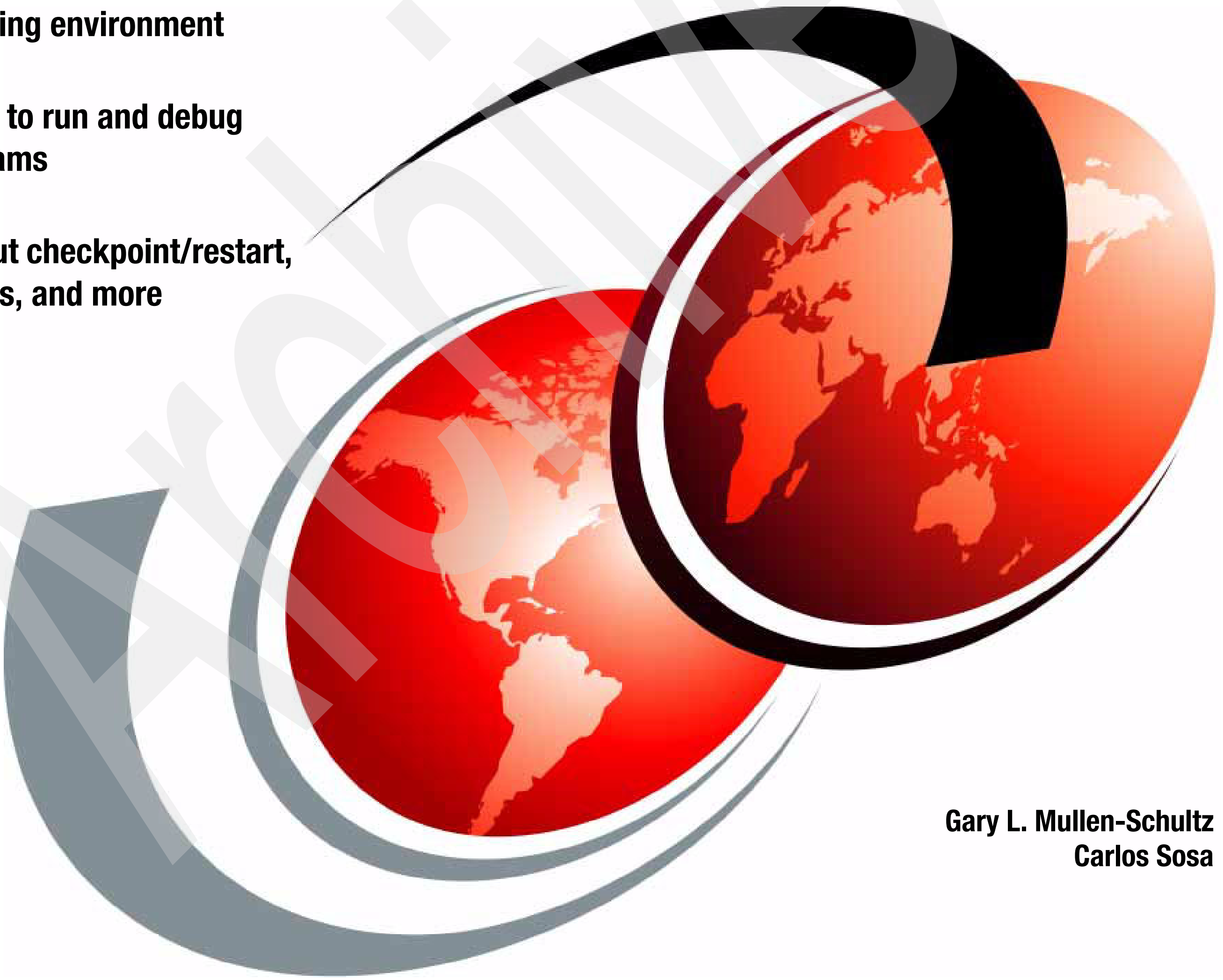
IBM

IBM System Blue Gene Solution: Application Development

Understand the Blue Gene/L programming environment

Learn how to run and debug MPI programs

Learn about checkpoint/restart, Bridge APIs, and more



Gary L. Mullen-Schultz
Carlos Sosa

Redbooks

ibm.com/redbooks

International Technical Support Organization

IBM System Blue Gene Solution: Application Development

June 2007

SG24-7179-04

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

Fifth Edition (June 2007)

This edition applies to Version 1, Release 3, Modification 4 of Blue Gene/L (product number 5733-BG1).

Contents

Notices

Trademarks

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

AIX 5L™	General Parallel File System™	POWER™
AIX®	GPFS™	Redbooks®
Blue Gene®	IBM®	Redbooks (logo) ®
Blue Gene/L™	LoadLeveler®	WebSphere®
DB2®	PowerPC®	

The following terms are trademarks of other companies:

Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. 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, and service names may be trademarks or service marks of others.

Preface

This IBM® Redbooks® publication is one in a series of IBM publications written specifically for the IBM System Blue Gene® Solution, Blue Gene/L™, which was developed by IBM in collaboration with Lawrence Livermore National Laboratory (LLNL). It provides an overview of the application development environment for Blue Gene/L.

This book explains the instances where Blue Gene/L is unique in its programming environment. It does not delve into great depth about the technologies that are commonly used in the supercomputing industry, such as Message Passing Interface (MPI) and Aggregate Remote Memory Copy Interface (ARMCI). References are provided in those instances so you can find more information if desired.

Prior to reading this book, you must have a strong background in MPI programming.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Rochester Center.

Gary L. Mullen-Schultz is a Consulting IT Specialist at the ITSO, Rochester Center. He leads the team responsible for producing Blue Gene/L documentation, and is the primary author of this book. Gary also focuses on Java™ and WebSphere®. He is a Sun Certified Java Programmer, Developer and Architect, and has three issued patents.

Carlos Sosa is a Senior Technical Staff Member in the IBM Systems and Technology Group. He is currently a member of the Blue Gene development team. His work involves enablement of the Chemistry and Life Sciences application on Blue Gene. His areas of interest include future POWER™ architectures and Blue Gene. 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.

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

Mark Mendell
Kara Moscoe
IBM Toronto, Canada

Ed Barnard
Todd Kelsey
Gary Lakner
James Milano
Jenifer Servais
Janet Willis
ITSO, Rochester Center

Paul Allen
Charles Archer
Peter Bergner
Lynn Boger
Mike Brutman
Jay Bryant

Tom Budnik
Kathy Cebell
Jeff Chauvin
Roxanne Clarke
Darwin Dumonceaux
David Hermsmeier
Mike Hjalmervik
Frank Ingram
Kerry Kaliszewski
Brant Knudson
Glenn Leckband
Matt Light
Dave Limpert
Chris Marroquin
Randall Massot
Curt Mathiowetz
Pat McCarthy
Mark Megerian
Sam Miller
Marv Misgen
Jose Moreira
Mike Mundy
Mike Nelson
Jeff Parker
Kurt Pinnow
Scott Plaetzer
Ruth Poole
Joan Rabe
Joseph Ratterman
Don Reed
Harold Rodakowski
Richard Shok
Brian Smith
Karl Solie
Wayne Wellik
Nancy Whetstone
Mike Woiwood
IBM Rochester

Tamar Domany
Edi Shmueli
IBM Israel

Gary Sutherland
Ed Varella
IBM Poughkeepsie

Gheorghe Almasi
Bob Walkup
IBM T.J. Watson Research Center

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbooks publication that deals with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners. and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll 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 IBM Redbooks publications to be as helpful as possible. Send us your comments about this or other IBM Redbooks publications in one of the following ways:

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

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@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

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition might also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-7179-04
for *IBM System Blue Gene Solution: Application Development*
as created or updated on July 16, 2008

June 2007, Fifth Edition

This revision reflects the addition, deletion, or modification of new and changed information described in the following sections.

