)
2 : 0xffffffff (96)
3 : 0x010b37f4 (96)
4 : 0x010b3578 (96)
5 : _wordcopy_bwd_aligned (1)
6 : _nl_find_msg (1)
7 : _nl_load_domain (1)
8 : __add_to_environ (1)
9 : vfprintf (1)
10 : find_module (1)
11 : __gconv_transliterate (1)
12 : __quicksort (1)
13 : __mktime_internal (1)
14 : 0x01085324 (1)
15 : 0x0109b95c (1)
16 : 0x01099fac (1)
17 : 0x00000000 (1)
5 : _wordcopy_bwd_dest_aligned (95)
6 : 0x010b92c4 (95)
7 : 0x010dada4 (95)
```

Figure 9-8 Core file condensed stack trace

Figure 9-9 shows the detailed version of the same trace.

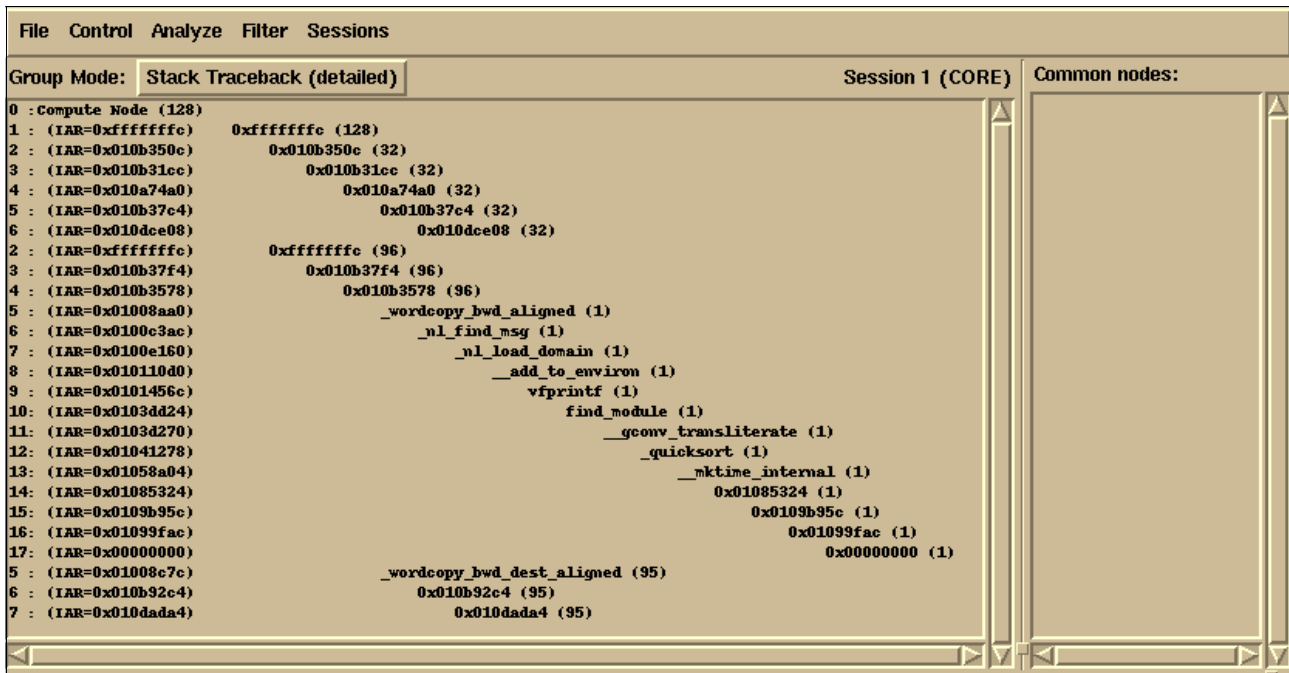


Figure 9-9 Core file detailed stack trace

The Survey option is less useful for core files because speed is not such a concern.

When you select a stack frame in the Traceback output (Figure 9-10), two additional pieces of information are displayed. The core files that share that stack frame are displayed in the Common nodes pane. The Location field under the Traceback pane displays the location of that function and the line number represented by the stack frame. If you select one of the core files in the Common nodes pane, the contents of that core file are displayed in the bottom pane.

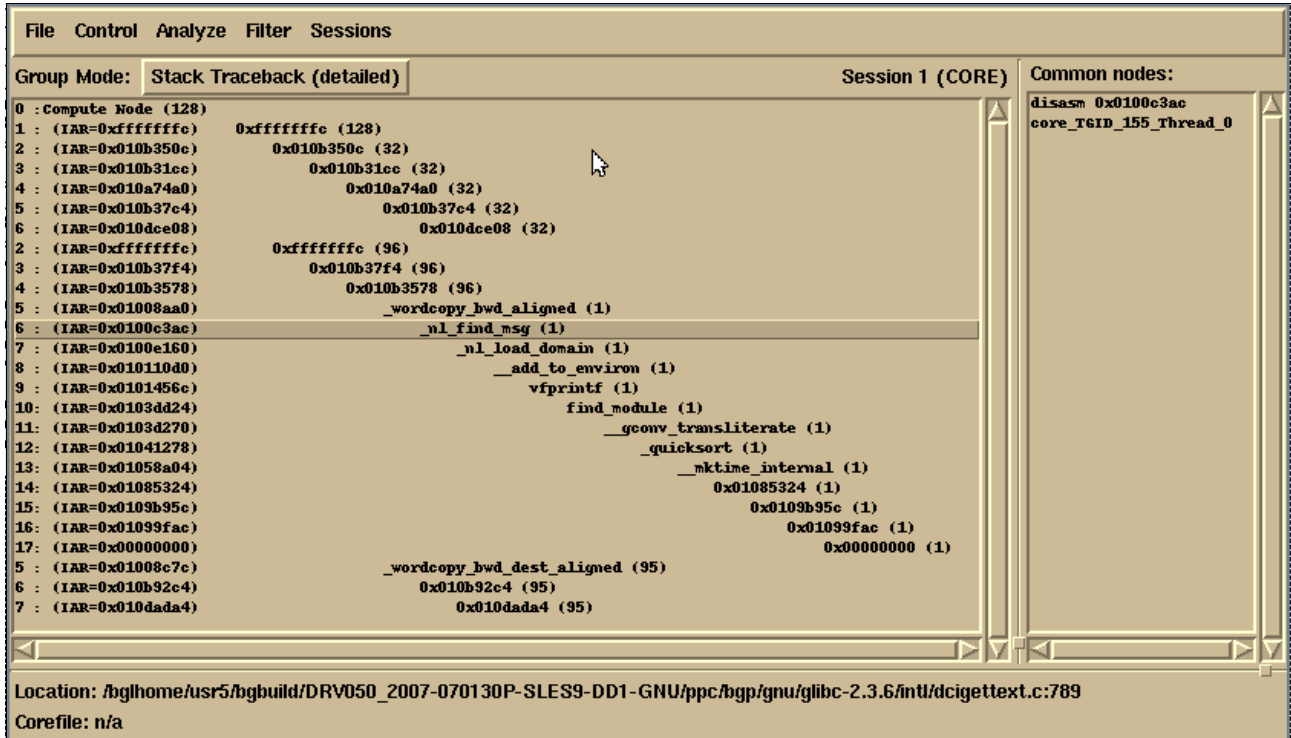


Figure 9-10 Core files common nodes

9.2.9 The addr2line utility

The **addr2line** utility is a standard Linux program. You can find additional information about this utility in any Linux manual as well as at the following Web site:

http://www.linuxcommand.org/man_pages/addr2line1.html

The **addr2line** utility translates an address into file names and line numbers. Using an address and an executable, this utility uses the debugging information in the executable to provide information about the file name and line number. To take advantage of this utility, compile your program with the **-g** option. On the Blue Gene/P system, the core file is a plain text file that you can view with the **vi** editor.

You can use the Linux **addr2line** command on the front-end node and enter the address found in the core file and the **-g** executable. Then the utility points you to the source line where the problem occurred.

Example 9-6 on page 149 shows a core file and how to use the **addr2line** utility to identify potential problems in the code. In this particular case, the program was *not* compiled with the **-g** flag option since this was a production run. However, notice in Example 9-6 that **addr2line** points to **malloc()**. This can be a hint that perhaps there is not enough memory to run this particular calculation or some other problems might be related to the usage of **malloc()** in the code.

Example 9-6 Using addr2line to identify potential problems in your code

vi core.0 and select the addresses between +++STACK and ---STACK and use them as input for addr2line

```
+++STACK  
0x01342cb8  
0x0134653c  
0x0106e5f8  
0x010841ec  
0x0103946c  
0x010af40c  
0x010b5e44  
0x01004fa0  
0x010027cc  
0x0100c028  
0x0100133c  
0x013227ec  
0x01322a4c  
0xffffffffc  
---STACK
```

Run addr2line with your executable

```
$addr2line -e a.out
```

```
0x01342cb8  
0x0134653c  
0x0106e5f8  
0x010841ec  
0x0103946c  
0x010af40c  
0x010b5e44  
0x01004fa0  
0x010027cc  
0x0100c028  
0x0100133c  
0x013227ec  
0x01322a4c  
0xffffffffc/bgldhome/usr6/bgbuid/DRV360_2007-070906P-SLES10-DD2-GNU10/ppc/bgp/gnu/glibc-2.4/mallo  
c/malloc.c:3377  
/bgldhome/usr6/bgbuid/DRV360_2007-070906P-SLES10-DD2-GNU10/ppc/bgp/gnu/glibc-2.4/malloc/malloc.c  
:3525  
modify.cpp:0  
?:0  
?:0  
?:0  
?:0  
main.cpp:0  
main.cpp:0  
main.cpp:0  
?:0  
../csu/libc-start.c:231  
../sysdeps/unix/sysv/linux/powerpc/libc-start.c:127
```



Checkpoint and restart support for applications

In this chapter, we provide details about the checkpoint and restart support provided by the Blue Gene/P system. The contents of this chapter reflect the information that was presented in *IBM System Blue Gene Solution: Application Development*, SG24-7179, but have been updated for the Blue Gene/P system.

Nowadays scientific and engineering applications tend to consume most of the compute cycles on high-performance computers. This is certainly the case on the Blue Gene/P system. Many of the simulations run for extended periods of time.

Checkpoint and restart capabilities are critical for fault recovery. If an application is running for a long period of time, you do not want it to fail after consuming many hours of compute cycles, losing all the calculations made up until the failure. By using checkpoint and restart, you can restart the application at the last checkpoint position, losing a much smaller slice of processing time. In addition, checkpoint and restart are helpful in cases where the given access to a Blue Gene/P system is in relatively small increments of time and you know that your application run will take longer than your allotted amount of processing time. With checkpoint and restart capabilities, you can execute your application in fragmented periods of time rather than an extended interval of time.

We discuss the following topics in this chapter:

- ▶ “Checkpoint and restart” on page 152
- ▶ “Technical overview” on page 152
- ▶ “Checkpoint API” on page 155
- ▶ “Directory and file naming conventions” on page 157
- ▶ “Restart” on page 157

10.1 Checkpoint and restart

Checkpoint and restart are among the primary techniques for fault recovery. A special user-level checkpoint library has been developed for Blue Gene/P applications. Using this library, application programs can take a checkpoint of their program state at the appropriate stages. Then the program can be restarted later from the last successful checkpoint.

10.2 Technical overview

The *checkpoint library* is a user-level library that provides support for user-initiated checkpoints in parallel applications. The current implementation requires application developers to insert calls manually to checkpoint library functions at proper places in the application code. However, the restart is transparent to the application and requires only the user or system to set specific environment variables while launching the application.

The application is expected to make a call to the `BGCheckpointInit()` function at the beginning of the program, to initialize the checkpoint related data structures, and to carry out an automated restart when required. The application can then make calls to the `BGCheckpoint()` function to store a snapshot of the program state in stable storage (files on a disk). The current model assumes that, when an application must take a checkpoint, all of the following points are true:

- ▶ All processes of the application make a call to the `BGCheckpoint()` function.
- ▶ When a process makes a call to `BGCheckpoint()`, no outstanding messages are in the network or buffers. That is the `recv` that corresponds to all the send calls has occurred.
- ▶ After a process has made a call to `BGCheckpoint()`, other processes do not send messages to the process until their checkpoint is complete. Typically, applications are expected to place calls to `BGCheckpoint()` immediately after a barrier operation, such as `MPI_Barrier` or after a collective operation, such as `MPI_Allreduce`, when there are no outstanding messages in the Message Passing Interface (MPI) buffers and the network.

`BGCheckpoint()` can be called multiple times. Successive checkpoints are identified and distinguished by a checkpoint sequence number. A program state that corresponds to different checkpoints is stored in separate files. It is possible to safely delete the old checkpoint files after a newer checkpoint is complete.

The data that corresponds to the checkpoints is stored in a user-specified directory. A separate checkpoint file is made for each process. This checkpoint file contains header information and a dump of the process's memory, including its data and stack segments, but excluding its text segment and read-only data. It also contains information that pertains to the input/output (I/O) state of the application, including open files and the current file positions.

For restart, the same job is launched again with the environment variables `BG_CHKPTRESTARTSEQNO` and `BG_CHKPTDIRPATH` set to the appropriate values. The `BGCheckpointInit()` function checks for these environment variables and, if specified, restarts the application from the desired checkpoint.

10.2.1 Input/output considerations

All the external I/O calls made from a program are shipped to the corresponding I/O Node using a function shipping procedure implemented in the Compute Node Kernel.

The checkpoint library intercepts calls to the following main file I/O functions:

- ▶ `open()`
- ▶ `close()`
- ▶ `read()`
- ▶ `write()`
- ▶ `lseek()`

The function name `open()` is a weak alias that maps to the `_libc_open` function. The checkpoint library intercepts this call and provides its own implementation of `open()` that internally uses the `_libc_open` function.

The library maintains a file state table that stores the file name, current file position, and the mode of all the files that are currently open. The table also maintains a translation that translates the file descriptors used by the Compute Node Kernel to another set of file descriptors to be used by the application. While taking a checkpoint, the file state table is also stored in the checkpoint file. Upon a restart, these tables are read. Also the corresponding files are opened in the required mode, and the file pointers are positioned at the desired locations as given in the checkpoint file.

The current design assumes that the programs either always read the file or write the files sequentially. A read followed by an overlapping write, or a write followed by an overlapping read, is not supported.

10.2.2 Signal considerations

Applications can register handlers for signals using the `signal()` function call. The checkpoint library intercepts calls to `signal()` and installs its own signal handler instead. It also updates a signal-state table that stores the address of the signal handler function (**sig handler**) registered for each signal (**signal**). When a signal is raised, the checkpoint signal handler calls the appropriate application handler given in the signal-state table.

While taking checkpoints, the signal-state table is also stored in the checkpoint file in its signal-state section. At the time of restart, the signal-state table is read, and the checkpoint signal handler is installed for all the signals listed in the signal state table. The checkpoint handler calls the required application handlers when needed.

Signals during checkpoint

The application can potentially receive signals while the checkpoint is in progress. If the application signal handlers are called while a checkpoint is in progress, it can change the state of the memory that is being checkpointed. This can make the checkpoint inconsistent. Therefore, the signals arriving while a checkpoint is under progress must be handled carefully.

For certain signals, such as SIGKILL and SIGSTOP, the action is fixed, and the application terminates without much choice. The signals without any registered handler are simply ignored. For signals with installed handlers, there are two choices:

- ▶ Deliver the signal immediately.
- ▶ Postpone the signal delivery until the checkpoint is complete.

All signals are classified into one of these two categories as shown in Table 10-1. If the signal must be delivered immediately, the memory state of the application might change, making the current checkpoint file inconsistent. Therefore, the current checkpoint must be aborted. The checkpoint routine periodically checks if a signal has been delivered since the current checkpoint began. In case a signal has been delivered, it aborts the current checkpoint and returns to the application.

For signals that are to be postponed, the checkpoint handler simply saves the signal information in a pending signal list. When the checkpoint is complete, the library calls application handlers for all the signals in the pending signal list. If more than one signal of the same type is raised while the checkpoint is in progress, the checkpoint library ensures that the handler registered by the application will be called at least once. However, it does not guarantee in-order-delivery of signals.

Table 10-1 Action taken on signal

Signal name	Signal type	Action to be taken
SIGINT	Critical	Deliver
SIGXCPU	Critical	Deliver
SIGILL	Critical	Deliver
SIGABRT/SIGIOT	Critical	Deliver
SIGBUS	Critical	Deliver
SIGFPE	Critical	Deliver
SIGSTP	Critical	Deliver
SIGSEGV	Critical	Deliver
SIGPIPE	Critical	Deliver
SIGSTP	Critical	Deliver
SIGSTKFLT	Critical	Deliver
SIGTERM	Critical	Deliver
SIGHUP	Non-critical	Postpone
SIGALRM	Non-critical	Postpone
SIGUSR1	Non-critical	Postpone
SIGUSR2	Non-critical	Postpone
SIGTSTP	Non-critical	Postpone
SIGVTALRM	Non-critical	Postpone
SIGPROF	Non-critical	Postpone
SIGPOLL/SIGIO	Non-critical	Postpone
SIGSYS/SIGUNUSED	Non-critical	Postpone
SIGTRAP	Non-critical	Postpone

Signals during restart

The pending signal list is not stored in the checkpoint file. Therefore, if an application is restarted from a checkpoint, the handlers for pending signals received during the checkpoint are not called. If some signals are raised while the restart is in progress, they are ignored. The checkpoint signal handlers are installed only in the end after the memory state, I/O state, and signal-state table have been restored. This ensures that, when the application signal handlers are called, they see a consistent memory and I/O state.

10.3 Checkpoint API

The checkpoint interface consists of the following items:

- ▶ A set of library functions that are used by the application developer to *checkpoint enable* the application
- ▶ A set of conventions used to name and store the checkpoint files
- ▶ A set of environment variables used to communicate with the application

In the following section, we describe each of these components in detail.

10.3.1 Checkpoint library API

To ensure minimal overhead, the basic interface has been kept fairly simple. Ideally, a programmer must call only two functions, one at the time of initialization and the other at the places where the application needs to be checkpointed. Restart is done transparently using the environment variable `BG_CHKPTRESTARTSEQNO` specified at the time of job launch. Alternatively, an explicit restart API is also provided to the programmer to manually restart the application from a specified checkpoint. The remainder of this section describes the checkpoint API in detail.

void BGCheckpointInit(char * ckptDirPath)

BGCheckpointInit is a *mandatory function* that must be invoked at the *beginning of* the program. You use this function to initialize the data structures of the checkpoint library. In addition, you use this function for transparent restart of the application program.

The `ckptDirPath` parameter specifies the location of checkpoint files. If `ckptDirPath` is `NULL`, then the default checkpoint file location is assumed as explained in 10.4, “Directory and file naming conventions” on page 157.

int BGCheckpoint()

BGCheckpoint takes a *snapshot of the program state* at the instant at which it is called. All the processes of the application must make a call to **BGCheckpoint** to take a consistent global checkpoint.

When a process makes a call to **BGCheckpoint**, no outstanding messages should be in the network or buffers. That is, the **recv** that corresponds to all the send calls should have occurred. In addition, after a process has made a call to **BGCheckpoint**, other processes must not send messages to the process until their call to **BGCheckpoint** is complete. Typically, applications are expected to place calls to **BGCheckpoint** immediately after a barrier operation, such as `MPI_Barrier`, or after a collective operation, such as `MPI_Allreduce`, when there is no outstanding message in the MPI buffers and the network.

The state that corresponds to each application process is stored in a separate file. The location of checkpoint files is specified by `ckptDirPath` in the call to **BGCheckpointInit**. If

ckptDirPath is NULL, then the checkpoint file location is decided by the storage rules mentioned in 10.4, “Directory and file naming conventions” on page 157.

void BGCheckpointRestart(int restartSqNo)

BGCheckpointRestart restarts the application from the checkpoint given by the argument restartSqNo. The directory where the checkpoint files are searched is specified by ckptDirPath in the call to **BGCheckpointInit**. If ckptDirPath is NULL, then the checkpoint file location is decided by the storage rules provided in 10.4, “Directory and file naming conventions” on page 157.

An application developer does not need to explicitly invoke this function. **BGCheckpointInit** automatically invokes this function whenever an application is restarted. The environment variable BG_CHKPTRESTARTSEQNO is set to an appropriate value. If the **restartSqNo**, the environment variable BG_CHKPTRESTARTSEQNO, is zero, then the system picks up the most recent consistent checkpoint files. However, the function is available for use if the developer chooses to call it explicitly. The developer must know the implications of using this function.

int BGCheckpointExcludeRegion(void *addr, size_t len)

BGCheckpointExcludeRegion marks the specified region (addr to addr + len - 1) to be excluded from the program state, while a checkpoint is being taken. The state that corresponds to this region is not saved in the checkpoint file. Therefore, after restart the corresponding memory region in the application is not overwritten. You can use this facility to protect critical data that should not be restored at the time of restart such as personality and checkpoint data structures. An application programmer can also use this call to exclude a scratch data structure that does not need to be saved at checkpoint time.

int BGAtCheckpoint((void *) function(void *arg), void *arg)

BGAtCheckpoint registers the functions to be called just before taking the checkpoint. You can use this function to take some action at the time of checkpoint. For example, a user can call this function to close all the communication states open at the time of checkpoint. The functions registered are called in the reverse order of their registration. The argument **arg** is passed to the function that is being called.

int BGAtRestart((void *) function (void *arg), void *arg)

BGAtRestart registers the functions to be called during restart after the program state has been restored, but before jumping to the appropriate position in the application code. The functions that are registered are called in the reverse order of their registration. You can use this function to resume or re-initialize functions or data structures at the time of restart. For example, in the symmetrical multiprocessing Node Mode (SMP Node Mode), the SMP needs to be re-initialized at the time of restart. The argument **arg** is passed to the function that is being called.

int BGAtContinue((void *) function (void *arg), void *arg)

BGAtContinue registers the functions to be called when continuing after a checkpoint. You can use this function to re-initialize or resume some functions or data structures that were closed or stopped at the time of checkpoint. The functions that are registered are called in the reverse order of their registration. The argument **arg** is passed to the function that is being called.

10.4 Directory and file naming conventions

By default, all the checkpoint files are stored, and retrieved during restart, in the directory specified by `ckptDirPath` in the initial call to `BGCheckpointInit()`. If `ckptDirPath` is not specified (or is null), the directory is picked from the environment variable `BG_CHKPTDIRPATH`. This environment variable can be set by the job control system at the time of job launch to specify the default location of the checkpoint files. If this variable is not set, the Blue Gene/P system looks for a `$(HOME)/checkpoint` directory. Finally, if this directory is also not available, `$(HOME)` is used to store all checkpoint files.

The checkpoint files are automatically created and named with the following convention:

```
<ckptDirPath>/ckpt.<xxx-yyy-zzz>.<seqNo>
```

Note the following explanation:

- ▶ `<ckptDirPath>`: Name of the executable, for example, `sweep3d` or `mg.W.2`
- ▶ `<xxx-yyy-zzz>`: Three-dimensional torus coordinates of the process
- ▶ `<seqNo>`: The checkpoint sequence number

The checkpoint sequence number starts at one and is incremented after every successful checkpoint.

10.5 Restart

A transparent restart mechanism is provided through the use of the `BGCheckpointInit()` function and the `BG_CHKPTRESTARTSEQNO` environment variable. Upon startup, an application is expected to make a call to `BGCheckpointInit()`. The `BGCheckpointInit()` function initializes the checkpoint library data structures.

Moreover the `BGCheckpointInit()` function checks for the environment variable `BG_CHKPTRESTARTSEQNO`. If the variable is not set, a job launch is assumed and the function returns normally. In case the environment variable is set to zero, the individual processes restart from their individual latest consistent global checkpoint. If the variable is set to a positive integer, the application is started from the specified checkpoint sequence number.

10.5.1 Determining the latest consistent global checkpoint

Existence of a checkpoint file does not guarantee consistency of the checkpoint. An application might have crashed before completely writing the program state to the file. We have changed this by adding a *checkpoint write complete flag* in the header of the checkpoint file. As soon as the checkpoint file is opened for writing, this flag is set to zero and written to the checkpoint file. When complete checkpoint data is written to the file, the flag is set to one indicating the consistency of the checkpoint data. The job launch subsystem can use this flag to verify the consistency of checkpoint files and delete inconsistent checkpoint files.

During a checkpoint, some of the processes can crash, while others might complete. This can create consistent checkpoint files for some processes and inconsistent or non-existent checkpoint files for other processes. The latest consistent global checkpoint is determined by the latest checkpoint for which all the processes have consistent checkpoint files.

It is the responsibility of the job launch subsystem to make sure that `BG_CHKPTRESTARTSEQNO` corresponds to a consistent global checkpoint. In case `BG_CHKPTRESTARTSEQNO` is set to zero, the job launch subsystem must make sure that files with the highest checkpoint sequence number correspond to a consistent global checkpoint. The behavior of the checkpoint library is undefined if `BG_CHKPTRESTARTSEQNO` does not correspond to a global consistent checkpoint.

10.5.2 Checkpoint and restart functionality

It is often desirable to enable or disable the checkpoint functionality at the time of job launch. Application developers are not required to provide two versions of their programs: one with checkpoint enabled and another with checkpoint disabled. We have used environment variables to transparently enable and disable the checkpoint and restart functionality.

The checkpoint library calls `check` for the environment variable `BG_CHKPT_ENABLED`. The checkpoint functionality is invoked only if this environment variable is set to a value of "1." Table 10-2 summarizes the checkpoint-related function calls.

Table 10-2 Checkpoint and restart APIs

Function name	Usage
<code>BGCheckpointInit(char *ckptDirPath);</code>	Sets the checkpoint directory to <code>ckptDirPath</code> . Initializes the checkpoint library data structures. Carries out restart if environment variable <code>BG_CHKPTRESTARTSEQNO</code> is set.
<code>BGCheckpoint();</code>	Takes a checkpoint. Stores the program state in the checkpoint directory.
<code>BGCheckpointRestart(int rstartSqNo);</code>	Carries out an explicit restart from the specified sequence number.
<code>BGCheckpointExcludeRegion(void *addr, size_t len);</code>	Excludes the specified region from the checkpoint state.

Table 10-3 summarizes the environment variables.

Table 10-3 Checkpoint and restart environment variables

Environment variables	Usage
<code>BG_CHKPT_ENABLED</code>	Is set (to 1) if checkpoints are desired; otherwise it is not specified.
<code>BG_CHKPTDIRPATH</code>	Default path to keep checkpoint files.
<code>BG_CHKPTRESTARTSEQNO</code>	Set to a desired checkpoint sequence number from where a user wants the application to restart. If set to zero, each process restarts from its individual latest consistent checkpoint. This option must not be specified, if no restart is desired.

The most common environment variable settings are:

- ▶ `BG_CHKPT_ENABLED=1`
- ▶ `BG_CHKPTDIRPATH=` checkpoint directory
- ▶ `BG_CHKPTRESTARTSEQNO=0`

A combination of `BG_CHKPT_ENABLED` and `BG_CHKPTRESTARTSEQNO` (as in Table 10-3) automatically signifies that after restart, further checkpoints are taken. A developer can restart an application but disable further checkpoints by simply unsetting (removing altogether) the `BG_CHKPT_ENABLED` variable.



Control system (Bridge) APIs

In this chapter, we define a list of application programming interfaces (APIs) into the Midplane Management Control System (MMCS) that can be used by a job management system. The `mpirun` program that ships with the Blue Gene/P software is an application that uses these APIs to manage partitions, jobs, and other similar aspects of the Blue Gene/P system. You can use these APIs to write applications to manage Blue Gene/P partitions and control Blue Gene/P job execution, as well as other similar administrative tasks.

In this chapter, we present an overview of the support provided by the APIs and discuss the following topics:

- ▶ “API requirements” on page 160
- ▶ “APIs” on page 162
- ▶ “Small partition allocation” on page 191
- ▶ “API examples” on page 192

11.1 API requirements

There are several requirements for writing programs to the Bridge API as explained in the following sections.

- ▶ Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- ▶ C and C++ are supported with the GNU gcc 4.1.1 level compilers. For more information and downloads, refer to the following Web address:
<http://gcc.gnu.org/>
- ▶ All required include files are installed in the `/bgsys/drivers/ppcfloor/include` directory. See Appendix B, “Header files and libraries” on page 271, for additional information about include files. The include file for the Bridge API is `rm_api.h`.
- ▶ The Bridge API supports 64-bit applications that use dynamic linking using shared objects. The required library files are installed in the `/bgsys/drivers/ppcfloor/lib64` directory.

The shared object for linking to the Bridge API is `libbgpbridge.so`. The `libbgpbridge.so` library has dependencies on other libraries that are included with the Blue Gene/P software, including:

- `libbgpconfig.so`
- `libbgpdb.so`
- `libsaymessage.so`
- `libtableapi.so`

These files are installed with the standard system installation procedure. They are contained in the `bgpbase.rpm` file.

11.1.1 Configuring environment variables

Table 11-1 provides information about the environment variables that are used to control the Bridge API.

Table 11-1 Environment variables that control the Bridge API

Environment variable	Required	Description
DB_PROPERTY	Yes	This variable must be set to the path of the <code>db.properties</code> file with database connection information. For default installation, the path to this file is <code>/bgsys/local/etc/db.properties</code> .
BRIDGE_CONFIG	Yes	This variable must be set to the path of the <code>bridge.config</code> file that contains the Bridge API configuration values. For a default installation, the path to this file is <code>/bgsys/local/etc/bridge.config</code> .
BRIDGE_DUMP_XML	No	When set to any value, this variable causes the Bridge API to dump its in-memory XML streams to files in <code>/tmp</code> for debugging. When this variable is not set, the Bridge API does not dump its in-memory XML streams.

For more information about the `db.properties` and `bridge.config` files, see *Blue Gene System Administration*, SG24-7417.

11.1.2 General comments

All of the APIs that are used have general considerations that apply to all calls. In the following list, we highlight the common features:

- ▶ All the API calls return a `status_t` indicating either success or an error code.
- ▶ The get APIs that retrieve a compound structure include accessory functions to retrieve relevant nested data.
- ▶ The get calls allocate new memory for the structure to be retrieved and return a pointer to the allocated memory in the corresponding argument.
- ▶ To add information to MMCS, use new functions as well as `rm_set_data()`. The new functions allocate memory for new data structures, and the `rm_set_data()` API is used to fill these structures.
- ▶ For each get and new function, a corresponding free function frees the memory allocated by these functions. For instance, `rm_get_BG(rm_BG_t **bg)` is complemented by `rm_free_BG(rm_BG_t *bg)`.
- ▶ The caller is responsible for matching the calls to the get and new allocators to the corresponding free deallocators. Memory leaks result if this is not done.

Memory allocation and deallocation

Some API calls result in memory being allocated on behalf of the user. The user must call the corresponding free function to avoid memory leaks, which can cause the process to run out of memory.

For the `rm_get_data()` API, see 11.2.8, “Field specifications for the `rm_get_data()` and `rm_set_data()` APIs” on page 178, for a complete list of the fields that require calls to free memory.

Avoiding invalid pointers

Some APIs return a pointer to an offset in a data structure, or object, that was previously allocated (based on *element* in `rm_get_data()`). An example of this is the `rm_get_data()` API call using the `RM_PartListNextPart` specification. In this example, *element* is a partition list, and it returns a pointer to the first or next partition in the list. If the caller of the API frees the memory of the partition list (*element*) and *data* is pointing to a subset of that freed memory, then the *data* pointer is invalid. The caller must make sure that no further calls are made against a data structure returned from an `rm_get_data()` call after it is freed.

First and next calls

Before a *next* call can be made against a data structure returned from an `rm_get_data()` call, the *first* call must have been made. Failure to do so results in an invalid pointer, either pointing at nothing or at invalid data.

Example 11-1 shows correct usage of the first and next API calls. Notice how memory is freed after the list is consumed.

Example 11-1 Correct usage of first and next API calls

```
status_t stat;
int list_size = 0;
rm_partition_list_t * bgp_part_list = NULL;
rm_partition_t * bgp_part = NULL;

// Get all information on existing partitions
stat = rm_get_partitions_info(PARTITION_ALL_FLAG, &bgp_part_list);
```

```

if (stat != STATUS_OK) {
    // Do some error handling here...
    return;
}

// How much data (# of partitions) did we get back?
rm_get_data(bgp_part_list, RM_PartListSize, &list_size);

for (int i = 0; i < list_size; i++) {
    // If this is the first time through, use RM_PartListFirstPart
    if (i == 0){
        rm_get_data(bgp_part_list, RM_PartListFirstPart, &bgp_part);
    }
    // Otherwise, use RM_PartListNextPart
    else {
        rm_get_data(bgp_part_list, RM_PartListNextPart, &bgp_part);
    }
}