New information

- mpirun APIs
- Dynamic partition allocator APIs
- Ability to allocate transient L1 data cache memory
- L1 data cache parity error recovery: sample code
- Addition of HTC documentation

Modified information

- Reorganization of system call (syscall) chapters

June 2006, Third Edition

This revision reflects the addition, deletion, or modification of new and changed information described in the following sections.

New information

- One-sided communications (Aggregate Remote Memory Copy Interface (ARMCI) and Global Arrays)
- Small partition application programming interfaces (APIs) added to the control system (Bridge) API set
- The addition of 64-bit support to the control system (Bridge) API set
- New system calls added (see Chapter 4, “Blue Gene/L-specific system calls” on page 31)

Deleted information

- Performance tooling and analysis information moved to *Blue Gene/L: Performance Analysis Tools*, SG24-7278

Modified information

- Changes to compilers and toolchain

1

Application development overview

This chapter provides an overview of the programming environment on Blue Gene/L. It discusses general items of interest to an application developer, while answering the following questions:

- What is the Message Passing Interface (MPI) implementation on Blue Gene/L?
- What are the major concerns an application developer should keep in mind when writing applications for Blue Gene/L?
- Where are the supporting files (compilers, include, and link files) located, and what versions are they?

1.1 Message Passing Interface on Blue Gene/L

The implementation of MPI on Blue Gene/L is the MPICH2 standard developed by Argonne National Labs. For more information about MPICH2, see:

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

Some functions of the MPI-2 standard that are not supported by Blue Gene/L include:

- Dynamic Process Management (creating new MPI processes) is not supported. In addition, the thread model supported is MPI_THREAD_SINGLE, which means that threads cannot be created.
- Remote Memory Operations (MPI-2 one-sided communications). However, ARMCI and Global Arrays are supported; see Chapter 8, “One-sided communications” on page 91, for more information.

1.2 Memory considerations

Give careful consideration to memory when writing applications for Blue Gene/L. It is important to remember that each compute node has 512 MB of memory. Of that memory, some is used by the Compute Node Kernel, and some is used by communications buffers. The following sections cover some points to remember when writing your MPI application.

You can use the Linux® **`size`** command to gain an idea of the memory size of the program. However, this command does not provide any information about the runtime memory usage of the application.

1.2.1 Memory leaks

Given that there is no virtual paging on Blue Gene/L, any memory leaks in your application can quickly consume available memory. When writing applications for Blue Gene/L, you must be especially diligent that you release all memory that you allocate.

1.2.2 Memory management

The Blue Gene/L 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.

The Blue Gene/L computer uses memory distributed across the nodes and uses networks to provide high bandwidth and low-latency communication. If the memory requirement per MPI task is greater than 256 MB in virtual node mode or greater than 512 MB in coprocessor mode, then the application will not run on Blue Gene/L. 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.

1.2.3 Uninitialized pointers

Blue Gene/L applications run in the same address space as the Compute Node Kernel and the communications buffers. You can create a pointer that doesn't 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.

The results can range from inserting malformed packets into the *torus*, causing spurious and unpredictable errors, to hanging the node. The main message here is to be extremely careful with pointers and strings in your application.

Torus network: In a torus network, each processor is directly connected to six other processors: two in the "X" dimension, two in the "Y" dimension, and two in the "Z" dimension. An easy way to picture a torus is to think of a 3-D "cube" of processors, where every processor on an edge has "wraparound" connections to link to other similar edge processors. To learn more about the torus network, see *Blue Gene/L: Hardware Overview and Planning*, SG24-6796.

1.2.4 Forcing MPI to allocate too much memory

Forcing MPI to allocate too much memory is relatively easy to do with innocent-looking code. For example, the snippets of legal MPI code shown in Example 1-1 and Example 1-2 run the risk of forcing the MPI support to allocate too much memory, resulting in failure, as it forces excessive buffering of messages.

Example 1-1 CPU1 MPI code that can cause excessive memory allocation

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

Example 1-2 CPU2 MPI code that can cause excessive memory allocation

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

You can accomplish the same goal and avoid memory allocation issues by recoding as shown in Example 1-3 and Example 1-4.

Example 1-3 CPU1 MPI code that can avoid excessive memory allocation

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

Example 1-4 CPU2 MPI code that can avoid excessive memory allocation

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

1.2.5 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 might accumulate over time and cause a memory overrun. This code displays the problem shown in Example 1-5.

Example 1-5 Potential memory overrun caused by not waiting for MPI_Test

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

Remember to wait, or test, for MPI_Test to return *true*.

1.2.6 Flooding of messages

The code shown in Example 1-6, while legal, floods the network with messages. It can cause CPU 0 to run out of memory. Even though it might work, it does not prove to be scalable.

Example 1-6 Flood of messages resulting in 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]);
```

1.2.7 Poor choice of programming mode

When you choose to run in Virtual Node Mode, your application only has half the memory (and cache) available. If your application is memory intensive, either in calculation or communication, you can easily run out of available memory.

Before you try Virtual Node Mode, make sure that your application runs well in Communication Coprocessor Mode.

1.2.8 Performance considerations

Consideration should be given for buffer alignment and the eager protocol threshold.

Buffer alignment

The MPI implementation on Blue Gene/L is sensitive to the alignment of the buffers that are being sent and received. Aligning buffers on 32-byte boundaries may improve performance.

For buffers that are dynamically allocated, for example via `malloc`, two techniques can be used:

- Instead of using `malloc()`, use `int posix_memalign(void **memptr, size_t alignment, size_t size)`, and specify 32 for the alignment parameter. This returns a 32-byte aligned pointer to the allocated memory. `free()` can be used to free the memory.
- Use `malloc()`, but request 32 bytes more storage than is required. Then, round the returned address up to a 32-byte boundary. See Example 1-7.

Example 1-7 malloc with 32 bytes for more storage

```
buffer_ptr_original = malloc(size + 32);
buffer_ptr          = (char*)( ( (unsigned)buffer_ptr + 32 ) & 0xFFFFFFE0 );
.
.
.
/* 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. See Example 1-8.

Example 1-8 Usage of __attribute__((aligned(32)))

```
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 or 32-byte aligned static (global) storage instead. If that is not possible, explicitly specify to use 16-byte alignment, `__attribute__((aligned(16)))` on the automatic storage declaration so that buffers are at least consistently aligned.

Eager or rendezvous protocol threshold

Different protocols are used to send messages, depending on the message length. The BGLMPI_EAGER environment variable can be used to change the message length threshold that controls which protocol is used. This threshold setting can affect performance. Try changing the threshold to see how it affects the performance of your application. Refer to Appendix E, “Environment variables” on page 149, for details about the BGLMPI_EAGER environment variable.

1.3 Torus and MPI communications

This section details some Blue Gene/L specific features related to MPI communications via the torus network.

1.3.1 Bandwidth considerations

Blue Gene/L has implemented bandwidth improvements for planar and diagonal torus communications. Enhancements have been made to allow more efficient use of available bandwidth when sending messages between nodes, which are connected in a planar or diagonal manner.

Let us say that the given application is sending a relatively large message from node 1 to node 4 in the two-dimensional diagram shown in Figure 1-1.