// Make sure we free the memory when finished
stat = rm_free_partition_list(bgp_part_list);
if (stat != STATUS_OK) {
    // Do some error handling here...
    return;
}

```

11.2 APIs

In the following sections, we describe details about the APIs.

11.2.1 API to the Midplane Management Control System

The Bridge API contains an `rm_get_BG()` function to retrieve current configuration and status information about all the physical components of the Blue Gene/P system from the MMCS database. The Bridge API also includes functions that add, remove, or modify information about transient entities, such as jobs and partitions.

The `rm_get_BG()` function returns all the necessary information to define new partitions in the system. The information is represented by three lists: a list of base partitions (BPs), a list of wires, and a list of switches. This representation does not contain redundant data. In general, it allows manipulation of the retrieved data into any desired format. The information is retrieved using a structure called `rm_BG_t`. It includes the three lists that are accessed using iteration functions and the various configuration parameters, for example, the size of a base partition in Compute Nodes.

All the data that is retrieved by using the get functions can be accessed using `rm_get_data()` with one of the specifications listed in 11.2.8, “Field specifications for the `rm_get_data()` and `rm_set_data()` APIs” on page 178. There are additional get functions to retrieve information about the partitions and jobs entities.

The `rm_add_partition()` and `rm_add_job()` functions add and modify data in the MMCS. The memory for the data structures is allocated by the *new* functions and updated using the `rm_set_data()` function. The specifications that can be set using the `rm_set_data()` function are shown in 11.2.8, “Field specifications for the `rm_get_data()` and `rm_set_data()` APIs” on page 178.

Deprecated: Some specifications can be marked as “deprecated”. A deprecated specification may be removed in future versions of the Blue Gene supercomputer.

11.2.2 Asynchronous APIs

Some APIs that operate on partitions or jobs are documented as being asynchronous. Asynchronous means that control returns to your application before the operation requested is complete.

Before you perform additional operations on the partition or job, make sure that it is in a valid state by using the `rm_get_partition_info()` or `rm_get_job()` APIs to check the current state of the partition or job.

11.2.3 State sequence IDs

For most Blue Gene objects that have a *state* field, there is a corresponding sequence ID field for the state value. MMCS guarantees that any time the state field changes for a given object, the associated sequence ID will be incremented.

The sequence ID fields can be used to determine which state value is more recent. A state value with a higher corresponding sequence ID is the more recent value. This comparison can be helpful for applications that retrieve state information from multiple sources such as the Bridge API and the real-time APIs.

The function to increment sequence IDs only occurs if the real-time APIs are configured for the system. For information about configuring the real-time APIs, see *Blue Gene System Administration*, SG24-7417.

11.2.4 Bridge API return codes

When a failure occurs, an API invocation returns an error code. You can use the error code to take corrective actions within your application. In addition, a failure always generates a log message, which provides more information for the possible cause of the problem and an optional corrective action. These log messages are used for debugging and programmed recovery of failures.

The design aims at striking a balance between the number of error codes detected and the different error paths per return code. Thus, some errors have specific return codes, while others have more generic ones. The Bridge API has the following return codes:

- ▶ `STATUS_OK`: The invocation completed successfully.
- ▶ `PARTITION_NOT_FOUND`: The required partition specified by the ID cannot be found in the control system.
- ▶ `JOB_NOT_FOUND`: The required job specified by the ID cannot be found in the control system.
- ▶ `BP_NOT_FOUND`: One or more of the base partitions in the `rm_partition_t` structure do not exist.
- ▶ `SWITCH_NOT_FOUND`: One or more of the switches in the `rm_partition_t` structure do not exist.
- ▶ `JOB_ALREADY_DEFINED`: A job with the same name already exists.
- ▶ `PARTITION_ALREADY_DEFINED`: A partition already exists with the ID specified.

- ▶ **CONNECTION_ERROR**: The connection with the control system has failed or could not be established.
- ▶ **INVALID_INPUT**: The input to the API invocation is invalid, which is due to missing required data, illegal data, and so on.
- ▶ **INCOMPATIBLE_STATE**: The state of the partition or job prohibits the specific action. See Figure 11-1 on page 171, Figure 11-2 on page 175, Figure 11-3 on page 176, and Figure 11-4 on page 177 for state diagrams.
- ▶ **INCONSISTENT_DATA**: The data retrieved from the control system is not valid.
- ▶ **INTERNAL_ERROR**: Such errors do not belong to any of the previously listed categories, such as a memory allocation problem or failures during the manipulation of internal XML streams.

11.2.5 Blue Gene hardware resource APIs

In this section, we describe the APIs that are used to manage the hardware resources in the Blue Gene system:

- ▶ `status_t rm_get_BG(rm_BG_t **BG);`

This function retrieves a snapshot of the Blue Gene/P machine, held in the `rm_BG_t` data structure.

The following return codes are possible:

- **STATUS_OK**
- **CONNECTION_ERROR**
- **INCONSISTENT_DATA**
 - List of base partitions is empty.
 - Wire list is empty, and the number of base partitions is greater than one.
 - Switch list is empty, and the number of base partitions is greater than one.
- **INTERNAL_ERROR**

- ▶ `status_t rm_get_data(rm_element_t *rme, enum RMSpecification spec, void *result);`

This function returns the content of the requested field from a valid `rm_element_t` (Blue Gene object, base partition object, wire object, switch object, and so on). The specifications that are available when using `rm_get_data()` are listed in 11.2.8, “Field specifications for the `rm_get_data()` and `rm_set_data()` APIs” on page 178, and are grouped by the object type that is being accessed.

The following return codes are possible:

- **STATUS_OK**
- **INVALID_INPUT**
 - The specification *spec* is unknown.
 - The specification *spec* is illegal (per the “rme” element).
- **INTERNAL_ERROR**

▶ `status_t rm_get_nodetcards(rm_bp_id_t bpid, rm_nodetcard_list_t **nc_list);`

This function returns all node cards in the specified base partition.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INCONSISTENT_DATA
 - The Base Partition was not found.
- INTERNAL_ERROR

▶ `status_t rm_get_serial(rm_serial_t *serial);`

This function gets the machine serial number that was set previously by `rm_set_serial()`.

The following return codes are possible:

- STATUS_OK
- INTERNAL_ERROR

▶ `status_t rm_set_data(rm_element_t *rme, enum RMSpecification spec, void *result);`

This function sets the value of the requested field in the `rm_element_t` (Blue Gene/P object, base partition object, wire object, switch object, and so on). The specifications, which are available when using `rm_set_data()`, are listed in 11.2.8, “Field specifications for the `rm_get_data()` and `rm_set_data()` APIs” on page 178, and are grouped by the object type that is being accessed.

The following return codes are possible:

- STATUS_OK
- INVALID_INPUT
 - The specification `spec` is unknown.
 - The specification `spec` is illegal (per the `rme` element).
- INTERNAL_ERROR

▶ `status_t rm_set_serial(rm_serial_t serial);`

This function sets the machine serial number to be used in all the API calls following this call. The database can contain more than one machine. Therefore, it is necessary to specify which machine to work with.

The following return codes are possible:

- STATUS_OK
- INVALID_INPUT
 - The machine serial number `serial` is null.
 - The machine serial number is too long.

11.2.6 Partition-related APIs

In this section, we describe the APIs that are used to create and manage partitions in the Blue Gene system:

▶ `status_t rm_add_partition(rm_partition_t* p);`

This function adds a partition record to the database. The partition structure includes an ID field that is filled by the resource manager.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT: The data in the `rm_partition_t` structure is invalid.
 - No base partition nor switch list is supplied.
 - Base partition or switches do not construct a legal partition.
 - No boot images nor boot image name is too long.
 - No user nor user name is too long.
- BP_NOT_FOUND:
 - One or more of the base partitions in the `rm_partition_t` structure does not exist.
- SWITCH_NOT_FOUND:
 - One or more of the switches in the `rm_partition_t` structure does not exist.
- INTERNAL_ERROR

▶ `status_t rm_add_part_user (pm_partition_id_t partition_id, const char *user);`

This function adds a new user to the partition. The partition's owner can add users who are allowed to use this partition. Adding users to the partition can be done only by the partition owner and only to partitions in the INITIALIZE state.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT
 - `partition_id` is null or the length exceeds the limitations of the control system.
 - `user` is null or the length exceeds the limitations of the control system.
 - `user` is already defined as the partition's user.
- INTERNAL_ERROR

▶ `status_t rm_assign_job(pm_partition_id_t partition_id, db_job_id_t jid);`

This function assigns a job to a partition. A job can be created and simultaneously assigned to a partition by calling `rm_add_job()` with a partition ID. If a job is created and not assigned to a specific partition, it can be assigned later by calling `rm_assign_job()`.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT
 - `partition_id` is null, or the length exceeds control system limitations.

- PARTITION_NOT_FOUND
 - JOB_NOT_FOUND
 - INCOMPATIBLE_STATE
 - The current state of the partition is not FREE. See Figure 11-1 on page 171.
 - INTERNAL_ERROR
- ▶ `status_t pm_create_partition(pm_partition_id_t partition_id);`
 This function allocates the necessary hardware for a partition, boots the partition, and updates the resulting status in the MMCS database.

Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:

- STATUS_OK
 - CONNECTION_ERROR
 - INVALID_INPUT
 - `partition_id` is null, or the length exceeds control system limitations.
 - PARTITION_NOT_FOUND
 - INCOMPATIBLE_STATE
 - The current state of the partition prohibits its creation. See Figure 11-1 on page 171.
 - INTERNAL_ERROR
- ▶ `status_t pm_destroy_partition(pm_partition_id_t partition_id);`
 This function shuts down a currently booted partition and updates the database accordingly.

Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:

- STATUS_OK
 - CONNECTION_ERROR
 - INVALID_INPUT
 - `partition_id` is null, or the length exceeds the limitations of the control system.
 - PARTITION_NOT_FOUND
 - INCOMPATIBLE_STATE
 - The state of the partition prohibits its destruction. See Figure 11-1 on page 171.
 - INTERNAL_ERROR
- ▶ `status_t rm_get_partition(pm_partition_id_t partition_id, rm_partition_t **p);`
 This function retrieves a partition, according to its ID.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR

- INVALID_INPUT
 - partition_id is null, or the length exceeds the limitations of the control system.
 - PARTITION_NOT_FOUND
 - INCONSISTENT_DATA
 - The base partition or switch list of the partition is empty.
 - INTERNAL_ERROR
- ▶ status_t rm_get_partitions(rm_partition_state_t flag_t flag, rm_partition_list_t **part_list);

This function is useful for status reports and diagnostics. It returns a list of partitions whose current state matches the flag. The possible flags are contained in the rm_api.h include file and listed in Table 11-2. You can use OR on these values to create a flag for including partitions with different states.

The following return codes are possible:

- STATUS_OK
 - CONNECTION_ERROR
 - INCONSISTENT_DATA
 - At least one of the partitions has an empty base partition list.
 - INTERNAL_ERROR
- ▶ status_t rm_get_partitions_info(rm_partition_state_t flag_t flag, rm_partition_list_t ** part_list);

This function is useful for status reports and diagnostics. It returns a list of partitions whose current state matches the flag. This function returns the partition information without their base partitions, switches, and node cards.

The possible flags are contained in the rm_api.h include file and are listed in Table 11-2. You can use OR on these values to create a flag for including partitions with different states.

Table 11-2 Flags for partition states

Flag	Value
PARTITION_FREE_FLAG	0x01
PARTITION_CONFIGURING_FLAG	0x02
PARTITION_READY_FLAG	0x04
PARTITION_DEALLOCATING_FLAG	0x10
PARTITION_ERROR_FLAG	0x20
PARTITION_REBOOTING_FLAG	0x40
PARTITION_ALL_FLAG	0xFF

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INCONSISTENT_DATA
 - At least one of the partitions has an empty base partition list.
- INTERNAL_ERROR

- ▶ `status_t rm_modify_partition(pm_partition_id_t partition_id, enum rm_modify_op modify_option, const void *value);`

This function makes it possible to change a set of fields in an already existing partition. Only partitions whose state is `RM_PARTITION_FREE` can be modified. The fields that can be modified are owner, description, options, and the partition boot images. The `modify_option` parameter identifies the field to be modified.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `INVALID_INPUT`
 - `partition_id` is null, or the length exceeds the limitations of the control system.
 - The value for the `modify_option` parameter is not valid.
- `PARTITION_NOT_FOUND`
- `INCOMPATIBLE_STATE`
 - The partition's current state forbids its modification. See Figure 11-1 on page 171.
- `INTERNAL_ERROR`

- ▶ `status_t rm_reboot_partition(pm_partition_id_t partition_id);`

This function sends a request to reboot a partition and update the resulting status in the database.

Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `INVALID_INPUT`
 - `partition_id` is null, or the length exceeds the limitations of the control system.
- `PARTITION_NOT_FOUND`
- `INCOMPATIBLE_STATE`
 - The partition's current state forbids its removal. See Figure 11-1 on page 171.
- `INTERNAL_ERROR`

- ▶ `status_t rm_release_partition(pm_partition_id_t partition_id);`

This function is the opposite of `rm_assign_job()`, because it releases the partition from all jobs. Only jobs that are in an `RM_JOB_IDLE` state have their partition reference removed.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `INVALID_INPUT`
 - `partition_id` is null, or the length exceeds the limitations of the control system (configuration parameter).

- PARTITION_NOT_FOUND
 - INCOMPATIBLE_STATE
 - The current state of one or more jobs assigned to the partition prevents this release. See Figure 11-1 on page 171 and Figure 11-2 on page 175.
 - INTERNAL_ERROR
- ▶ `status_t rm_remove_partition(pm_partition_id_t partition_id);`
 This function removes the specified partition record from MMCS.
 The following return codes are possible:
- STATUS_OK
 - CONNECTION_ERROR
 - INVALID_INPUT
 - `partition_id` is null, or the length exceeds the limitations of the control system (configuration parameter).
 - PARTITION_NOT_FOUND
 - INCOMPATIBLE_STATE
 - The partition's current state forbids its removal. See Figure 11-1 on page 171 and Figure 11-2 on page 175.
 - INTERNAL_ERROR
- ▶ `status_t rm_remove_part_user(pm_partition_id_t partition_id, const char *user);`
 This function removes a user from a partition. The partition's owner can remove users from the partition's user list. Removing a user from a partition can be done only by the partition owner and only to partitions in the `RM_PARTITION_FREE` state.
 The following return codes are possible:
- STATUS_OK
 - CONNECTION_ERROR
 - INVALID_INPUT
 - `partition_id` is null, or the length exceeds the limitations of the control system (configuration parameter).
 - `user` is null, or the length exceeds the limitations of the control system.
 - `user` is already defined as the partition's user.
 - Current user is not the partition owner.
 - INTERNAL_ERROR
- ▶ `status_t rm_set_part_owner(pm_partition_id_t partition_id, const char *user);`
 This function sets the new owner of the partition. Changing the partition's owner can be done only to a partition in the `RM_PARTITION_FREE` state.
 The following return codes are possible:
- STATUS_OK
 - CONNECTION_ERROR
 - INVALID_INPUT
 - `partition_id` is null, or the length exceeds the limitations of the control system (configuration parameter).
 - `owner` is null, or the length exceeds the limitations of the control system.
 - INTERNAL_ERROR

State transition diagram for partitions

Figure 11-1 illustrates the states that a partition goes through during its life cycle.

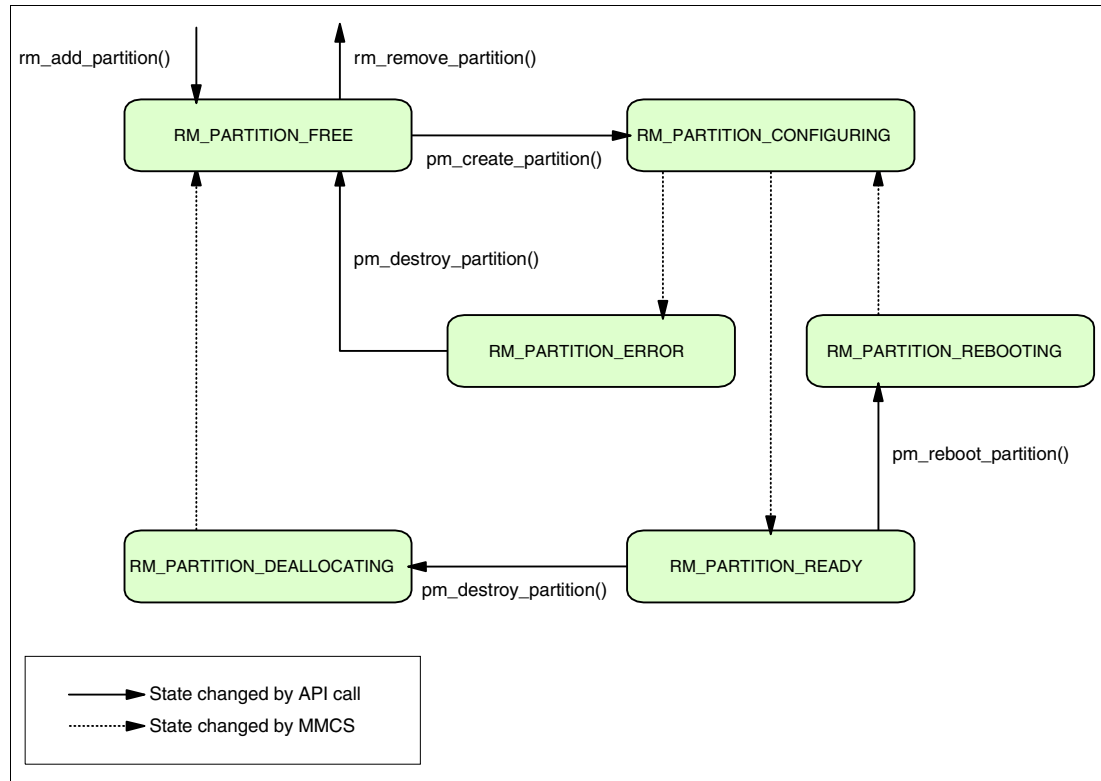


Figure 11-1 Partition state diagram

11.2.7 Job-related APIs

In this section, we describe the APIs to create and manage jobs in the Blue Gene system:

▶ `status_t rm_add_job(rm_job_t *job);`

This function adds a job record to the database. The job structure includes an ID field that will be filled by the resource manager.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT:
 - Data in the `rm_job_t` structure is invalid.
 - There is no job name, or a job name is too long.
 - There is no user, or the user name is too long.
 - There is no executable, or the executable name is too long.
 - The output or error file name is too long.
- JOB_ALREADY_DEFINED
 - A job with the same name already exists.
- INTERNAL_ERROR

▶ `status_t jm_attach_job(jobid);`

This function initiates the spawn of debug servers to job in the `RM_JOB_LOADED` state.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `JOB_NOT_FOUND`
- `INCOMPATIBLE_STATE`

- The job's state prevents it from being attached. See Figure 11-2 on page 175.

- `INTERNAL_ERROR`

▶ `status_t jm_begin_job(jobid);`

This function begins a job that is already loaded.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `JOB_NOT_FOUND`
- `INCOMPATIBLE_STATE`

- The job's state prevents it from beginning. See Figure 11-2 on page 175.

- `INTERNAL_ERROR`

▶ `status_t jm_cancel_job(db_job_id_t jid);`

This function sends a request to cancel the job identified by the `jid` parameter.

Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `JOB_NOT_FOUND`
- `INCOMPATIBLE_STATE`

- The job's state prevents it from being canceled. See Figure 11-2 on page 175.

- `INTERNAL_ERROR`

▶ `status_t jm_debug_job(jobid);`

This function initiates the spawn of debug servers to job in the `RM_JOB_RUNNING` state.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `JOB_NOT_FOUND`
- `INCOMPATIBLE_STATE`

- The job's state prevents it from being debugged. See Figure 11-2 on page 175.

- `INTERNAL_ERROR`

- ▶ `status_t rm_get_job(db_job_id_t jid, rm_job_t **job);`

This function retrieves the specified job object.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INTERNAL_ERROR

- ▶ `status_t rm_get_jobs(rm_job_state_flag_t flag, rm_job_list_t **job_list);`

This function returns a list of jobs whose current state matches the flag.

The possible flags are contained in the `rm_api.h` include file and are listed in Table 11-3. You can use OR on these values to create a flag for including jobs with different states.

Table 11-3 *Flags for job states*

Flag	Value
JOB_IDLE_FLAG	0x001
JOB_STARTING_FLAG	0x002
JOB_RUNNING_FLAG	0x004
JOB_TERMINATED_FLAG	0x008
JOB_ERROR_FLAG	0x010
JOB_DYING_FLAG	0x020
JOB_DEBUG_FLAG	0x040
JOB_LOAD_FLAG	0x080
JOB_LOADED_FLAG	0x100
JOB_BEGIN_FLAG	0x200
JOB_ATTACH_FLAG	0x400
JOB_KILLED_FLAG	0x800

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INTERNAL_ERROR

- ▶ `status_t jm_load_job(jobid);`

This function sets the job state to LOAD.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INCOMPATIBLE_STATE
 - The job's state prevents it from being loaded. See Figure 11-2 on page 175.
- INTERNAL_ERROR

- ▶ `status_t rm_query_job(db_job_id_t db_job_id, MPIR_PROCDesc **proc_table, int *proc_table_size);`

This function fills the `proc_table` with information about the specified job.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `JOB_NOT_FOUND`
- `INTERNAL_ERROR`

- ▶ `status_t rm_remove_job(db_job_id_t jid);`

This function removes the specified job record from MMCS.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `JOB_NOT_FOUND`
- `INCOMPATIBLE_STATE`
 - The job's state prevents its removal. See Figure 11-2.
- `INTERNAL_ERROR`

- ▶ `status_t jm_signal_job(db_job_id_t jid, rm_signal_t signal);`

This function sends a request to signal the job identified by the `jid` parameter.

Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `JOB_NOT_FOUND`
- `INCOMPATIBLE_STATE`
 - The job's state prevents it from being signaled.
- `INTERNAL_ERROR`

- ▶ `status_t jm_start_job(db_job_id_t jid);`

This function starts the job identified by the `jid` parameter. Note that the partition information is referenced from the job record in MMCS.

Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:

- `STATUS_OK`
- `CONNECTION_ERROR`
- `JOB_NOT_FOUND`
- `INCOMPATIBLE_STATE`
 - The job's state prevents its execution. See Figure 11-2.
- `INTERNAL_ERROR`

State transition diagrams for jobs

Figure 11-2 illustrates the states that a job goes through during its life cycle. It also illustrates the order of API calls for creating, running, and canceling a job.

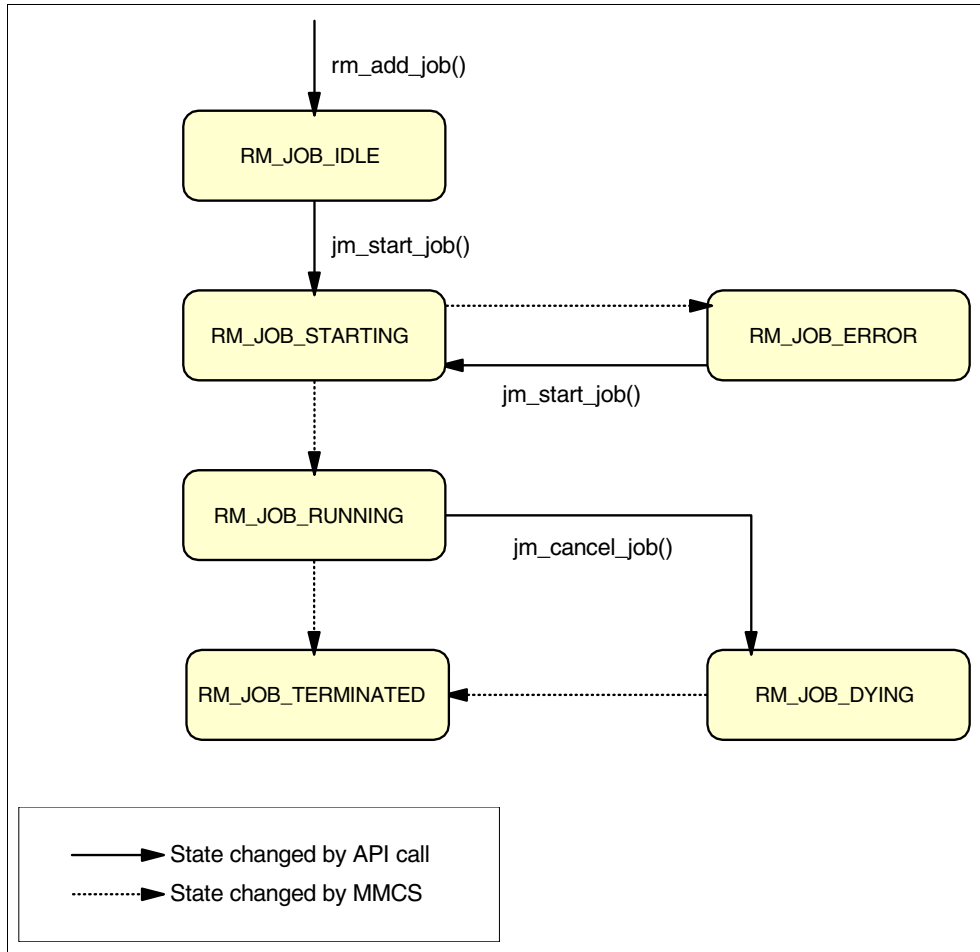


Figure 11-2 Job state diagram for running a Blue Gene/P job

Figure 11-3 illustrates the main states that a job goes through when debugging a new job.

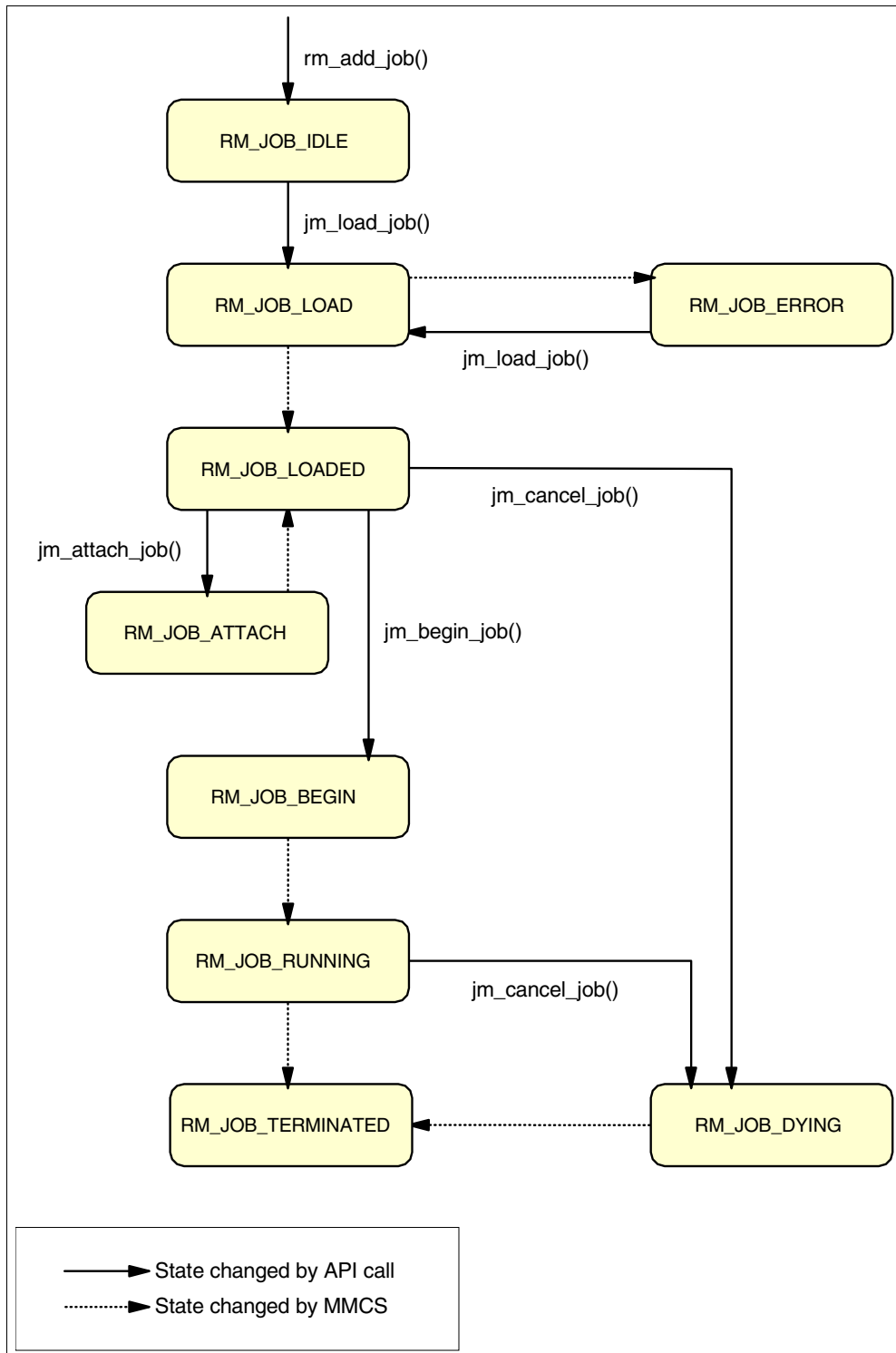


Figure 11-3 Job state diagram for debugging a running job

Figure 11-4 illustrates the states that a job goes through when debugging an already running job.

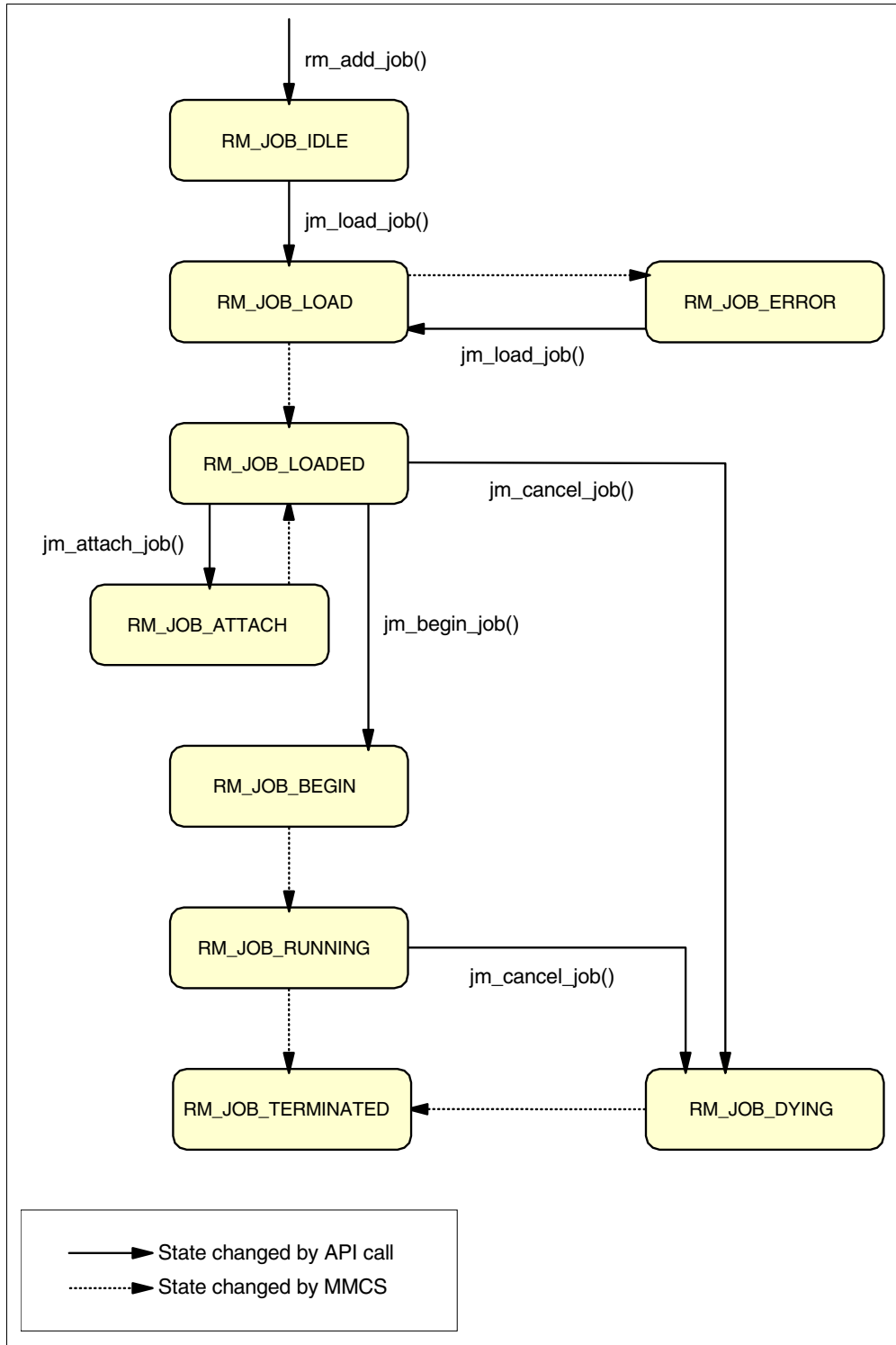


Figure 11-4 Job state diagram for debugging a new job

11.2.8 Field specifications for the `rm_get_data()` and `rm_set_data()` APIs

In this section, we describe all the field specifications that can be used to get and set fields from various objects using the `rm_get_data()` and `rm_set_data()` APIs.

Blue Gene object

The Blue Gene/P object (`rm_BG_t`) represents the Blue Gene/P system. You can use this object to retrieve information and status for other components in the system, such as base partitions, node cards, I/O Nodes, switches, wires, and port (Table 11-4). The Blue Gene object is retrieved by calling the `rm_get_BG()` API.

Table 11-4 Values retrieved from a Blue Gene object using `rm_get_data()`

Description	Specification	Argument type	Notes
Size of a base partition (in Compute Nodes) in each dimension	RM_BPsize	<code>rm_size3D_t *</code>	
Size of the machine in base partition units	RM_Msize	<code>rm_size3D_t *</code>	
Number of base partitions in the machine	RM_BPNum	<code>int *</code>	
First base partition in the list	RM_FirstBP	<code>rm_BP_t **</code>	
Next base partition in the list	RM_NextBP	<code>rm_BP_t **</code>	
Number of switches in the machine	RM_SwitchNum	<code>int *</code>	
First switch in the list	RM_FirstSwitch	<code>rm_switch_t **</code>	
Next switch in the list	RM_NextSwitch	<code>rm_switch_t **</code>	
Number of wires in the machine	RM_WireNum	<code>int *</code>	
First wire in the list	RM_FirstWire	<code>rm_wire_t **</code>	
Next wire in the list	RM_NextWire	<code>rm_wire_t **</code>	

Base partition object

The base partition object (`rm_BP_t`) represents one base partition in the Blue Gene system. The base partition object is retrieved from the Blue Gene object using either the `RM_FirstBP` or `RM_NextBP` specification. See Table 11-5.

Table 11-5 Values retrieved from a base partition object using `rm_get_data()`

Description	Specification	Argument type	Notes
Base partition identifier	RM_BPID	<code>rm_bp_id_t *</code>	free required
Base partition state	RM_BPState	<code>rm_BP_state_t *</code>	
Sequence ID for the base partition state	RM_BPStateSeqID	<code>rm_sequence_id_t *</code>	
Location of the base partition in the 3D machine	RM_BPLoc	<code>rm_location_t *</code>	
Identifier of the partition associated with the base partition	RM_BPPartID	<code>pm_partition_id_t *</code>	free required. If no partition is associated, NULL is returned.

Description	Specification	Argument type	Notes
State of the partition associated with the base partition	RM_BPPartState	rm_partition_state_t *	
Sequence ID for the state of the partition associated with the base partition	RM_BPStateSeqID	rm_sequence_id_t *	
Flag indicating whether this base partition is being used by a small partition (smaller than a base partition)	RM_BPSDB	int *	0=No 1=Yes
Flag indicating whether this base partition is being divided into one or more small partitions	RM_BPSD	int *	0=No 1=Yes
Compute Node memory size for the base partition	RM_BPComputeNodeMemory	rm_BP_computenode_memory_t *	
Number of available node cards	RM_BPAvailableNodeCards	int *	
Number of available I/O Nodes	RM_BPNumberIONodes	int *	

Table 11-6 shows the values that are set in the base partition object using `rm_set_data()`.

Table 11-6 Values set in a base partition object using `rm_set_data()`

Description	Specification	Argument type	Notes
Base partition identifier	RM_BPID	rm_bp_id_t	free required

Node card list object

The node card list object (`rm_nodecard_list_t`) contains a list of node card objects. The node card list object is retrieved by calling the `rm_get_nodecards()` API for a given base partition. See Table 11-7.

Table 11-7 Values retrieved from a node card list object using `rm_get_data()`

Description	Specification	Argument type	Notes
Number of node cards in the list	RM_NodeCardListSize	int *	
First node card in the list	RM_NodeCardListFirst	rm_nodecard_t **	
Next node card in the list	RM_NodeCardListNext	rm_nodecard_t **	

Node card object

The node card object (`rm_nodecard_t`) represents a node card within a base partition. The node card object is retrieved from the node card list object using the `RM_NodeCardListFirst` and `RM_NodeCardListNext` specifications. See Table 11-8.

Table 11-8 Values retrieved from a node card object using `rm_get_data()`

Description	Specification	Argument type	Notes
Node card identifier	<code>RM_NodeCardID</code>	<code>rm_nodecard_id_t *</code>	free required; possible values: N00..N15
The quadrant of the base partition that this node card is installed	<code>RM_NodeCardQuarter</code>	<code>rm_quarter_t *</code>	
Node card state	<code>RM_NodeCardState</code>	<code>rm_nodecard_state_t *</code>	
Sequence ID for the node card state	<code>RM_NodeCardStateSeqID</code>	<code>rm_sequence_id_t *</code>	
Number of I/O Nodes on the node card (can be 0, 1, or 2)	<code>RM_NodeCardIONodes</code>	<code>int *</code>	
Identifier of the partition associated with the node card	<code>RM_NodeCardPartID</code>	<code>pm_partition_id_t *</code>	free required. If no partition is associated, NULL is returned.
State of the partition associated with the node card	<code>RM_NodeCardPartState</code>	<code>rm_partition_state_t *</code>	
Sequence ID for the state of the partition associated with the node card	<code>RM_NodeCardPartStateSeqID</code>	<code>rm_sequence_id_t *</code>	
Flag indicating whether the node card is being used by a partition whose size is smaller than a node card	<code>RM_NodeCardSDB</code>	<code>int *</code>	0=No 1=Yes
Number of I/O Nodes in a list	<code>RM_NodeCardIONodeNum</code>	<code>int *</code>	
First I/O Node in the node card	<code>RM_NodeCardFirstIONode</code>	<code>rm_ionode_t **</code>	
Next I/O Node in the node card	<code>RM_NodeCardNextIONode</code>	<code>rm_ionode_t **</code>	

Table 11-9 shows the values that are set in a node card object when using `rm_set_data()`.

Table 11-9 Values set in a node card object using `rm_set_data()`

Description	Specification	Argument type	Notes
Node card identifier	<code>RM_NodeCardID</code>	<code>rm_nodecard_id_t</code>	
Number of I/O Nodes in list	<code>RM_NodeCardIONodeNum</code>	<code>int *</code>	
First I/O Node in the node card	<code>RM_NodeCardFirstIONode</code>	<code>rm_ionode_t *</code>	
Next I/O Node in the node card	<code>RM_NodeCardNextIONode</code>	<code>rm_ionode_t *</code>	

I/O Node object

The I/O Node object (`rm_ionode_t`) represents an I/O Node within a node card. The I/O Node object is retrieved from the node card object using the `RM_NodeCardFirstIONode` and `RM_NodeCardNextIONode` specifications. See Table 11-10.

Table 11-10 Values retrieved from an I/O Node object using `rm_get_data()`

Description	Specification	Argument type	Notes
I/O Node identifier	<code>RM_IONodeID</code>	<code>rm_ionode_id_t *</code>	Possible values: J00, J01; free required
Node card identifier	<code>RM_IONodeNodeCardID</code>	<code>rm_nodocard_id_t *</code>	Possible values: N00..N15; free required
IP address	<code>RM_IONodeIPAddress</code>	<code>char **</code>	free required
MAC address	<code>RM_IONodeMacAddress</code>	<code>char **</code>	free required
Identifier of the partition associated with the I/O Node	<code>RM_IONodePartID</code>	<code>pm_partition_id_t *</code>	free required. If no partition is associated with this I/O Node, NULL is returned.
State of the partition associated with the I/O Node	<code>RM_IONodePartState</code>	<code>rm_partition_state_t *</code>	
Sequence ID for the state of the partition associated with the I/O Node	<code>RM_IONodePartStateSeqID</code>	<code>rm_sequence_id_t *</code>	

Table 11-11 shows the values that are set in an I/O Node object by using `rm_set_data()`.

Table 11-11 Values set in an I/O Node object using `rm_set_data()`

Description	Specification	Argument type	Notes
I/O Node identifier	<code>RM_IONodeID</code>	<code>rm_ionode_id_t</code>	Possible values: J00, J01

Switch object

The switch object (`rm_switch_t`) represents a switch in the Blue Gene system. The switch object is retrieved from the following specifications:

- ▶ The Blue Gene object using the `RM_FirstSwitch` and `RM_NextSwitch` specifications
- ▶ The partition object using the `RM_PartitionFirstSwitch` and `RM_PartitionNextSwitch` specifications

Table 11-12 shows the values that are retrieved from a switch object using `rm_get_data()`.

Table 11-12 Values retrieved from a switch object using `rm_get_data()`

Description	Specification	Argument type	Notes
Switch identifier	RM_SwitchID	rm_switch_id_t *	free required
Identifier of the base partition connected to the switch	RM_SwitchBPID	rm_BP_id_t *	free required
Switch state	RM_SwitchState	rm_switch_state_t *	
Sequence ID for the switch state	RM_SwitchStateSeqID	rm_sequence_id_t *	
Switch dimension	RM_SwitchDim	rm_dimension_t *	Values: ▶ RM_DIM_X ▶ RM_DIM_Y ▶ RM_DIM_Z
Number of connections in the switch	RM_SwitchConnNum	int *	A connection is a pair of ports that are connected internally in the switch.
First connection in the list	RM_SwitchFirstConnection	rm_connection_t *	
Next connection in the list	RM_SwitchNextConnection	rm_connection_t *	

Table 11-13 shows the values that are set in a switch object using `rm_set_data()`.

Table 11-13 Values set in a switch object using `rm_set_data()`

Description	Specification	Argument type	Notes
Switch identifier	RM_SwitchID	rm_switch_id_t *	
Number of connections in the switch	RM_SwitchConnNum	int *	A connection is a pair of ports that are connected internally in the switch.
First connection in the list	RM_SwitchFirstConnection	rm_connection_t *	
Next connection in the list	RM_SwitchNextConnection	rm_connection_t *	

Wire object

The wire object (`rm_wire_t`) represents a wire in the Blue Gene/P system. The wire object is retrieved from the Blue Gene/P object using the `RM_FirstWire` and `RM_NextWire` specifications. See Table 11-14.

Table 11-14 Values retrieved from a wire object using `rm_get_data()`

Description	Specification	Argument type	Notes
Wire identifier	<code>RM_WireID</code>	<code>rm_wire_id_t *</code>	free required
Wire state	<code>RM_WireState</code>	<code>rm_wire_state_t *</code>	The state can be UP or DOWN.
Source port	<code>RM_WireFromPort</code>	<code>rm_port_t **</code>	
Destination port	<code>RM_WireToPort</code>	<code>rm_port_t **</code>	
Identifier of the partition associated with the wire	<code>RM_WirePartID</code>	<code>pm_partition_id_t *</code>	free required. If no partition is associated, NULL is returned.
State of the partition associated with the wire	<code>RM_WirePartState</code>	<code>rm_partition_state_t *</code>	
Sequence ID for the state of the partition associated with the wire	<code>RM_WirePartStateSeqID</code>	<code>rm_sequence_id_t *</code>	

Port object

The port object (`rm_port_t`) represents a port for a switch in the Blue Gene System. The port object is retrieved from the wire object using the `RM_WireFromPort` and `RM_WireToPort` specifications. See Table 11-15.

Table 11-15 Values retrieved from a port object using `rm_get_data()`

Description	Specification	Argument type	Notes
Identifier of the base partition or switch associated with the port	<code>RM_PortComponentID</code>	<code>rm_component_id_t *</code>	free required
Port identifier	<code>RM_PortID</code>	<code>rm_port_id_t *</code>	Possible values for base partitions: <code>plus_x minus_x</code> , <code>plus_y</code> , <code>minus_y</code> , <code>plus_z minus_z</code> . Possible values for switches: <code>s0..S5</code>

Partition list object

The partition list object (`rm_partition_list_t`) contains a list of partition objects. The partition list object is retrieved by calling the `rm_get_partitions()` or `rm_get_partitions_info()` API. See Table 11-16.

Table 11-16 Values retrieved from a partition list object using `rm_get_data()`

Description	Specification	Argument type	Notes
Number of partitions in the list	<code>RM_PartListSize</code>	<code>int *</code>	
First partition in the list	<code>RM_PartListFirstPart</code>	<code>rm_partition_t **</code>	
Next partition in the list	<code>RM_PartListNextPart</code>	<code>rm_partition_t **</code>	

Partition object

The partition object (`rm_partition_t`) represents a partition that is defined in the Blue Gene system. The partition object is retrieved from the partition list object using the `RM_PartListFirstPart` and `RM_PartListNextPart` specifications. A new partition object is created using the `rm_new_partition()` API. After setting the appropriate fields in a new partition object, the partition can be added to the system using the `rm_add_partition()` API. See Table 11-17.

Table 11-17 Values retrieved from a partition object using `rm_get_data()`

Description	Specification	Argument type	Notes
Partition identifier	<code>RM_PartitionID</code>	<code>pm_partition_id_t *</code>	free required
Partition state	<code>RM_PartitionState</code>	<code>rm_partition_state_t *</code>	
Sequence ID for the partition state	<code>RM_PartitionStateSeqID</code>	<code>rm_sequence_id_t *</code>	
Connection type of the partition	<code>RM_PartitionConnection</code>	<code>rm_connection_type_t *</code>	Values: TORUS or MESH
Partition description	<code>RM_PartitionDescription</code>	<code>char **</code>	free required
Flag indicating whether this partition is a partition smaller than the base partition	<code>RM_PartitionSmall</code>	<code>int *</code>	0=No 1=Yes
Number of used processor sets (psets) per base partition	<code>RM_PartitionPsetsPerBP</code>	<code>int *</code>	
Job identifier of the current job	<code>RM_PartitionJobID</code>	<code>int *</code>	If no job is currently on the partition, zero is returned
User name of the user who submitted the job	<code>RM_PartitionUserName</code>	<code>char **</code>	free required
Partition options	<code>RM_PartitionOptions</code>	<code>char **</code>	free required
File name of the machine loader image	<code>RM_PartitionMloaderImg</code>	<code>char **</code>	free required
Comma-separated list of images to load on the Compute Nodes	<code>RM_PartitionCnloadImg</code>	<code>char **</code>	free required
Comma-separated list of images to load on the I/O Nodes	<code>RM_PartitionIoloadImg</code>	<code>char **</code>	free required
Number of base partitions in the partition	<code>RM_PartitionBPNum</code>	<code>int *</code>	
First base partition in the partition	<code>RM_PartitionFirstBP</code>	<code>rm_BP_t **</code>	
Next base partition in the partition	<code>RM_PartitionNextBP</code>	<code>rm_BP_t **</code>	
Number of switches in the partition	<code>RM_PartitionSwitchNum</code>	<code>int *</code>	
First switch in the partition	<code>RM_PartitionFirstSwitch</code>	<code>rm_switch_t **</code>	
Next switch in the partition	<code>RM_PartitionNextSwitch</code>	<code>rm_switch_t **</code>	
Number of node cards in the partition	<code>RM_PartitionNodeCardNum</code>	<code>int *</code>	
First node card in the partition	<code>RM_PartitionFirstNodeCard</code>	<code>rm_nodocard_t **</code>	

Description	Specification	Argument type	Notes
Next node card in the partition	RM_PartitionNextNodeCard	rm_nodecard_t **	
Number of users of the partition	RM_PartitionUsersNum	int *	
First user name for the partition	RM_PartitionFirstUser	char **	free required
Next user name for the partition	RM_PartitionNextUser	char **	free required

Table 11-18 shows the values that are set in a partition object using `rm_set_data()`.

Table 11-18 Values set in a partition object using `rm_set_data()`

Description	Specification	Argument type	Notes
Partition identifier	RM_PartitionID	pm_partition_id_t	Up to 32 characters for a new partition ID, or up to 16 characters followed by an asterisk (*) for a prefix for a unique name
Connection type of the partition	RM_PartitionConnection	rm_connection_type_t *	Values: TORUS or MESH
Partition description	RM_PartitionDescription	char *	
Flag indicating whether this partition is a partition smaller than the base partition	RM_PartitionSmall	int *	0=No 1=Yes
Number of used processor sets (psets) per base partition	RM_PartitionPsetsPerBP	int *	
User name who submitted the job	RM_PartitionUserName	char *	
File name of the machine loader image	RM_PartitionMloaderImg	char *	
Comma-separated list of images to load on the Compute Nodes	RM_PartitionCnloadImg	char *	
Comma-separated list of images to load on the I/O Nodes	RM_PartitionIoloadImg	char *	
Number of base partitions in the partition	RM_PartitionBPNum	int *	
First base partition in the partition	RM_PartitionFirstBP	rm_BP_t *	
Next base partition in the partition	RM_PartitionNextBP	rm_BP_t *	
Number of switches in the partition	RM_PartitionSwitchNum	int *	
First switch in the list in the partition	RM_PartitionFirstSwitch	rm_switch_t *	
Next switch in the partition	RM_PartitionNextSwitch	rm_switch_t *	

Description	Specification	Argument type	Notes
Number of node cards in the partition	RM_PartitionNodeCardNum	int *	
First node card in the partition	RM_PartitionFirstNodecard	rm_nodecard_t *	
Next node card in the partition	RM_PartitionNextNodecard	rm_nodecard_t *	

Job list object

The job list object (`rm_job_list_t`) contains a list of job objects. The job list object is retrieved by calling the `rm_get_jobs()` API. See Table 11-19.

Table 11-19 Values retrieved from a job list object using `rm_get_data()`

Description	Specification	Argument type	Notes
Number of jobs in the list	RM_JobListSize	int *	
First job in the list	RM_JobListFirstJob	rm_job_t **	
Next job in the list	RM_JobListNextJob	rm_job_t **	

Job object

The job object (`rm_job_t`) represents a job defined in the Blue Gene system. The job object is retrieved from the job list object using the `RM_JobListFirstJob` and `RM_JobListNextJob` specifications. A new job object is created using the `rm_new_job()` API. After setting the appropriate fields in a new job object, the job can be added to the system using the `rm_add_job()` API. See Table 11-20.

Table 11-20 Values retrieved from a job object using `rm_get_data()`

Description	Specification	Argument type	Notes
Job identifier	RM_JobID	rm_job_id_t *	free required Identifier is unique across all jobs on the system.
Identifier of the partition assigned for the job	RM_JobPartitionID	pm_partition_id_t *	free required
Job state	RM_JobState	rm_job_state_t *	
Sequence ID for the job state	RM_JobStateSeqID	rm_sequence_id_t *	
Executable file name for the job	RM_JobExecutable	char **	free required
Name of the user who submitted the job	RM_JobUserName	char **	free required
Integer containing the ID given to the job by the database	RM_JobDBJobID	db_job_id_t *	
Job output file name	RM_JobOutFile	char **	free required
Job error file name	RM_JobErrFile	char **	free required
Job output directory name	RM_JobOutDir	char **	free required This directory contains the output files if a full path is not given.

Description	Specification	Argument type	Notes
Error text returned from the control daemons	RM_JobErrText	char **	free required
Arguments for the job executable	RM_JobArgs	char **	free required
Environment parameter needed for the job	RM_JobEnvs	char **	free required
Flag indicating whether the job was retrieved from the history table	RM_JobInHist	int *	0=No 1=Yes
Job mode	RM_JobMode	rm_job_mode_t *	Indicates Virtual Node, SMP, or DUAL mode
System call trace indicator for Compute Nodes	RM_JobStrace	rm_job_strace_t *	
Job start time The format is yyyy-mm-dd-hh.mm.ss.nnnnnn. If the job never went to running state, it will be an empty string. Data is only valid for completed jobs. The rm_get_data() specification RM_JobInHist can be used to determine whether a job has completed. If the job is an active job, then the value returned is meaningless.	RM_JobStartTime	char **	free required
Job end time Format is yyyy-mm-dd-hh.mm.ss.nnnnnn. Data is valid only for completed jobs. The rm_get_data() specification RM_JobInHist can be used to determine whether a job has completed. If the job is an active job, then the value returned is meaningless.	RM_JobEndTime	char **	free required
Job run time in seconds Data is only valid for completed jobs. The rm_get_data() specification RM_JobInHist can be used to determine whether a job has completed. If the job is an active job, then the value returned is meaningless.	RM_JobRunTime	rm_job_runtime_t *	

Description	Specification	Argument type	Notes
Number of Compute Nodes used by the job Data is only valid for completed jobs. The <code>rm_get_data()</code> specification <code>RM_JobInHist</code> can be used to determine whether a job has completed. If the job is an active job, then the value returned is meaningless.	<code>RM_JobComputeNodesUsed</code>	<code>rm_job_computenodes_used_t *</code>	
Job exit status Data is only valid for completed jobs. The <code>rm_get_data()</code> specification <code>RM_JobInHist</code> can be used to determine whether a job has completed. If the job is an active job, then the value returned is meaningless.	<code>RM_JobExitStatus</code>	<code>rm_job_exitstatus_t *</code>	
User UID	<code>RM_JobUserUid</code>	<code>rm_job_user_uid_t *</code>	Zero is returned when querying existing jobs.
User GID	<code>RM_JobUserGid</code>	<code>rm_job_user_gid_t *</code>	Zero is returned when querying existing jobs.

Table 11-21 shows the values that are set in a job object using `rm_set_data()`.

Table 11-21 Values set in a job object using `rm_set_data()`

Description	Specification	Argument type	Notes
Job identifier	<code>RM_JobID</code>	<code>rm_job_id_t</code>	This must be unique across all jobs on the system; if not, return code <code>JOB_ALREADY_DEFINED</code> is returned.
Partition identifier assigned for the job	<code>RM_JobPartitionID</code>	<code>pm_partition_id_t</code>	This field can be left blank when adding a new job to the system.
Executable file name for the job	<code>RM_JobExecutable</code>	<code>char *</code>	
Name of the user who submitted the job	<code>RM_JobUserName</code>	<code>char *</code>	
Job output file name	<code>RM_JobOutFile</code>	<code>char *</code>	
Job error file name	<code>RM_JobErrFile</code>	<code>char *</code>	
Job output directory	<code>RM_JobOutDir</code>	<code>char *</code>	This directory contains the output files if a full path is not given.
Arguments for the job executable	<code>RM_JobArgs</code>	<code>char *</code>	
Environment parameter needed for the job	<code>RM_JobEnvs</code>	<code>char *</code>	
Job mode	<code>RM_JobMode</code>	<code>rm_job_mode_t *</code>	Possible values: Virtual Node, SMP or DUAL mode

Description	Specification	Argument type	Notes
System call trace indicator for Compute Nodes	RM_JobStrace	rm_job_strace_t *	
User UID	RM_JobUserUid	rm_job_user_uid_t *	This value can be set when adding a job.
User GID	RM_JobUserGid	rm_job_user_gid_t *	This value can be set when adding a job.

11.2.9 Object allocator APIs

In this section, we describe the APIs that are used to allocate memory for objects used with other API calls:

- ▶ `status_t rm_new_BP(rm_BP_t **bp);`
Allocates storage for a new base partition object.
- ▶ `status_t rm_new_ionode(rm_ionode_t **io);`
Allocates storage for a new I/O Node object.
- ▶ `status_t rm_new_job(rm_job_t **job);`
Allocates storage for a new job object.
- ▶ `status_t rm_new_nodecard(rm_nodecard_t **nc);`
Allocates storage for a new node card object.
- ▶ `status_t rm_new_partition(rm_partition_t **partition);`
Allocates storage for a new partition object.
- ▶ `status_t rm_new_switch(rm_switch_t **switch);`
Allocates storage for a new switch object.

11.2.10 Object deallocator APIs

In this section, we describe the APIs that used to deallocate memory for objects that are created by other API calls:

- ▶ `status_t rm_free_BG(rm_BG_t *bg);`
Frees storage for a Blue Gene object.
- ▶ `status_t rm_free_BP(rm_BP_t *bp);`
Frees storage for a base partition object.
- ▶ `status_t rm_free_ionode(rm_ionode_t *io);`
Frees storage for an I/O Node object.
- ▶ `status_t rm_free_job(rm_job_t *job);`
Frees storage for a job object.
- ▶ `status_t rm_free_job_list(rm_job_list_t *job_list);`
Frees storage for a job list object.
- ▶ `status_t rm_free_nodecard(rm_nodecard_t *nc);`
Frees storage for a node card object.

- ▶ `status_t rm_free_nodocard_list(rm_nodocard_list_t *nc_list);`
Frees storage for a node card list object.
- ▶ `status_t rm_free_partition(rm_partition_t *partition);`
Frees storage for a partition object.
- ▶ `status_t rm_free_partition_list(rm_partition_list_t *part_list);`
Frees storage for a partition list object.
- ▶ `status_t rm_free_switch(rm_switch_t *switch);`
Frees storage for a switch object.

11.2.11 Messaging APIs

In this section, we describe the set of thread-safe Messaging APIs. These APIs are used by the Bridge as well as by other components of the job management system, such as the `mpirun` program that ships with the Blue Gene software. Each message is written using the following format:

```
<Timestamp> Component (Message type): Message text
```

Here is an example:

```
<Mar 9 04:24:30> BRIDGE (Debug): rm_get_BG()- Completed Successfully
```

The message can be one of the following types:

- ▶ `MESSAGE_ERROR`: Error messages
- ▶ `MESSAGE_WARNING`: Warning messages
- ▶ `MESSAGE_INFO`: Informational messages
- ▶ `MESSAGE_DEBUG1`: Basic debug messages
- ▶ `MESSAGE_DEBUG2`: More detailed debug messages
- ▶ `MESSAGE_DEBUG3`: Very detailed debug messages

The following verbosity levels, to which the messaging APIs can be configured, define the policy:

- ▶ Level 0: Only error or warning messages are issued.
- ▶ Level 1: Level 0 messages and informational messages are issued.
- ▶ Level 2: Level 1 messages and basic debug messages are issued.
- ▶ Level 3: Level 2 messages and more debug messages are issued.
- ▶ Level 4: The highest verbosity level. All messages that will be printed are issued.

By default, only error and warning messages are written. To have informational and minimal debug messages written, set the verbosity level to 2. To obtain more detailed debug messages, set the verbosity level to 3 or 4.

In the following list, we describe the Message APIs:

- ▶ `int isSayMessageLevel(message_type_t m_type);`
Tests the current messaging level. Returns 1 if the specified message type is included in the current messaging level; otherwise returns 0.
- ▶ `void closeSayMessageFile();`
Closes the messaging log file.

Note: Any messaging output after calling this method is sent to `stderr`.

- ▶ `int sayFormattedMessage(FILE * curr_stream, const void * buf, size_t bytes);`
Logs a preformatted message to the messaging output without a time stamp.
- ▶ `void sayMessage(const char * component, message_type_t m_type, const char * curr_func, const char * format, ...);`
Logs a message to the messaging output.
The format parameter is a format string that specifies how subsequent arguments are converted for output. This value must be compatible with `printf` format string requirements.
- ▶ `int sayPlainMessage(FILE * curr_stream, const char * format, ...);`
Logs a message to the messaging output without a time stamp.
The format parameter is a format string that specifies how subsequent arguments are converted for output. This value must be compatible with the `printf` format string requirements.
- ▶ `void setSayMessageFile(const char* oldfilename, const char* newfilename);`
Opens a new file for messaging logging.

Note: This method can be used to atomically rotate log files.

- ▶ `void setSayMessageLevel(unsigned int level);`
Sets the messaging verbose level.
- ▶ `void setSayMessageParams(FILE * stream, unsigned int level);`
Uses the provided file for message logging and sets the logging level.

Note: This method has been deprecated in favor of the `setSayMessageFile()` and `setSayMessageLevel()` methods.

11.3 Small partition allocation

The base allocation unit in the Blue Gene/P system is a base partition. Partitions are composed of whole numbers of base partitions, except in two special cases concerning small partitions. A *small partition* is a partition that is comprised of a fraction of a base partition. Small partitions can be created in the following sizes:

- ▶ 16 Compute Nodes
A 16-node partition is comprised of 16 Compute Nodes from a single node card. The node card must have two installed I/O Nodes in order to be used for a 16-node partition.
- ▶ 32 Compute Nodes
A 32-node partition is comprised of all the Compute Nodes in a single node card. The node card must have at least one installed I/O Node in order to be used for a 32-node partition.
- ▶ 64 Compute Nodes
A 64-node partition is comprised of two adjacent node cards beginning with N00, N02, N04, N06, N08, N10, N12, or N14. The first node card in the pair must have at least one installed I/O Node in order to be used for a 64-node partition.

- ▶ 128 Compute Nodes

A 128-node partition is comprised of set of four adjacent node cards beginning with N00, N04, N08, or N12. The first node card in the set must have at least one installed I/O Node in order to be used for a 128-node partition.
- ▶ 256 Compute Nodes

A 256-node partition is comprised of a set of eight adjacent node cards beginning with N00 or N08. The first node card in the set must have at least one installed I/O Node in order to be used for a 256-node partition.

11.3.1 Subdivided busy base partitions

It is important that you understand the concept of *subdivided busy base partitions* when working with small partitions. A base partition is considered subdivided busy if at least one partition, defined for a subset of its node cards, is busy. A partition is busy if its state is not free (RM_PARTITION_FREE).

A base partition that is subdivided busy cannot be booted as a whole because some of its hardware is unavailable. A base partition can have small partitions and full midplane partitions (multiples of 512 Compute Nodes) defined for it in the database. If the base partition has small partitions defined, they do not have to be in use, and a full midplane partition can use the actual midplane. In this case, the partition name that is using the base partition is returned on the RM_BPPartID specification.

For small partitions, multiple partitions can use the same base partition. This is the subdivided busy (SDB) example. In this situation, the value returned for the RM_BPPartID specification is meaningless. You must use the RM_BPSDB specification to determine whether the base partition is subdivided busy (small partition in use).

11.4 API examples

In this section, we provide example API calls for several common situations.

11.4.1 Retrieving base partition information

The code in Example 11-2 retrieves the Blue Gene/P hardware information and prints some information about each base partition in the system.

Example 11-2 Retrieving base partition information

```
#include "rm_api.h"
int main(int argc, char *argv[]) {
    status_t rmmc;
    rm_BG_t *rmbg;
    int bpNum;
    enum rm_specification getOption;
    rm_BP_t *rmbp;
    rm_bp_id_t bpid;
    rm_BP_state_t state;
    rm_location_t loc;
    rmmc = rm_set_serial("BGP");
    rmmc = rm_get_BG(&rmbg);
    if (rmmc) {
        printf("Error occured calling rm_get_BG: %d\n", rmmc);
        return -1;
    }
}
```



```

    }
    rm_get_data(rmbg, RM_BPNum, &bpNum);
    printf("Number of base partitions: %d\n", bpNum);
    getOption = RM_FirstBP;
    for (int ii = 0; ii < bpNum; ++ii) {
        rm_get_data(rmbg, getOption, &rmbp);
        rm_get_data(rmbp, RM_BPID, &bpid);
        rm_get_data(rmbp, RM_BPState, &state);
        rm_get_data(rmbp, RM_BPLoc, &loc);
        printf("    BP %s with state %d at location <%d,%d,%d>\n", bpid, state, loc.X,
loc.Y, loc.Z);
        free(bpid);
        getOption = RM_NextBP;
    }
    rm_free_BG(rmbg); // Deallocate memory from rm_get_BG()
}

```

The example code can be compiled and linked with the commands shown in Figure 11-5.

```

g++ -m64 -pthread -I/bgsys/drivers/ppcfloor/include -c sample1.cc -o sample1.o_64
g++ -m64 -pthread -o sample1 sample1.o_64 -L/bgsys/drivers/ppcfloor/lib64 -lbgpbridge

```

Figure 11-5 Example compile and link commands

11.4.2 Retrieving node card information

The code in Example 11-3 shows how to retrieve information about the node cards for a base partition. The `rm_get_nodecards()` function retrieves a list of all the node cards in a base partition. The list always contains exactly 16 node cards.

Example 11-3 Retrieving node card information

```

int getNodeCards(rm_bp_id_t bpid) {
    int rmmc;
    rm_nodecard_list_t *ncList;
    int ncNum;
    enum rm_specification getOption;
    rm_nodecard_t *rmnc;
    rm_nodecard_id_t ncid;
    rm_nodecard_state_t ncState;
    int ioNum;
    rmmc = rm_get_nodecards(bpid, &ncList);
    if (rmmc) {
        printf("Error occured calling rm_get_nodecards: %d\n", rmmc);
        return -1;
    }
    rmnc = rm_get_data(ncList, RM_NodeCardListSize, &ncNum);
    printf("    Base partition %s has %d nodecards\n", bpid, ncNum);
    getOption = RM_NodeCardListFirst;
    for (int ii = 0; ii < ncNum; ++ii) {
        rmnc = rm_get_data(ncList, getOption, &rmnc);
        rmnc = rm_get_data(rmnc, RM_NodeCardID, &ncid);
        rmnc = rm_get_data(rmnc, RM_NodeCardState, &ncState);
        rmnc = rm_get_data(rmnc, RM_NodeCardIONodes, &ioNum);
        printf("        Node card %s with state %d has %d I/O nodes\n", ncid, ncState,
ioNum);
        free(ncid);
    }
}

```

```

        getOption = RM_NodeCardListNext;
    }
    rm_free_nodecard_list(ncList);
}

```

11.4.3 Defining a new small partition

Example 11-4 contains pseudo code that shows how to allocate a new small partition.

Example 11-4 Allocating a new small partition

```

int isSmall = 1;

rm_new_partition(&newpart); //Allocate space for new partition

// Set the descriptive fields
rm_set_data(newpart, RM_PartitionUserName, username);
rm_set_data(newpart, RM_PartitionMloaderImg, BGP_MLOADER_IMAGE);
rm_set_data(newpart, RM_PartitionCnloadImg, BGP_CNLOAD_IMAGE);
rm_set_data(newpart, RM_PartitionIoloadImg, BGP_IOLOAD_IMAGE);
rm_set_data(newpart, RM_PartitionSmall, &isSmall); // Mark partition as a small partition

// Add a single BP
rm_new_BP(rm_BP_t **BP);
rm_set_data(BP, RM_BPID, "R01-M0");
rm_set_data(newpart, RM_PartitionFirstBP, BP);

// Add the node card(s) comprising the partition
ncNum = 4; // The number of node cards is 4 for 128 compute nodes
rm_set_data(newpart, RM_PartitionNodeCardNum, &ncNum); // Set the number of node cards
for (1 to ncNum) {
    // all four node cards must belong to same quarter!
    rm_new_nodecard(rm_nodecard_t **nc); // Allocate space for new node card
    rm_set_data(nc, RM_NodeCardID, ncid);
    rm_set_data(newpart, RM_PartitionFirstNodeCard, nc); // Add the node card to the
partition
    or
    rm_set_data(newpart, RM_PartitionNextNodeCard, nc);
    rm_free_nodecard(nc);
}

rm_add_partition(newpart);

```

11.4.4 Querying a small partition

Example 11-5 contains pseudo code that shows how to query a small partition for its node cards.

Example 11-5 Querying a small partition

```
rm_get_partition(part_id, &mypart); // Get the partition
rm_get_data(mypart, RM_PartitionSmall, &small); // Check if this is a "small" partition
if (small) {
    rm_get_data(mypart, RM_PartitionFirstBP, &BP); // Get the First (and only) BP
    rm_get_data(mypart, RM_PartitionNodeCardNum, &nc_num); // Get the number of node cards

    for (1 to nc_num) {
        rm_get_data(mypart, RM_PartitionFirstNodeCard, &nc);
        or
        rm_get_data(mypart, RM_PartitionNextNodeCard, &nc);

        rm_get_data(nc, RM_NodeCardID, &ncid); // Get the id
        rm_get_data(nc, RM_NodeCardQuarter, &quarter); // Get the quarter
        rm_get_data(nc, RM_NodeCardState, &state); // Get the state
        rm_get_data(nc, RM_NodeCardIONodes, &ionodes); // Get num of I/O nodes
        rm_get_data(nc, RM_NodeCardPartID, &partid); // Get the partition ID
        rm_get_data(nc, RM_NodeCardPartState, &partstate); // Get the partition state

        print node card information
    }
}
```



Real-time Notification APIs

With the Blue Gene/P system, two programming models can handle state transitions for jobs, blocks, base partitions, switches, and node cards. The first model is based on a polling model, where the Bridge application programming interface (API) caller is responsible for the continuous polling of state information. The second model consists of Real-time Notification APIs that allow callers to register for state transition event notifications.

The Real-time Notification APIs are designed to eliminate the need for a resource management system to constantly have to read in all of the machine state in order to detect changes. The APIs allow the caller to be notified in real-time of state changes to jobs, blocks and hardware, such as base partitions, switches, and node cards. After a resource management application has obtained an initial snapshot of the machine state using the Bridge APIs, the Bridge APIs can then determine to only get notified of changes, and the Real-time Notification APIs provides that mechanism.

In this chapter, we describe the thread-safe Real-time Notification APIs for the Blue Gene/P system that can be used by a resource management application.

12.1 API support overview

In the following sections, we present an overview of the support provided by the APIs.

12.1.1 Requirements

There are several requirements for writing programs to the Real-time Notification APIs as explained in the following sections:

- ▶ Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- ▶ When the application calls `rt_init`, the API looks for the `DB_PROPERTY` environment variable. The corresponding `db.properties` file indicates the port on which the real-time server is listening and that the real-time client will use to connect to the server. The environment variable should be set to point to the actual `db.properties` file location as follows:
 - On a bash shell

```
export DB_PROPERTY=/bgsys/drivers/ppcfloor/bin/db.properties
```
 - On a csh shell

```
setenv DB_PROPERTY /bgsys/drivers/ppcfloor/bin/db.properties
```
- ▶ C and C++ are supported with the GNU gcc 4.1.1 level compilers. For more information and downloads, refer to the following Web address:
<http://gcc.gnu.org/>
- ▶ The include file is `/bgsys/drivers/ppcfloor/include/rt_api.h`.
- ▶ Only 64-bit shared library support is provided. Link your real-time application with the file `/bgsys/drivers/ppcfloor/lib64/libbgrealtime.so`.
Both the include and shared library files are installed as part of the standard system installation. They are contained in the `bgpbase.rpm` file.

Makefile excerpt

Example 12-1 shows a possible excerpt from a makefile that you might want to create to help automate builds of your application. This sample is shipped in the `/bgsys/drivers/ppcfloor/doc/realtime/simple/Makefile` directory. In this makefile, the program that is being built is `rt_sample_app`, and the source is in the `rt_sample_app.cc` file.

Example 12-1 Makefile excerpt

```
...
ALL_APPS = rt_sample_app

CXXFLAGS += -w -Wall -g -m64 -pthread
CXXFLAGS += -I/bgsys/drivers/ppcfloor/include

LDFLAGS += -L/bgsys/drivers/ppcfloor/lib64 -lbgrealtime
LDFLAGS += -pthread

.PHONY: all clean default distclean

default: $(ALL_APPS)

all: $(ALL_APPS)
```

```
clean:
    $(RM) $(ALL_APPS) *.o

distclean: clean

...
```

Real-time server

Before using these functions, the Blue Gene/P administrator must start the real-time server. Otherwise the `rt_init` API returns an `RT_CONNECTION_ERROR` status code. Configuring and starting the real-time server is documented in *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417.

12.1.2 General comments

All of the real-time APIs have general considerations that apply to all calls. We highlight the common features here:

- ▶ All the API calls return an `rt_status_t`, which indicates either success or a status code. Successful status codes are non-negative, where failure status codes are negative.
- ▶ All the API calls take a pointer to `rt_handle_t`, which is an opaque structure that represents a stream of real-time messages.
- ▶ The real-time APIs use `sayMessage` APIs for printing debug and error messages. The application should set the `sayMessage` logging level before calling the real-time APIs.

Blocking mode versus non-blocking mode

A real-time handle can be in *blocking* or *non-blocking* mode. In blocking mode, the `rt_request_realtime` API blocks until it can send the request, and the `rt_read_msgs` API blocks until there is an event to receive. In non-blocking mode, the `rt_request_realtime` API returns `RT_WOULD_BLOCK` if it cannot send the request. If you get this return code from `rt_request_realtime`, you must call it again until it returns `RT_FINISHED_PREV`. In non-blocking mode, the `rt_read_msgs` API returns `RT_NO_REALTIME_MSGS` immediately if no real-time event is ready to be processed.

The `rt_get_socket_descriptor` API can be used to get a file descriptor that can be used with a `select()`-type function to wait for a real-time event when a handle is in blocking mode.

The initial blocking or non-blocking mode is set using the `rt_init` API. An initialized handle can be set to blocking mode by using the `rt_set_blocking` API or set to non-blocking mode by using the `rt_set_nonblocking` API.

Filtering events

A real-time handle can be configured so that only partition events that affect certain partitions, job events, or both that affect certain jobs are passed to the application.

Setting the partition filter is done by using the `rt_set_filter` API with `RT_PARTITION` as the `filter_type` parameter. The `filter_names` parameter can specify one or more partition IDs separated by spaces. When the `rt_get_msgs` API is called, partition events are delivered only to the application if the partition ID matches any of the partition IDs in the filter. If the `filter_names` parameter is set to `NULL`, then the partition filter is removed and all partition events are delivered to the application. An example of the value to use for the `filter_names` parameter for partition IDs `R00-M0` and `R00-M1` is `R00-M0 R00-M1`.

Setting the job filter is done by using the `rt_set_filter` API with `RT_JOB` as the `filter_type` parameter. The `filter_names` parameter can specify one or more job IDs (as strings) separated by spaces. When the `rt_get_msgs` API is called, job events are delivered only to the application if the job ID matches any of the job IDs in the filter. If the `filter_names` parameter is set to `NULL`, then the job filter is removed and all job events are delivered to the application. An example of the value to use for the `filter_names` parameter for job IDs 10030 and 10031 is `10030 10031`.

The other use of the `rt_set_filter` API is to remove both types of filter by passing `RT_CLEAR_ALL` in the `filter_type` parameter.

12.2 Real-time Notification APIs

In this section, we describe the Real-time Notification APIs:

- ▶ `rt_status_t rt_init(rt_handle_t **handle_out, rt_block_flag_t blocking_flag, rt_callbacks_t* callbacks);`
Initializes a real-time handle. This function gets port of the real-time server from the `db.properties` file. The name of the `db.properties` file must be in the `DB_PROPERTY` environment variable, or `RT_DB_PROPERTY_ERROR` will be returned.
If this function is successful, `*handle_out` is set to a valid handle that is connected to the real-time server. The blocking state for the handle is set based on the blocking flag parameter. The callbacks for the handle are set to the `callbacks` parameter. If this function is not successful and `handle_out` is not `NULL`, then `*handle_out` is set to `NULL`.
- ▶ `rt_status_t rt_close(rt_handle_t **handle);`
Closes a real-time handle. The handle must not be used after calling this function.
- ▶ `rt_status_t rt_set_blocking(rt_handle_t **handle);`
Sets a real-time handle to blocking mode.
- ▶ `rt_status_t rt_set_nonblocking(rt_handle_t **handle);`
Sets a real-time handle to non-blocking mode.
- ▶ `rt_status_t rt_set_filter(rt_handle_t **handle, rt_filter_type_t filter_type, const char* filter_names);`
Sets the filter on a real-time handle. The filter names consists of a C-style string that contains a space-separated list of names to filter on. If removing filter entries, then set this to `NULL`. For filtering on partition names, consider this example of `R01-M0 R02-M1 R03`.
- ▶ `rt_status_t rt_request_realtime(rt_handle_t **handle);`
Requests real-time events for this handle. If this function returns `RT_WOULD_BLOCK`, then the request has not been sent. Call this function again until it returns `RT_FINISHED_PREV`, which indicates that the previous request has been sent.
If this function returns `RT_FINISHED_PREV`, then a new request was not sent.
- ▶ `rt_status_t rt_get_socket_descriptor(rt_handle_t **handle, int *sd_out);`
Gets the socket descriptor that is used by the real-time APIs. You can use this socket descriptor with the `select()` or `poll()` Linux APIs to wait until a real-time message is ready to be read. Other file or socket descriptor APIs, such as `close()`, should not be used on the socket descriptor returned by this API.
- ▶ `rt_status_t rt_read_msgs(rt_handle_t **handle, void* data);`
Receives real-time events on a handle. If the handle is blocking, this function blocks as long as there are no events waiting. If the handle is non-blocking, the function returns

immediately with `RT_NO_REALTIME_MSGS` if no events are waiting. If an event is waiting to be processed, the callback associated with the event type is called. If the callback returns `RT_CALLBACK_CONTINUE`, then events continue to be processed.

12.3 Real-time callback functions

Developers who use the Real-time Notification APIs must write functions that will be called when real-time events are received. These functions are callback functions because the application calls the `rt_read_msgs` API, which then calls the function that is supplied by the application.

Pointers to the callback functions must be set in an `rt_callbacks_t` structure. When a real-time event is received, the corresponding function is called using that pointer. The application passes its `rt_callbacks_t` into `rt_init`, which is stored for use when `rt_read_msgs` is called. If the pointer to the callback function in the `rt_callbacks_t` structure is `NULL`, then the event is discarded.

In addition to setting the callback functions in the `rt_callbacks_t` structure, the application must also set the version field to `RT_CALLBACK_VERSION_0`. With a later version of the real-time APIs, you can allow different callbacks and provide a different version for this field.

From inside your callback function, you cannot call a real-time API using the same handle on which the event occurred. Otherwise your application will deadlock.

The return type of the callback functions is an indicator of whether `rt_read_msgs` should continue to attempt to receive another real-time event on the handle or whether it should stop. If the callback function returns `RT_CALLBACK_CONTINUE`, then `rt_read_msgs` continues to attempt to receive real-time events. If the callback function returns `RT_CALLBACK_QUIT`, then `rt_read_msgs` does not attempt to receive another real-time event but returns `RT_STATUS_OK`.

Sequence identifiers (IDs) are associated with the state of each partition, job, base partition, node card, and switch. A state with a higher sequence ID is newer. If your application gets the state for an object from the Bridge APIs in addition to the real-time APIs, you will want to discard any state that has a lower sequence ID for the same object.

These APIs provide the *raw state* for partitions, jobs, base partitions, node cards, and switches in addition to the state. The raw state is the status value that is stored in the Blue Gene database as a single character, rather than the state enumeration that the Bridge APIs use. Several raw state values map to a single state value, so that your application might receive real-time event notifications where the state does not change but the raw state does. For example, the partition raw states of “A” (allocating), “C” (configuring), and “B” (booting) all map to the Bridge enumerated state of `RM_PARTITION_CONFIGURING`.

Real-time callbacks structure

In this section, we describe each of the callbacks that are available to applications in the `rt_callbacks_t` structure. We list each field of the structure along with the following information:

- ▶ The description of the event that causes the callback to be invoked
- ▶ The signature of the callback function
 - Your function must match the signature. Otherwise your program will fail to compile.
- ▶ A description of each argument to the callback function

Field end_cb

The field `end_cb` callback function is called when a real-time ended event occurs. Your application does not receive any more real-time events on this handle until you request real-time events from the server again by calling the `rt_request_realtime` API.

The function uses the following signature:

```
cb_ret_t my_rt_end(rt_handle_t **handle, void* extended_args, void* data);
```

Table 12-1 lists the arguments to the field `end_cb` function.

Table 12-1 Field end_cb

Argument	Description
handle	Real-time handle on which the event occurred
extended_args	Not used; will be NULL for now
data	Application data forwarded by <code>rt_read_msgs</code>

Field partition_added_cb

The field `partition_added_cb` function is called when a partition added event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_partition_added(rt_handle_t **handle, rm_sequence_id_t seq_id, pm_partition_id_t partition_id, rm_partition_state_t partition_new_state, rt_raw_state_t partition_raw_new_state, void* extended_args, void* data);
```

Table 12-2 lists the arguments to the field `partition_added_cb` function.

Table 12-2 Field partition_added_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for this partition's state
partition_id	The partition's ID
partition_new_state	The partition's new state
partition_raw_new_state	The partition's new raw state
extended_args	Not used; will be NULL for now
data	Application data forwarded by <code>rt_read_msgs</code>

Field partition_state_changed_cb

The field `partition_state_changed_cb` function is called when a partition state changed event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_partition_state_changed(rt_handle_t **handle, rm_sequence_id_t seq_id, rm_sequence_id_t previous_seq_id, pm_partition_id_t partition_id, rm_partition_state_t partition_new_state, rm_partition_state_t partition_old_state, rt_raw_state_t partition_raw_new_state, rt_raw_state_t partition_raw_old_state, void* extended_args, void* data);
```

Table 12-3 lists the arguments to the field `partition_state_changed_cb` function.

Table 12-3 Field `partition_state_changed_cb`

Argument	Description
<code>handle</code>	Real-time handle on which the event occurred
<code>seq_id</code>	Sequence ID for this partition's new state
<code>previous_seq_id</code>	Sequence ID for this partition's old state
<code>partition_id</code>	The partition's ID
<code>partition_new_state</code>	The partition's new state
<code>partition_old_state</code>	The partition's old state
<code>partition_raw_new_state</code>	The partition's new raw state
<code>partition_raw_old_state</code>	The partition's old raw state
<code>extended_args</code>	Not used; will be NULL for now
<code>data</code>	Application data forwarded by <code>rt_read_msgs</code>

Field `partition_deleted_cb`

The field `partition_deleted_cb` is called when a partition deleted event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_partition_deleted(rt_handle_t **handle, rm_sequence_id_t
previous_seq_id, pm_partition_id_t partition_id, void* extended_args, void* data);
```

Table 12-4 lists the arguments to the field `partition_deleted_cb` function.

Table 12-4 Field `partition_deleted_cb`

Argument	Description
<code>handle</code>	Real-time handle on which the event occurred
<code>previous_seq_id</code>	Sequence ID for this partition's state when removed
<code>partition_id</code>	The partition's ID
<code>extended_args</code>	Not used; will be NULL for now
<code>data</code>	Application data forwarded by <code>rt_read_msgs</code>

Field `job_added_cb`

The field `job_added_cb` function is called when a job added event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_job_added(rt_handle_t **handle, rm_sequence_id_t seq_id,
db_job_id_t job_id, pm_partition_id_t partition_id, rm_job_state_t job_new_state,
rt_raw_state_t job_raw_new_state, void* extended_args, void* data);
```

Table 12-5 lists the arguments to the field `job_added_cb` function.

Table 12-5 Field `job_added_cb`

Argument	Description
<code>handle</code>	Real-time handle on which the event occurred
<code>seq_id</code>	Sequence ID for the job's state
<code>job_id</code>	The new job's ID
<code>partition_id</code>	ID of the partition to which the job is assigned
<code>job_new_state</code>	The job's new state
<code>job_raw_new_state</code>	The job's new raw state
<code>extended_args</code>	Not used; will be NULL for now
<code>data</code>	Application data forwarded by <code>rt_read_msgs</code>

Field `job_state_changed_cb`

The field `job_state_changed_cb` function is called when a job state changed event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_job_state_changed(rt_handle_t **handle, rm_sequence_id_t
seq_id, rm_sequence_id_t previous_seq_id, db_job_id_t job_id, pm_partition_id_t
partition_id, rm_job_state_t job_new_state, rm_job_state_t job_old_state,
rt_raw_state_t job_raw_new_state, rt_raw_state_t job_raw_old_state, void*
extended_args, void* data);
```

Table 12-6 lists the arguments to the field `job_state_changed_cb` function.

Table 12-6 Field `job_state_changed_cb`

Argument	Description
<code>handle</code>	Real-time handle on which the event occurred
<code>seq_id</code>	Sequence ID for the job's new state
<code>previous_seq_id</code>	Sequence ID of the job's previous state
<code>job_id</code>	The job's ID
<code>partition_id</code>	ID of the partition to which the job is assigned
<code>job_new_state</code>	The job's new state
<code>job_old_state</code>	The job's old state
<code>job_raw_new_state</code>	The job's new raw state
<code>job_raw_old_state</code>	The job's old raw state
<code>extended_args</code>	Not used; will be NULL for now
<code>data</code>	Application data forwarded by <code>rt_read_msgs</code>

Field `job_deleted_cb`

The field `job_deleted_cb` function is called when a job-deleted event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_job_deleted(rt_handle_t **handle, rm_sequence_id_t previous_seq_id,
db_job_id_t job_id, pm_partition_id_t partition_id, void* extended_args, void*
data);
```

Table 12-7 lists the arguments to the field `job_deleted_cb` function.

Table 12-7 Field `job_deleted_cb`

Argument	Description
handle	Real-time handle on which the event occurred
previous_seq_id	Sequence ID of the job's previous state
job_id	The deleted job's ID
partition_id	ID of the partition to which the job was assigned
extended_args	Not used; will be NULL for now
data	Application data forwarded by <code>rt_read_msgs</code>

Field `bp_state_changed_cb`

The field `bp_state_changed_cb` is called when a base partition state changed event occurs.

The function uses the following signature:

```
cb_ret_t (*rt_BP_state_changed_fn_p)(rt_handle_t **handle, rm_sequence_id_t
seq_id, rm_sequence_id_t previous_seq_id, rm_bp_id_t bp_id, rm_BP_state_t
BP_new_state, rm_BP_state_t BP_old_state, rt_raw_state_t BP_raw_new_state,
rt_raw_state_t BP_raw_old_state, void* extended_args, void* data);
```

Table 12-8 lists the arguments to the field `bp_state_changed_cb` function.

Table 12-8 Field `bp_state_changed_cb`

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID of the base partition's new state
previous_seq_id	Sequence ID of the base partition's previous state
bp_id	The base partition's ID
BP_new_state	The base partition's new state
BP_old_state	The base partition's old state
BP_raw_new_state	The base partition's new raw state
BP_raw_old_state	The base partition's old raw state
extended_args	Not used; will be NULL for now
data	Application data forwarded by <code>rt_read_msgs</code>

Field switch_state_changed_cb

The field `switch_state_changed_cb` is called when a switch state changed event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_switch_state_changed(rt_handle_t **handle, rm_sequence_id_t seq_id,
rm_sequence_id_t previous_seq_id, rm_switch_id_t switch_id, rm_bp_id_t bp_id,
rm_switch_state_t switch_new_state, rm_switch_state_t switch_old_state,
rt_raw_state_t switch_raw_new_state, rt_raw_state_t switch_raw_old_state, void*
extended_args, void* data);
```

Table 12-9 lists the arguments to the field `switch_state_changed_cb` function.

Table 12-9 Field switch_state_changed_cb

Argument	Description
<code>handle</code>	Real-time handle on which the event occurred
<code>seq_id</code>	Sequence ID for the switch's new state
<code>previous_seq_id</code>	Sequence ID of the switch's previous state
<code>switch_id</code>	The switch's ID
<code>bp_id</code>	The switch's base partition's ID
<code>switch_new_state</code>	The switch's new state
<code>switch_old_state</code>	The switch's old state
<code>switch_raw_new_state</code>	The switch's new raw state
<code>switch_raw_old_state</code>	The switch's old raw state
<code>extended_args</code>	Not used; will be NULL for now
<code>data</code>	Application data forwarded by <code>rt_read_msgs</code>

Field nodecard_state_changed_cb

The field `nodecard_state_changed_cb` is called when a node card state changed event occurs.

The function uses the following signature:

```
b_ret_t my_rt_nodecard_state_changed(rt_handle_t **handle, rm_sequence_id_t
seq_id, rm_sequence_id_t previous_seq_id, rm_nodecard_id_t nodecard_id, rm_bp_id_t
bp_id, rm_nodecard_state_t nodecard_new_state, rm_nodecard_state_t
nodecard_old_state, rt_raw_state_t nodecard_raw_new_state, rt_raw_state_t
nodecard_raw_old_state, void* extended_args, void* data);
```

Table 12-10 lists the arguments to the field `nodecard_state_changed_cb` function.

Table 12-10 Field `nodecard_state_changed_cb`

Argument	Description
<code>handle</code>	Real-time handle on which the event occurred
<code>seq_id</code>	Sequence ID for the node card's new state
<code>previous_seq_id</code>	Sequence ID of the node card's previous state
<code>nodecard_id</code>	The node card's ID
<code>bp_id</code>	The node card's base partition's ID
<code>nodecard_new_state</code>	The node card's new state
<code>nodecard_old_state</code>	The node card's old state
<code>nodecard_raw_new_state</code>	The node card's new raw state
<code>nodecard_raw_old_state</code>	The node card's old raw state
<code>extended_args</code>	Not used; will be NULL for now
<code>data</code>	Application data forwarded by <code>rt_read_msgs</code>

12.4 Real-time Notification API status codes

When a failure occurs, an API invocation returns a status code. This status code helps apply automatic corrective actions within the resource management application. In addition, a failure always generates a log message, which provides more information for the possible cause of the problem and any corrective action. These log messages are used for debugging and non-automatic recovery of failures.

The design aims at striking a balance between the number of status codes detected and the different error paths per status code. Thus, some errors have specific status codes, while others have more generic ones.

The Real-time Notification API uses the following status codes:

- ▶ `RT_STATUS_OK`: API call completed successfully.
- ▶ `RT_NO_REALTIME_MSGS`: No events available.
- ▶ `RT_WOULD_BLOCK`: In non-blocking mode and request would block.
- ▶ `RT_FINISHED_PREV`: Previous request completed.
- ▶ `RT_CONNECTION_ERROR`: Connection to the real-time server failed.
- ▶ `RT_INTERNAL_ERROR`: Unexpected internal error. No recovery possible.
- ▶ `RT_INVALID_INPUT_ERROR`: The input to the API is bad due to missing required data, illegal data, and so on.
- ▶ `RT_DB_PROPERTY_ERROR`: Error trying to read the `db.properties` file.
- ▶ `RT_PROTOCOL_ERROR`: An incorrect message was received from the real-time server.
- ▶ `RT_HANDLE_CLOSED`: The handle passed to the API was previously closed.

12.4.1 Status code specification

The various API functions have the following status codes:

- ▶ `rt_status_t rt_init(rt_handle_t **handle_out, rt_block_flag_t blocking_flag, rt_callbacks_t* callbacks);`

This function initializes a real-time handle.

The status codes are:

- `RT_STATUS_OK`: The handle is initialized.
- `RT_INVALID_INPUT_ERROR`: One or more of the parameters are not valid.
- `RT_CONNECTION_ERROR`: Failed to connect to the real-time server.
- `RT_INTERNAL_ERROR`: There was an unexpected internal error in setting blocking or non-blocking mode on socket.
- `RT_DB_PROPERTY_ERROR`: Problem accessing the db.properties file.

- ▶ `rt_status_t rt_close(rt_handle_t **handle);`

This function closes a real-time handle.

The status codes are:

- `RT_STATUS_OK`: The handle was closed.
- `RT_INVALID_INPUT_ERROR`: The handle is not valid.
- `RT_INTERNAL_ERROR`: There was an unexpected internal error in closing the handle.

- ▶ `rt_status_t rt_set_blocking(rt_handle_t **handle);`

This function sets a real-time handle to blocking mode.

The status codes are:

- `RT_STATUS_OK`: Blocking mode was set for the handle.
- `RT_INVALID_INPUT_ERROR`: The handle is not valid.
- `RT_INTERNAL_ERROR`: There was an unexpected internal error in setting blocking mode.

- ▶ `rt_status_t rt_set_nonblocking(rt_handle_t **handle);`

This function sets a real-time handle to non-blocking mode.

The status codes are:

- `RT_STATUS_OK`: Non-blocking mode was set for the handle.
- `RT_INVALID_INPUT_ERROR`: The handle is not valid.
- `RT_INTERNAL_ERROR`: There was an unexpected internal error in setting non-blocking mode.

- ▶ `rt_status_t rt_set_filter(rt_handle_t **handle, rt_filter_type_t filter_type, const char* filter_names);`

This function sets the filter on a real-time handle.

The status codes are:

- `RT_STATUS_OK`: Filtering was set successfully.
- `RT_INVALID_INPUT_ERROR`: The handle is not valid.
- `RT_INTERNAL_ERROR`: There was an unexpected internal error in setting the filter.

- ▶ `rt_status_t rt_request_realtime(rt_handle_t **handle);`

This function requests real-time events for this handle.

The status codes are:

- `RT_STATUS_OK`: Request to start real-time updates was successful.
- `RT_INVALID_INPUT_ERROR`: The handle is not valid.
- `RT_CONNECTION_ERROR`: The connection to the server was lost.

- RT_WOULD_BLOCK: The handle is non-blocking and this request would block.
 - RT_FINISHED_PREV: A previous request finished.
- ▶ `rt_status_t rt_get_socket_descriptor(rt_handle_t **handle, int *sd_out);`
 This functions gets the socket descriptor used by the real-time APIs.
 The status codes are:
- RT_STATUS_OK: Socket descriptor was retrieved successfully.
 - RT_INVALID_INPUT_ERROR: The handle is not valid.
- ▶ `rt_status_t rt_read_msgs(rt_handle_t **handle, void* data);`
 This function receives real-time events.
 The status codes are:
- RT_STATUS_OK: Message or messages were read successfully.
 - RT_INVALID_INPUT_ERROR: The handle is not valid.
 - RT_CONNECTION_ERROR: The connection to the server was lost.
 - RT_INTERNAL_ERROR: There was an unexpected internal error when reading messages.
 - RT_HANDLE_CLOSED: The real-time handle was closed.
 - RT_NO_REALTIME_MSGS: Non-blocking mode and no messages to receive.

12.5 Sample real-time application code

Example 12-2 shows basic example code for calling the real-time APIs and programming the callback functions.

Example 12-2 Sample real-time call

```

/* ----- */
/*
/* (C)Copyright IBM Corp. 2007, 2007
/* All rights reserved.
/* US Government Users Restricted Rights -
/* Use, duplication or disclosure restricted
/* by GSA ADP Schedule Contract with IBM Corp.
/*
/* Licensed Materials-Property of IBM
/* ----- */
/*
  Title: Blue Gene Real-time Notification Interface - Sample Program

  The environment must have DB_PROPERTY set for the realtime apis.
*/

#include <rt_api.h>
#include <sayMessage.h>

#include <getopt.h>
#include <stdio.h>
#include <unistd.h>

#include <iostream>
#include <sstream>

using namespace std;
```

```

// Converts partition state enum to character string for messages
string partition_state_to_msg(rm_partition_state_t state)
{
    switch(state) {
        case RM_PARTITION_FREE:
            return "Free";
        case RM_PARTITION_CONFIGURING:
            return "Configuring";
        case RM_PARTITION_READY:
            return "Ready";
        case RM_PARTITION_DEALLOCATING:
            return "Deallocating";
        case RM_PARTITION_ERROR:
            return "Error";
        case RM_PARTITION_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
}

// Converts job state enum to character string for messages
string job_state_to_msg(rm_job_state_t state)
{
    switch(state) {
        case RM_JOB_IDLE:
            return "Queued/Idle";
        case RM_JOB_STARTING:
            return "Starting";
        case RM_JOB_RUNNING:
            return "Running";
        case RM_JOB_TERMINATED:
            return "Terminated";
        case RM_JOB_ERROR:
            return "Error";
        case RM_JOB_DYING:
            return "Dying";
        case RM_JOB_DEBUG:
            return "Debug";
        case RM_JOB_LOAD:
            return "Load";
        case RM_JOB_LOADED:
            return "Loaded";
        case RM_JOB_BEGIN:
            return "Begin";
        case RM_JOB_ATTACH:
            return "Attach";
        case RM_JOB_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
}

// Converts BP state enum to character string for messages
string BP_state_to_msg(rm_BP_state_t state)
{
    switch(state) {
        case RM_BP_UP:
            return "Available/Up";
        case RM_BP_MISSING:
            return "Missing";
    }
}

```

```

    case RM_BP_ERROR:
        return "Error";
    case RM_BP_DOWN:
        return "Service/Down";
    case RM_BP_NAV:
        return "Not a value (NAV)";
    }
    return "Unknown";
}

// Converts switch state enum to character string for messages
string switch_state_to_msg(rm_switch_state_t state)
{
    switch(state) {
        case RM_SWITCH_UP:
            return "Available/Up";
        case RM_SWITCH_MISSING:
            return "Missing";
        case RM_SWITCH_ERROR:
            return "Error";
        case RM_SWITCH_DOWN:
            return "Service/Down";
        case RM_SWITCH_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
}

// Converts nodecard state enum to character string for messages
string nodecard_state_to_msg(rm_nodecard_state_t state)
{
    switch(state) {
        case RM_NODECARD_UP:
            return "Available/Up";
        case RM_NODECARD_MISSING:
            return "Missing";
        case RM_NODECARD_ERROR:
            return "Error";
        case RM_NODECARD_DOWN:
            return "Service/Down";
        case RM_NODECARD_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
}

/* Definitions of the Real-time callback functions. */

cb_ret_t rt_end_callback(
    rt_handle_t **handle,
    void* extended_args,
    void* data)
{
    cout << "Received real-time end message." << endl;
    return RT_CALLBACK_QUIT;
}

cb_ret_t rt_partition_added_callback(

```

```

        rt_handle_t **handle,
        rm_sequence_id_t seq_id,
        pm_partition_id_t partition_id,
        rm_partition_state_t partition_new_state,
        rt_raw_state_t partition_raw_new_state,
        void* extended_args,
        void* data)
    {
        cout << "Received callback for add partition " << partition_id
              << " state of partition is " << partition_state_to_msg(partition_new_state) <<
endl
              << "Raw state=" << partition_raw_new_state << " sequence ID=" << seq_id << endl;
        return RT_CALLBACK_CONTINUE;
    }

cb_ret_t rt_partition_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    pm_partition_id_t partition_id,
    rm_partition_state_t partition_new_state,
    rm_partition_state_t partition_old_state,
    rt_raw_state_t partition_raw_new_state,
    rt_raw_state_t partition_raw_old_state,
    void* extended_args,
    void* data)
    {
        cout << "Received callback for partition " << partition_id
              << " state change, old state is " << partition_state_to_msg(partition_old_state)
              << ", new state is " << partition_state_to_msg(partition_new_state) << endl
              << "Raw old state=" << partition_raw_old_state
              << " Raw new state=" << partition_raw_new_state
              << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
endl;
        return RT_CALLBACK_CONTINUE;
    }

cb_ret_t rt_partition_deleted_callback(
    rt_handle_t **handle,
    rm_sequence_id_t prev_seq_id,
    pm_partition_id_t partition_id,
    void* extended_args,
    void* data)
    {
        cout << "Received callback for delete on partition " << partition_id
              << " Previous sequence ID=" << prev_seq_id << endl;
        return RT_CALLBACK_CONTINUE;
    }

cb_ret_t rt_job_added_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    db_job_id_t job_id,
    pm_partition_id_t partition_id,
    rm_job_state_t job_new_state,
    rt_raw_state_t job_raw_new_state,
    void* extended_args,

```

```

        void* data)
    {
        cout << "Received callback for add job " << job_id
              << " on partition " << partition_id << ", "
              << " state of job is " << job_state_to_msg(job_new_state) << endl
              << "Raw new state=" << job_raw_new_state << " New sequence ID=" << seq_id << endl;
        return RT_CALLBACK_CONTINUE;
    }

cb_ret_t rt_job_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    db_job_id_t job_id,
    pm_partition_id_t partition_id,
    rm_job_state_t job_new_state,
    rm_job_state_t job_old_state,
    rt_raw_state_t job_raw_new_state,
    rt_raw_state_t job_raw_old_state,
    void* extended_args,
    void* data)
{
    cout << "Received callback for job " << job_id
          << " state change on partition " << partition_id << ", old state is "
          << job_state_to_msg(job_old_state)
          << ", new state is " << job_state_to_msg(job_new_state) << endl
          << "Raw old state=" << job_raw_old_state << " Raw new state=" << job_raw_new_state
          << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_job_deleted_callback(
    rt_handle_t **handle,
    rm_sequence_id_t prev_seq_id,
    db_job_id_t job_id,
    pm_partition_id_t partition_id,
    void* extended_args,
    void* data)
{
    cout << "Received callback for delete of job " << job_id
          << " on partition " << partition_id << " Previous sequence ID=" << prev_seq_id <<
endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_BP_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    rm_bp_id_t bp_id,
    rm_BP_state_t BP_new_state,
    rm_BP_state_t BP_old_state,
    rt_raw_state_t BP_raw_new_state,
    rt_raw_state_t BP_raw_old_state,
    void* extended_args,
    void* data)

```

```

{
    cout << "Received callback for BP " << bp_id
          << " state change, old state is " << BP_state_to_msg(BP_old_state)
          << ", new state is " << BP_state_to_msg(BP_new_state) << endl
          << "Raw old state=" << BP_raw_old_state << " Raw new state=" << BP_raw_new_state
          << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_switch_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    rm_switch_id_t switch_id,
    rm_bp_id_t bp_id,
    rm_switch_state_t switch_new_state,
    rm_switch_state_t switch_old_state,
    rt_raw_state_t switch_raw_new_state,
    rt_raw_state_t switch_raw_old_state,
    void* extended_args,
    void* data)
{
    cout << "Received callback for switch " << switch_id
          << " state change on BP " << bp_id
          << ", old state is " << switch_state_to_msg(switch_old_state)
          << ", new state is " << switch_state_to_msg(switch_new_state) << endl
          << "Raw old state=" << switch_raw_old_state << " Raw new state=" <<
switch_raw_new_state
          << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_nodocard_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    rm_nodocard_id_t nodocard_id,
    rm_bp_id_t bp_id,
    rm_nodocard_state_t nodocard_new_state,
    rm_nodocard_state_t nodocard_old_state,
    rt_raw_state_t nodocard_raw_new_state,
    rt_raw_state_t nodocard_raw_old_state,
    void* extended_args,
    void* data)
{
    cout << "Received callback for node card " << nodocard_id
          << " state change on BP " << bp_id
          << ", old state is " << nodocard_state_to_msg(nodocard_old_state)
          << ", new state is " << nodocard_state_to_msg(nodocard_new_state) << endl
          << "Raw old state=" << nodocard_raw_old_state
          << " Raw new state=" << nodocard_raw_new_state
          << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
endl;
    return RT_CALLBACK_CONTINUE;
}

```

```

/* Program entry point */

int main( int argc, char *argv[] ) {

    string job_filter, *job_filter_p(0);
    string partition_filter, *partition_filter_p(0);
    int verbose(0);

    const int JOB_FILTER_PARAM_IND = 0;
    const int PARTITION_FILTER_PARAM_IND = 1;
    const int VERBOSE_PARAM_IND = 2;

    struct option long_options[] = {
        { "job_filter", 1, 0, JOB_FILTER_PARAM_IND },
        { "partition_filter", 1, 0, PARTITION_FILTER_PARAM_IND },
        { "verbose", 1, 0, VERBOSE_PARAM_IND },
        { 0, 0, 0, 0 }
    };

    int option_index = 0;

    while ( 1 ) {
        int getopt_ret = getopt_long(argc, argv, "", long_options, &option_index);
        if (-1 == getopt_ret) {
            break;
        }
        switch ( getopt_ret ) {
            case JOB_FILTER_PARAM_IND:
                job_filter = optarg;
                job_filter_p = &job_filter;
                break;
            case PARTITION_FILTER_PARAM_IND:
                partition_filter = optarg;
                partition_filter_p = &partition_filter;
                break;
            case VERBOSE_PARAM_IND:
                {
                    istringstream iss( optarg );
                    iss >> verbose;
                }
                break;
        }
    }

    setSayMessageParams(stdout, verbose);

    rt_handle_t *rt_handle;
    rt_callbacks_t rt_callbacks;

    rt_callbacks.version = RT_CALLBACK_VERSION_0;
    rt_callbacks.end_cb = &rt_end_callback;
    rt_callbacks.partition_added_cb = &rt_partition_added_callback;
    rt_callbacks.partition_state_changed_cb = &rt_partition_state_changed_callback;
    rt_callbacks.partition_deleted_cb = &rt_partition_deleted_callback;
    rt_callbacks.job_added_cb = &rt_job_added_callback;
    rt_callbacks.job_state_changed_cb = &rt_job_state_changed_callback;
    rt_callbacks.job_deleted_cb = &rt_job_deleted_callback;
    rt_callbacks.bp_state_changed_cb = &rt_BP_state_changed_callback;
    rt_callbacks.switch_state_changed_cb = &rt_switch_state_changed_callback;

```

```

rt_callbacks.nodiscard_state_changed_cb = &rt_nodiscard_state_changed_callback;

// Get a handle, set socket to block, and setup callbacks
if (rt_init(&rt_handle, RT_BLOCKING, &rt_callbacks) != RT_STATUS_OK)
{
    cout << "Failed on real-time initialize (rt_init), exiting program." << endl;
    return -1;
}

// Set the job filter if requested.
if (job_filter_p != 0) {
    if (rt_set_filter(&rt_handle, RT_JOB, job_filter_p->c_str()) != RT_STATUS_OK) {
        cout << "Failed to set job filter." << endl;
        rt_close(&rt_handle);
        return -1;
    }
}

// Set the partition filter if requested.
if (partition_filter_p != 0) {
    if (rt_set_filter(&rt_handle, RT_PARTITION, partition_filter_p->c_str()) !=
RT_STATUS_OK) {
        cout << "Failed to set partition filter." << endl;
        rt_close(&rt_handle);
        return -1;
    }
}

// Tell real-time server we are ready to handle messages
if (rt_request_realtime(&rt_handle) != RT_STATUS_OK)
{
    cout << "Failed to connect to real-time server, exiting program." << endl;
    rt_close(&rt_handle);
    return -1;
}

// Read messages
if (rt_read_msgs(&rt_handle, NULL) != RT_STATUS_OK)
{
    cout << "rt_read_msgs failed" << endl;
    rt_close(&rt_handle);
    return -1;
}

// Close the handle
rt_close(&rt_handle);
return 0;
} // main()

```



mpirun

mpirun is a software utility for launching, monitoring, and controlling programs (applications) that run on the BlueGene/ P system. **mpirun** on the Blue Gene/P system serves the same function as on the Blue Gene/L system.

The name **mpirun** comes from Message Passing Interface (MPI) since its primary use is to launch parallel jobs. This was certainly the case on the Blue Gene/L system. **mpirun** can be used as a stand-alone program by providing parameters either directly through a command line or environmental variable arguments, or indirectly through the framework of a scheduler that submits the job on the user's behalf. In the former case, **mpirun** can be invoked as a shell command and the user can interact with the running applications within the **mpirun** capabilities. In turn, **mpirun** acts as a shadow of the job by monitoring its status, as well as providing access to standard input, output, and error. After the job has terminated, **mpirun** terminates as well. If the user wants to prematurely end the job before it has terminated, **mpirun** provides a mechanism to do so explicitly or via a timeout period.

mpirun provides the capability to debug the job, currently only with **gdb**. In this chapter, we describe the *stand-alone interactive* use of **mpirun**. We also provide a brief overview of **mpirun** on the Blue Gene/P system. In addition, we define a list of application programming interfaces (APIs) that allow interaction with the **mpirun** program. These APIs are used by applications, such as external resource managers, that want to programmatically invoke jobs via **mpirun**.

We address the following topics in this chapter:

- ▶ “mpirun implementation on Blue Gene/P” on page 218
- ▶ “mpirun setup” on page 219
- ▶ “Invoking mpirun” on page 220
- ▶ “Environmental variables” on page 224
- ▶ “Return codes” on page 225
- ▶ “Examples” on page 227
- ▶ “mpirun application program interfaces” on page 234

13.1 mpirun implementation on Blue Gene/P

`mpirun` accepts a rich set of parameters, following the philosophy of the Blue Gene/L system, that describe its behavior prior to submitting the application for execution on the Compute Nodes and during execution of the application. These parameters can be divided into three groups. The first group identifies resources that are required to run the application. The second group identifies the application (binary) to execute and the environment settings for that particular run or executable. The third group identifies the level of verbosity that `mpirun` prints to `STDOUT` or `STDERR`.

Although `mpirun` has kept all the functionality that is available on the Blue Gene/L system, it has the following differences in its implementation on the Blue Gene/P system:

- ▶ The `rsh/ssh` mechanism has been eliminated for starting the back-end process due to security concerns of allowing users access to the Service Node. In the Blue Gene/P system, this is replaced with a daemon process that runs on the Service Node whose purpose is to handle connections from front-end `mpirun` processes and fork back-end `mpirun` processes as illustrated in Figure 12-1.

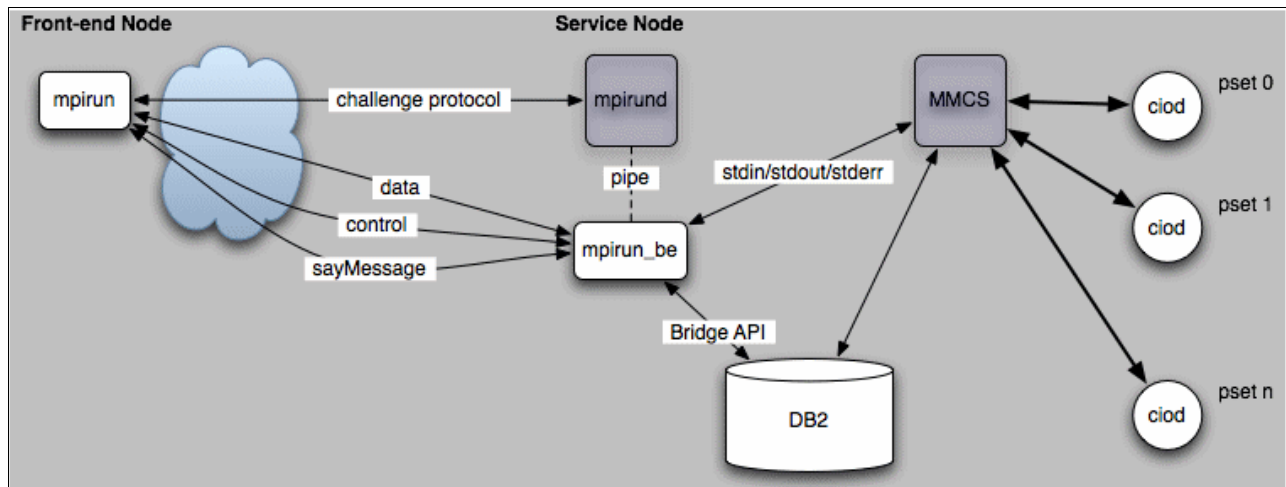


Figure 13-1 `mpirun` interacting with the rest of the control system on the Blue Gene/P system

- ▶ After `mpirun_be` is forked, the sequence of events for booting partitions, starting jobs, and collecting `stdout/stderr` is similar to the use of `mpirun` on the Blue Gene/L system.
- ▶ The `freepartition` program has been integrated as an option in `mpirun` for the Blue Gene/P system as illustrated in Example 13-1.

Example 13-1 mpirun example with -nofree option

```
mpirun -partition N01_32_1 -np 32 -cwd /bgusr/cpsosa -exe a.out -nofree
```

Example 13-2 shows how the `free` option is now used as part of `mpirun` on the Blue Gene/P system.

Example 13-2 *mpirun* example with *-free* option

```
[descartes:/bgusr/cpsosa/red/example.5_1] mpirun -partition N01_32_1 -free wait -verbose 1
<Jul 06 15:10:48.401421> FE_MPI (Info) : Invoking free partition
<Jul 06 15:10:48.414677> FE_MPI (Info) : freePartition() - connected to mpirun server at spinoza
<Jul 06 15:10:48.414768> FE_MPI (Info) : freePartition() - sent free partition request
<Jul 06 15:11:19.202335> FE_MPI (Info) : freePartition() - partition N01_32_1 was freed successfully
<Jul 06 15:11:19.202746> FE_MPI (Info) : == FE completed ==
<Jul 06 15:11:19.202790> FE_MPI (Info) : == Exit status: 0 ==
```

- ▶ Also new in **mpirun** for the Blue Gene/P system is the support for multiple program, multiple data (MPMD)³² style jobs where a different executable, arguments, environment, and current working directory can be supplied for a single job on a processor set (pset) basis. With this capability, a user can run four different executables on a partition with four psets.

This capability is handled by a new tool called **mpirexec**, which should not be confused with the **mpirexec** style of submitting a Single Program Multiple Data (SPMD) parallel MPI job.

13.2 mpirun setup

mpirun does not require setup from a user point of view or requires little setup up front. However, on the Service Node, **mpirun** requires slightly more setup for a system administrator. We have classified the setup of **mpirun** into the following types:

- ▶ User setup
- ▶ System administrator setup

13.2.1 User setup

In general, little has changed compared to **mpirun** for the Blue Gene/L system. The following changes are among those for user setup:

- ▶ It is not required to set up `.rhosts` or `ssh-agent`.
- ▶ It is not required to set up `.bashrc`, `.tcshrc`, or `.profile` to include `BRIDGE_CONFIG_FILE` or `DB_PROPERTY` environmentals.
- ▶ The **freepartition** program is now an option in **mpirun**.
- ▶ The `-backend` option is no longer available.

Due to the removal of the `ssh/rsh` mechanism to start a back-end **mpirun** process, users no longer are required to create an `.rhosts` file in their home directory for **mpirun** to work properly.

13.2.2 System administrator setup

System administrators can change the following configuration files for the **mpirun** daemon (**mpirund**):

db.properties	Contains information about the DB2 database
bridge.config	Contains locations of the default I/O Node and Compute Node images when allocating partitions
mpirun.cfg	Contains the shared secret that is used for challenge authentication between mpirun and mpirund

Database properties and Bridge configuration files

The location of the database properties and bridge configuration files can be changed by passing the appropriate arguments to `bgpmaster` when starting `mpirund`. The `mpirun` daemon then passes these locations to each `mpirun_be` forked. Example 13-3 shows a sample Bridge configuration file.

Example 13-3 Sample bridge configuration file

```
BGP_MACHINE_SN      BGP
BGP_MLOADER_IMAGE  /bgsys/drivers/ppcfloor/boot/uloader
BGP_CNLOAD_IMAGE
/bgsys/drivers/ppcfloor/boot/cns,/bgsys/drivers/ppcfloor/boot/cnk
BGP_ILOAD_IMAGE
/bgsys/drivers/ppcfloor/boot/cns,/bgsys/drivers/ppcfloor/boot/linux,/bgsys/drivers
/ppcfloor/boot/ramdisk
BGP_BOOT_OPTIONS
BGP_DEFAULT_CWD    $PWD
```

`BGP_DEFAULT_CWD` is used for `mpirun` jobs when a user does not give the `-cwd` argument or one of its environmentals. This value can be optionally changed to something more site specific, such as `/bgp/users`, `/gpfs/`, and so on. The special keyword `$PWD` is expanded to the user's current working directory from where they executed `mpirun`.

Challenge protocol

The challenge protocol, which is used to authenticate the `mpirun` front end when connecting to the `mpirun` daemon on the Service Node, is a challenge/response protocol. It uses a shared secret to create a hash of a random number, thereby verifying that the `mpirun` front end has access to the secret.

To protect the secret, the challenge protocol is stored in a configuration file that is accessible only by the `bgpadmin` user on the Service Node and by a special `mpirun` user on the front-end nodes. The front-end `mpirun` binary has its `setuid` flag enabled so that it can change its `uid` to match the `mpirun` user, and read the configuration file to access the secret. Several steps are necessary during the installation process for this setup to work.

13.3 Invoking mpirun

The first method of using `mpirun` is to specify the parameters explicitly as shown in the following example:

```
mpirun [options]
```

Here is a practical example of using `mpirun`:

```
mpirun -partition R00-M0 -mode VN -cwd /bgusr/tmp a.out --timeout 50
```

Alternatively, you can use the `mpiexec` style where the executable and arguments are implicit as shown in the following example:

```
mpirun [options] binary [arg1 arg2 ... argn]
```

Here is a practical example of using `mpiexec`:

```
mpirun -partition R00-M0 -mode VN -cwd /bgusr/tmp -exe a.out --args "--timeout 50"
```

Specifying parameters

You can specify parameters for the `mpirun` program in the following different ways:

- ▶ Command line arguments
- ▶ Environmental variables
- ▶ Scheduler interface plug-in

In general, users normally use the command line arguments and the environmental variables. Certain schedulers use the scheduler interface plug-in to restrict or enable `mpirun` features according to their environment. For example, the scheduler might have a policy where interactive job submission with `mpirun` can be allowed only during certain hours of the day.

Command line arguments

The `mpirun` arguments consist of the following categories:

- ▶ Job control
- ▶ Block control
- ▶ Output
- ▶ Other

Job control arguments

Table 13-1 lists the job control arguments.

Table 13-1 Job control arguments

Arguments	Description
<code>-args "program args"</code>	Passes "program args" to the BlueGene job on the Compute Nodes.
<code>-env "ENVVAR=value"</code>	Sets an environment variable in the environment of the job on the Compute Nodes.
<code>-exp_env ENVVAR</code>	Exports an environment variable in the current environment of <code>mpirun</code> to the job on the Compute Nodes.
<code>-env_all</code>	Exports all environment variables in the current environment of <code>mpirun</code> to the job on the Compute Nodes.
<code>-np <n></code>	Creates exactly <i>n</i> MPI ranks for the job. Aliases are <code>-nodes</code> and <code>-n</code> .
<code>-mode <SMP or DUAL or VN></code>	Specifies the mode in which the job will run. Choices are SMP (1 rank, 4 threads), DUAL (2 ranks, 2 threads each), or Virtual Node Mode (4 ranks, 1 thread each).
<code>-exe <executable></code>	Specifies the full path to the executable to run on the Compute Nodes. The path is specified as seen by the I/O and Compute Nodes.
<code>-cwd <path></code>	Specifies the full path to use as the current working directory on the Compute Nodes. The path is specified as seen by the I/O and Compute Nodes.
<code>-mapfile <mapfile></code>	Specifies an alternative MPI topology. The mapfile path must be fully qualified as seen by the I/O and Compute Nodes. ^a
<code>-timeout <n></code>	Timeout after <i>n</i> seconds. <code>mpirun</code> monitors the job and terminates it if the job runs longer than the time specified. The default is never to timeout.

a. For additional information about mapping, see Appendix E, "Mapping" on page 281.

Block control options

mpirun can also allocate partitions and create new partitions if necessary. Use the following general rules for block control:

- ▶ If **mpirun** is told to use a pre-existing partition and it is already booted, **mpirun** uses it as is without trying to boot it again.
- ▶ If **mpirun** creates a partition or is told to use a pre-existing partition that is not already allocated, **mpirun** allocates the partition.
- ▶ If **mpirun** allocates a partition, it deallocates the partition when it is done.

Table 13-2 summarizes the options that modify this behavior.

Table 13-2 Block control options

Arguments	Description
-partition <block>	Specifies a predefined block to use.
-nofree	If mpirun booted the block, it does not deallocate the block when the job is done. This is useful for when you want to run a string of jobs back-to-back on a block but do not want mpirun to boot and deallocate the block each time (which happens if you had not booted the block first using the console.) When your string of jobs is finally done, use the freepartition command to deallocate the block.
-free <wait nowait>	Frees the partition specified with -partition. No job is run. The wait parameter does not return control until the partition has changed state to free. The nowait parameter returns control immediately after submitting the free partition request.
-noallocate	This option is more interesting for job schedulers. It tells mpirun not to use a block that is not already booted.
-shape <XxYxZ>	Specifies a hardware configuration to use. The dimensions are in the Compute Nodes. If hardware matching is found, a new partition is created and booted. Implies that -partition is not specified.
-psets_per_bp <n>	Specifies the I/O Node to Compute Node ratio. The default is to use the best possible ratio of I/O Nodes to Compute Nodes. Specifying a higher number of I/O Nodes than what is available results in an error.
-connect <MESH TORUS>	Specifies a mesh or a torus when mpirun creates new partitions.
-reboot	Reboots all the Compute Nodes of an already booted partition that is specified with -partition before running the job. If the partition is in any other state, this is an error.
-boot_options <options>	Specifies boot options to use when booting a freshly created partition.

Output options

The output options (Table 13-3) control information that is sent to STDIN, STDOUT, and STDERR.

Table 13-3 Output options

Arguments	Description
-verbose [0-4]	Sets the verbosity level. The default is 0, which means that mpirun does not output any status or diagnostic messages unless a severe error occurs. If you are curious about what is happening, try levels 1 or 2. All mpirun generated status and error messages appear on STDERR.
-label	Use this option to have mpirun label the source of each line of output. The source is the MPI rank, and stderr or stdout from which the output originated.
-enable_tty_reporting	By default, mpirun tells the control system and the C runtime on the Compute Nodes that STDIN, STDOUT, and STDERR are tied to TTY type devices. While semantically correct for the BlueGene system, this prevents blocked I/O to these file descriptors, which can slow down operations. If you use this option, mpirun will sense if these file descriptors are tied to TTYs and report the results accurately to the control system.
-strace <all none n>	Use this argument to enable a syscall trace on all Compute Nodes, no Compute Nodes, or a specific Compute Node (identified by MPI rank). The extra output from the syscall trace appears on STDERR. The default is none.

Other options

Table 13-4 provides a list of other options. These options provide general information about selected software and hardware features.

Table 13-4 Other options

Arguments	Description
-h	Displays help text.
-version	Displays mpirun version information.
-host <host name>	Specifies the Service Node to use.
-port <port>	Specifies the listening port of mpirund .
-start_gdbserver <path_to_gdbserver>	Loads the job in such a way as to enable GDB debugging, either right from the first instruction or later on while the job is running. There is a separate set of instructions on GDB debugging.
-nw	Reports mpirun -generated return code instead of an application generated return code. Useful only for debugging mpirun .
-only_test_protocol	Simulates a job without using any hardware or talking to the control system. It is useful for making sure that mpirun can start mpirun_be correctly.

13.4 Environmental variables

An alternative way to control `mpirun` execution is to use environmental variables. Most command line options for `mpirun` can be specified using an environment variable. The variables are useful for options that are used in production runs. If you do need to alter the option, you can modify it on the command line to override the environment variable. Table 13-5 summarizes all the environmental variables. The variables must be defined before execution of `mpirun` starts.