```
1 - 2
|   |
3 - 4
```

Figure 1-1 Simple node diagram

In older versions of the Blue Gene/L system software, all packets were sent only along one path (all either via node 2 or via node 3). Now the packets that make up the message can be sent along both routes, potentially doubling bandwidth. In a three-dimensional (diagonal) example, which commonly occurs in a torus network, packets can be sent along three different routes, potentially tripling bandwidth.

This routing is primarily important for point-to-point messages; large-scale collectives (such as all-to-all) will likely benefit little from this enhancement, because there will be higher contention for channels. With this enhancement, however, developers might want to consider arranging sends of larger point-to-point messages so that the nodes involved are physically connected in a planar or diagonal fashion.

It important to understand that this discussion is strictly tied to bandwidth. When considering any such optimizations, you must take latency into consideration. Any gains made in improved bandwidth can easily be wiped out, or even made worse, if the nodes involved are a great distance apart.

1.3.2 Cartesian communicator functions

Three new APIs make it easier to map nodes to specific hardware or processor set (pset) configurations. We go over each one individually.

- `PMI_Cart_comm_create()`

  This function creates a four-dimensional 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 coprocessor mode and cardinality 2 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 a number of 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.

- `PMI_Pset_same_comm_create()`

  This 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 1-2.

  The most common use for this 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.

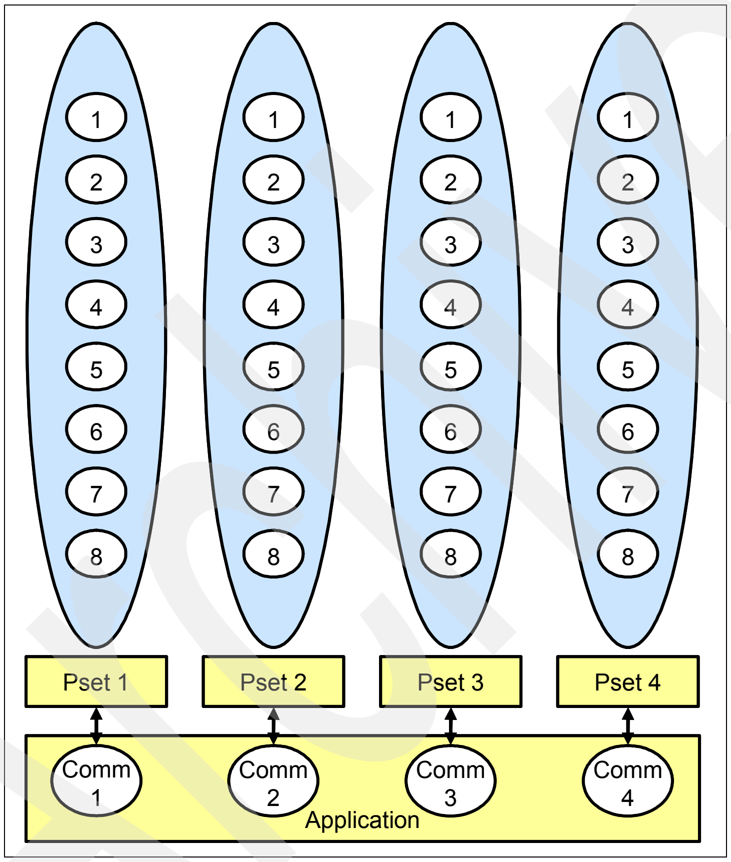


Figure 1-2 How PMI_Pset_same_comm_create() creates communicators

- `PMI_Pset_diff_comm_create()`

  This 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 1-3. The most common use for this 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 `PMI_Pset_same_comm_create()` and `PMI_Pset_diff_comm_create()`.

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

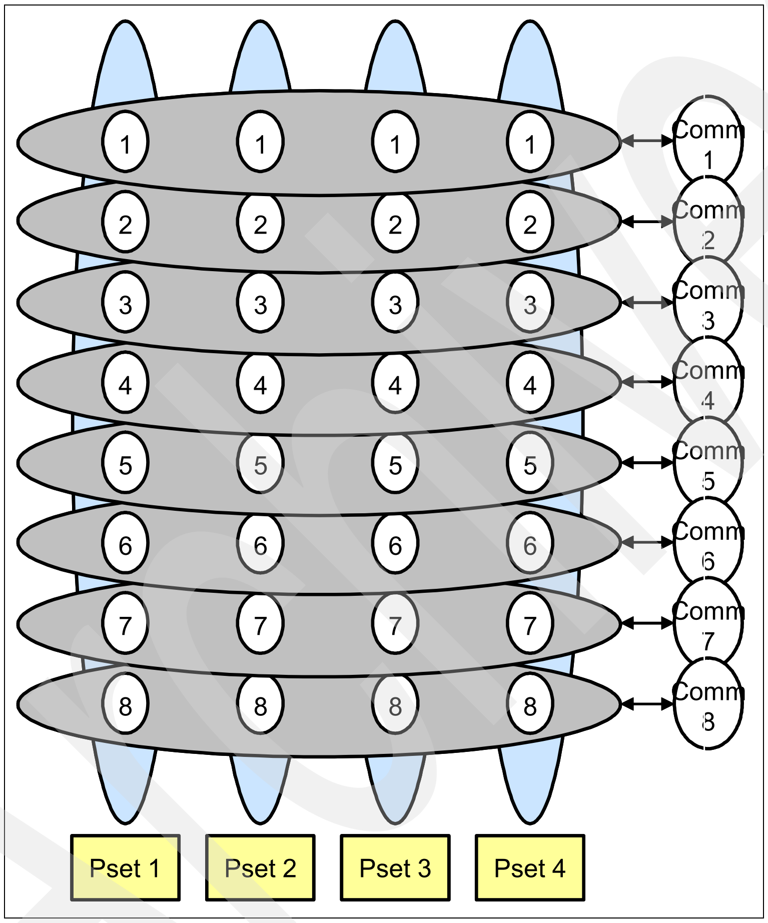


Figure 1-3 How PMI_Pset_diff_comm_create() creates communicators

1.4 Other considerations

It is important to understand that the operating system present on the Compute Node, the Compute Node Kernel, is not a full-fledged version of Linux. Because of this, there are a few areas in which you must use care when writing applications for Blue Gene/L. For a full list of supported system calls, see Chapter 3, "System calls supported by the Compute Node Kernel" on page 21.

1.4.1 Input/output

I/O is an area where you need to pay special attention in your application.

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.

You can find a full list of supported file I/O calls in 3.2.2, "List of supported system calls" on page 22.

Standard input (stdin)

As of Release 2 of software, standard input (stdin) is supported on Blue Gene/L. It is no longer necessary to pass input to your application using only file I/O.

Sockets calls

Sockets client-side system calls, such as `send()`, `recv()`, and `socket()`, are supported. However, server-side sockets calls, such as `accept()`, `bind()`, and `listen()`, are not supported.

For a full list of supported sockets calls, see 3.2.2, "List of supported system calls" on page 22.

1.4.2 Miscellaneous

You must also keep in mind the considerations presented in the following sections.

Linking

Dynamic linking is not supported on Blue Gene/L. You must statically link all code into your application.

Read-only memory

There is no true read-only memory in the Compute Node Kernel. This means that no segmentation violation will be signalled if an application attempts to write to a variable designated as a "const."

1.5 Include and link files

Include and link files are found in the main system path in the /bgl/BlueLight/ppcfloor/bglsys/ directory.

1.5.1 Include files

Include files for Blue Gene/L are found in the /bgl/BlueLight/ppcfloor/bglsys/include directory. Table 1-1 lists the include files.

Table 1-1 Include files on Blue Gene/L

File name	Description
allocator_api.h	Dynamic partition allocator APIs.
attach_bgl.h	The Blue Gene/L version of attach.h, which is described in the MPI debug specification.
bgl_errors.h	Master file containing rules for Blue Gene/L errors and error code bases.
bgl_perfctr.h	Function header file for the universal performance counters.
bgl_perfctr_events.h	Event list header file for the universal performance counters.
bglCheckpoint.h	Required when using the application programming interface (API) for the checkpoint and restore.
bglLinkCheckApi.h	Required when using the link verification function. This includes the APIs for both link checksums as well as link CRC verification.
bglLinkCheckApi.h	Link checksum support.
bgllockbox.h	Access memory-mapped program synchronization hardware (the “lockbox”).
bglmemmap.h	Memory constants for various APIs.
bglpersonality.h	The “personality” is static data given to every compute and I/O node at boot time by the control system. The data contains information specific to the node with respect to the block that is being booted.
mpi.h	Required for MPI applications.
mpicxx.h	Required for C++ MPI applications.
mpif.h	Required for Fortran MPI applications.
mpio.h	Required for MPI applications that perform MPI I/O.
mpiof.h	Required for MPI applications in Fortran that perform MPI I/O.
rm_api.h	Used for applications accessing the Bridge API.
rts.h	System-related functions for use by programs running under bgl-rts.
sayMessage.h	sayMessage is a general message facility to enable the generation of formatted messages.
sched_api.h	Required for applications accessing the scheduling system API.

1.5.2 Link (library) files

This section lists the different library files that are required to create Blue Gene/L applications.

32-bit static link files

The 32-bit static link files for Blue Gene/L are in the /bgl/BlueLight/ppcfloor/bglsys/lib directory.

Table 1-2 32-bit static link files on Blue Gene/L

File name	Description