Table 13-5 List of environmental variables

Arguments	Environmental variables
-partition	MPIRUN_PARTITION
-nodes	MPIRUN_NODES MPIRUN_N MPIRUN_NP
-mode	MPIRUN_MODE
-exe	MPIRUN_EXE
-cwd	MPIRUN_CWD MPIRUN_WDIR
-host	MMCS_SERVER_IP _MPIRUN_SERVER_HOSTNAME
-port	MPIRUN_SERVER_PORT
-env	MPIRUN_ENV
-exp_env	MPIRUN_EXP_ENV
-env_all	MPIRUN_EXP_ENV_ALL
-mapfile	MPIRUN_MAPFILE
-args	MPIRUN_ARGS
-timeout	MPIRUN_TIMEOUT
-start_gdbserver	MPIRUN_START_GDBSERVER
-label	MPIRUN_LABEL
-nw	MPIRUN_NW
-nofree	MPIRUN_NOFREE
-noallocate	MPIRUN_NOALLOCATE
-reboot	MPIRUN_REBOOT
-boot_options	MPIRUN_BOOT_OPTIONS MPIRUN_KERNEL_OPTIONS
-verbose	MPIRUN_VERBOSE
-only_test_protocol	MPIRUN_ONLY_TEST_PROTOCOL
-shape	MPIRUN_SHAPE
-psets_per_bp	MPIRUN_PSETS_PER_BP
-connect	MPIRUN_CONNECTION
-enable_tty_reporting	MPIRUN_ENABLE_TTY_REPORTING
-config	MPIRUN_CONFIG_FILE

13.5 Return codes

If `mpirun` fails for any reason, such as a bug, boot failure, job failure, and so on, it returns a return code to your shell if you supply the `-nw` argument. If you omit the `-nw` argument, it returns the job's return code if it is present in the job table. Table 13-6 lists the possible error codes.

Table 13-6 List of return codes

Return code	Description
0	OK; successful
10	Communication error
11	Version handshake failed
12	Front-end initialization failed
13	Failed to execute back-end <code>mpirun</code> on Service Node
14	Back-end initialization failed
15	Failed to locate <code>db.properties</code> file
16	Failed to get the machine serial number (bridge configuration file not found?)
17	Execution interrupted by message from the front end
18	Failed to prepare the partition
19	Failed to initialize allocator
20	Partition name already exists
21	No free space left to allocate partition for this job
22	Failed to allocate partition
23	Failed to allocate a partition; job has illegal requirements
24	Specified partition does not exist
25	Failed to get a partition state
26	Specified partition is in an incompatible state
27	Specified partition is not ready
28	Failed to get a partition owner
29	Failed to set a partition owner
30	Failed while checking to see if the partition is busy
31	Partition is occupied by another job
32	Failed while checking to see if the user is in the partition's user list
33	A user does not have permission to run the job on the specified partition
34	Failed while examining the specified partition
35	Failed while setting kernel options; the <code>rm_modify_partition()</code> API failed
36	Kernel options were specified but the partition is not in a FREE state

Return code	Description
37	Failed to boot the partition
38	Failed to reboot the partition
39	Failed to create MPMD configuration file on the Service Node
40	Found a zero-length line while writing to the MPMD configuration file
41	Failed to write a line to the MPMD configuration file
42	Failed to validate the MPMD configuration file
43	Failed to add the new job to the database
44	Failed to get an ID for the new job
45	Failed to start the job
46	An error occurred while mpirun was waiting for the job to terminate
47	Job timed out
48	The job was moved to the history table before it terminated
49	Job execution failed; job switched to an error state
50	Job execution interrupted; job queued
51	Failed to get a job exit status
52	Failed to get a job error text
53	Executable path for the debugger server is not specified
54	Failed to set debug information; unable to attach the debugger
55	Failed to get proctable; unable to attach the debugger
56	Failed while attaching to the job; unable to attach the debugger
57	Failed debugging job; unable to attach the debugger
58	Failed to begin a job
59	Failed to load a job
60	Failed to load a job
61	Failed to clean up a job, partition, or both
62	Failed to cancel a job
63	Failed to destroy a partition
64	Failed to remove a partition
65	Failed to reset kernel options; the <code>rm_modify_partition()</code> API failed
66	One or more threads died
67	Unexpected message
68	Failed to dequeue control message
69	Out of memory
70	Execution interrupted by signal

13.6 Examples

In this section, we present various examples of **mpirun** commands.

Display information

Example 13-4 shows how to display information using the **-h** flag.

*Example 13-4 Invoking **mpirun -h** or **-help** to list all the options available*

```
[descartes:/bgusr/cpsosa] mpirun -h
```

Usage:

```
    mpirun [options]
    or
    mpirun [options] binary [arg1 arg2 ... argn]
```

Options:

-h	Provides this extended help information; can also use -help
-version	Display version information
-partition <partition_id>	ID of the partition to run the job on
-np <compute_nodes>	The number of Compute Nodes to use for the job
-mode <SMP DUAL VN>	Execution mode, either SMP, DUAL, or Virtual Node Mode; the default is SMP
-exe <binary>	Full path to the binary to execute
-cwd <path>	Current working directory of the job, as seen by the Compute Nodes; can also use -wdir
-host <service_node_host>	Host name of the Service Node
-port <service_node_port>	Port of the mpirun server on the Service Node
-env <env=val>	Environment variable that should be set
-exp_env <env vars>	Environment variable in the current environment to export
-env_all	Export all current environment variables to the job environment
-mapfile <mapfile mapping>	mapfile contains a user specified MPI topology; mapping is a permutation of XYZT
-args <"<arguments>">	Arguments to pass to the job; must be enclosed in double quotation marks
-timeout <seconds>	The limit of the job execution time
-start_gdbserver <path>	Start gdbserver for the job; must specify the path to gdbserver
-label	Add labels (STDOUT, STDERR, and MPI rank) to the job output
-nw	Return mpirun job cycle status instead of the job exit status
-nofree	Do not deallocate the partition if mpirun allocated it
-free <wait nowait>	Free the partition specified by -partition ; no job will be run
-noallocate	Do not allocate the partition; the job will only start if the partition was already INITIALIZED or CONFIGURING
-reboot	Reboot all Compute Nodes of the specified partition before running the job; the partition must be INITIALIZED prior to rebooting
-backend	Use a specified mpirun backend binary on the Service Node
-boot_options <options>	Low-level options used when booting a partition
-verbose <0 1 2 3 4>	Verbosity level, default is 0
-trace <0-7>	Trace level; output is sent to a file in the current working directory; default level is 0
-only_test_protocol	Test the mpirun frontend to backend communication; no job will be run
-strace <all none n>	Enable syscall trace for all, none, or node with MPI rank n
-shape <XxYxZ>	Shape of job in XxYxZ format; if not specified, you must use -partition or -np


```
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3:~/bgp/control/mpirun/new>
```

Using a predefined partition and -np

Example 13-7 shows a simple script to invoke `mpirun`.

Example 13-7 csh script to invoke mpirun

```
[descartes:/bgusr/cpsosa/pallas] ./run.pallas >& pallas_july06_2007_bgp.out
where the script run.pallas is:
#!/bin/csh
set MPIRUN="mpirun"
set MPIOPT="-np 32"
set MODE="-mode VN"
set PARTITION="-partition N01_32_1"
set WDIR="-cwd /bgusr/cpsosa/pallas"
set EXE="-exe /bgusr/cpsosa/pallas/PMB-MPI1"
#
$MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE
#
echo "That's all folks!!"
```

Using of environmental variables

Example 13-8 shows use of `-env` to define environmental variables.

Example 13-8 Use of -env

```
[descartes:/bgusr/cpsosa]mpirun -partition N00_32_1 -np 32 -mode SMP -cwd
/bgusr/cpsosa -exe a.out -env "OMP_NUM_THREADS=4"
```

Using stdin from a terminal

In Example 13-9, the user types their name `bgp user` in response to the job's stdout. After a while, the job is terminated when the user presses `Ctrl+C` to send `mpirun` a `SIGINT`.

Example 13-9 Usage of stin from a terminal

```
dd2sys1fen3:~/bgp/control/mpirun/new> mpirun -partition R00-M0-N00 -verbose 0 -exe
/BGPhome/stdin.sh -np 1
What's your name?
bgp user
hello bgp user
What's your name?
<Aug 11 15:33:44.021105> FE_MPI (WARN) : SignalHandler() -
<Aug 11 15:33:44.021173> FE_MPI (WARN) : SignalHandler() -
!-----!
<Aug 11 15:33:44.021201> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking all the
necessary actions !
<Aug 11 15:33:44.021217> FE_MPI (WARN) : SignalHandler() - ! to terminate the job and to free
the resources !
<Aug 11 15:33:44.021233> FE_MPI (WARN) : SignalHandler() - ! occupied by this job. This might
take a while... !
<Aug 11 15:33:44.021261> FE_MPI (WARN) : SignalHandler() -
!-----!
<Aug 11 15:33:44.021276> FE_MPI (WARN) : SignalHandler() -
<Aug 11 15:33:44.050365> BE_MPI (WARN) : Received a message from frontend
<Aug 11 15:33:44.050465> BE_MPI (WARN) : Execution of the current command interrupted
<Aug 11 15:33:59.532817> FE_MPI (ERROR): Failure list:
<Aug 11 15:33:59.532899> FE_MPI (ERROR): - 1. Execution interrupted by signal (failure #71)
dd2sys1fen3:~/bgp/control/mpirun/new>
```

Using stdin from a file or pipe

Example 13-10 illustrates the use of stdin from a file or pipe.

Example 13-10 Usage of stin from a file or pipe

```
dd2sys1fen3:~/bgp/control/mpirun/new> cat ~/stdin.cc
#include <iostream>

using namespace std;

int main() {
    unsigned int lineno = 0;
    while (cin.good()) {
        string line;
        getline(cin, line);
        if (!line.empty()) {
            cout << "line " << ++lineno << ": " << line << endl;
        }
    }
}
dd2sys1fen3:~/bgp/control/mpirun/new> cat stdin.txt
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque id orci. Ut eleifend dui a erat varius
facilisis. Aliquam felis. Ut tincidunt, velit in pulvinar imperdiet, sem sapien sagittis neque, vitae bibendum
sapien erat vitae risus. Aenean suscipit. Aliquam molestie orci nec magna. Aliquam non enim. Integer dictum
magna quis orci. Praesent eget libero sed erat ultrices ullamcorper. Donec sodales hendrerit velit. Fusce
mattis. Suspendisse blandit ornare arcu. Pellentesque venenatis.

dd2sys1fen3:~/bgp/control/mpirun/new> cat stdin.txt | mpirun -partition R00-M0-N00 -verbose 0 -exe /BGPhome/stdin_test
-np 1
line 1: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque id orci. Ut eleifend dui a erat varius
```

line 2: facilisis. Aliquam felis. Ut tincidunt, velit in pulvinar imperdiet, sem sapien sagittis neque, vitae bibendum
line 3: sapien erat vitae risus. Aenean suscipit. Aliquam molestie orci nec magna. Aliquam non enim. Integer dictum
line 4: magna quis orci. Praesent eget libero sed erat ultrices ullamcorper. Donec sodales hendrerit velit. Fusce
line 5: mattis. Suspendisse blandit ornare arcu. Pellentesque venenatis.
dd2sys1fen3:~/bgp/control/mpirun/new>

Using the tee utility

To send stdout, stderr, or both to a file in addition to your terminal, use the tee utility. Give tee the `-i` argument, so that it ignores any signals that are sent, such as Ctrl+C to terminate a job prematurely. See Example 13-11.

Example 13-11 Using tee

```
dd2sys1fen3:~/bgp/control/mpirun/new> mpirun -partition R00-M0-N00 -verbose 1 -exe
/BGPhome/datespinner.sh -np 1 | tee -i datespinner.out
<Aug 12 10:27:10.997374> FE_MPI (Info) : Invoking mpirun backend
<Aug 12 10:27:11.155416> BRIDGE (Info) : rm_set_serial() - The machine serial
number (alias) is BGP
<Aug 12 10:27:11.194557> FE_MPI (Info) : Preparing partition
<Aug 12 10:27:11.234550> BE_MPI (Info) : Examining specified partition
<Aug 12 10:27:11.823425> BE_MPI (Info) : Checking partition R00-M0-N00 initial
state ...
<Aug 12 10:27:11.823499> BE_MPI (Info) : Partition R00-M0-N00 initial state =
READY ('I')
<Aug 12 10:27:11.823516> BE_MPI (Info) : Checking partition owner...
<Aug 12 10:27:11.823532> BE_MPI (Info) : partition R00-M0-N00 owner is 'userX'
<Aug 12 10:27:11.824744> BE_MPI (Info) : Partition owner matches the current user
<Aug 12 10:27:11.824870> BE_MPI (Info) : Done preparing partition
<Aug 12 10:27:11.864539> FE_MPI (Info) : Adding job
<Aug 12 10:27:11.864876> BE_MPI (Info) : No CWD specified ('-cwd' option)
<Aug 12 10:27:11.864903> BE_MPI (Info) :   - it will be set to
'/BGPhome/usr3/bgp/control/mpirun/new'
<Aug 12 10:27:11.865046> BE_MPI (Info) : Adding job to database...
<Aug 12 10:27:11.944540> FE_MPI (Info) : Job added with the following id: 15
<Aug 12 10:27:11.944593> FE_MPI (Info) : Starting job 15
<Aug 12 10:27:12.004492> FE_MPI (Info) : Waiting for job to terminate
<Aug 12 10:27:12.816792> BE_MPI (Info) : IO - Threads initialized
Sun Aug 12 10:27:13 CDT 2007
Sun Aug 12 10:27:18 CDT 2007
Sun Aug 12 10:27:23 CDT 2007
Sun Aug 12 10:27:28 CDT 2007
Sun Aug 12 10:27:33 CDT 2007
Sun Aug 12 10:27:38 CDT 2007
Sun Aug 12 10:27:43 CDT 2007
Sun Aug 12 10:27:48 CDT 2007
Sun Aug 12 10:27:53 CDT 2007
Sun Aug 12 10:27:58 CDT 2007
Sun Aug 12 10:28:03 CDT 2007
Sun Aug 12 10:28:08 CDT 2007
<Aug 12 10:28:11.159680> FE_MPI (Info) : SignalHandler() -
<Aug 12 10:28:11.159737> FE_MPI (Info) : SignalHandler() - ! Received signal
SIGINT
<Aug 12 10:28:11.159760> FE_MPI (WARN) : SignalHandler() -
<Aug 12 10:28:11.159773> FE_MPI (WARN) : SignalHandler() -
!-----!
```

```

<Aug 12 10:28:11.159788> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking
all the necessary actions !
<Aug 12 10:28:11.159801> FE_MPI (WARN) : SignalHandler() - ! to terminate the job
and to free the resources !
<Aug 12 10:28:11.159815> FE_MPI (WARN) : SignalHandler() - ! occupied by this job.
This might take a while... !
<Aug 12 10:28:11.159829> FE_MPI (WARN) : SignalHandler() -
!-----!
<Aug 12 10:28:11.159842> FE_MPI (WARN) : SignalHandler() -
<Aug 12 10:28:11.201498> FE_MPI (Info) : Termination requested while waiting for
backend response
<Aug 12 10:28:11.201534> FE_MPI (Info) : Starting cleanup sequence
<Aug 12 10:28:11.201794> BE_MPI (WARN) : Received a message from frontend
<Aug 12 10:28:11.201863> BE_MPI (WARN) : Execution of the current command
interrupted
<Aug 12 10:28:11.201942> BE_MPI (Info) : Starting cleanup sequence
<Aug 12 10:28:11.201986> BE_MPI (Info) : cancel_job() - Cancelling job 15
<Aug 12 10:28:11.204567> BE_MPI (Info) : cancel_job() - Job 15 state is RUNNING
('R')
<Aug 12 10:28:11.230352> BE_MPI (Info) : cancel_job() - Job 15 state is DYING
('D'). Waiting...
<Aug 12 10:28:16.249665> BE_MPI (Info) : cancel_job() - Job 15 has been moved to
the history table
<Aug 12 10:28:16.255793> BE_MPI (Info) : cleanupDatabase() - Partition was
supplied with READY ('I') initial state
<Aug 12 10:28:16.255996> BE_MPI (Info) : cleanupDatabase() - No need to destroy
the partition
<Aug 12 10:28:16.591667> FE_MPI (ERROR): Failure list:
<Aug 12 10:28:16.591708> FE_MPI (ERROR): - 1. Execution interrupted by signal
(failure #71)
<Aug 12 10:28:16.591722> FE_MPI (Info) : == FE completed ==
<Aug 12 10:28:16.591736> FE_MPI (Info) : == Exit status: 1 ==
dd2sys1fen3:~/bgp/control/mpirun/new> cat datespinner.out
Sun Aug 12 10:28:49 CDT 2007
Sun Aug 12 10:28:54 CDT 2007
Sun Aug 12 10:28:59 CDT 2007
Sun Aug 12 10:29:04 CDT 2007
Sun Aug 12 10:29:09 CDT 2007
Sun Aug 12 10:29:14 CDT 2007
Sun Aug 12 10:29:19 CDT 2007
dd2sys1fen3:~/bgp/control/mpirun/new>

```

Terminating a job prematurely

Example 13-12 shows a case where the user specified a value for `-np` larger than the number provided in the partition.

Example 13-12 Error due to requesting an -np value greater than the partition size

```
dd2sys1fen3:~/bgp/control/mpirun/new> .mpirun -partition R00-M0-N00 -verbose 0 -exe
/bin/hostname -np 55
<Aug 11 15:28:46.797523> BE_MPI (ERROR): Job execution failed
<Aug 11 15:28:46.797634> BE_MPI (ERROR): Job 8 is in state ERROR ('E')
<Aug 11 15:28:46.842559> FE_MPI (ERROR): Job execution failed (error code - 50)
<Aug 11 15:28:46.842738> FE_MPI (ERROR): - Job execution failed - job switched to an error
state
<Aug 11 15:28:46.851840> BE_MPI (ERROR): The error message in the job record is as follows:
<Aug 11 15:28:46.851900> BE_MPI (ERROR): "BG_SIZE of 55 is greater than block 'R00-M0-N00'
size of 32"
```

Killing a hung job or a running job

`mpirun` has the capability to kill the job and free your partition if it was booted by `mpirun`. To kill your job, we recommend that you send `mpirun` a SIGINT (kill -2) while the job is running or hung. We recommend that you do *not* use SIGKILL since subsequent jobs might experience problems.

Be aware that using SIGINT is somewhat time consuming depending on the state of the job. Therefore, do not expect it to return control instantaneously. Alternatively, if you do not want to wait, try sending `mpirun` three SIGINTs in succession. In this case, it immediately returns control to your shell. However, as the warning messages indicate, your job, partition, or both might be left in a bad state. Ensure that they are cleaned up correctly before you attempt to use them again. Example 13-13 illustrates this procedure.

Example 13-13 Proper way to kill hung or running jobs

From window 2: (open another window to kill a job)

```
ps -ef | grep cpsosa
```

```
cpsosa 23393 23379 0 13:21 pts/13 00:00:00 /bgsys/drivers/ppcfloor/bin/mpirun -partition
N04_32_1 -np 32 -mode VN -cwd /bgusr/cpsosa/red/pallas -exe /bgusr/cpsosa/red/pallas/PMB-MPI1
```

From window 1: (where the job is running)

```
.
. ! Output generated by the program
.
32768          1000          95.49          95.49          95.49          654.50
          65536          640          183.20          183.20          183.20          682.31
<Oct 18 13:22:10.804667> FE_MPI (WARN) : SignalHandler() -
<Oct 18 13:22:10.804743> FE_MPI (WARN) : SignalHandler() -
!-----!
<Oct 18 13:22:10.804769> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking all the
necessary actions !
<Oct 18 13:22:10.804794> FE_MPI (WARN) : SignalHandler() - ! to terminate the job and to free
the resources !
<Oct 18 13:22:10.804818> FE_MPI (WARN) : SignalHandler() - ! occupied by this job. This might
take a while... !
<Oct 18 13:22:10.804841> FE_MPI (WARN) : SignalHandler() -
!-----!
```

```
<Oct 18 13:22:10.804865> FE_MPI (WARN) : SignalHandler() -  
131072      320      357.97      357.97      357.97      698.38  
<Oct 18 13:21:10.936378> BE_MPI (WARN) : Received a message from frontend  
<Oct 18 13:21:10.936449> BE_MPI (WARN) : Execution of the current command interrupted  
<Oct 18 13:21:16.140631> BE_MPI (ERROR): The error message in the job record is as follows:  
<Oct 18 13:21:16.140678> BE_MPI (ERROR): "killed with signal 9"  
<Oct 18 13:22:16.320232> FE_MPI (ERROR): Failure list:  
<Oct 18 13:22:16.320406> FE_MPI (ERROR): - 1. Execution interrupted by signal (failure #71)
```

13.7 mpirun application program interfaces

When writing programs to the **mpirun** APIs, you must consider these requirements:

- ▶ Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- ▶ C and C++ are supported with the GNU gcc 4.1 level compilers. For more information and downloads, refer to the following Web address:
<http://gcc.gnu.org/>
- ▶ The include file is include/sched_api.h.
- ▶ Regarding the library files, support for both 64-bit dynamic libraries is provided. The 64-bit dynamic library file called by **mpirun** must be called libsched_if.so.

mpirun can retrieve runtime information directly from the scheduler without using command-line parameters or environment variables. Each time **mpirun** is invoked, it attempts to load a dynamically loaded library called libsched_if.so. **mpirun** looks for this library in a set of directories as described by the **dlopen** command manual pages.

If the plug-in library is found and successfully loaded, **mpirun** calls the `get_parameters()` function within that library to retrieve the information from the scheduler. The `get_parameters()` function returns the information in a data structure of type `sched_params`. This data structure contains a set of fields that describe the block that the scheduler has allocated the job to run. Each field corresponds to one of the command-line parameters or environment variables.

mpirun complements the information that is retrieved by `get_parameters()` with values from its command-line parameters and environment variables. It gives precedence to the information that is retrieved by `get_parameters()` first, then to its command line parameters, and finally to the environment variables. For example, if the number of processors retrieved by `get_parameters()` is 256, the `-np` command-line parameter is set to 512, and the environment variable `MPIRUN_NP` is set to 448, **mpirun** runs the job on 256 Compute Nodes.

The block ID to use for that job can be the one specified by the `MPIRUN_PARTITION` environment variable, if both the `get_parameters()` function does not retrieve the block ID and the `-partition` command line parameter is not specified.

If **mpirun** is invoked with the `-verbose` parameter with a value greater than 0, it displays information that describes the loading of the dynamically loaded library. The message "Scheduler interface library loaded" indicates that **mpirun** found the library, loaded it, and is using it.

The implementation of the `libsched_if.so` library is scheduling-system specific. In general, this library should use the scheduler's APIs to retrieve the required information and convert it to the `sched_params` data type for `mpirun` to use. The only requirement is that the library interface conforms to the definitions in the `sched_api.h` header file distributed with the `mpirun` binaries. This interface may be modified with future releases of `mpirun`.

The `mpirun` plug-in interface also requires the implementer provide an `mpirun_done()` function (`void mpirun_done(int res);`). This function is called by `mpirun` just before it does an exit. It is used to signal the plug-in implementer that `mpirun` is terminating.

You can find more information about the library implementation and data structures in the `sched_api.h` header file.

The following APIs are supported for `mpirun`:

► `int get_parameters(sched_params_t *params);`

This function is used to provide input parameters to `mpirun` from your application. If a value of 1 (failure) is returned on the `get_parameters()` call, then `mpirun` proceeds to terminate. Some external resource managers use this technique to prevent stand-alone `mpirun` from being used. If the plug-in provider wants `mpirun` processing to continue, then they must return a 0 (success) value on the `get_parameters()` call.

► `void mpirun_done(int res);`

This function is called by `mpirun` just before it calls the `exit()` function. It can be used to signal the scheduler that `mpirun` is terminating.



Dynamic Partition Allocator APIs

The Dynamic Partition Allocator application programming interface (API) provides an easy-to-use interface for the dynamic creation of partitions. This API inspects the current state of the Blue Gene/P machine and attempts to create a partition based on available resources. If no resources are available that match the partition requirements, then the partition is not created. It is expected that any job scheduler that uses the partition allocator does so from a centralized process to avoid conflicts in finding free resources to build the partition. Dynamic Partition Allocator APIs are thread safe. Only 64-bit shared libraries are being provided.

In this chapter, we define a list of APIs into the Midplane Management Control System (MMCS) Dynamic Partition Allocator. See Chapter 11, “Control system (Bridge) APIs” on page 159, for details about the Bridge API.

14.1 Overview of API support

In the following sections, we provide an overview of the support provided by the APIs.

14.1.1 Requirements

When writing programs to the Dynamic Partition Allocator APIs, you must follow the requirements as explained in the following sections.

Operating system supported

Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.

Languages supported

C and C++ are supported with the GNU gcc 4.1.1 level compilers. For more information and downloads, refer to the following Web address:

<http://gcc.gnu.org/>

Include files

All required include files are installed in the `/bgsys/drivers/ppcfloor/include` directory. The include file for the dynamic allocator API is `allocator_api.h`.

Library files

The Dynamic Partition Allocator APIs support 64-bit applications using dynamic linking with shared objects.

64-bit libraries

The required library files are installed in the `/bgsys/drivers/ppcfloor/lib64` directory. The shared object for linking to the Bridge API is `libbgpallocator.so`.

The `libbgpallocator.so` library has dependencies on other libraries included with the Blue Gene software, including the following objects:

- ▶ `libbgpbridge.so`
- ▶ `libbgpconfig.so`
- ▶ `libbgpdb.so`
- ▶ `libsaymessage.so`
- ▶ `libtableapi.so`

These files are installed with the standard system installation procedure. They are contained in the `bgpbase.rpm` file.

Configuring environment variables

The environment variables in Table 14-1 are used to control the dynamic allocator and Bridge API.

Table 14-1 Environment variables that control the Bridge API

Environment variable	Required	Description
DB_PROPERTY	Yes	This variable must be set to the path of the db.properties file with database connection information. For a default installation, the path to this file is /bgsys/local/etc/db.properties.
BRIDGE_CONFIG	Yes	This variable must be set to the path of the bridge.config file that contains the Bridge API configuration values. For a default installation, the path to this file is /bgsys/local/etc/bridge.config.
ALLOCATOR_DRAIN_LIST	No	This variable can be set to the path of the base partition drain list to be used if one is not specified on the call to <code>rm_init_allocator()</code> . When this variable is not set, the file /etc/allocator_drain.lst is used as a default if it exists.
BRIDGE_DUMP_XML	No	When set to any value, this variable causes the Bridge API to dump its in-memory XML streams to files in /tmp for debugging. When this variable is not set, the Bridge API does not dump its in-memory XML streams.

14.2 API details

In this section, we provide details about the APIs and return codes for dynamic partition allocation.

14.2.1 APIs

The following APIs are used for dynamic partition allocation and are all thread safe:

- ▶ `BGALLOC_STATUS rm_init_allocator(const char * caller_desc, const char * drain_list);`

A program should call `rm_init_allocator()` and pass a description that will be used as the text description for all partitions used by subsequent `rm_allocate_partition()` calls. For example, passing in *ABC job scheduler* causes any partitions that are created by `rm_allocate_partition()` to have *ABC job scheduler* as the partition description.

The caller can also optionally specify a drain list file name that identifies the base partitions (midplanes) that will be excluded from the list of resources to consider when allocating new partitions. If NULL is passed in for the drain list file name, a default drain list is set first from the following locations:

- The path in the environment variable `ALLOCATOR_DRAIN_LIST` if it exists
- The `/etc/allocator_drain.lst` file if it exists

If no drain list file is established, no base partitions are excluded. If an invalid file name is passed in, the call fails. For example, a drain list file with the following content excludes base partitions R00-M0, R00-M1, and R01-M0 when allocating resources for a partition:

```
R00-M0
R00-M1
R01-M0
```

The list of resources can contain items separated by any white-space character (space, tab, new line, vertical tab or form feed). Items found that do not match an existing resource are ignored, but an error message is logged.

```
▶ BGALLOC_STATUS rm_allocate_partition(  
    const rm_size_t size,  
    const rm_connection_type_t conn,  
    const rm_size3D_t shape,  
    const rm_job_mode_t mode,  
    const rm_psetsPerBP_t psetsPerBP,  
    const char * user_name,  
    const char * caller_desc,  
    const char * boot_options,  
    const char * ignoreBPs,  
    const char * partition_id,  
    char ** newpartition_id);
```

The caller to `rm_allocate_partition()` provides input parameters that describe the characteristics of the partition that should be created from available Blue Gene/P machine resources. If resources are available that match the requirements, a partition is created and allocated, and the partition name is returned to the caller along with a return code of `BGALLOC_OK`.

If both `size` and `shape` values are provided, the allocation is based on the `shape` value only.

The `user_name` parameter is required.

If the `caller_desc` value is `NULL`, the caller description specified on the call to `rm_init_allocator` is used.

The `boot_options` parameter is optional and can be `NULL`.

If the `ignoreBPs` parameter is not `NULL`, it must be a string of blank separated base partition identifiers to be ignored. The base partitions listed in the parameter is ignored as though the partitions were included in the drain list file currently in effect.

If the `partition_id` parameter is not `NULL`, it can specify one of the following options:

- The name of the new partition
The name can be from 1 to 32 characters. Valid characters are `a..z`, `A..Z`, `0..9`, `-` (hyphen), and `_` (underscore).
- The prefix to be used for generating a unique partition name
The prefix can be from 1 to 16 characters, followed by an asterisk (*). Valid characters are the same as those for a new partition name. For example, if `ABC-Scheduler*` is specified as a prefix, the resulting unique partition name can be `ABC-Scheduler-27Sep1519514155`.

Important: The returned `char *` value for `newpartition_id` should be freed by the caller when it is no longer needed to avoid memory leaks.

14.2.2 Return codes

When a failure occurs, the API invocation returns an error code. In addition, a failure always generates a log message, which provides more information about the possible cause of the problem and an optional corrective action. These log messages are used for debugging and non-automatic recovery of failures.

The BGALLOC_STATUS return codes for the Dynamic Partition Allocator can be one of the following types:

- ▶ BGALLOC_OK: Invocation completed successfully.
- ▶ BGALLOC_ILLEGAL_INPUT: The input to the API invocation is invalid. This result is due to missing required data, illegal data, and similar problems.
- ▶ BGALLOC_ERROR: An error occurred, such as a memory allocation problem or failure on low-level call.
- ▶ BGALLOC_NOT_FOUND: The request to dynamically create a partition failed because required resources are not available.
- ▶ BGALLOC_ALREADY_EXISTS: A partition already exists with the name specified. This error occurs only when the caller indicates a specific name for the new partition.

14.3 Sample program

The sample program in Example 14-1 shows how to allocate a partition from resources on base partition R001.

Example 14-1 Sample allocator API program

```
#include <iostream>
#include <sstream>
#include <cstring>
#include "allocator_api.h"

using std::cout;
using std::cerr;
using std::endl;

int main() {
    rm_size3D_t shape;
    rm_connection_type_t conn = RM_MESH;
    char * ignoreBPs = "R00-M0";
    char* new_partition_id;
    shape.X = 0;
    shape.Y = 0;
    shape.Z = 0;
    BGALLOC_STATUS alloc_rc;

    //set lowest level of verbosity
    setSayMessageParams(stderr, MESSAGE_DEBUG1);
    alloc_rc = rm_init_allocator("test", NULL);
    alloc_rc = rm_allocate_partition(256, conn, shape, RM_SMP_MODE, 0,
                                   "user1",
                                   "New partition description",
                                   ignoreBPs,
                                   "",
                                   "ABC-Scheduler*",
                                   &new_partition_id);

    if (alloc_rc == BGALLOC_OK) {
        cout << "successfully allocated partition: " << new_partition_id << endl;
        free(new_partition_id);
    } else {
        cerr << "could not allocate partition: " << endl;
    }
}
```

```

    if (alloc_rc == BGALLOC_ILLEGAL_INPUT) {
        cerr << "illegal input" << endl;
    } else if (alloc_rc == BGALLOC_ERROR) {
        cerr << "unknown error" << endl;
    } else if (alloc_rc == BGALLOC_NOT_FOUND) {
        cerr << "not found" << endl;
    } else if (alloc_rc == BGALLOC_ALREADY_EXISTS) {
        cerr << "partition already exists" << endl;
    } else {
        cerr << "internal error" << endl;
    }
}
}

```

Example 14-2 shows the commands that are used to compile and link the sample program.

Example 14-2 The compile and link commands

```

g++ -m64 -pthread -I/bgsys/drivers/ppcfloor/include -c sample1.cc -o sample1.o_64

g++ -m64 -pthread -o sample1 sample1.o_64 -L/bgsys/drivers/ppcfloor/lib64
-lbgallocator

```



Part 4

Applications

In this part, we discuss applications that are being used on the Blue Gene/L or Blue Gene/P system. This part includes Chapter 15, “Performance overview of engineering and scientific applications” on page 245.



Performance overview of engineering and scientific applications

In this chapter, we briefly describe a series of scientific and engineering applications that are currently being used on either the Blue Gene/L or Blue Gene/P system. For a comprehensive list of applications, refer to the IBM Blue Gene Web page at:

<http://www-03.ibm.com/servers/deepcomputing/bluegene/siapps.html>

The examples in this chapter emphasize the benefits of using the Blue Gene supercomputer as a highly scalable parallel system. They present results for running applications in various modes that exploit the architecture of the system.

15.1 Blue Gene/P system from an applications perspective

This book has been dedicated to describing the Blue Gene/P massively parallel supercomputer from IBM. In this section, we summarize the benefits of the Blue Gene/P system from an applications point of view. We recall that at the core of the system is the IBM PowerPC (PowerPC 450) processor with the addition of two floating-point units (FPU). This system uses a distributed memory, message-passing programming model.

To achieve a high level of integration and quantity of micro-processors with low power consumption, the machine was developed based on a processor with moderate frequency. The Blue Gene/P system uses system-on-a-chip (SoC) technology to allow a high level of integration, low power, and low design cost. Each processor core runs at a frequency of 850 MHz giving a theoretical peak performance of 3.4 gigaflops/core or 13.6 gigaflops/chip. The chip constitutes the Compute Node.

The next building blocks are the compute and I/O cards. A single Compute Node attached to a processor card with 2 GB of memory (RAM) creates the compute and I/O cards. The compute cards and I/O cards are plugged into a node card. There are two rows of sixteen compute cards on the node card. There can be up to two I/O cards per node card.

A midplane consists of 16 node cards stacked in a rack. A rack holds two midplanes, for a total of 32 node cards. A system with 72 racks consisting of 294,912 processor cores.

In 2005, running a real application on the Blue Gene/L system broke the barrier of 100 teraflops/second (TF/s), sustaining performance using the domain decomposition molecular-dynamics code (ddcMD) from the Lawrence Livermore National Laboratory.³³ In 2006, first system to break the barrier of 200 TF/s was Qbox running at 207.3 TF/s.³⁴ Real applications are currently achieving two orders of magnitude higher performance than previously possible. Successful scaling has pushed from O(1000) processors to O(100,000) processors by the Gordon Bell Prize finalists at Supercomputing 2006.³⁵ Out of six finalists, three ran on the Blue Gene/L system.

In silico experimentation plays a crucial role in many scientific disciplines. It provides a fingerprint to experiment. In engineering applications, such as automotive crash studies, numerical simulation is much cheaper than physical experimentation. In other applications, such as global climate change where experiments are impossible, simulations are used to explore the fundamental scientific issues.³⁶ This is certainly true in Life Sciences as well as in Materials Science. Figure 15-1 illustrates a landscape of a few selected areas and techniques where high-performance computing is important to carry out simulations.

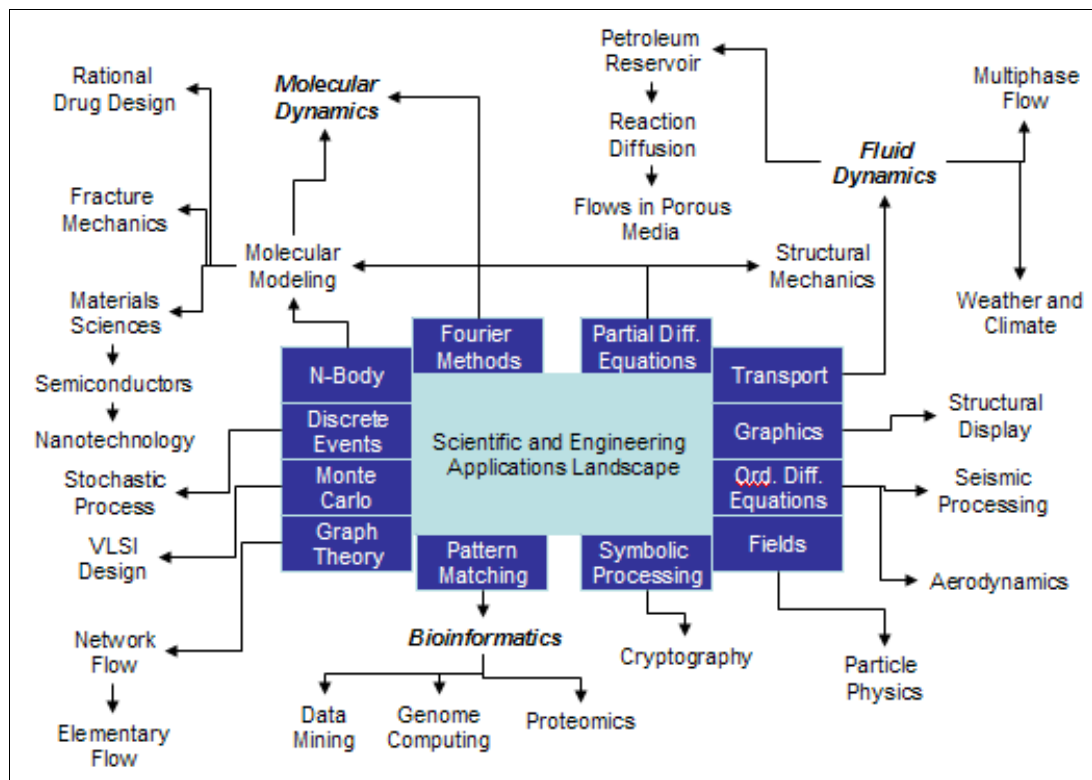


Figure 15-1 High-performance computing landscape for selected Scientific and Engineering applications

In the rest of this chapter, we summarize the performance that has appeared in the literature for a series of applications in Life Sciences and Materials Science. A comprehensive list of applications is available for the Blue Gene/L and Blue Gene/P systems. For more information, see the IBM Blue Gene Applications Web page at:

<http://www-03.ibm.com/servers/deepcomputing/bluegene/siapps.html>

15.2 Selected Chemistry and Life Sciences applications

In this section, we provide a brief overview of the performance characteristics of a selected set of Chemistry and Life Sciences applications. In particular, we focus on what is known as *Computational Chemistry*. However, as other disciplines in sciences that traditionally relied almost exclusively on experimental observation began to fully incorporate Information Technology (IT) as one of their tools, the area of Computational Chemistry has expanded to new disciplines such as Bioinformatics, Systems Biology and several other areas that have emerged after the post-genomic era.

In order to understand or define the kind of molecular systems that can be studied with these techniques, Figure 15-2 on page 248 defines the Computational Chemistry landscape as a function of the size of the systems and the methodology. It illustrates that Classical Molecular Mechanics/Molecular Dynamics (MM/MD) are commonly used to simulate large biomolecules that cannot be treated with more accurate methods. The next level corresponds to semi-empirical methods. Finally *Ab Initio* methods (also called *electronic structure methods*) provide a more accurate description of the system, but the computational demands in terms of compute cycles increase rapidly.

Alternatively, Bioinformatics techniques rely mainly on string manipulations in an effort to carry out data mining of large databases. These applications tend to be data intensive.

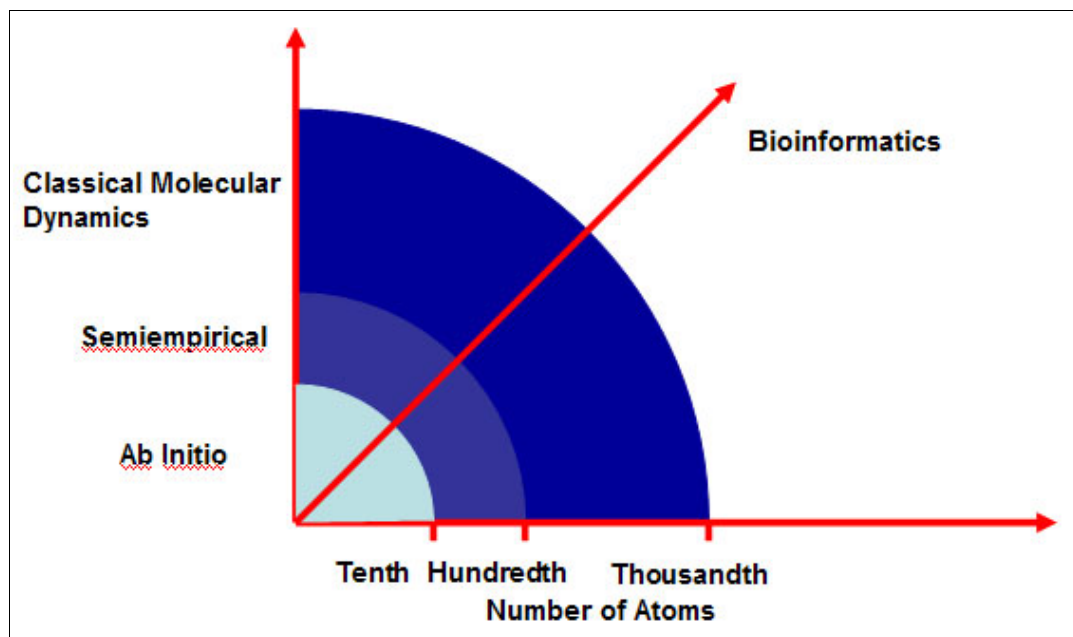


Figure 15-2 Computational methods landscape in Computational Chemistry

Although, Density Functional Theory-based approaches are not fully represented in Figure 15-2, nowadays, these types of methods are being used to simulate biologically important systems.³⁷ These techniques allow for the calculation of larger systems. In this chapter, we briefly describe Car-Parrinello Molecular Dynamics (CPMD).³⁸ In the same vein, use of mixed Quantum Mechanical/Molecular Mechanical (QM/MM) methods³⁹ can simulate larger systems.

15.2.1 Classical Molecular Mechanics and Molecular Dynamics applications

Applications in such areas as Chemistry and Life Sciences can benefit from the type of architecture used in the Blue Gene supercomputer.⁴⁰ In particular, software packages based on molecular dynamics have been considered good candidates for the Blue Gene architecture. Classical MD simulations compute atomic trajectories by solving equations of motion numerically by using empirical force fields. The overall MD energy equation is broken into three components: bonded, van der Waals, and electrostatic. The first two components are local in nature and therefore do not make a significant contribution to the overall running time.

The quadratic scaling of the electrostatics force terms, however, requires a high level of optimization of the MD application.⁴¹ To improve performance on simulations in which the solvent is modeled at the atomic level (that is, explicit solvent modeling), the four Blue Gene MD applications of AMBER,⁴² Blue Matter,⁴³ LAMMPS,⁴⁴ and NAMD⁴⁵ employ a reciprocal-space technique called *Ewald sums*, which enables the evaluation of long-range electrostatic forces to a pre-selected level of accuracy. In addition to the *particle mesh Ewald (PME) method*, LAMMPS offers the particle particle/particle-mesh (PPPM) technique with characteristics that make it scale well on massively parallel processing (MPP) machines such as the Blue Gene system.

AMBER

AMBER⁴⁶ is the collective name for a suite of programs that are developed by the Scripps Research Institute. With these programs, users can carry out molecular dynamics simulations, particularly on biomolecules. The primary AMBER module, called *sander*, was designed to run on parallel systems and provides direct support for several force fields for proteins and nucleic acids. AMBER includes an extensively-modified version of *sander*, called *pmemd* (particle mesh). For complete information about AMBER as well as benchmarks, refer to the AMBER Web site at:

<http://amber.scripps.edu/>

For implicit solvent (continuum) models, which rely on variations of the Poisson equation of classical electrostatics, AMBER offers the Generalized Born (GB) method. This method uses an approximation to the Poisson equation that can be solved analytically and allows for good scaling (Figure 15-3). In Figure 15-3, the experiment is with an implicit solvent (GB) model of 120,000 atoms (Aon benchmark).

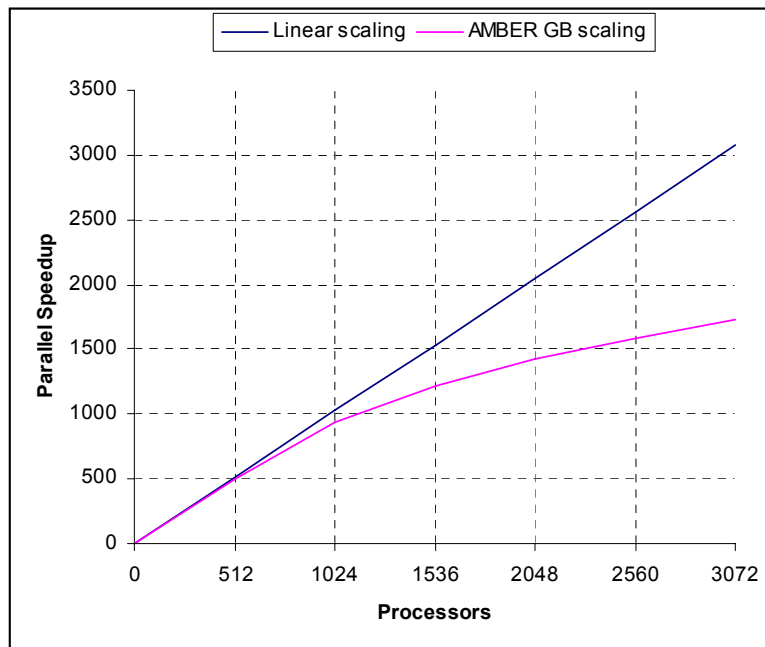


Figure 15-3 Parallel scaling of AMBER on the Blue Gene/L system

AMBER also incorporates the PME algorithm, which takes the full electrostatic interactions into account and to improve the performance of electrostatic force evaluation (Figure 15-4). In Figure 15-4, the experiment is with an explicit solvent (PME) model of 290,000 atoms (Rubisco).

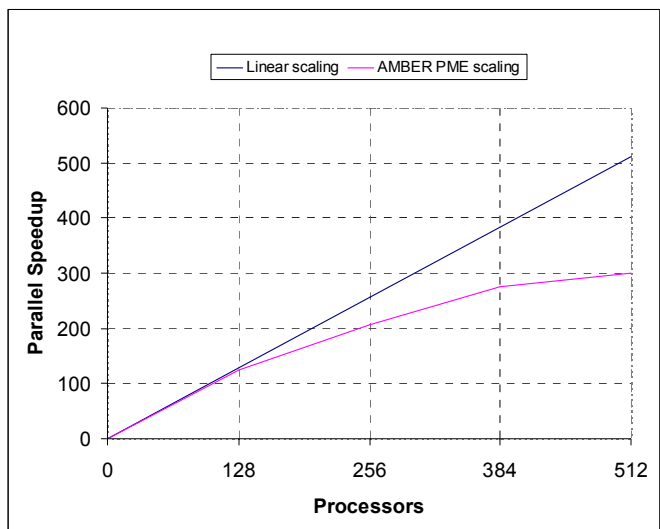


Figure 15-4 Parallel scaling of AMBER on the Blue Gene/L system

Blue Matter

Blue Matter⁴⁷ is a classical molecular dynamics application that has been under development as part of the IBM Blue Gene project. The effort serves two purposes:

- ▶ Enable scientific work in the area of biomolecular simulation that IBM announced in December 1999.
- ▶ Act as an experimental platform for the exploration of programming models and algorithms for massively parallel machines in the context of a real application.

Blue Matter has been implemented via spatial-force decomposition for N-body simulations uses the PME method for handling electrostatic interactions. The Ewald summation method and particle mesh techniques are approximated by a finite range cut-off and a reciprocal space portion for the charge distribution. This is done in Blue Matter via the Particle-Particle-Particle-Mesh (P3ME) method.⁴⁸

The results presented by Fitch et al.⁴⁹ show impressive scalability on the Blue Gene/L system. Figure 15-5 shows scalability as a function of the number of nodes. It illustrates that the performance in time/time step as a function of the number of processors for β -Hairpin contains a total of 5,239 atoms. SOPE contains 13,758 atoms. In this case, the timings that are reported here correspond to a size of 64^3 FFT. Rhodopsin contains 43,222 atoms, and ApoA contains 92,224 atoms. All runs were carried out using the P3ME method, which was implemented in Blue Matter at constant particle number, volume, and energy (NVE).⁵¹

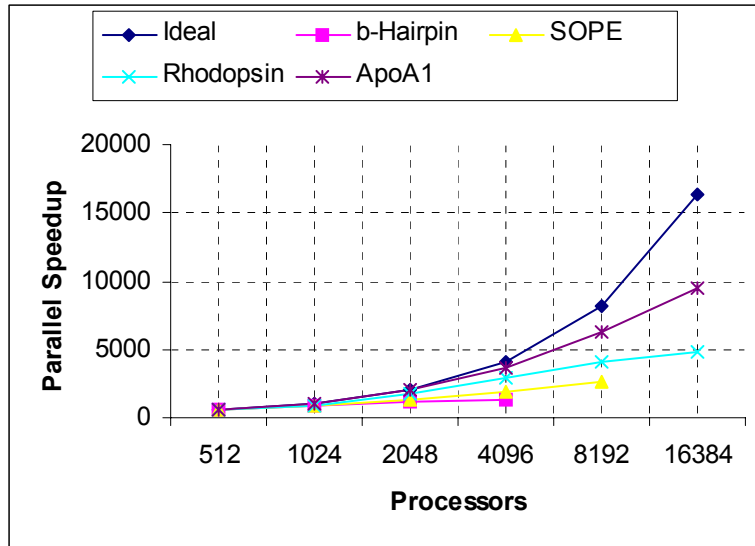


Figure 15-5 Performance in time/time step as a function of number of processors (from Fitch, et al.⁵⁰)

LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)⁵² is an MD program from Sandia National Laboratories that is designed specifically for MPP. LAMMPS is implemented in C++ and is distributed freely as open-source software under the GNU Public License (GPL).⁵³ LAMMPS can model atomic, polymeric, biological, metallic, or granular systems using a variety of force fields and boundary conditions. The parallel efficiency of LAMMPS varies from the size of the benchmark data and the number of steps being simulated. In general, LAMMPS can scale to more processors on larger systems (Figure 15-6).

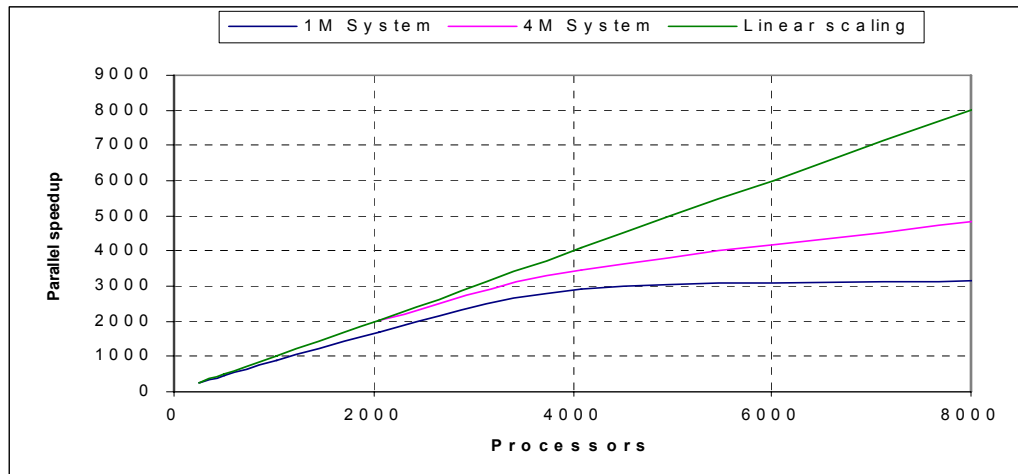


Figure 15-6 Parallel scaling of LAMMPS on Blue Gene/L (1M System: 1-million atom scaled rhodopsin, 4M System: 4-million atom scaled rhodopsin)

For a 1-million atom system, LAMMPS can scale up to 4096 nodes. For a larger system, such as a 4-million atom system, LAMMPS can scale up to 4096 nodes as well. As the size of the system increases, the scalability increases as well.

NAMD

NAMD is a parallel molecular dynamics application that was developed for high-performance calculations of large biological molecular systems.⁵⁴ NAMD supports the force fields used by AMBER, CHARMM,⁵⁵ and X-PLOR⁵⁶ and is also file compatible with these programs. This commonality allows simulations to migrate between these four programs. The C++ source for NAMD and Charm++ are freely available from UIUC. For additional information about NAMD, see the official NAMD Web site at:

<http://www.ks.uiuc.edu/Research/namd/>

NAMD incorporates the PME algorithm, which takes the full electrostatic interactions into account and reduces computational complexity. To further reduce the cost of the evaluation of long-range electrostatic forces, a multiple time step scheme is employed. The local interactions (bonded, van der Waals, and electrostatic interactions within a specified distance) are calculated at each time step. The longer range interactions (electrostatic interactions beyond the specified distance) are computed less often. An incremental load balancer monitors and adjusts the load during the simulation.

Due to the good balance of network and processor speed of the Blue Gene system, NAMD is able to scale to large processor counts (see Figure 15-7). While scalability is affected by many factors, many simulations can make use of multiple Blue Gene racks. Work by Kumar et al.⁵⁷ has reported scaling up to 8192 processors. Timing comparisons often use the “benchmark time” metric instead of wallclock time to completion. The benchmark time metric omits setup, I/O, and load balance overhead. While benchmark scaling can be considered a guide to what is possible, ideal load balance and I/O parameters for each case must be found for the wallclock time to scale similarly. Careful consideration of these parameters might be necessary to achieve the best scalability.

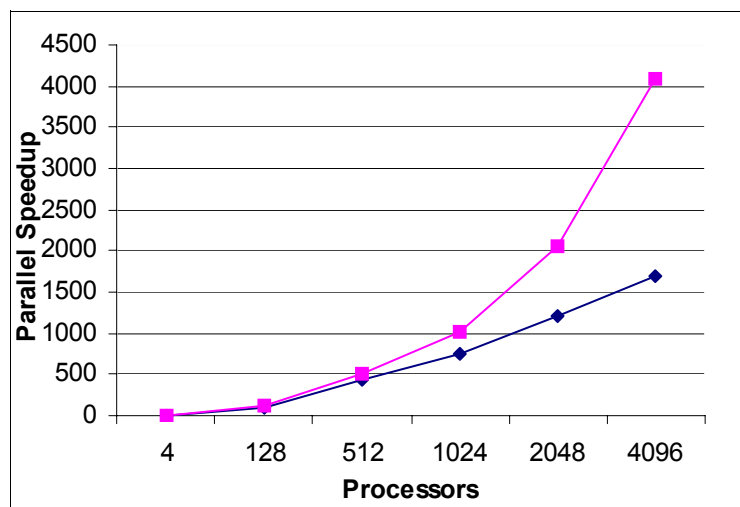


Figure 15-7 Parallel speedup on the Blue Gene/L system for the NAMD standard apoA1 benchmark

15.2.2 Molecular Docking applications

Applications in the area of Molecular Docking are becoming important in high-performance computing. In particular, in *silico screening* using molecular docking has been recognized as an approach that benefits from high-performance computing to identify novel small molecules that can then be used for drug design.⁵⁸ This process consists of the identification or selection of compounds that show activity against a biomolecule that is of interest as a drug target.⁵⁹

Docking programs place molecules into the active site of the receptor (or target biomolecule) in a non-covalent fashion and then rank them by the ability of the small molecules to interact with the receptor.⁶⁰ There is an extensive family of molecular docking software packages.⁶¹

DOCK6

DOCK is an open-source molecular docking software package that is frequently used in structure-based drug design.⁶² The computational aspects of this program can be divided into two parts. The first part consists of the *ligand atoms* that are located inside the cavity or binding pocket of a receptor, which is a large biomolecule. This step is carried out by a search algorithm.⁶³ The second part corresponds to scoring or identifying the most favorable interactions, which is normally done by means of a scoring function.⁶⁴

The latest version of the DOCK software package is version 6.1. However, in our work, we used version 6.0. This version is written in C++ to exploit code modularity and has been parallelized using the Message Passing Interface (MPI) paradigm. DOCK 6.0 is parallelized using a master-worker scheme.⁶⁵ The master handles I/O and tasks management, while each worker is given an individual molecule to perform simultaneous independent docking.⁶⁶

Recently, Peters, et al. have shown that DOCK6 is well suited for doing virtual screening on the Blue Gene/L or Blue Gene/P system.⁶⁷ Figure 15-8 shows the receptor HIV-1 reverse transcriptase in complex with nevirapine as used and described in in the Official UCSF DOCK Web site. The ligand library corresponds to a subset of 27,005 drug-like ligands from the ZINC database.⁶⁸ The scalability of the parallel version of the code is illustrated by constructing a set of ligands with 128,000 copies of nevirapine as recommended in the Official UCSF DOCK Web site to remove dependence on the order and size of the compound. You can find this Web site at:

<http://dock.compbio.ucsf.edu>

In Figure 15-8, the original code is the dark bar. Sorting by total number of atoms per ligand is the bar with horizontal lines. Sorting by total number of rotatable bonds per ligand is the white bar.⁶⁹

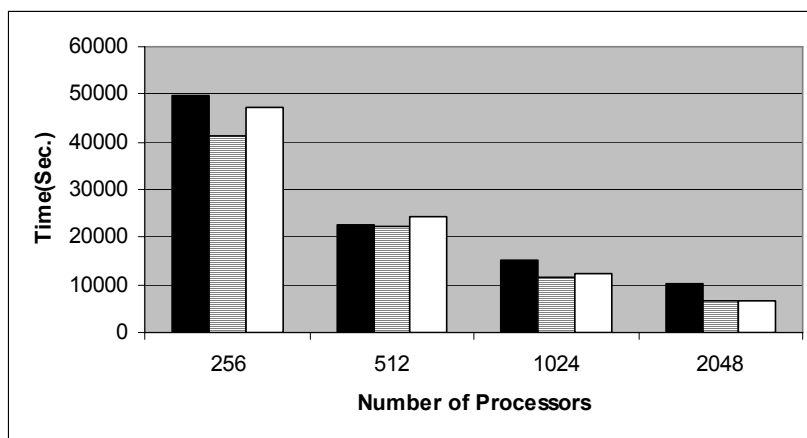


Figure 15-8 The effect of load balancing optimization for 27,005 ligands on 2048 processors

15.2.3 Electronic structure (Ab Initio) applications

Electronic structure calculations, such as the Hartree-Fock (HF) method, represent one of the simplest techniques in this area. However, even this first approximation tends to be computationally demanding. Many types of calculations begin with a Hartree-Fock calculation

and subsequently correct for electron-electron repulsion, which is also referred to as *electronic correlation*. The Møller-Plesset perturbation theory (MPn) and coupled cluster theory (CC) are examples of these post-Hartree-Fock methods.⁷⁰

A common characteristic of these techniques is that they are used to accurately compute molecular properties. As such, they tend to be widely available in high-performance computing. However, in addition to traditional electronic structure methods, Density Functional Theory-based methods have proven to be an attractive alternative to include correction effects and still treat large systems.

CPMD

The CPMD code is based on the original computer code written by Car and Parrinello.⁷¹ It was developed first at the IBM Research Zurich laboratory, in collaboration with many groups worldwide. It is a production code with many unique features written in Fortran and has grown from its original size of approximately 10,000 lines to currently close to 200,000 lines of code. Since January 2002, the program has been freely available for non-commercial use.⁷²

The basics of the implementation of the Kohn-Sham method using a plane-wave basis set and pseudopotentials are described in several review articles,⁷³ and the CPMD code follows them closely. All standard gradient-corrected density functionals are supported, and preliminary support for functionals that depend on the kinetic energy density is available. Pseudopotentials used in CPMD are either of the norm-conserving or the ultra-soft type.⁷⁴ Norm-conserving pseudopotentials have been the default method in CPMD, and only some of the rich functionality has been implemented for ultra-soft pseudopotentials.

The emphasis of CPMD on MD simulations of complex structures and liquids led to the optimization of the code for large supercells and a single k-point (the $k = 0$ point) approximation. Therefore, many features have only been implemented for this special case. CPMD has a rich set of features, many of which are unique. For a complete overview, refer to the CPMD manual.⁷⁵ The basic electronic structure method implemented uses fixed occupation numbers, either within a spin-restricted or an unrestricted scheme. For systems with a variable occupation number (small gap systems and metals), the free energy functional3 can be used together with iterative diagonalization methods.

15.2.4 Bioinformatics applications

The list of molecular biology databases is constantly increasing and more scientists rely on this information. The NAR Molecular Biology Database collection reported an increase of 139 more databases for 2006 compared to the previous year. enBank doubles its size approximately every 18 months. However, the increase in microprocessor clock speed is not changing at the same rate. Therefore, scientists try to leverage the use of multiple processors. In this section, we introduce some of these applications that are currently running on the Blue Gene supercomputer.

HMMER

For a complete discussion of hidden Markov models, refer to the work by Krogh et al.⁷⁶ HMMER 2.3.2 consists of nine different programs: hmalign, hmmbuild, hmmcalibrate, hmmconvert, hmmit, hmfetch, hmindex, hmmpfam, and hmmsearch.⁷⁷ Out of these nine programs, hmmcalibrate, hmmpfam, and hmmsearch have been parallelized. hmmcalibrate is used to identify statistical significance parameters for profile HMM. hmmpfam is used to search a profile HMM database. hmmsearch is used to carry out sequence database searches.⁷⁸

The first module tested corresponds to hmmcalibrate. Figure 15-9 on page 255 summarizes the performance of this module up to 2048 nodes.⁷⁹ Although this module was not optimized,

the parallel efficiency is still 75% on 2048 nodes. The graph in Figure 15-9 illustrates the performance of `hmmcalibrate` using only the first 327 entries in the Pfam database.⁸⁰

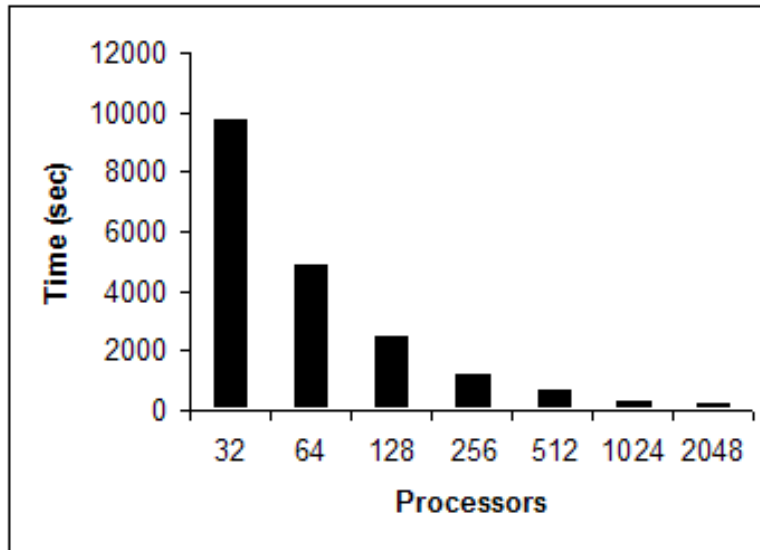


Figure 15-9 `hmmcalibrate` parallel performance using the first 327 entries of the Pfam database

Figure 15-10 illustrates the work presented by Jiang, et al.⁸¹ for optimizing `hmmsearch` parallel performance using 50 proteins of the globin family from different organisms and the UniProt release 8 database. For each processor count, the left bar shows the original PVM to MPI port. Notice scaling stops at 64 nodes. The second bar shows the multiple master implementation. The third bar shows the dynamic data collection implementation, and the right bar shows the load balancing implementation.

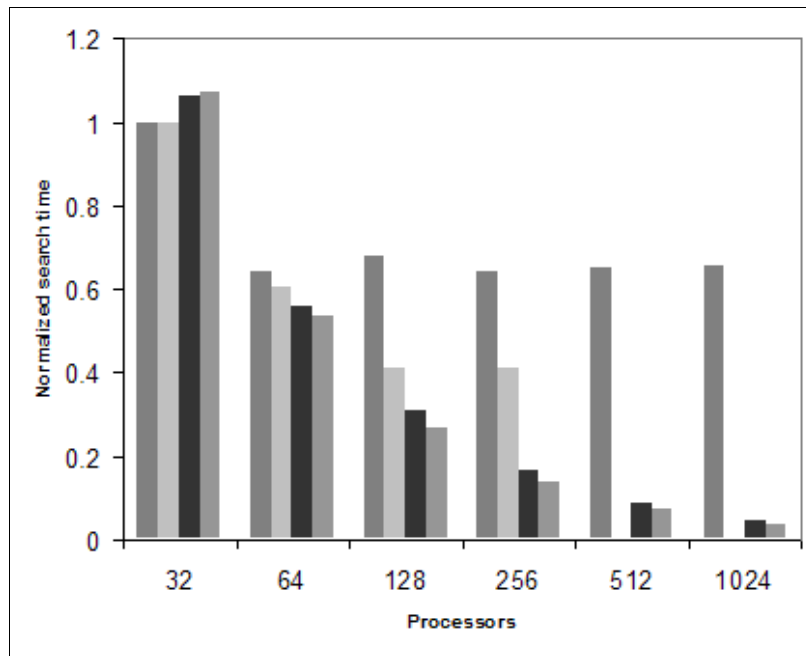


Figure 15-10 `hmmsearch` parallel performance

mpiBLAST-PIO

mpiBLAST is an open-source parallelization of BLAST that uses MPI.⁸² One of the key features of the initial parallelization of mpiBLAST is its ability to fragment and distribute databases.

Thorsen et al.⁸³ have compared the query *Arabidopsis thaliana*. This is a model organism for studying plant genetics. This query was further subdivided into small, medium, and large query sets that contain 200, 1168, and 28014 sequences, respectively.

Figure 15-11 illustrates the results of comparing three queries of three different sizes. We labeled them small, medium, and large. The database corresponds to NR. This figure shows that scalability is a function of the query size. The small query scales to approximately 1024 nodes in coprocessor mode with a parallel efficiency of 72% were the large query scales to 8,192 nodes with a parallel efficiency of 74%.

From the top of Figure 15-11, the thick solid line corresponds to ideal scaling. The thin solid line corresponds to the large query. The dashed line corresponds to the medium query. The dotted line corresponds to the small query.

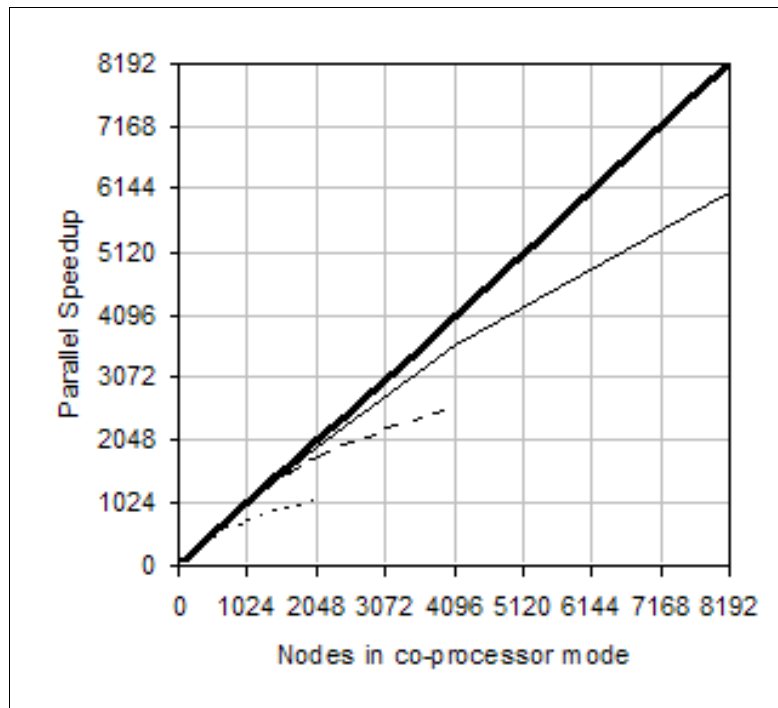


Figure 15-11 Scaling chart for queries run versus the nr database

15.2.5 Performance kernel benchmarks

Communication performance is an important aspect when running parallel applications, particularly, when running on a distributed-memory system such as the Blue Gene/P system. On both the Blue Gene/L and Blue Gene/P systems, instead of implementing a single type of network that is capable of transporting all protocols needed, these two systems have separate networks for different types of communications.

Usually two measurements provide information about the network and can be used to look at the parallel performance of applications:

Bandwidth	The number of MB of data that can be sent from a node to another node in one second
Latency	The amount of time it takes for the first byte sent from one node to reach its target node

These two values provide information about communication. In this section, we illustrate two simple cases. The first case corresponds to a benchmark that involves a single transfer. The second case corresponds to a collective as defined in the Intel MPI Benchmarks. Intel MPI Benchmarks is formerly known as “Pallas MPI Benchmarks” - PMB-MPI1 (for MPI1 standard functions only). Intel MPI Benchmarks - MPI1 provides a set of elementary MPI benchmark kernels.

For more details, see the product documentation included in the package that you can download from the Web at:

<http://www.intel.com/cd/software/products/asm-na/eng/219848.htm>

Intel MPI Benchmarks

The Intel MPI Benchmarks kernel or elementary set of benchmarks was reported as part of *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Here we describe and perform the same benchmarks. You can run all of the supported benchmarks, or just a subset, specified via the command line. The rules, such as time measurement, message lengths, selection of communicators to run a particular benchmark, are program parameters. For more information, see the product documentation that is included in the package, which you can download from the Web at:

http://www.intel.com/software/products/cluster/mpi/mpi_benchmarks_lic.htm

This set of benchmarks has the following objectives:

- ▶ Provide a concise set of benchmarks targeted at measuring important MPI functions: point-to-point message-passing, global data movement and computation routines, and one-sided communications and file I/O.
- ▶ Set forth precise benchmark procedures: run rules, set of required results, repetition factors, and message lengths.
- ▶ Avoid imposing an interpretation on the measured results: execution time, throughput, and global operations performance.

15.2.6 MPI point-to-point

In the Intel MPI Benchmarks, single transfer corresponds to PingPong and PingPing benchmarks. Here we illustrate a comparison between the Blue Gene/L and Blue Gene/P system for the case of PingPong. This benchmark illustrates a single message that was transferred between two MPI tasks, which in our case, is on two different nodes.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/L system, the benchmark was run in co-processor mode, which is defined in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. On the Blue Gene/P system, we used the SMP Node Mode.

Example 15-1 shows how `mpirun` was invoked on the Blue Gene/L system.

Example 15-1 mpirun on the Blue Gene/L system

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0
BGL_APP_L1_SWOA=0" -partition R000 -cwd /bglscratch/pallas -exe
/bglscratch/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512
PingPong" | tee IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Example 15-2 shows how `mpirun` was invoked on the Blue Gene/P system.

Example 15-2 mpirun on the Blue Gene/P system

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP -partition R01-M1 -cwd
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB
.perf.rts -args "-msglen 4194304.txt -npmin 512 PingPong" | tee
IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Figure 15-12 shows the bandwidth on the torus network as a function of the message size, for one simultaneous pair of nearest neighbor communications. The protocol switch from short to eager is visible in these two cases, where the eager to rendezvous switch is most pronounced on the Blue Gene/L system. This figure also shows the improved performance on the Blue Gene/P system. Notice in Figure 15-12 that the diamonds corresponds to the Blue Gene/P system and the asterisks (*) to correspond to the Blue Gene/L system.

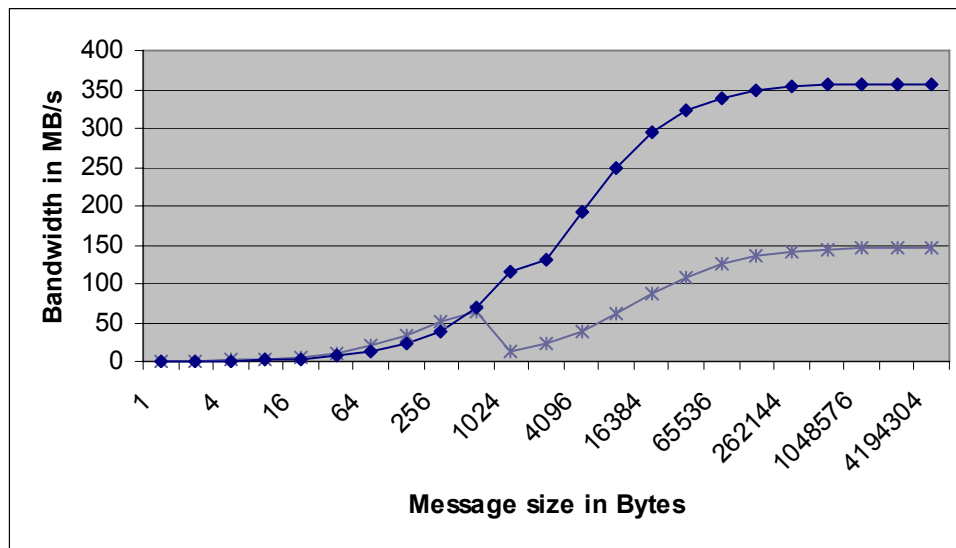


Figure 15-12 Bandwidth versus message size

MPI collective benchmarks

In the Intel MPI Benchmarks, collective benchmarks correspond to Bcast, Allgather, Allgatherv, Alltoall, Alltoallv, Reduce, Reduce_scatter, Allreduce, and Barrier benchmarks. Here we illustrate a comparison between the Blue Gene/L and Blue Gene/P system for the case of Allreduce, which is a popular collective that is used in certain scientific applications. These benchmarks measure the message passing power of a system as well as the quality of the implementation.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/P system, the benchmark was run in co-processor mode, which is defined in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. On the Blue Gene/P system, we used SMP Node Mode.

Example 15-3 shows how `mpirun` was invoked on the Blue Gene/L system.

Example 15-3 mpirun on the Blue Gene/L system

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0
BGL_APP_L1_SWOA=0" -partition R000 -cwd
/bglscratch/BGTH/testsmall1512nodeBGL/pallas -exe
/bglscratch/BGTH/testsmall1512nodeBGL/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen
4194304.txt -npmin 512 Allreduce" | tee
IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Example 15-4 shows how `mpirun` was invoked on the Blue Gene/P system.

Example 15-4 mpirun on the Blue Gene/P system

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP -partition R01-M1 -cwd
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB
.perf.rts -args "-msglen 4194304.txt -npmin 512 Allreduce" | tee
IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Collective operations are more efficient on the Blue Gene/P system. You should try to use these operations instead of point-to-point communication wherever possible. The overhead for point-to-point communications is much larger than those for collectives. Unless all your point-to-point communication is purely the nearest neighbor, it is also difficult to avoid network congestion on the torus network.

Alternatively, collective operations can use the barrier (global interrupt) network or the torus network. If they run over the torus network, they can still be optimized by using specially designed communication patterns that achieve optimum performance. Doing this manually with point-to-point operations is possible in theory, but in general, the implementation in the Blue Gene/P MPI library offers superior performance.

With point-to-point communication, the goal of reducing the point-to-point Manhattan distances necessitates a good mapping of MPI tasks to the physical hardware. For collectives, mapping is equally important because most collective implementations prefer certain communicator shapes to achieve optimum performance. The technique of mapping is illustrated in Appendix E, "Mapping" on page 281.

Similar to point-to-point communications, collective communications also works best if you do not use complicated derived data types and if your buffers are aligned to 16-byte boundaries.

While the MPI standard explicitly allows for MPI collective communications to occur at the same time as point-to-point communications (on the same communicator), we generally do not recommend that you allow this to happen for performance reasons.

Table 15-1 summarizes the MPI collectives that have been optimized on the Blue Gene/P system, together with their performance characteristics when executed on the various networks of the Blue Gene/P system.

Table 15-1 MPI collectives that have been optimized on the Blue Gene/P system

MPI routine	Condition	Network	Performance
MPI_Barrier	MPI_COMM_WORLD	Barrier (global interrupt) network	1.2 μ s
MPI_Barrier	Any communicator	Torus network	30 μ s
MPI_Broadcast	MPI_COMM_WORLD	Collective network	817 MBps
MPI_Broadcast	Rectangular communicator	Torus network	934 MBps
MPI_Allreduce	MPI_COMM_WORLD fixed-point	Collective network	778 MBps
MPI_Allreduce	MPI_COMM_WORLD floating point	Collective network	98 MBps
MPI_Alltoall[v]	Any communicator	Torus network	84-97% peak
MPI_Allgatherv	N/A	Torus network	Same as broadcast

Figure 15-13 shows a comparison between the Blue Gene/L and Blue Gene/P systems for the MPI_Allreduce() type of communication.