bglbootload.a	
bglsp440supt.a	
lib_ido_api.a	
libbgl_perfctr.rts.a	Universal performance counter library.
libbglbridge.a	The API set provided for an external scheduler to interact with Midplane Management Control System (MMCS) low-level components. These APIs can be used to interact with MMCS and create, boot, and destroy partitions. The APIs also provide functions for gathering information about the topology of the machine, such as base partitions, wire, and switches.
libbgldb.a	
libbglmachine.a	
libbglsim_counters.rts.a	
libbglsp.a	
libbglupc.rts.a	
libchkpt.rts.a	Contains the user-initiated checkpoint/restart bindings for parallel applications written in C++, C or Fortran. Provides APIs to save program state in stable storage at a synchronizing point (typically after a barrier, assumes no messages are in transit), and restart from this point at a later stage.
libcxxmpich.rts.a	Contains the C++ bindings for MPICH. Required for C++ MPI applications.
libdevices.rts.a	
libfmpich.rts.a and libfmpich_.rts.a	Contains the Fortran bindings for MPICH. Required for Fortran MPI applications.
liblinkcheck.rts.a	Library to facilitate link error verification at the user level. The link-checksum verification APIs allow users to periodically store checksums of injected data (on stable storage), and compare it later to ensure that there were no undetected transient faults in memory or internal state of a node. The link-CRC verification APIs allow users to periodically verify sent and received CRCs on network links (both torus and tree) to detect and isolate link faults. The library is typically used for diagnostics or debugging purposes, but users can optionally use the APIs to build safety checks in their application.

File name	Description
liblmpe.a, libmpe.a, libmpe_collchk.a, libmpe_f2cmpi.a, libmpe_nompi.a, libmpe_null.a, libtmpe.a, mpe_prof.o	Required for applications that perform MPE profiling.
libmpich.rts.a	This is the main MPI library. Required for any MPI application.
libmpirun_secure.a	
libmsglayer.rts.a	Contains many of the “glue” functions (hardware<->MPICH) and collectives. Required for any MPI application.
libpavtrace.a	
libprinters.a	
librts.rts.a	Blue Gene/L-specific functions for compute node applications.
libsaymessage.a	sayMessage is a general message facility to enable the generation of formatted messages.
libtableapi.a	The API set for MMCS components to work directly with the DB2® tables on the service node.

32-bit dynamic link files

There are currently no 32-bit dynamic link files for Blue Gene/L.

64-bit static link files

The 64-bit static link files for Blue Gene/L are in the /bgl/BlueLight/ppcfloor/bglsys/lib64 directory.

Table 1-3 64-bit static link files on Blue Gene/L

File name	Description
libbglallocator_s.a	Required when using the dynamic partition allocator APIs.
libbglbridge_s.a	Required when using the Bridge APIs.
libbgldb_s.a	
libbglmachine_s.a	
libsaymessage_s.a	sayMessage is a general message facility to enable the generation of formatted messages.
libtableapi_s.a	The API set for MMCS components to work directly with the DB2 tables on the service node.

64-bit dynamic link files

The 64-bit dynamic link files for Blue Gene/L are in the /bgl/BlueLight/ppcfloor/bglsys/lib64 directory.

Table 1-4 64-bit dynamic link files on Blue Gene/L

File name	Description
libbglallocator.so	Required when using the dynamic partition allocator APIs.
libbglbridge.so	Required when using the Bridge APIs.
libbgldb.so	
libbglmachine.so	
libsaymessage.so	sayMessage is a general message facility to enable the generation of formatted messages.
libtableapi.so	The API set for MMCS components to work directly with the DB2 tables on the service node.

1.6 Compilers overview

Two compiler families are supported on Blue Gene/L: GNU compilers and IBM XL compilers. We expect most users to take advantage of the additional performance optimizations found in the IBM XL compilers.

1.6.1 Programming environment overview

The diagram in Figure 1-4 provides a quick view into the software stack that supports the execution of Blue Gene/L applications.

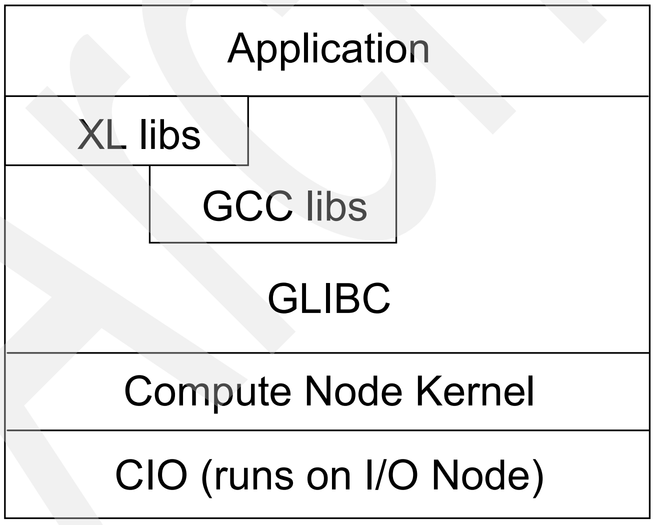


Figure 1-4 Software stack supporting execution of Blue Gene/L applications

1.6.2 GNU

The standard GNU version 3.4.3 C, C++, and Fortran77 compilers are modified during installation to support Blue Gene/L.

The toolchain has been updated to allow for support of SLES9 applications. The current versions are:

- gcc 3.4.3
- binutils 2.16.1
- glibc 2.3.6

You can find the GNU compilers in the /bgl/BlueLight/ppcfloor/blrts-gnu/bin directory.

For more information about how the XL compilers, GNU, and the toolchain all fit together, see 5.1, "Compiling and linking applications on Blue Gene/L" on page 44.

1.6.3 IBM XL compilers

The following IBM XL compilers are supported when developing Blue Gene/L applications:

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

See Chapter 5, "Developing applications with IBM XL compilers" on page 43, for more information.

Installation

For detailed steps about installing the IBM XL Fortran compiler, refer to the following Web addresses:

http://www.ibm.com/software/awdtools/fortran/xlfortran/features/bg/index.html
http://www.ibm.com/software/awdtools/fortran/xlfortran/features/bg/xlf-bg.html

For details about installing the XL C and C++ compilers, refer to the following Web addresses:

http://www.ibm.com/software/awdtools/xlcpp/features/bg/index.html
http://www.ibm.com/software/awdtools/xlcpp/features/bg/xlcpp-bg.html

Message catalogs

When you are running Blue Gene/L applications, you can make message catalogs work in several different ways. The key to success is that the I/O Nodes need to know where the message catalogs are.

Here are a couple examples on how to make this to work:

- Export /opt/ibmcmp from the Front End Node where the compilers were installed, and then mount /opt/ibmcmp onto the I/O Nodes, so the message catalogs are visible.

  This is the default location for the message catalogs, so no further action is needed.

- Place the compiler somewhere in /bgl or some other directory that is already mounted on the I/O Node. Do this by installing the catalogs into an alternate directory:

```
rpm -ivh xlf.msg.rte-9.1.0-3.ppc64.rpm --prefix /opt/ibmcmp --dbpath /bgl/rpm  --nodeps
rpm -ivh xlsmp.msg.rte-1.5.0-3.ppc64.rpm --prefix /opt/ibmcmp --dbpath /bgl/rpm
--nodeps
rpm -ivh vacpp.cmp-7.0.0-3.ppc64.rpm --prefix /opt/ibmcmp --dbpath /bgl/rpm  --nodeps
rpm -ivh vac.cmp-7.0.0-3.ppc64.rpm --prefix /opt/ibmcmp --dbpath /bgl/rpm --nodeps
```

Because this is not the default directory, set up your NLSPATH like this:

vacpp export
"NLSPATH=/opt/ibmcmp/vacpp/bg/9.0/msg/%L/%N:/opt/ibmcmp/vacpp/bg/9.0/opt.ibmcmp.msg/%L/%N"

vac export
"NLSPATH=/opt/ibmcmp/vac/bg/9.0/msg/%L/%N:/opt/ibmcmp/vac/bg/9.0/opt.ibmcmp.msg/%L/%N"

xlf export
"NLSPATH=/opt/ibmcmp/xlf/bg/11.1/msg/%L/%N:/opt/ibmcmp/xlf/bg/11.1/opt.ibmcmp.msg/%L/%N.cat"

Important: Support for creating applications targeting Blue Gene/L is *no longer* provided with the IBM XL for Linux compilers. Both compilers can exist simultaneously on the same Front End Node; however, you must purchase them separately.

The compilers are in the following directories:

- The XL C compilers are in the /opt/ibmcmp/vac/bg/8.0/bin/blrts_xlc directory.
- The XL C++ compilers are in the /opt/ibmcmp/vacpp/bg/8.0/bin/blrts_xlc++ directory.
- The XL Fortran compilers are in the /opt/ibmcmp/xlf/bg/10.1/bin/blrts_xlf directory.

You can find other versions of each of these compilers in the same directory as those that are listed. For example, compilers are available for Fortran77, Fortran90, and so on.

Mathematical Acceleration Subsystem libraries

The complete Mathematical Acceleration Subsystem (MASS) scalar and vector libraries for Blue Gene/L are available for free download on the Web at the following address:

http://www.ibm.com/software/awdtools/mass/bgl/

The libmassv.a and libmass.a libraries are also shipped with the compilers in /opt/ibmcmp/xlmass/bg/4.3/blrts_lib. The difference between the MASS libraries obtained from the referenced Web page compared to those delivered with the compilers is that the Web page has the most recent version, but is not officially supported. The libraries delivered with the compiler are not as current, but are more stable and supported.

XL runtime libraries

The libraries listed in Table 1-5 on page 16 are linked into your application automatically by the XL linker when you create your application.

Important: The exception to this statement is for the libmassv.a file (the MASS libraries). It must be explicitly specified on the linker command.

Table 1-5 XL libraries

File name	Description
libibmc++.a	IBM C++ library
libxlf90.a	IBM XLF runtime library
libxlfmath.a	IBM XLF stubs for math routines in system library libm, for example, _sin() for sin(), _cos() for cos(), and so on
libxlfpmt4.a	IBM XLF to be used with -qautobdl=dbl4 (promote floating-point objects that are single precision)
libxlfpad.a	IBM XLF run-time routines to be used with -qautobdl=dblpad (promote floating-point objects and pad other types if they can share storage with promoted objects)
libxlfpmt8.a	IBM XLF run-time routines to be used with -qautobdl=dbl8 (promote floating-point objects that are double precision)
libxl.a	IBM low-level runtime library
libxlopt.a	IBM XL optimized intrinsic library ► Vector intrinsic functions ► BLASS routines
libmass.a	IBM XL MASS library: Scalar intrinsic functions
libmassv.a	IBM XL MASSV library: Vector intrinsic functions
ibxlomp_ser.a	IBM XL Open MP compatibility library

Engineering and Scientific Subroutine Library libraries

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

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

Important: When using the IBM XL Fortran V10.1 for IBM System Blue Gene, customers must use ESSL 4.2.3 and not ESSL 4.2.2. If an attempt is made to install a wrong mix of ESSL and XLF, the rpm install fails with a dependency error message.

Programming modes

This chapter provides information about the way in which Message Passing Interface (MPI) is implemented and used on Blue Gene/L.

There are two main modes in which you can use Blue Gene/L:

- Communication Coprocessor Mode
- Virtual Node Mode

This chapter explores both of these modes in detail.

2.1 Communication Coprocessor Mode

In the default mode of operation of Blue Gene/L, named Communication Coprocessor Mode, each physical compute node executes a single compute process. The Blue Gene/L system software treats those two processors in a compute node asymmetrically. One of the processors (CPU 0) behaves as a main processor, running the main thread of the compute process. The other processor (CPU 1) behaves as an offload engine (coprocessor) that only executes specific operations.

The coprocessor is used primarily for offloading communication functions. It can also be used for running application-level coroutines.

2.2 Virtual Node Mode

The Compute Node Kernel in the compute nodes also supports a Virtual Node Mode of operation for the machine. In that mode, the kernel runs two separate processes in each compute node. Node resources (primarily the memory and the torus network) are shared by both processes.

In Virtual Node Mode, an application can use both processors in a node simply by doubling its number of MPI tasks, without explicitly handling cache coherence issues. The now distinct MPI tasks running in the two CPUs of a compute node have to communicate to each other. This problem was solved by implementing a virtual torus device, serviced by a virtual packet layer, in the scratchpad memory.

In Virtual Node Mode, the two 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 two processors in a compute node cannot be done over the network hardware. Instead, it is done via a region of memory, called the *scratchpad* to which both processors have access.

Virtual FIFOs make portions of the scratchpad look like a *send FIFO* to one of the processors and a *receive FIFO* to the other. Access to the virtual FIFOs is mediated with help from the hardware lockboxes.

From an application perspective, virtual nodes behave like physical nodes, but with less memory. Each virtual node executes one compute process. Processes in different virtual nodes, even those allocated in the same compute node, only communicate through messages. Processes running in virtual node mode cannot invoke coroutines.

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

2.3 Deciding which mode to use

Whether you choose to use Communication Coprocessor Mode or Virtual Node Mode depends largely on the type of application you plan to execute.

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

2.4 Specifying a mode

You specify which mode to use when booting a Blue Gene/L partition. How you do this depends on the mechanism, such as LoadLeveler® or **mpirun**, that you use to perform this function.

The default for **mpirun** is Communication Coprocessor Mode. To specify Virtual Node Mode, you use the following command:

```
mpirun ... -mode vn ...
```

See *Blue Gene/L: System Administration*, SG24-7178, for more information about the **mpirun** command.

3

System calls supported by the Compute Node Kernel

This chapter discusses the system calls (syscalls) that are supported by the Compute Node Kernel. It is important for you to understand which functions can be called, and perhaps more importantly, which ones cannot be called, by your application running on Blue Gene/L.

See Chapter 4, "Blue Gene/L-specific system calls" on page 31, for details about system calls that are specific to Blue Gene/L.

3.1 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 which 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 which are expected to run on Blue Gene/L, 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 7, "Checkpoint and restart support" on page 83.

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.

3.2 System calls

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

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

3.2.1 Return codes

As is true for return codes on a standard Linux system, a return code of 0 from a syscall indicates success. A value of -1 (negative one) indicates failure. In this case, errno contains further information about exactly what caused the problem.

3.2.2 List of supported system calls

Table 3-1 lists all system calls that are supported on Blue Gene/L.

Table 3-1 List of supported system calls

System call	Category	Description
access	File I/O	Determines the accessibility of a file
brk	Miscellaneous	Changes the data segment size
chdir	Directory	Changes the working directory
chmod	File I/O	Changes the mode of a file
chown	File I/O	Changes the owner and group of a file
close	File I/O	Closes a file descriptor
connect	Sockets	Connects a socket

System call	Category	Description
dup	File I/O	Duplicates an open descriptor
dup2	File I/O	Duplicates an open descriptor
execve	Miscellaneous	Runs a new program in the process. See 3.2.3, “execve system call” on page 26, for more details.
exit	Miscellaneous	Terminates a process
fchmod	File I/O	Changes the mode of a file
fchown	File I/O	Changes the owner and group of a file
fcntl	File I/O	Performs the following operations (commands) on an open file. These operations are in the <fnctl.h> include file. ▶ F_GETFL ▶ F_DUPFD ▶ F_GETLK ▶ F_SETLK ▶ F_SETLKW ▶ F_GETLK64 ▶ F_SETLK64 ▶ F_SETLKW64
fstat	File I/O	Gets the file status
fstat64	File I/O	Gets the file status
fstatfs	File I/O	Gets file system statistics
fstatfs64	File I/O	Gets file system statistics
fsync	File I/O	Synchronizes changes to a file
ftruncate	File I/O	Truncates a file to a specified length
ftruncate64	File I/O	Truncates a file to a specified length
getcwd	Directory	Gets the path name of the current working directory
getdents	Directory	Gets the directory entries
getdents64	Directory	Gets the directory entries
getgid	Process info	Gets the real group ID
getitimer	Time	Gets the value of the interval timer
getpeername	Sockets	Gets the name of the peer socket
getpid	Process info	Gets the process ID. The value is the MPI rank of the node, meaning that 0 is a valid value.
getrlimit	Process info	Gets information about resource limits
getrusage	Process info	Gets information about resource utilization. All time reported is attributed to the user application so the reported system time is always zero.
getsockname	Sockets	Gets the socket name
gettimeofday	Time	Gets the date and time

System call	Category	Description
getuid	Process info	Gets the real user ID
kill	Signal	Sends a signal. A signal can only be sent to the same process. See 3.2.5, “kill system call” on page 26, for more details.
lchown	File I/O	Changes the owner and group of a symbolic link
link	File I/O	Links to a file
lseek	File I/O	Moves the read/write file offset
llseek	File I/O	Moves the read/write file offset
lstat	File I/O	Gets the symbolic link status
lstat64	File I/O	Gets the symbolic link status
mkdir	Directory	Makes a directory
open	File I/O	Opens a file
read	File I/O	Reads from a file
readlink	File I/O	Reads the contents of a symbolic link
readv	File I/O	Reads a vector
recv	Sockets	Receives a message from a connected socket
recvfrom	Sockets	Receives a message from a socket
rename	File I/O	Renames a file
rmdir	Directory	Removes a directory
rts_alloc_lockbox	Compute Node Kernel	Allocate a lockbox object (mutex or barrier)
rts_coordinatesForRank	Compute Node Kernel	Get the physical coordinates given the logical rank
rts_dcache_evict_normal	Compute Node Kernel	Write modified normal lines from the L1 data cache
rts_dcache_invalidate	Compute Node Kernel	Invalidate a region in the L1 data cache
rts_dcache_store	Compute Node Kernel	Write a region of modified lines from the L1 data cache
rts_dcache_store_invalidate	Compute Node Kernel	Write and invalidate a region of modified lines from the L1 data cache
rts_free_sram	Compute Node Kernel	Free storage allocated from SRAM
rts_get_dram_window	Compute Node Kernel	Get access to DRAM with the given attributes
rts_get_jobid	Compute Node Kernel	Get the job identifier
rts_get_personality	Compute Node Kernel	Get the personality information
rts_get_processor_id	Compute Node Kernel	Get the processor id
rts_get_processor_version	Compute Node Kernel	Get the value of the processor version register
rts_get_scratchpad_window	Compute Node Kernel	Get access to the DRAM scratchpad storage
rts_get_timebase	Compute Node Kernel	Get the value of the timebase register
rts_get_virtual_process_window	Compute Node Kernel	Get access to the other processor’s address space

System call	Category	Description
rts_interrupt_control	Compute Node Kernel	Control interrupts for devices and exceptions
rts_malloc_sram	Compute Node Kernel	Allocate storage from SRAM
rts_rankForCoordinates	Compute Node Kernel	Get the logical rank given the physical coordinates
send	Sockets	Sends a message on a connected socket
sendto	Sockets	Sends a message on a socket
setitimer	Time	Sets the value of an interval timer. Only the following operations are supported: ▶ ITIMER_PROF ▶ ITIMER_REAL **Note**: An application can only set one active timer at a time.
setrlimit	Process info	Sets resource limits. Only RLIMIT_CORE can be set.
sigaction	Signals	Manages signals. The only flags supported are SA_RESETHAND and SA_NODEFER.
signal	Signals	Manages signals
sigreturn	Signals	Returns from a signal handler
socket	Sockets	Opens a socket
stat	File I/O	Gets the file status
stat64	File I/O	Gets the file status
statfs	File I/O	Gets file system statistics
statfs64	File I/O	Gets file system statistics
symlink	File I/O	Makes a symbolic link to a file
time	Time	Gets time
times	Process info	Gets the process times. All time reported is attributed to the user application so the reported system time is always zero.
truncate	File I/O	Truncates a file to a specified length
truncate64	File I/O	Truncates a file to a specified length
umask	File I/O	Sets and gets the file mode creation mask
uname	Miscellaneous	Gets the name of the current system, and other information (for example, version and release)
unlink	File I/O	Removes a directory entry
utime	File I/O	Sets file access and modification times
write	File I/O	Writes to a file
writev	File I/O	Writes a vector

3.2.3 execve system call

The Compute Node Kernel supports the `execve` system call, which enables the user to run a different program on the compute node. The `execve` system call has the following limitations:

- The control system does not know about the new program that is running after calling the `execve` system call. The job table and job history table contain the name of the initial program that was submitted to the partition. Any interfaces that use the program name field from the two job tables return the name of the submitted program.
- If a debugger is currently attached to the job, the `execve` system call returns an error and sets errno to ENOTSUP.
- The `execve` system call can be used only once per job. After the first call to `execve`, any subsequent calls return an error and set errno to ENOTSUP.
- There is no support for changing the effective user ID and effective group ID based on the SetUserID and SetGroupID mode bits. The Compute Node Kernel does not support this on job submission either.
- There is no support for the close-on-exec flag for file descriptors, so all open file descriptors remain open after the `execve` system call.
- There is a limit of 32 KB for the size of the arguments and environment variables passed to the new program.
- The new program must use the same origin address as the submitted program. If the submitted program obtains storage using the `rts_get_dram_window()` system call, the storage is still available and mapped in the same way after the `execve` system call.
- The submitted program must not use any MPI functions. The MPI library does not allow `MPI_Init()` to be called more than once per job.
- Extreme caution must be used if the application uses checkpoint or restart APIs. A checkpoint uses a barrier to sync all of the nodes in a partition and quiesce the networks. With separate programs running, it is highly variable as to when each program gets to the barrier and the checkpoint can start.

The `execve` system call can be used by an application to implement Multiple Program Multiple Data (MPMD) support. The application can provide a "launcher" program that is submitted and that in turn uses either the `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`, or `execve()` function to run other programs based on the MPI rank of the node.

3.2.4 fork system call

Because the Compute Node Kernel is a single-process operating system, no support is provided for an application to use multiple processes or threads. Calls to `fork()` and `pthread_create()` return -1, with errno set to ENOSYS.

In addition, no shell utilities are supported in the Compute Node Kernel. Only the system calls listed in this document are supported in the Compute Node Kernel. For example, no utility commands provided by such shells as BASH or Bourne can be invoked by applications running in the Compute Node Kernel.

3.2.5 kill system call