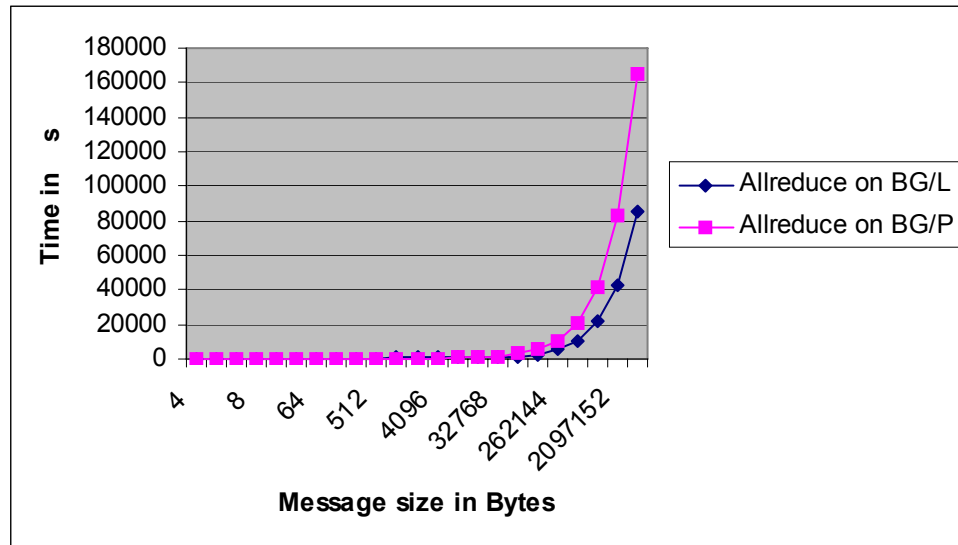


Figure 15-13 MPI_Allreduce () performance on 512 nodes

Figure 15-14 illustrates the performance of the barrier on Blue Gene/P for up to 32 nodes.

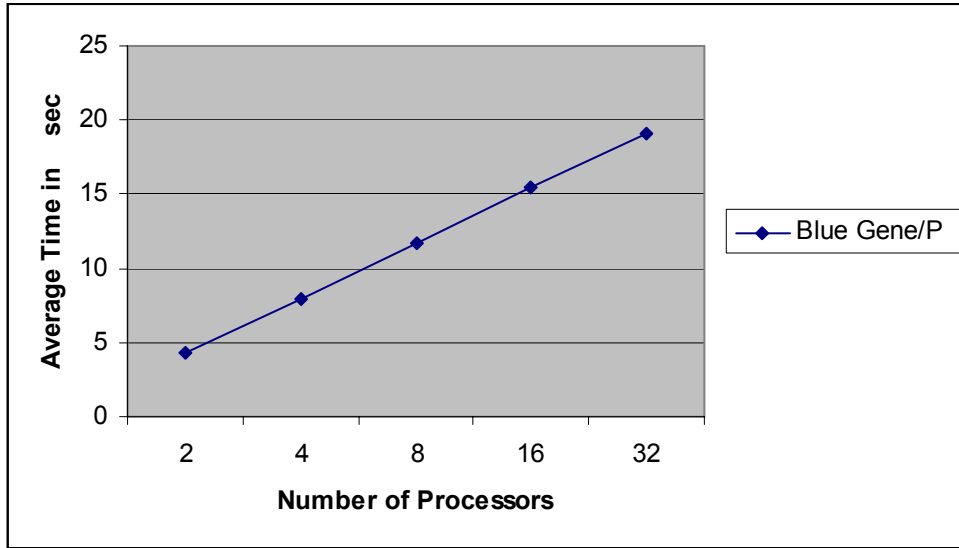


Figure 15-14 Barrier performance on the Blue Gene/P system

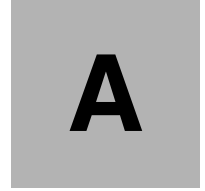


Part 5

Appendixes

In this part, we provide additional information about system administration for the Blue Gene/P system. This part includes the following appendixes:

- ▶ Appendix A, “Blue Gene/P hardware naming convention” on page 265
- ▶ Appendix B, “Header files and libraries” on page 271
- ▶ Appendix C, “Files on architectural features” on page 275
- ▶ Appendix D, “Porting applications” on page 279
- ▶ Appendix E, “Mapping” on page 281
- ▶ Appendix F, “Statement of completion” on page 285



Blue Gene/P hardware naming convention

In this appendix, we present an overview of how the Blue Gene/P hardware locations are assigned. This naming is used consistently throughout both the hardware and software.

Figure A-1 shows the conventions that are used when assigning locations to all hardware except the various cards in a Blue Gene/P system. Using the charts and diagrams that follow, consider an example where you have an error in the fan named R23-M1-A3-0. This naming convention tells you where to look for the error. In Figure A-1, in the upper left corner, you see that racks use the convention Rxx. Looking at our error message, we can see that the rack involved is R23. From the chart in Figure A-1, we see that R23 is the fourth rack in row two. (Remember that all numbering starts with 0). The bottom midplane of any rack is 0. Therefore, we are dealing with the top midplane (R23-M1).

In the chart, you can see in the fan assemblies description that assemblies zero through four are on the front of the rack, bottom to top, respectively. Therefore, we check for an attention light (Amber LED) on the fan assembly second from the top, because the front-most fan is the one that is causing the error message to surface. Service, link, and node cards use a similar form of addressing.

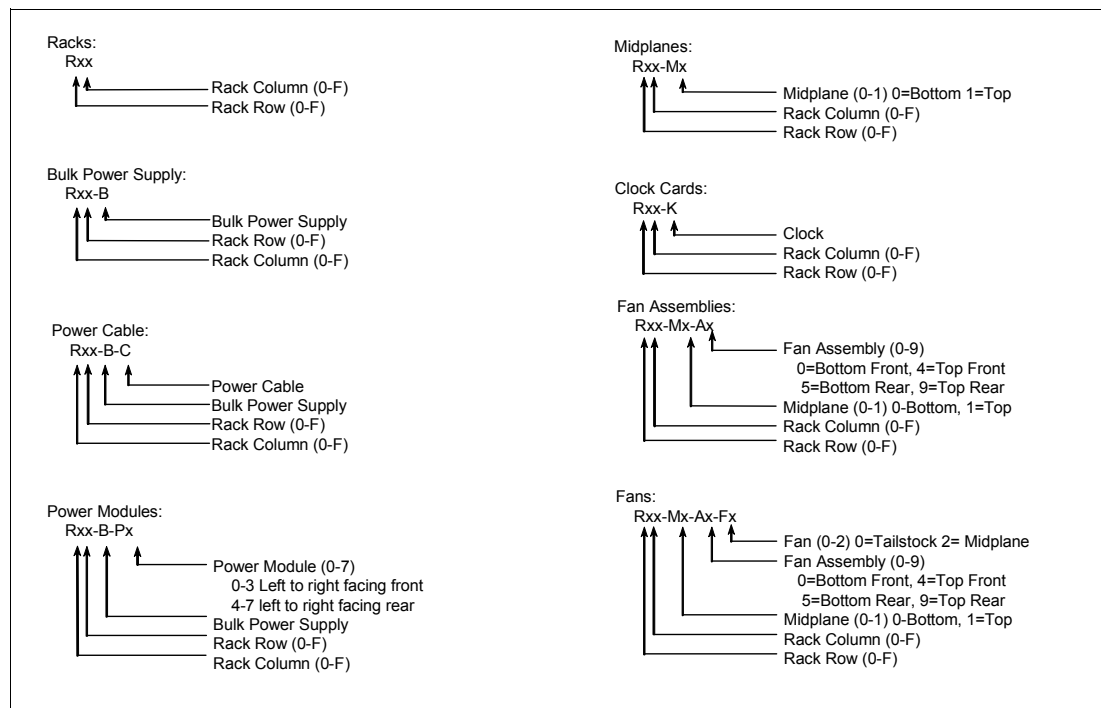


Figure A-1 Hardware naming conventions

Figure A-2 shows the conventions used for the various card locations.

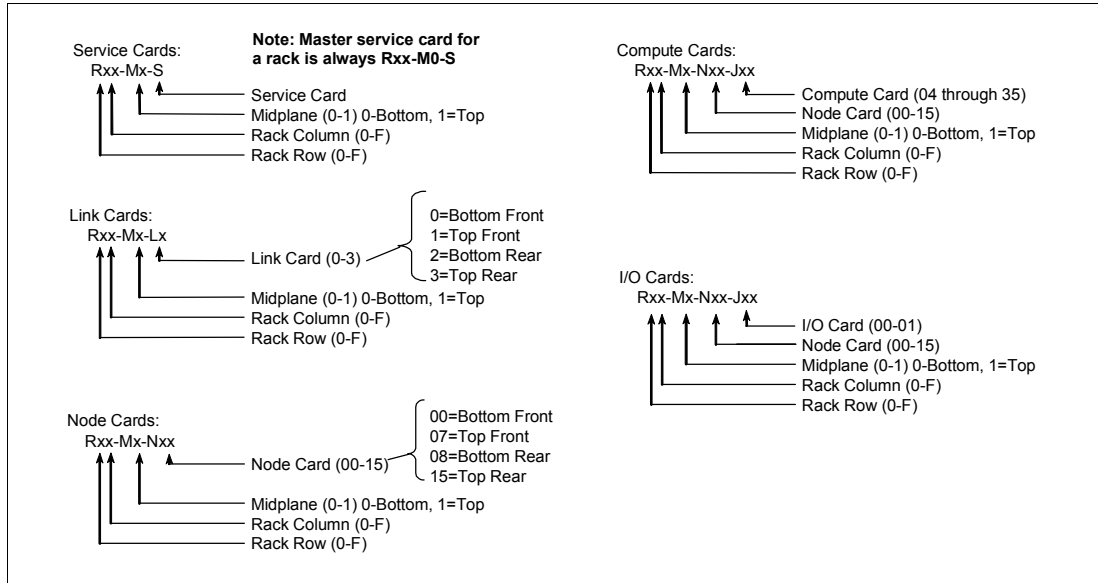


Figure A-2 Card naming conventions

Table A-1 contains examples of various hardware conventions. The figures that follow the table provide illustrations of the actual hardware.

Table A-1 Example of hardware name conventions

Card	Element	Name	Example
Compute	Card	J04 through J35	R23-M10-N02-J09
I/O	Card	J00 through J01	R57-M1-N04-J00
I/O & Compute	Module	U00	R23-M0-N13-J08-U00
Link	Module	U00 through U05 (00 left most, 05 right most)	R32-M0-L2_U03
Link	Port	TA through TF	R01-M0-L1-U02-TC
Link data cable	Connector	J00 through J15 (as labeled on link card)	R21-M1-L2-J13
Node Ethernet	Connector	EN0, EN1	R16-M1-N14-EN1
Service	Connector	Control FPGA, control network, Clock R, Clock B	R05-M0-S-Control FPGA
Clock	Connector	Input, Output 0 through Output 9	R13-K- Output 3

Figure A-3 shows the layout of a 64-rack system.

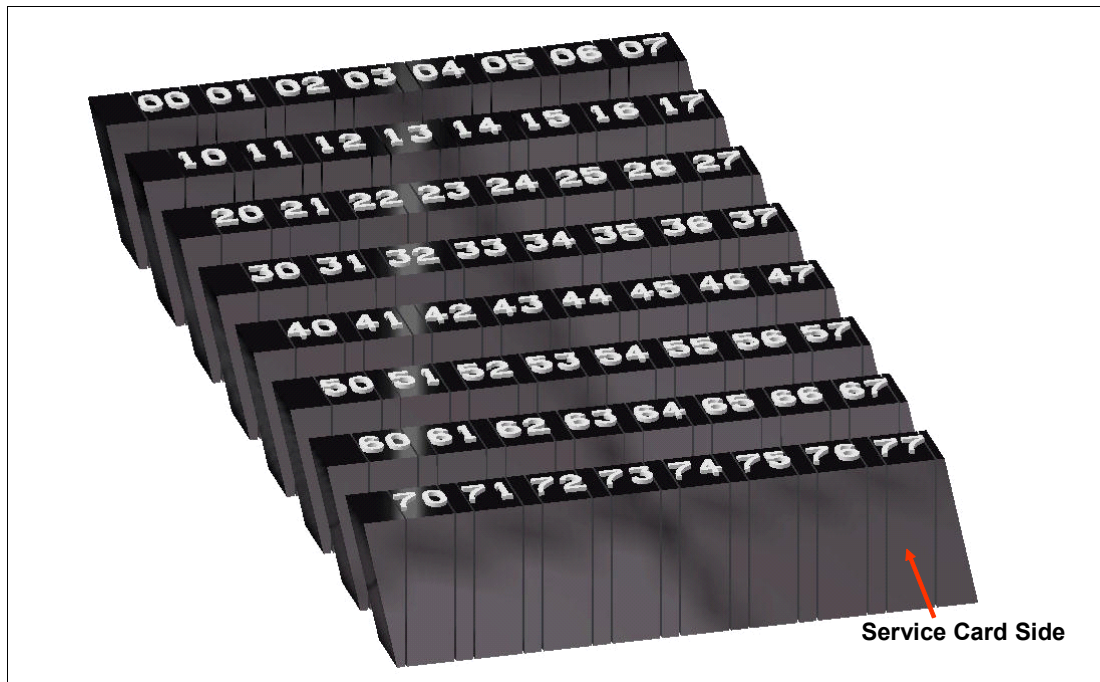


Figure A-3 Rack numbering

Note: The fact that Figure A-3 shows numbers 00 through 77 does not imply that this is the largest configuration possible. The largest configuration possible is 256 racks numbered 00 through FF.

Figure A-4 identifies each of the cards in a single midplane.

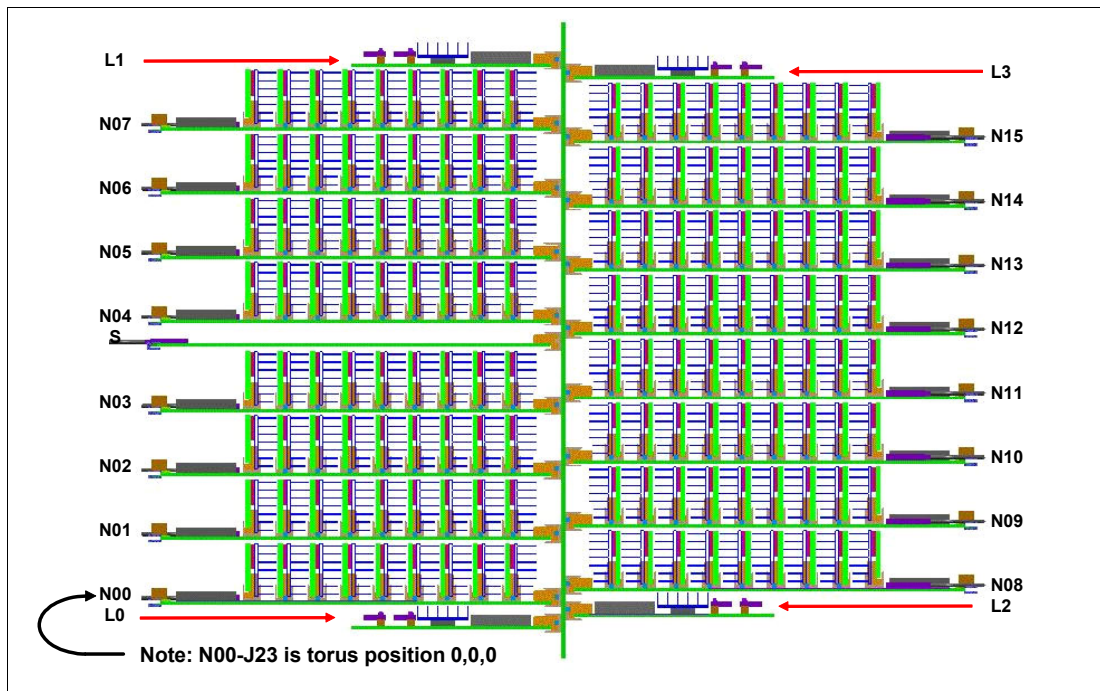


Figure A-4 Positions of the node, link, and service cards

Figure A-5 shows a diagram of a node card. On the front of the card are Ethernet ports EN0 and EN1. The first nodes behind the Ethernet ports are the I/O Nodes. In this diagram, the node card is fully populated with I/O Nodes, meaning that it has two I/O Nodes. Behind the I/O Nodes are the Compute Nodes.

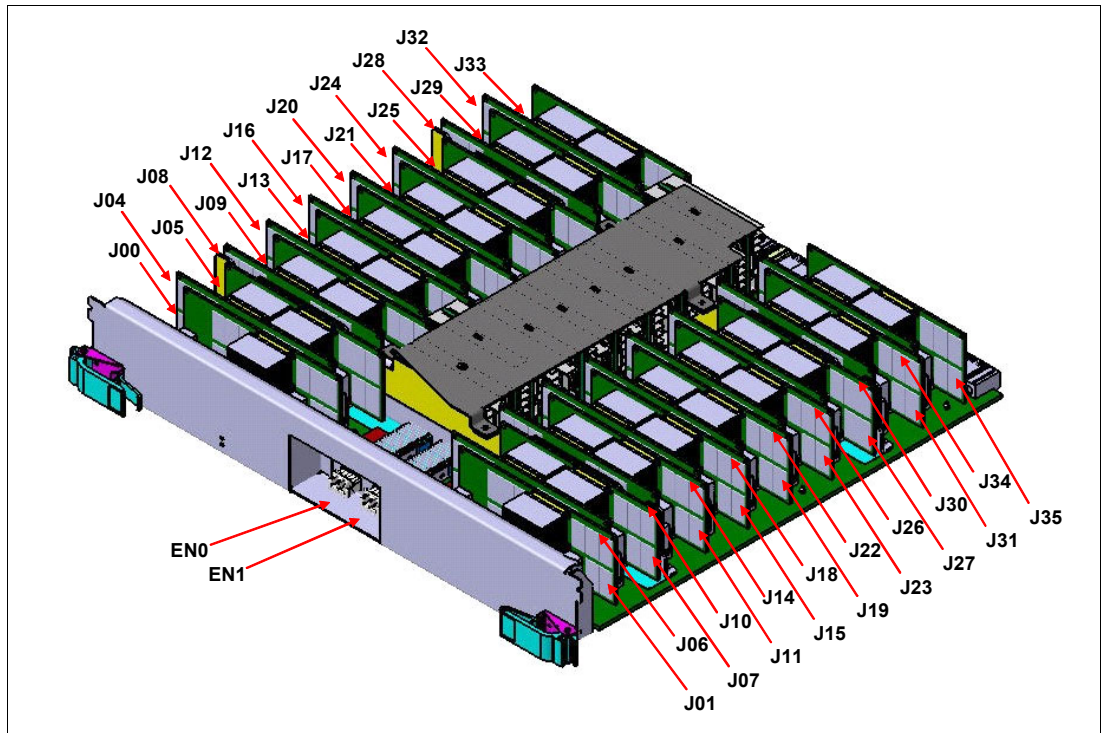


Figure A-5 Node card diagram

Figure A-6 is an illustration of a service card.

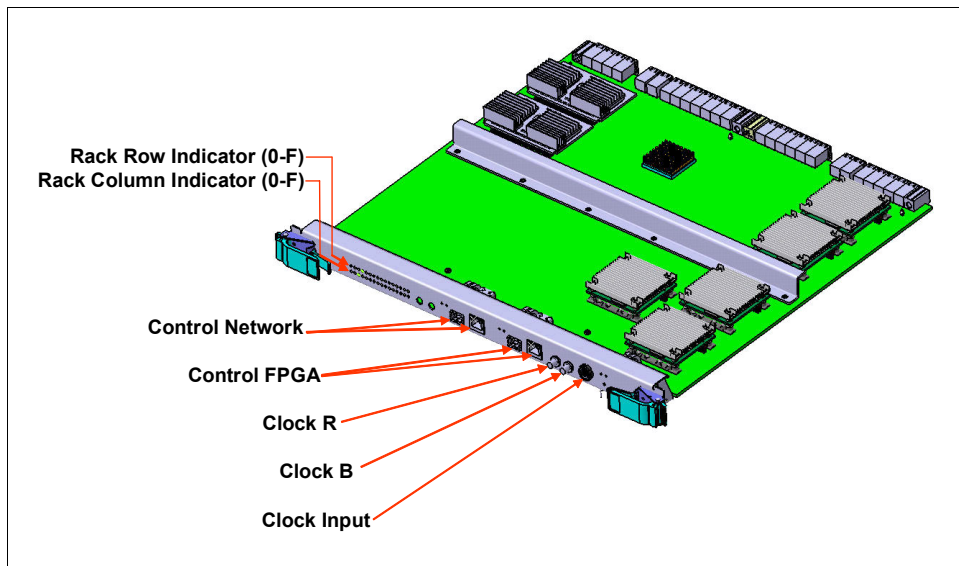


Figure A-6 Service card

Figure A-7 shows the link card. The locations identified as J00 through J15 are the link card connectors. The link cables are routed from one link card to another to form the torus network between the midplanes.

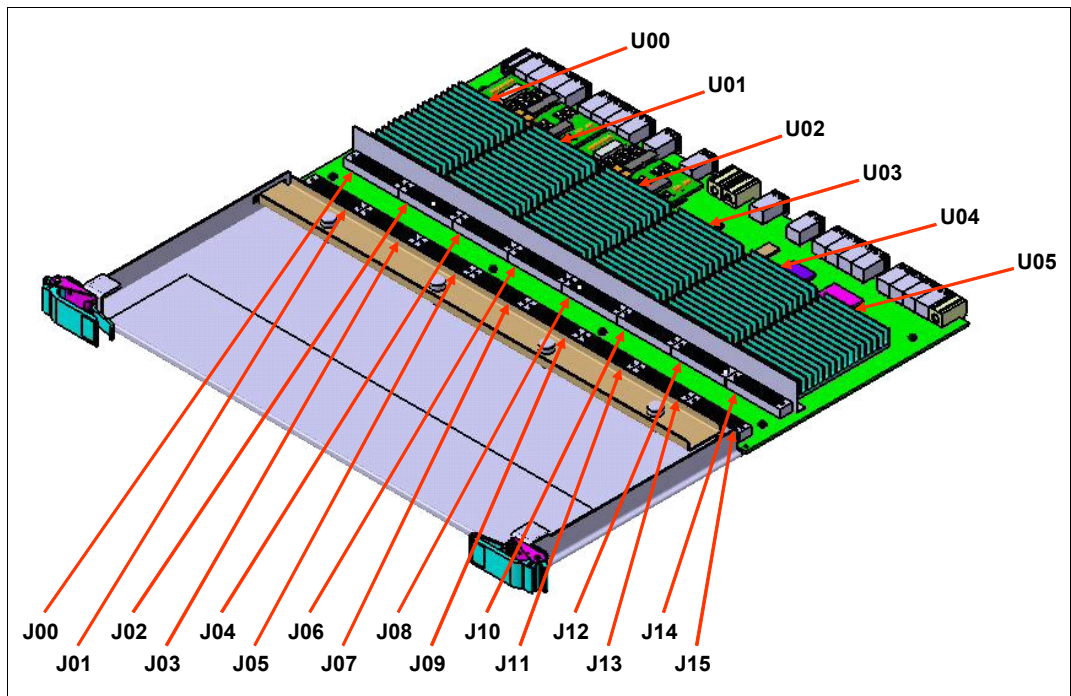


Figure A-7 Link card

Figure A-8 shows the clock card. If the clock is a secondary or tertiary clock, there will be a cable coming to the input connector on the far right. Next to the input (just to the left) is the master and worker toggle switch. All clock cards are built with the capability of filling either role. If the clock is a secondary or tertiary clock, this must be set to *worker*. Output zero through nine can be used to send signals to midplanes throughout the system.

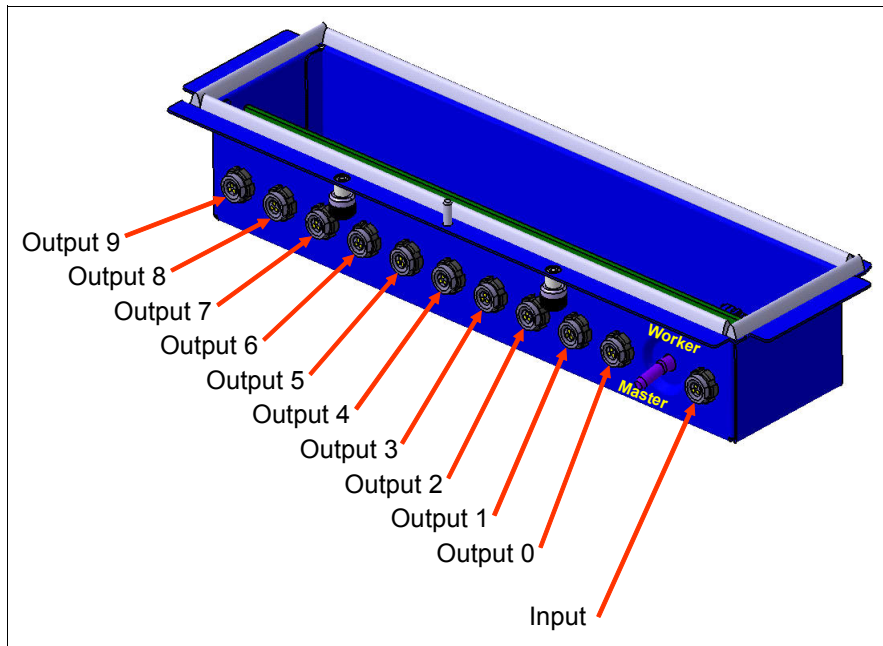


Figure A-8 Clock card



B

Header files and libraries

In this appendix, we provide information about selected header files and libraries for the Blue Gene/P system. Directories that contain header files and libraries for the Blue Gene/P system are under the main system path in the `/bgsys/drivers/ppcfloor` directory.

Blue Gene/P applications

Blue Gene/P applications run on the Blue Gene/P compute or I/O Nodes. Table B-1 describes the header files in the `/bgsys/drivers/ppcfloor/comm/include` directory.

Table B-1 Header files in /bgsys/drivers/ppcfloor/comm/include

File name	Description
dcmf.h	Common BGP message layer interface
dcmf_collectives.h	Common BGP message layer interface for general collectives
mpe_thread.h	Multi-processing environment (MPE) routines
mpicxx.h	MPI GCC script routine naming
mpif.h	MPI Fortran parameters
mpi.h	MPI C defines
mpiof.h	MPI I/O Fortran programs
mpio.h	MPI I/O C includes
mpix.h	Blue Gene/P extensions to the MPI specifications

Table B-2 describes the header files in the `/bgsys/drivers/ppcfloor/arch/include/common` directory.

Table B-2 Header files in /bgsys/drivers/ppcfloor/arch/include/common

File name	Description
bgp_personality.h	Defines personality
bgp_personality_inlines.h	Static inline for personality
bgp_personalityP.h	Defines personality processing

Table B-3 describes the 32-bit static and dynamic libraries in the `/bgsys/drivers/ppcfloor/comm/lib` directory.

Table B-3 32-bit static and dynamic libraries in /bgsys/drivers/ppcfloor/comm/lib/

File name	Description
libdcmf.cnk.a, libdcmf.cnk.so	Common BGP message layer interface in C
libdcmfcoll.cnk.a, libdcmfcoll.cnk.so	Common BGP message layer interface for general collectives in C
libmpich.cnk.a, libmpich.cnk.so	C bindings for MPI
libcxxmpich.cnk.a,	C++ bindings for MPI
libfmpich.cnk.a, libfmpich.cnk.so	Fortran bindings for MPI
libfmpich_.cnk.a	Fortran bindings for MPI with extra underscoring

Resource management APIs

Blue Gene/P resource management applications run on the Service Node. Table B-4 describes the header files used by resource management applications. They are located in the `/bgsys/drivers/ppcfloor/include` directory.

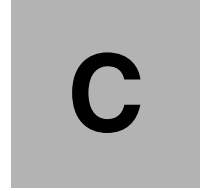
Table B-4 Header files for resource management APIs

File name	Description
<code>allocator_api.h</code>	Available for applications using the Dynamic Partition Allocator APIs
<code>attach_bg.h</code>	The Blue Gene/P version of <code>attach.h</code> , which is described in the Message Passing Interface (MPI) debug specification
<code>rm_api.h</code>	Available for applications that use Bridge APIs
<code>rt_api.h</code>	Available for applications that use Real-time Notification APIs
<code>sayMessage.h</code>	Available for applications that use <code>sayMessage</code> APIs
<code>sched_api.h</code>	Available for applications that use the <code>mpirun</code> plug-in interface

Table B-5 describes the 64-bit dynamic libraries that are available to resource management applications. They are located in the `/bgsys/drivers/ppcfloor/lib64` directory.

Table B-5 64-bit dynamic libraries for resource management APIs

File Name	Description
<code>libbgpallocator.so</code>	Required when using the Dynamic Partition Allocator APIs
<code>libbgprealtime.so</code>	Required when using the Real-time Notification APIs
<code>libbgpbridge.so</code>	Required when using the Bridge APIs
<code>libsaymessage.so</code>	Required when using the <code>sayMessage</code> APIs



Files on architectural features

System calls that provide access to certain hardware or system features can be accessed by applications. In this appendix, we illustrate how to obtain hardware-related information.

Personality of Blue Gene/P

The personality of a Blue Gene/P node is static data given to every Compute Node and I/O Node at boot time by the control system. This data contains information that is specific to the node, with respect to the block that is being booted.

The personality is a set of C language structures that contain such items as the node's coordinates on the torus network. This kind of information can be useful if the application programmer wants to determine, at run time, where the tasks of the application are running. It can also be used to tune certain aspects of the application at run time, such as determining which set of tasks share the same I/O Node and then optimizing the network traffic from the Compute Nodes to that I/O Node.

Example of running personality on Blue Gene/P

Example C-1 illustrates how to invoke and print selected hardware features.

Example: C-1 personali.c architectural features program

```
/* ----- */
/* Example: architectural features */
/* Written by: Bob Walkup */
/* IBM Watson, Yorktown, NY */
/* September 17, 2007 */
/* ----- */

#include <mpi.h>
#include <stdio.h>

#include <spi/kernel_interface.h>
#include <common/bgp_personality.h>
#include <common/bgp_personality_inlines.h>

int main(int argc, char * argv[])
{
    int taskid, ntasks;
    int memory_size_MBytes;
    _BGP_Personality_t personality;
    int torus_x, torus_y, torus_z;
    int pset_size, pset_rank, node_config;
    int xsize, ysize, zsize, procid;
    char location[128];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

    Kernel_GetPersonality(&personality, sizeof(personality));

    if (taskid == 0)
    {
        memory_size_MBytes = personality.DDR_Config.DDRSizeMB;
        printf("Memory size = %d MBytes\n", memory_size_MBytes);

        node_config = personality.Kernel_Config.ProcessConfig;
```

```

if      (node_config == _BGP_PERS_PROCESSCONFIG_SMP) printf("SMP mode\n");
else if (node_config == _BGP_PERS_PROCESSCONFIG_VNM) printf("Virtual-node mode\n");
else if (node_config == _BGP_PERS_PROCESSCONFIG_2x2) printf("Dual mode\n");
else
    printf("Unknown mode\n");

printf("number of MPI tasks = %d\n", ntasks);

xsize = personality.Network_Config.Xnodes;
ysize = personality.Network_Config.Ynodes;
zsize = personality.Network_Config.Znodes;

pset_size = personality.Network_Config.PSetSize;
pset_rank = personality.Network_Config.RankInPSet;

printf("number of processors in the pset = %d\n", pset_size);
printf("torus dimensions = <%d,%d,%d>\n", xsize, ysize, zsize);
}

torus_x = personality.Network_Config.Xcoord;
torus_y = personality.Network_Config.Ycoord;
torus_z = personality.Network_Config.Zcoord;

BGP_Personality_getLocationString(&personality, location);

procid = Kernel_PhysicalProcessorID();

/*-----*/
/* print torus coordinates and the node location */
/*-----*/
printf("MPI rank %d has torus coords <%d,%d,%d>  cpu = %d, location = %s\n",
    taskid, torus_x, torus_y, torus_z, procid, location);

MPI_Finalize();
return 0;
}

```

Example C-2 illustrates the makefile that is used to build `personality.c`. This particular file uses the GNU compiler.

Example: C-2 Makefile to build the `personality.c` program

```

BGP_FLOOR = /bgsys/drivers/ppcfloor
BGP_IDIRS = -I$(BGP_FLOOR)/arch/include

CC        = /bgsys/drivers/ppcfloor/comm/bin/mpicc

EXE       = personality
OBJ       = personality.o
SRC       = personality.c
FLAGS    =
FLD       =

$(EXE): $(OBJ)
    ${CC} $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC)

```

```
 ${CC} $(FLAGS) $(BGP_IDIRS) -c $(SRC)
```

clean:

```
 rm personality.o personality
```

Example C-3 shows a section of the output that is generated after running **personality** using XYZ mapping. (See Appendix E, “Mapping” on page 281). Notice that the output has been ordered by MPI rank for readability.

Example: C-3 Output generated after running personality

```
 /bgsys/drivers/ppcfloor/bin/mpirun -partition N04_32_1 -label -env "BG_MAPPING=XYZ" -mode VN  
 -np 8 -cwd `pwd` -exe personality | tee personality_VN_8_XYZ.out
```

Memory size = 2048 MBytes

Virtual-node mode

number of MPI tasks = 128

number of processors in the pset = 32

torus dimensions = <4,4,2>

```
 MPI rank 0 has torus coords <0,0,0>  cpu = 0, location = R00-M0-N04-J23  
 MPI rank 1 has torus coords <0,0,0>  cpu = 1, location = R00-M0-N04-J23  
 MPI rank 2 has torus coords <0,0,0>  cpu = 2, location = R00-M0-N04-J23  
 MPI rank 3 has torus coords <0,0,0>  cpu = 3, location = R00-M0-N04-J23  
 MPI rank 4 has torus coords <1,0,0>  cpu = 0, location = R00-M0-N04-J04  
 MPI rank 5 has torus coords <1,0,0>  cpu = 1, location = R00-M0-N04-J04  
 MPI rank 6 has torus coords <1,0,0>  cpu = 2, location = R00-M0-N04-J04  
 MPI rank 7 has torus coords <1,0,0>  cpu = 3, location = R00-M0-N04-J04
```

Example C-4 illustrates running **personality** with XYZT mapping for a comparison. Notice that the output has been ordered by MPI rank for readability.

Example: C-4 Output generated after running personality

```
 /bgsys/drivers/ppcfloor/bin/mpirun -partition N04_32_1 -label -env "BG_MAPPING=XYZT" -mode VN  
 -np 8 -cwd `pwd` -exe personality | tee personality_VN_8_XYZT.out
```

Memory size = 2048 MBytes

Virtual-node mode

number of MPI tasks = 128

number of processors in the pset = 32

torus dimensions = <4,4,2>

```
 MPI rank 0 has torus coords <0,0,0>  cpu = 0, location = R00-M0-N04-J23  
 MPI rank 1 has torus coords <1,0,0>  cpu = 0, location = R00-M0-N04-J04  
 MPI rank 2 has torus coords <2,0,0>  cpu = 0, location = R00-M0-N04-J12  
 MPI rank 3 has torus coords <3,0,0>  cpu = 0, location = R00-M0-N04-J31  
 MPI rank 4 has torus coords <0,1,0>  cpu = 0, location = R00-M0-N04-J22  
 MPI rank 5 has torus coords <1,1,0>  cpu = 0, location = R00-M0-N04-J05  
 MPI rank 6 has torus coords <2,1,0>  cpu = 0, location = R00-M0-N04-J13  
 MPI rank 7 has torus coords <3,1,0>  cpu = 0, location = R00-M0-N04-J30
```



Porting applications

In this appendix, we summarize Appendix A, “BG/L prior to porting code,” in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Porting applications to massively parallel systems requires special considerations to take full advantage of this specialized architectures. Never underestimate the effort that is required to port a code to any new hardware. The amount of effort depends on the nature of the way in which the code has been implemented.

Answer the following questions to help you in the decision process of porting applications:

1. Is the code already running in parallel?
2. Is the application addressing 32-bit?
3. Does the application relies on system calls, for example, **system**?
4. Does the code use the Message Passing Interface (MPI), specifically MPICH? Although there are several parallel programming application programming interfaces (APIs), the only one supported on the Blue Gene/P system that is portable is MPICH. OpenMP is supported only on individual nodes.
5. Is the code Single Process, Multiple Data (SPMD) and not multiple process, multiple data (MPMD)? The Blue Gene/P system only supports SPMD, same program everywhere, style of parallel programming.
6. Is the memory requirement per MPI task less than 1 GB?
7. Is the code computational-intensive? That is, is there a small amount of I/O compared to computation?
8. Is the code floating point-intensive? This allows the double floating-point capability of the Blue Gene/P system to be exploited.
9. Does the algorithm allow for distributing the work to a large number of nodes?
10. Have you ensured that the code does not use flex_{lm} licensing? At present, there is no flex_{lm} library support for Linux on System p™.

If you have answered “yes” to all of these questions, then you must answer the following questions:

- ▶ Has the code been ported to Linux on System p?
- ▶ Is the code Open Source Software (OSS)? This type of applications require the use of the GNU standard **configure** and special considerations are required.⁸⁴
- ▶ Can the problem size be increased with increased numbers of processors?
- ▶ Do you use standard input? If yes, can this be changed to single file input?



Mapping

Mapping Message Passing Interface (MPI) tasks to Blue Gene/L nodes are discussed in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. In this appendix, we summarize and discuss mapping of these tasks with respect to the Blue Gene/P system.

In general, *mapping* is the process of assigning tasks to processors.⁸⁵ In this appendix, we define mapping as an assignment of MPI ranks onto Blue Gene processors. For both the Blue Gene/L and Blue Gene/P systems, the network topology is a three-dimensional (3D) torus or mesh, with direct links between the nearest neighbors in the +/-x, +/-y, and +/-z directions. When communication involves the nearest neighbors on the torus network, you can obtain a large fraction of the theoretical peak bandwidth. However, when MPI ranks communicate with many hops between the neighbors, the effective bandwidth is reduced by a factor that is equal to the average number of hops that messages take on the torus network. In a number of cases, it is possible to control the placement of MPI ranks so that communication remains local. This can significantly improve scaling for a number of applications, particularly at large processor counts.

The default mapping is to place MPI ranks on the system in XYZT order, where $\langle X, Y, Z \rangle$ are torus coordinates and T is the processor number within each node ($T=0, 1, 2, 3$). If the job uses symmetrical multiprocessing (SMP) Node Mode on the Blue Gene/P system, only one MPI rank is assigned to each node that is using processor 0. For SMP Node Mode and the default mapping, we get the following results:

- ▶ MPI rank 0 is assigned to $\langle X, Y, Z, T \rangle$ coordinates $\langle 0, 0, 0, 0 \rangle$.
- ▶ MPI rank 1 is assigned to $\langle X, Y, Z, T \rangle$ coordinates $\langle 1, 0, 0, 0 \rangle$.
- ▶ MPI rank 2 is assigned to $\langle X, Y, Z, T \rangle$ coordinates $\langle 2, 0, 0, 0 \rangle$.

The results continue like this, first incrementing the X coordinate, then the Y coordinate, and then the Z coordinate. In Virtual Node Mode and in Dual Node Mode, the same XYZT order remains the default.

For example, in Virtual Node Mode, the system first places one MPI rank using processor 0 on each of the nodes in XYZ order. The next MPI ranks are assigned to processor 1, again in XYZ order, and so forth. In many cases, it might be better to change this assignment so that the first four MPI ranks use processors 0,1,2,3 on the first node, then the next four ranks use processors 0,1,2,3 on the second node, where the nodes are populated in XYZ order. This ordering is called *TXYZ order* (first increment T, then X, then Y, and then Z).

Table E-1 illustrates this type of mapping using the output from the personality program presented in Appendix C, “Files on architectural features” on page 275.

Table E-1 Topology mapping 4x4x2 with TXYZ and XYZT

Mapping option	Topology	Coordinates	Processor
TXYZ	4x4x2	0,0,0	0
		0,0,0	1
		0,0,0	2
		0,0,0	3
		1,0,0	0
		1,0,0	1
		1,0,0	2
		1,0,0	3
XYZT	4x4x2	0,0,0	0
		1,0,0	0
		2,0,0	0
		3,0,0	0
		0,1,0	0
		1,1,0	0
		2,1,0	0
		3,1,0	0

The way to specify a mapping depends on the method that is used for job submission. The `mpi run` command for the Blue Gene/P system includes two methods to specify the mapping. You can add `-mapfile TXYZ` to request TXYZ order. Other permutations of XYZT are also permitted. You can also create a map file, and use `-mapfile my.map`, where *my.map* is the name of your map file. Alternatively, you can specify the environment variable `-env BG_MAPPING=TXYZ` to obtain one of the predefined non-default mappings.

The use of a customized map file provides the most flexibility. The syntax for the map file is simple. It must contain one line for each MPI rank in the Blue Gene partition, with four integers on each line separated by spaces, where the four integers specify the <X,Y,Z,T> coordinates for each MPI rank. The first line in the map file assigns MPI rank 0, the second line assigns MPI rank 1, and so forth. It is important to ensure that your map file is consistent, with a unique relationship between MPI rank and <X,Y,Z,T> location.

General guidance

For applications that use a 1D, 2D, 3D, or 4D (D for dimensional) logical decomposition scheme, it is often possible to map MPI ranks onto the Blue Gene torus network in a way that preserves locality for nearest-neighbor communication. For example, in a one-dimensional processor topology, where each MPI rank communicates with its rank +/- 1, the default XYZT mapping is sufficient at least for partitions that are large enough to use torus wrap-around.

Torus wrap-around is enabled for partitions that are one midplane = 8x8x8 512 nodes, or multiples of one midplane. With torus wrap-around, the XYZT order keeps communication local, except for one extra hop at the torus edges. For smaller partitions, such as a 64-node partition with a 4x4x4 mesh topology, it is better to create a map file that assigns ranks that go down the X-axis in the +x direction and then for the next Y-value, fold the line to return in the -x direction, making a snake-like pattern that winds back and forth, filling out the 4x4x4 mesh. It is worthwhile to note that, for a random placement of MPI ranks onto a 3D torus network, the average number of hops is one-quarter of the torus length, in each of the three dimensions. Thus mapping is generally more important for large or elongated torus configurations.

Two-dimensional logical processes topologies are more challenging. In some cases, it is possible to choose the dimensions of the logical 2D process mesh so that one can fold the logical 2D mesh to fit perfectly in the 3D Blue Gene torus network. For example, if you want to use one midplane (8x8x8 nodes) in virtual-node mode, a total of 2048 CPUs are available. A 2D process mesh is 32x64 for this problem. The 32 dimension can be lined up along one edge of the torus, say the X-axis, using TX order to fill up processors (0,1,2,3) on each of the eight nodes going down the X-axis, resulting in 32 MPI ranks going down the X-axis.

The simplest good mapping, in this case is to specify `-mapfile TXYZ`. This keeps nearest-neighbor communication local on the torus, except for one extra hop at the torus edges. You can do slightly better by taking the 32x64 logical 2D process mesh, aligning one edge along the X-axis with TX order and then folding the 64 dimension back and forth to fill the 3D torus in a seamless manner. It is straightforward to construct small scripts or programs to generate the appropriate map file. Not all 2D process topologies can be neatly folded onto the 3D torus.

For 3D logical process topologies, it is best to choose a decomposition or mapping that fits perfectly onto the 3D torus if possible. For example, if your application uses SMP Node Mode on one Blue Gene rack (8x8x16 torus), then it is best to choose a 3D decomposition with 8 ranks in the X-direction, 8 ranks in the Y-direction, and 16 ranks in the Z-direction. If the application requires a different decomposition, for example 16x8x8, you might be able to use mapping to maintain locality for nearest-neighbor communication. In this case, ZXY order works.

Quantum chromodynamics (QCD) applications often use a 4D process topology. This can fit perfectly onto Blue Gene/P using Virtual Node Mode. For example, with one full rack, there are 4096 CPUs in Virtual Node Mode, with a natural layout of 8x8x16x4 (X, Y, Z, T order). By choosing a decomposition of 8x8x16x4, communication remains entirely local for nearest neighbors in the logical 4D process mesh. In contrast, a more balanced decomposition of 8x8x8x8 results in a significant amount of link sharing, and thus degraded bandwidth in one of the dimensions.

In summary, it is often possible to choose a mapping that keeps communication local on the Blue Gene torus network. This is recommended for cases where a natural mapping can be identified based on the parallel decomposition strategy used by the application. The mapping can be specified using the `-mapfile` argument for the `mpirun` command.



F

Statement of completion

IBM considers installation to be complete when the following activities have taken place:

- ▶ The Blue Gene/P rack or racks have been physically placed in position.
- ▶ The cabling is complete, including power, Ethernet, and torus cables.
- ▶ The Blue Gene/P racks can be powered on.
- ▶ All hardware is displayed in the Navigator and is available.

References

1. TOP500 Supercomputer Sites
<http://www.top500.org/>
2. The MPI Forum. The MPI message-passing interface standard. May 1995
<http://www.mcs.anl.gov/mpi/standard.html>
3. OpenMP application program interface (API):
<http://www.openmp.org>
4. IBM XL family of compilers
 - XL C/C++
<http://www-306.ibm.com/software/awdtools/xlcpp/>
 - XL Fortran
<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/features/bg/>
5. GCC, the GNU Compiler Collection
<http://gcc.gnu.org/>
6. *IBM System Blue Gene Solution: Configuring and Maintaining Your Environment*, SG24-7352
7. *GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters*, REDP-4168
8. Engineering and Scientific Subroutine Library (ESSL)
<http://www-03.ibm.com/systems/p/software/essl.html>
9. See note 2 above.
10. See note 3 above.
11. See note 4 above.
12. See note 5 above.
13. See note 6 above.
14. See note 7 above.
15. See note 8 above.
16. Gropp, W. and Lusk, E. "Dynamic Process Management in an MPI Setting." *7th IEEE Symposium on Parallel and Distributed Processing*. p. 530, 1995.
<http://www.cs.uiuc.edu/homes/wgropp/bib/papers/1995/sanantonio.pdf>
17. See note 2 above.
18. See note 3 above.
19. See note 5 above.
20. See note 8 above.
21. Ganier, CJ. "What is Direct Memory Access (DMA)?"
<http://cnx.org/content/m11867/latest/>
22. See note 2 above.

23. See note 3 on page 287.
24. Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, 2004. ISBN 0-072-82256-2.
25. Snir, Marc; Otto, Steve; Huss-Lederman, Steven; Walker, David; Dongarra, Jack. *MPI: The Complete Reference, 2nd Edition, Volume 1*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0-262-69215-5.
26. Gropp, William; Huss-Lederman, Steven; Lumsdaine, Andrew; Lusk, Ewing; Nitzberg, Bill; Saphir, William; Snir, Marc. *MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0-262-69216-3.
27. See note 3 on page 287.
28. See note 24 on page 288.
29. Ibid.
30. Ibid.
31. See note 3 on page 287.
32. Flynn's taxonomy in Wikipedia
http://en.wikipedia.org/wiki/Flynn%27s_Taxonomy
33. Rennie, Gabriele. "Keeping an Eye on the Prize." *Science and Technology Review*, July/August 2006.
http://www.llnl.gov/str/JulAug06/pdfs/07_06.3.pdf
34. Rennie, Gabriele. "Simulating Materials for Nanostructural Designs." *Science and Technology Review*, January/February 2006.
<http://www.llnl.gov/str/JanFeb06/Schwegler.html>
35. SC06 Supercomputing Web site, press release from 16 November 2006.
http://sc06.supercomputing.org/news/press_release.php?id=14
36. *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686
37. Sebastiani, D. and Rothlisberger, U. "Advances in Density-functional-based Modeling Techniques of the Car-Parinello Approach," chapter in *Quantum Medicinal Chemistry*, ed. by P. Carloni and F. Alber. Wiley-VCH, Germany, 2003. ISBN 9-783-52730-456-1.
38. Car, R. and Parrinello, Mi. "Unified Approach for Molecular Dynamics and Density-Functional Theory." *Physical Review Letter* 55, 2471 (1985).
http://prola.aps.org/abstract/PRL/v55/i22/p2471_1
39. See note 33 above.
40. Suits, F., et al. *Overview of Molecular Dynamics Techniques and Early Scientific Results from the Blue Gene Project*. IBM Research & Development, 2005. 49, 475 (2005).
<http://www.research.ibm.com/journal/rd/492/suits.pdf>
41. Ibid.
42. Case, D. A., et al. "The Amber biomolecular simulation programs." *Journal of Computational Chemistry*. 26, 1668 (2005).
43. Fitch, B. G., et al. "Blue Matter, an application framework for molecular simulation on Blue Gene." *Journal of Parallel and Distributed Computing*. 63, 759 (2003).
<http://portal.acm.org/citation.cfm?id=952903.952912&d1=GUIDE&d1=ACM>
44. Plimpton, S. "Fast parallel algorithms for short-range molecular dynamics." *Journal of Computational Physics*. 117, 1 (1995).

45. Phillips, J., et al. "Scalable molecular dynamics with NAMD." *Journal of Computational Chemistry*. 26, 1781 (2005).
46. See note 42 on page 288.
47. See note 43 on page 288.
48. Ibid.
49. Ibid.
50. Ibid.
51. Ibid.
52. See note 44 on page 288.
53. LAMMPS Molecular Dynamics Simulator:
<http://lammps.sandia.gov/>
54. See note 45 on page 289.
55. Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. "CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations." *Journal of Computational Chemistry*. 4, 187 (1983).
56. Brünger, A. I. "X-PLOR, Version 3.1, A System for X-ray Crystallography and NMR." 1992: The Howard Hughes Medical Institute and Department of Molecular Biophysics and Biochemistry, Yale University. 405.
57. Kumar, S., et al. "Achieving Strong Scaling with NAMD on Blue Gene/L." *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2006.
58. Waszkowycz, B., et al. "Large-scale Virtual Screening for Discovering Leads in the Postgenomic Era." *IBM Systems Journal*. 40, 360 (2001).
59. Patrick, G. L. *An Introduction to Medicinal Chemistry, 3rd Edition*. Oxford University Press, Oxford, UK, 2005. ISBN 0-199-27500-9.
60. Kontoyianni, M., et al. "Evaluation of Docking Performance: Comparative Data on Docking Algorithms." *Journal of Medical Chemistry*. 47, 558 (2004).
61. Kuntz, D., et al. "A Geometric Approach to Macromolecule-ligand Interactions." *Journal of Molecular Biology*. 161, 269 (1982); Morris, G. M., et al. "Automated Docking Using a Lamarckian Genetic Algorithm and Empirical Binding Free Energy Function." *Journal of Computational Chemistry*. 19, 1639 (1998); Jones, G., et al. "Development and Validation of a Genetic Algorithm to Flexible Docking." *Journal of Molecular Biology*. 267, 904 (1997); Rarey, M., et al. "A Fast Flexible Docking Method Using an Incremental Construction Algorithm." *Journal of Molecular Biology*. 261, 470 (1996), Schrödinger, Portland, OR 972001; Pang, Y. P., et al. "EUDOC: A Computer Program for Identification of Drug Interaction Sites in Macromolecules and Drug Leads from Chemical Databases." *Journal of Computational Chemistry*. 22, 1750 (2001).
62. (a) <http://dock.compbio.ucsf.edu> (b) Moustakas, D. T., et al. "Development and Validation of a Modular, Extensible Docking Program: DOCK5." *Journal of Computational Aided Molecular Design*. 20, 601 (2006).
63. Ibid.
64. Ibid.
65. Ibid.
66. Ibid.
67. Peters, A., et al., "High Throughput Computing Validation for Drug Discovery using the DOCK Program on a Massively Parallel System." *1st Annual MSCBB - Location: Northwestern University - Evanston, IL, September, 2007.*

68. Irwin, J. J. and Shoichet, B. K. "ZINC - A Free Database of Commercially Available Compounds for Virtual Screening." *Journal of Chemical Information and Modeling*. 45, 177 (2005).
69. Ibid.
70. Pople, J. A. *Approximate Molecular Orbital Theory (Advanced Chemistry)*. McGraw-Hill, NY. June 1970. ISBN 0-070-50512-8.
71. See note 38 on page 288.
72. (a) CPMD V3.9, Copyright IBM Corp. 1990-2003, Copyright MPI fur Festkorperforschung, Stuttgart, 1997-2001. (b) See also <http://www.cpmc.org>
73. Marx, D. and Hutter, J. *Ab-initio molecular dynamics: Theory and implementation*, in: *Modern Methods and Algorithms of Quantum Chemistry*. J. Grotendorst (Ed.), NIC Series, 1, FZ Julich, Germany, 2000; see also <http://www.fz-juelich.de/nic-series/Volume> and references therein.
74. Vanderbilt, D. "Soft self-consistent pseudopotentials in a generalized eigenvalue formalism." *Physical Review B*. 1990, 41, 7892 (1990).
http://prola.aps.org/abstract/PRB/v41/i11/p7892_1
75. See note 72 above.
76. Eddy, S. R., *HMMER User's Guide. Biological Sequence Analysis Using Profile Hidden Markov Models*, Version 2.3.2, October 1998.
77. Ibid.
78. Ibid.
79. Jiang, K., et al. "An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence Search on a Massively Parallel System." *IEEE Transactions On Parallel and Distributed Systems*. 19, 1 (2008).
80. Bateman, A., et al. "The Pfam Protein Families Database." *Nucleic Acids Research*. 30, 276 (2002).
81. Ibid.
82. Darling, A., et al. "The Design, Implementation, and Evaluation of mpiBLAST." Proceedings of 4th International Conference on Linux Clusters (in conjunction with ClusterWorld Conference & Expo), 2003.
83. Thorsen, O., et al. "Parallel genomic sequence-search on a massively parallel system." *Conference On Computing Frontiers: Proceedings of the 4th international conference on Computing frontiers*. ACM, 2007, pp. 59-68.
84. Heyman, J. *Porting Open Source Software (OSS) to Blue Gene/P*, white paper WP101152.
<http://www-03.ibm.com/support/techdocs/atmsastr.nsf/WebIndex/WP101152>
85. See note 24 on page 288.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 294. Note that some of the documents referenced here might be available in softcopy only.

- ▶ *Blue Gene Safety Considerations*, REDP-4257
- ▶ *Blue Gene/L: Hardware Overview and Planning*, SG24-6796
- ▶ *Blue Gene/L: Performance Analysis Tools*, SG24-7278
- ▶ *Evolution of the IBM System Blue Gene Solution*, REDP-4247
- ▶ *GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters*, REDP-4168
- ▶ *IBM System Blue Gene Solution: Application Development*, SG24-7179
- ▶ *IBM System Blue Gene Solution: Configuring and Maintaining Your Environment*, SG24-7352
- ▶ *IBM System Blue Gene Solution: Hardware Installation and Serviceability*, SG24-6743
- ▶ *IBM System Blue Gene Solution Problem Determination Guide*, SG24-7211
- ▶ *IBM System Blue Gene Solution: System Administration*, SG24-7178
- ▶ *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686

Other publications

These publications are also relevant as further information sources:

- ▶ Bateman, A., et al. “The Pfam Protein Families Database.” *Nucleic Acids Research*. 30, 276 (2002).
- ▶ Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. “CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations.” *Journal of Computational Chemistry*. 4, 187 (1983).
- ▶ Brünger, A. I. “X-PLOR, Version 3.1, A System for X-ray Crystallography and NMR.” 1992: The Howard Hughes Medical Institute and Department of Molecular Biophysics and Biochemistry, Yale University. 405.
- ▶ Car, R. and Parrinello, Mi. “Unified Approach for Molecular Dynamics and Density-Functional Theory.” *Physical Review Letter* 55, 2471 (1985).
- ▶ Case, D. A., et al. “The Amber biomolecular simulation programs.” *Journal of Computational Chemistry*. 26, 1668 (2005).
- ▶ Darling, A., et al. “The Design, Implementation, and Evaluation of mpiBLAST.” Proceedings of 4th International Conference on Linux Clusters (in conjunction with ClusterWorld Conference & Expo), 2003.

- ▶ Eddy, S. R., *HMMER User's Guide. Biological Sequence Analysis Using Profile Hidden Markov Models*, Version 2.3.2, October 1998.
- ▶ Fitch, B. G., et al. "Blue Matter, an application framework for molecular simulation on Blue Gene." *Journal of Parallel and Distributed Computing*. 63, 759 (2003).
- ▶ Gropp, W. and Lusk, E. "Dynamic Process Management in an MPI Setting." *7th IEEE Symposium on Parallel and Distributed Processing*. p. 530, 1995.
- ▶ Gropp, William; Huss-Lederman, Steven; Lumsdaine, Andrew; Lusk, Ewing; Nitzberg, Bill; Saphir, William; Snir, Marc. *MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0-262-69216-3.
- ▶ Heyman, J. *Porting Open Source Software (OSS) to Blue Gene/P*, white paper WP101152.
<http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101152>
- ▶ Irwin, J. J. and Shoichet, B. K. "ZINC - A Free Database of Commercially Available Compounds for Virtual Screening." *Journal of Chemical Information and Modeling*. 45, 177 (2005).
- ▶ Jiang, K., et al. "An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence Search on a Massively Parallel System." *IEEE Transactions On Parallel and Distributed Systems*. 19, 1 (2008).
- ▶ Jones, G., et al. "Development and Validation of a Genetic Algorithm to Flexible Docking." *Journal of Molecular Biology*. 267, 904 (1997).
- ▶ Kontoyianni, M., et al. "Evaluation of Docking Performance: Comparative Data on Docking Algorithms." *Journal of Medical Chemistry*. 47, 558 (2004).
- ▶ Kumar, S., et al. "Achieving Strong Scaling with NAMD on Blue Gene/L." *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2006.
- ▶ Kuntz, D., et al. "A Geometric Approach to Macromolecule-ligand Interactions." *Journal of Molecular Biology*. 161, 269 (1982).
- ▶ Marx, D. and Hutter, J. *Ab-initio molecular dynamics: Theory and implementation*, in: *Modern Methods and Algorithms of Quantum Chemistry*. J. Grotendorst (Ed.), NIC Series, 1, FZ Julich, Germany, 2000
- ▶ Mendell, Mark, "Exploiting the Dual Floating Point Units in Blue Gene/L":
<http://www-1.ibm.com/support/docview.wss?uid=swg27007511>
- ▶ Morris, G. M., et al. "Automated Docking Using a Lamarckian Genetic Algorithm and Empirical Binding Free Energy Function." *Journal of Computational Chemistry*. 19, 1639 (1998).
- ▶ Pang, Y. P., et al. "EUDOC: A Computer Program for Identification of Drug Interaction Sites in Macromolecules and Drug Leads from Chemical Databases." *Journal of Computational Chemistry*. 22, 1750 (2001).
- ▶ Patrick, G. L. *An Introduction to Medicinal Chemistry, 3rd Edition*. Oxford University Press, Oxford, UK, 2005. ISBN 0-199-27500-9.
- ▶ Peters, A., et al., "High Throughput Computing Validation for Drug Discovery using the DOCK Program on a Massively Parallel System." *1st Annual MSCBB - Location: Northwestern University - Evanston, IL, September, 2007*.
- ▶ Phillips, J., et al. "Scalable molecular dynamics with NAMD." *Journal of Computational Chemistry*. 26, 1781 (2005).
- ▶ Plimpton, S. "Fast parallel algorithms for short-range molecular dynamics." *Journal of Computational Physics*. 117, 1 (1995).

- ▶ Pople, J. A. *Approximate Molecular Orbital Theory (Advanced Chemistry)*. McGraw-Hill, NY. June 1970. ISBN 0-070-50512-8.
- ▶ Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, 2004. ISBN 0-072-82256-2.
- ▶ Rarey, M., et al. "A Fast Flexible Docking Method Using an Incremental Construction Algorithm." *Journal of Molecular Biology*. 261, 470 (1996), Scrödinger, Portland, OR 972001.
- ▶ Sebastiani, D. and Rothlisberger, U. "Advances in Density-functional-based Modeling Techniques of the Car-Parinello Approach," chapter in *Quantum Medicinal Chemistry*, edited by P. Carloni and F. Alber. Wiley-VCH, Germany, 2003. ISBN 9-783-52730-456-1.
- ▶ Snir, Marc; Otto, Steve; Huss-Lederman, Steven; Walker, David; Dongarra, Jack. *MPI: The Complete Reference, 2nd Edition, Volume 1*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0-262-69215-5.
- ▶ Suits, F., et al. *Overview of Molecular Dynamics Techniques and Early Scientific Results from the Blue Gene Project*. IBM Research & Development, 2005. 49, 475 (2005).
- ▶ Thorsen, O., et al. "Parallel genomic sequence-search on a massively parallel system." *Conference On Computing Frontiers: Proceedings of the 4th international conference on Computing frontiers*. ACM, 2007, pp. 59-68.
- ▶ Vanderbilt, D. "Soft self-consistent pseudopotentials in a generalized eigenvalue formalism." *Physical Review B*. 1990, 41, 7892 (1990).
- ▶ Waszkowycz, B., et al. "Large-scale Virtual Screening for Discovering Leads in the Postgenomic Era." *IBM Systems Journal*. 40, 360 (2001).

Online resources

These Web sites are also relevant as further information sources:

- ▶ Compiler Related topics:
 - XL C/C++:
<http://www-306.ibm.com/software/awdtools/xlcpp/>
 - XL C/C++ library
<http://www.ibm.com/software/awdtools/xlcpp/library/>
 - XL Fortran Advanced Edition for Blue Gene
<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/features/bg/>
 - XL Fortran library
<http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/>
- ▶ Debugger-related topics:
 - The GNU Project Debugger
<http://www.gnu.org/software/gdb/gdb.html>
 - GDB documentation:
<http://www.gnu.org/software/gdb/documentation/>
- ▶ Engineering and Scientific Subroutine Library (ESSL)
<http://www-03.ibm.com/systems/p/software/essl.html>

- ▶ GCC, the GNU Compiler Collection
<http://gcc.gnu.org/>
- ▶ Intel MPI Benchmarks is formerly known as "Pallas MPI Benchmarks" - PMB-MPI1?
<http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm>
- ▶ Mathematical Acceleration Subsystem (MASS)
<http://www-306.ibm.com/software/awdtools/mass/index.html>
- ▶ The MPI Forum
<http://www.mpi-forum.org/>
- ▶ MPI Performance Topics
http://www.llnl.gov/computing/tutorials/mpi_performance/
- ▶ OpenMP application program interface (API):
<http://www.openmp.org>
- ▶ Danier, CJ, "What is Direct Memory Access (DMA)?"
<http://cnx.org/content/m11867/latest/>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Numerics

10 Gb Ethernet network 11
32-bit static link files 273
3D torus network 10

A

Ab Initio method 247
adaptive routing 68
address space 22
allocate block 130
AMBER 249
ANSI-C 59
Argonne National Labs 18
arithmetic functions 114
asynchronous API 163
asynchronous file I/O 22

B

base partition 191
 definitions 192
BG_CHKPT_ENABLED 158
BG_SHAREDMEMPOOLSIZ 40
bgpmaster daemon 26
binary functions 116
binutils 92
block 130
blrts_xlc 94
blrts_xlc++ 94
blrts_xlf 94
Blue Gene specifications 12
Blue Gene XL compilers 91
Blue Gene/L PowerPC 440d processor 91
Blue Gene/P
 environment 6
 hardware 3
 memory considerations 18
 MPI 18
 programs 11
 software 16
 system architecture 4
 threading support 18
Blue Matter 250
Bridge API 160
 asynchronous API 163
 deprecated APIs 163
 examples 192
 first and next calls 161
 invalid pointers 161
 library files 198, 238
 memory allocation and deallocation 161
 requirements 160, 198
 return codes 163, 207
 sample makefile 198

bss 19
built-in floating-point functions 106

C

cache
 L1 44–45, 72
 L2 44, 46
 L3 44, 46
Car-Parrinello Molecular Dynamics (CPMD) 248
Cartesian communicator functions 74
checkpoint and restart
 API 155
 BG_CHKPT_ENABLED 158
 BGLAtCheckpoint 156
 BGLAtContinue 156
 BGLAtRestart 156
 BGLCheckpoint 155
 BGLCheckpointExcludeRegion 156
 BGLCheckpointInit 155
 BGLCheckpointRestart 156
 directory and file naming conventions 157
 I/O considerations 153
 restarting application 157
 signal considerations 153
 technical overview 152
checkpoint library 152
checkpoint write complete flag 157
chip 4
CIOD (control and I/O daemon) 31, 33
 threading 35
Classical Molecular Mechanics/Molecular Dynamics (MM/MD) 247
collective 11
collective MPI 81, 260
collective network 68
Common Node Services 32
Communication Coprocessor Mode 17, 37–38, 47
compilers
 GNU 23
 IBM XL 24
complex type manipulation functions 109
compute card 4
Compute Node 5–6, 9
 card 4
Compute Node Kernel 6, 17, 22, 30, 52
 socket services 25
control and I/O daemon (CIOD) 31, 33, 35
control network 6, 11
control system 25
 bgpmaster 26
 Bridge API 25
 ciodb 26
 MMCS 25
 mpirun 25

Control System APIs

- jm_attach_job 172
- jm_begin_job 172
- jm_cancel_job 172
- jm_debug_job 172
- jm_load_job 173
- jm_signal_job 174
- jm_start_job 174
- job state flags 173
- message types 190
- messaging API 190
- partition state flags 168
- pm_create_partition 167
- pm_destroy_partition 167
- requirements 160, 238
- rm_add_job 171
- rm_add_part_user 166, 200, 208
- rm_add_partition 166, 200, 208
- rm_assign_job 166
- rm_free_BGL 189
- rm_free_BP 189
- rm_free_job 189
- rm_free_job_list 189
- rm_free_nodecard 189
- rm_free_nodecard_list 190
- rm_free_partition 190
- rm_free_partition_list 190
- rm_free_switch 190
- rm_get_BGL 164
- rm_get_data 162, 164
- rm_get_job 173
- rm_get_jobs 173
- rm_get_partitions 167–168, 200, 208–209
- rm_get_partitions_info 168, 200, 209
- rm_get_serial 165
- rm_new_BP 189
- rm_new_job 189
- rm_new_nodecard 189
- rm_new_partition 189
- rm_new_switch 189
- rm_query_job 174
- rm_release_partition 169
- rm_remove_job 174
- rm_remove_part_user 170, 200, 208
- rm_remove_partition 170
- rm_set_data 162, 165
- rm_set_part_owner 170
- rm_set_serial 165

copy-primary operations 107

copy-secondary operations 108

core files 146

Core Processor tool 138

cores, computation of 5

CPMD (Car-Parrinello Molecular Dynamics) 248

critical pragma 87

cross operations 107

cross-copy operations 108

D

data 19

DB_PROPERTY 198

DCMF_EAGER 68

DDR (double data RAM) 46

debug client 133

debug server 133

debugging applications 132

deterministic routing 68

DOCK6 253

double data RAM (DDR) 46

Double Hummer FPU 94

double precision square matrix multiply example 125

DUAL mode (2X2) 39

Dual Node Mode 39, 49

dynamic linking 23

Dynamic Partition Allocator API 237

- requirements 238
- return codes 240
- sample program 241

E

eager protocol 68, 71

electronic correlation 254

electronic structure method 247

Engineering and Scientific Subroutine Library (ESSL) 96

entities 25

ESSL (Engineering and Scientific Subroutine Library) 96

Ewald sums 248

extended basic blocks 101

F

fault recovery 152

- see checkpoint and restart

file I/O 22

flood of messages 70

freepartition 218

Front End Node 6, 13

functional Ethernet (10 Gigabit) 11

functional network 6

G

GDB (GNU Project debugger) 133

gdbserver 133

General Parallel File System (GPFS) 13

gid 52

global collective network 11

global interrupt network 11

GNU

- 3.2 C 23
- C++ 23
- Fortran77 23
- GDB 133

GNU Project debugger (GDB) 133

GPFS (General Parallel File System) 13

H

heap 19

HMMER 254

host system 13

host system software 14

I

I/O (input/output) 22
I/O Node 5–6, 10, 24, 32
 daemons 25
 file system services 24
 Kernel boot 24
 software 24
IBM XL compiler 24, 91
inline functions 101
inlining 101
input/output (I/O) 22
 file 22
 sockets calls 22
installation 285
Intel MPI Benchmarks 79

L

LAMMPS 251
libbgrealtime.so 198
ligand atoms 253
load and store functions 111
LoadLeveler 131
 cluster 131

M

mapping 281
MASS (Mathematical Acceleration Subsystem) 96
Mathematical Acceleration Subsystem (MASS) 96
mcServer daemon 26
memory 18
 address space 22
 addressing 19
 considerations 9
 distributed 44, 66
 leaks 21
 management 22
 model 22
 virtual 44
message layer 39
Message Passing Interface (MPI) 18, 66
 bandwidth 79
 collective 81, 260
 eager protocol 68
 latency 79
 point-to-point 80
 rendezvous protocol 68
 short protocol 68
 too much memory 69
microprocessor 8
midplane 7
Midplane Management Control System (MMCS) 25–26
Midplane Management Control System APIs 237
MM/MD (Classical Molecular Mechanics/Molecular Dynamics) 247
mmap 40
MMCS (Midplane Management Control System) 25–26

MMCS console 130
MMCS daemon 26
MPI (Message Passing Interface) 18, 66
 bandwidth 79
 collective 81, 260
 communications 74
 eager protocol 68
 latency 79
 point-to-point 80
 rendezvous protocol 68
 short protocol 68
 too much memory 69
MPI_Barrier 73
MPI_COMM_WORLD 75
MPI_Irecv 69
MPI_Isend 69
MPI_SUCCESS 74
MPI_Test 70
MPI_Wait 71
MPI-2 18
mpiBLAST-PIO 256
MPICH2 18, 67
mpiexec 219
mpirun 25, 131, 217
 challenge protocol 220
 -env 229
 freepartition 218
 get_parameters() 234
 mpirun_done() 235
 multiple program, multiple data (MPMD) 219
 -psets_per_bp 228
 SIGINT 233
mpirun.cfg 219
mpix.h file 74
MPMD (multiple program, multiple data) 66
multiple program, multiple data (MPMD) 66, 219
multiply-add functions 117

N

NAMD 252
natural alignment 97
network 10
 10 Gb Ethernet 11
 3D torus 10
 collective 11, 68
 control 11
 functional Ethernet 11
 global collective 11
 global interrupt 11
 torus 10, 68
Node
 Front End 13
 I/O 24
 Service 13
 Storage 13

O

OpenMP 83, 94, 134
other system calls 57

P

- parallel operations 106
- particle mesh Ewald (PME) method 248
- personality 31
- PingPong 257
- PME (particle mesh Ewald) method 248
- pmemd 249
- PMI_Cart_comm_create() 74
- PMI_Pset_diff_comm_create() 75
- PMI_Pset_same_comm_create() 75
- pointer aliasing 102
- pointers, uninitialized 22
- point-to-point MPI 80
- PowerPC 440d Double Hummer dual FPU 106
- PowerPC 440d processor 91
- PowerPC 450 microprocessor 8
- processor set (pset) 74
- pset (processor set) 74

Q

- q64 94
- qaltivec 94
- qarch 93
- qcache 93
- qfltrap 94
- qinline 102
- qipa 102
- QM/MM (Quantum Mechanical/Molecular Mechanical) 248
- qmshrojb 94
- qnoautoconfig 93
- qpics 94
- qtune 93
- Quantum Mechanical/Molecular Mechanical (QM/MM) 248

R

- rack 4
- raw state 201
- real-time application code 209
- Real-time Notification API 197
 - blocking or non-blocking 199
 - libbgrealttime.so 198
 - rt_api.h 198
 - RT_CALLBACK_CONTINUE 201
 - RT_CALLBACK_QUIT 201
 - RT_CALLBACK_VERSION_0 201
 - rt_callbacks_t 201
 - RT_CONNECTION_ERROR 199, 209
 - RT_DB_PROPERTY_ERROR 208
 - rt_get_msgs 199
 - RT_HANDLE_CLOSE 209
 - rt_handle_t 199
 - rt_init 198
 - RT_INVALID_INPUT_ERROR 208–209
 - rt_set_blocking 199
 - rt_set_filter 200
 - rt_set_nonblocking 199
 - RT_STATUS_OK 201

- RT_WOULD_BLOCK 209
- real-time server 199
- Redbooks Web site 294
 - Contact us xiii
- reduction clause 87
- rendezvous protocol 68, 71
- rm_modify_partition 169
- running applications 130

S

- segfaults 47
- service actions 26
- Service Node 6, 13
- shared memory 40
- shm_open 40
- SIMD (single-instruction, multiple-data) 7, 46, 96
- SIMD computation 106
- single program, multiple data (SPMD) 66
- single-instruction, multiple-data (SIMD) 7, 46, 96
- size command 19
- small partition
 - allocation 191, 194, 209
 - query 195
- SMP mode (1X4) 38
- SMP Node Mode 47
- socket support 57
- sockets calls 22
- specifications, deprecated 163
- SPI (System Programming Interface) 57
- SPMD (single program, multiple data) 66
- stack 19
- standard input 22
- stdin 22
- Storage Node 13
- structure alignment 100
- Symmetrical Multi-Processing (SMP) Node Mode 38, 47
- system 4
- system architecture 4
- system calls 51
 - other 57
 - return codes 52
- System Programming Interface (SPI) 57

T

- threading support 18
- TLB (translation look-aside buffer) 47
- torus communications 74
- torus network 68
- torus wrap-around 283
- translation look-aside buffer (TLB) 47
- WXYZ order 281

U

- uid 52
- unary functions 114
- uninitialized pointers 22
- unsupported system calls 57

V

vectorizable basic blocks 101
virtual FIFO 39
virtual memory 44
Virtual Node Mode 17, 37–38, 48
virtual paging 21
VN mode (4X1) 38

X

XL

#pragma disjoint directive 103
__alignx function 104
__attribute__(always_inline) extension 102
__cimag 110
__cimagf 110
__cimagl 110
__cmplx 109
__cmplxf 109
__cmplxl 109
__creal 110
__crealf 110
__creall 110
__fpabs 115
__fpadd 116
__fpctiw 114
__fpctiwz 114
__fpmadd 117
__fpmsub 118
__fpmul 116
__fpnabs 116
__fpneg 115
__fpmadd 118
__fpmnsb 118
__fpre 115
__fprsp 114
__fprsqte 115
__fpse 123
__fpsub 116
__fxcpmadd 120
__fxcpmsub 121
__fxcpnmadd 120
__fxcpnmsub 121
__fxcpnpma 121
__fxcpnsma 122
__fxcsadd 120
__fxcsmsub 121
__fxcsnmadd 120
__fxcsnmsub 121
__fxcsnpma 121
__fxcsnsma 122
__fxcxma 122
__fxcxms 122
__fxcxnpma 123
__fxcxnsma 123
__fxmadd 119
__fxmr 113
__fxmsub 119
__fxmul 117
__fxnmadd 119
__fxnmsub 120
__fxpmul 117
__fxsmul 117
__lfpd 111
__lfps 111
__lfxd 112
__lfxs 111
__stfpd 112
__stfpiw 113
__stfps 112
__stfxd 113
__stfxs 112
ALIGNX 104
arithmetic functions 114
basic blocks 101
batching computations 103
binary functions 116
built-in floating-point functions 106
built-in functions usage 124
CIMAG 110
CIMAGF 110
CIMAGL 110
CMPLX 109
CMPLXF 109
compiling and linking 92
complex type manipulation functions 109
copy-primary operations 107
copy-secondary operations 108
CREAL 110
CREALF 110
CREALL 110
cross operations 107
cross-copy operations 108
data alignment 104
defining data objects 100
FPABS 115
FPADD 116
FPCTIW 114
FPCTIWZ 114
FPMADD 118
FPMSUB 118
FPMUL 117
FPNABS 116
FPNEG 115
FPNMADD 118
FPNMSUB 118
FPRE 115
FPRSP 114
FPRSQRTE 115
FPSEL 124
FPSUB 116
FXCPMADD 120
FXCPMSUB 121
FXCPNMADD 120
FXCPNMSUB 121
FXCPNPMA 121
FXCSMADD 120
FXCSMSUB 121
FXCSNMADD 120
FXCSNMSUB 121

- FXCSNPMA 121
- FXCXMA 122
- FXCXNMS 122
- FXCXNPMA 123
- FXCXNSMA 123
- FXMADD 119
- FXMR 113
- FXMSUB 119
- FXMUL 117
- FXNMADD 119
- FXNMSUB 120
- FXPMUL 117
- FXSMUL 117
- inline functions 101
- load and store functions 111
- LOADFP 111
- LOADFX 111–112
- move functions 113
- multiply-add functions 117
- optimization 96
- parallel operations 106
- pointer aliasing 102
- scripts 93
- select functions 123
- SIMD 106
- STOREFP 112–113
- STOREFX 112
- unary functions 114
- using complex types 101
- vectorizable basic blocks 101
- XL C/C++ Advanced Edition V8.0 for Blue Gene 91
- XL compiler options 93
- XL compilers 24, 91
 - developing applications 91
- XL Fortran Advanced Edition V10.1 for Blue Gene 91
- XL linker 95



Redbooks

IBM System Blue Gene Solution: Blue Gene/P Application Development

(0.5" spine)
0.475" x 0.873"
250 <-> 459 pages



IBM System Blue Gene Solution: Blue Gene/P Application Development



Understand the Blue Gene/P programming environment

Learn how to run and debug MPI programs

Learn about Bridge and Real-time APIs

This IBM Redbooks publication is one in a series of IBM books written specifically for the IBM System Blue Gene/P Solution. The Blue Gene/P system is the second generation of a massively parallel supercomputer from IBM in the IBM System Blue Gene Solution series. This book provides an overview of the application development environment for the Blue Gene/P system. It is intended to help programmers understand the requirements to develop applications on this high-performance massively parallel supercomputer.

In this book, we explain instances where the Blue Gene/P system is unique in its programming environment. We also attempt to look at the differences between the IBM System Blue Gene/L Solution and the Blue Gene/P Solution. This book does not delve into great depth about the technologies that are commonly used in the supercomputing industry, such as Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) nor tries to teach parallel programming. References are provided in those instances for you to find more information if desired.

Prior to reading this book, you must have a strong background in high-performance computing (HPC) programming. The high-level programming languages that are used throughout this book are C/C++ and Fortran95.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7287-00

ISBN 0738488674