An application can send signals only to itself. For example, an application instance running on one node cannot directly send a signal to another node. Internode communication should be achieved using Message Passing Interface (MPI) support.

3.2.6 Other system calls

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

- Blue Gene/L does not support the use of the `system()` function. Therefore, for example, you cannot use something such as the `system('chmod -w file')` call.
- Blue Gene/L does not provide the same support for `gethostname()` and `getlogin()` as Linux provides.
- Blue Gene/L does not support calls to `signal(SIGTRAP, xl__trce)` or `signal(SIGNAL, xl__trbk)`.
- Calls to `usleep()` are not supported.

3.3 Additional Compute Node Kernel application support

This section provides details about additional support provided to application developers by the Compute Node Kernel.

3.3.1 Allocating memory regions with specific L1 cache attributes

Each PowerPC® 440 core on a compute node has a 32 KB L1 data cache that is 64-way set associative and uses a 32-byte cache line. A load or store operation to a virtual address is translated by the hardware to a real address using the Translation Lookaside Buffer (TLB). The real address is used to select the "set." If the address is available in the data cache, it is returned to the processor without needing to access lower levels of the memory subsystem. If the address is not in the data cache (a cache miss), a cache line is evicted from the data cache and is replaced with the cache line containing the address. The "way" to evict from the data cache is selected using a round-robin algorithm. The L1 data cache can be divided into two regions: a normal region and a transient region. The number of "ways" to use for each region can be configured by the application.

The Blue Gene/L memory subsystem supports the following L1 data cache attributes:

- Cache-inhibited or cached

  Memory with the cache-inhibited attribute causes all load and store operations to access the data from lower levels of the memory subsystem. Memory with the cached attribute might use the data cache for load and store operations. The default attribute for application memory is cached.
- Store without allocate (SWOA) or store with allocate (SWA)

  Memory with the SWOA attribute bypasses the L1 data cache on a cache miss for a store operation, and the data is stored directly to lower levels of the memory subsystem. Memory with the SWA attribute allocates a line in the L1 data cache when there is a cache miss for a store operation on the memory. The default attribute for application memory is SWA.
- Write-through or write-back

  Memory with the write-through attribute is written through to the lower levels of the memory subsystem for store operations. If the memory also exists in the L1 data cache, it is written to the data cache and the cache line is marked as clean. Memory with the write-back attribute is written to the L1 data cache, and the cache line is marked as dirty. The default attribute for application memory is write-back.

- Transient or normal

  Memory with the transient attribute uses the transient region of the L1 data cache. Memory with the normal attribute uses the normal region of the L1 data cache. By default, the L1 data cache is configured without a transient region and all application memory uses the normal region.

The Compute Node Kernel supports setting some L1 data cache attributes for all of application memory. The user can choose SWOA mode by setting the BGL_APP_L1_SWOA environment variable to a non-zero value and write-through mode by setting the BGL_APP_L1_WRITE_THROUGH environment variable to a non-zero value. For more details, see Appendix E, “Environment variables” on page 149.

The Compute Node Kernel supports allocating only a region of memory with a particular L1 data cache attribute with the `rts_get_dram_window()` system call. To make memory available for the system call, the application must be linked with a larger starting address.

Application developers must understand well the way that their code interacts with memory and cache when using `rts_get_dram_window()`; misuse can degrade performance. One example of where this feature might improve performance is during matrix multiplication operations where one matrix is used repeatedly and another is calculated, written to disk (or to another node), and then discarded. The latter matrix might best occupy a region of memory that has the SWOA attribute, because using valuable L1 data cache for this transient data can push other data (such as the repeatedly used matrix data) out of the L1 data cache. An alternative is to specify that the latter matrix use the transient region.

3.3.2 Allocating storage from SRAM

SRAM is a low- latency memory that is accessible by both PPC440 processors on a compute node. No caching is performed on SRAM, and coherence is maintained between both processors. It can be used as a fast communications buffer to pass data between the processors. There is 8 KB of SRAM available to applications, of which 4 KB is used by the MPI libraries.

The Compute Node Kernel now supports two new system calls:

- `rts_malloc_sram()` to allocate storage from SRAM
- `rts_free_sram()` to free storage from SRAM

3.3.3 Balancing performance and parity error recovery

For Blue Gene/L hardware, most data in the memory subsystem is Error Correction Code (ECC) protected. This protection allows for hardware recovery of single-bit errors along with detecting all double-bit errors. However, the L1 instruction cache, L1 data cache, and TLBs are only parity protected.

For sample code that shows how to register a callback and simulate a cache parity error, see Appendix F, “L1 data cache parity error recovery: Sample code” on page 155.

The Compute Node Kernel provides extensive support for handling parity errors and allowing applications to choose a balance between performance and parity error recovery. *Parity errors* are soft errors that do not indicate a hardware failure that requires replacement. Parity errors happen on all systems, but the large number of processors on Blue Gene/L increases the probability and impact of a parity error. Users are encouraged to benchmark their applications with the various memory configurations to find the right balance between performance and the ability to recover from L1 data cache parity errors.

Compute Node Kernel automatically handles L1 instruction cache parity errors by invalidating the instruction cache and allowing the instructions to be reloaded from lower levels of the memory subsystem.

For L1 data cache parity errors, Compute Node Kernel provides several options:

- Running in write-back mode with no parity error recovery
- Running in write-through mode with automatic parity error recovery
- Running in write-back mode with application assisted parity error recovery

The default option is *write-back mode* with no parity error recovery. When a parity error occurs, Compute Node Kernel ends the job because there is no way to recover. An application can use the checkpoint library to take regular checkpoints during the execution of the job and restore from the checkpoint after a parity error. This option provides the best runtime performance from the memory subsystem. However, frequent checkpoints cause additional I/O, which can impact overall application performance, and the disk usage for the checkpoint files can be large. In addition, the default option has the greatest risk for an abnormal termination because there is no way to recover from parity errors.

When running in *write-through mode*, the L1 data cache is always clean; the Compute Node Kernel handles all parity errors by invalidating the L1 data cache and allowing the data to be reloaded from lower levels of the memory subsystem. There is additional overhead in using the write-through mode, which impacts application performance.

You choose the write-through option by setting the environment variable BGL_APP_L1_WRITE_THROUGH to a non-zero value when submitting a job. Compute Node Kernel configures the L1 data cache before the application starts and automatically handles the parity errors as they occur during the execution of the job. This option is the simplest way to avoid parity errors but has the biggest performance impact to applications. Some applications can get back some of the performance by also using the store without allocate mode. You set the BGL_APP_L1_SWOA environment variable to a non-zero value to turn on the store without allocate attribute. The default is for application memory to use the store with allocate attribute.

Compute Node Kernel has support for handling L1 data cache parity errors when running in *write-back mode with application assisted parity error recovery*. If the parity error occurs on a clean cache line, Compute Node Kernel can automatically handle the parity error. If the parity error occurs on a dirty cache line, Compute Node Kernel flushes any other dirty cache lines in the L1 data cache, invalidates the data cache, and transfers control to an application supplied handler. If no handler is specified, Compute Node Kernel ends the job.

You choose the write-back mode with application assisted parity error recovery option by setting the environment variable BGL_APP_L1_WRITEBACK_RECOVERY to a non-zero value when submitting a job. This option provides good runtime performance from the memory subsystem. Additional overhead is added by the processor's load pipeline to enable proper recovery of parity errors, which impacts application performance. In addition, the application support is relatively complicated to implement and does not allow all parity errors to be recovered. An application can get automatic recovery of clean cache lines even if a handler is not specified.

Important: If you have a long running job that you do not want interrupted by an unrecoverable parity error, set the environment variable BGL_APP_L1_WRITE_THROUGH to a non-zero value when submitting the job. See the following example using the `mpirun` command:

```
$ mpirun -env "BGL_APP_L1_SWOA=1" -env "BGL_APP_L1_WRITE_THROUGH=1" -partition R001 -cwd
`pwd` -exe `pwd`/bg_application
```

4

Blue Gene/L-specific system calls

The following pages describe the list of Blue Gene/L-specific system calls (syscalls). They are listed in alphabetical order. In some cases, this section provides additional details to the list of system calls documented in Chapter 3, “System calls supported by the Compute Node Kernel” on page 21.

4.1 rts_alloc_lockbox

```
#include <bgllockbox.h>
#include <rts.h>

int rts_alloc_lockbox(
   int type,
   void **ptr,
   int flags);
```

`rts_alloc_lockbox` allocates a barrier or mutex object from the hardware lockbox. There are eight barrier objects and 64 mutex objects available for the application to allocate. The barrier and mutex objects can be used to synchronize the two processors on a compute node.

The `type` parameter identifies the object to allocate. The BGL_BARRIER_001 through BGL_BARRIER_008 and BGL_MUTEX_001 through BGL_MUTEX_064 constants are provided in the rts.h header file for each of the valid objects.

The `ptr` parameter is the address of an opaque pointer to a lockbox object. Upon successful completion, `ptr` is set to the lockbox object pointer. The pointer to a barrier object can be used as input to the `BGL_Barrier_Lower()`, `BGL_Barrier_Is_Raised()`, `BGL_Barrier_Pass()`, or `BGL_Barrier_Pass_fn()` functions. The pointer to a mutex object can be used as input to the `BGL_Mutex_Try()`, `BGL_Mutex_Acquire()`, `BGL_Mutex_Release()`, or `BGL_Mutex_Is_Locked()` functions. See the bgllockbox.h header file for more information.

The `flags` parameter controls how the lockbox object is allocated. When the flag BGL_LOCKBOX_ORDERED_ALLOC is set, `rts_alloc_lockbox` does not return until both processors have successfully allocated an object.

`rts_alloc_lockbox` returns 0 when successful. Otherwise it returns -1 and sets errno to one of the following values:

- EFAULT if the `ptr` parameter is an invalid application address
- EINVAL if either the `type` or `flags` parameter is invalid

4.2 rts_coordinatesForRank

```
#include <rts.h>

int rts_coordinatesForRank(
unsigned logicalRank,
unsigned *x,
unsigned *y,
unsigned *z,
unsigned *t);
```

`rts_coordinatesForRank` returns the physical coordinates associated with a logical rank. The `logicalRank` parameter specifies the logical rank of the processor.

The `x`, `y`, `z`, and `t` parameters are pointers to unsigned integers. Upon successful completion, they are set to the X, Y, Z, and T physical coordinates, respectively.

`rts_coordinatesForRank` returns 0 when successful. Otherwise it returns -1 and sets errno to one of the following values:

- EFAULT if any of the `x`, `y`, `z`, or `t` parameters is an invalid application address
- EINVAL if the `logicalRank` parameter is invalid

4.3 rts_dcache_evict_normal

```
#include <rts.h>

int rts_dcache_evict_normal(void);
```

`rts_dcache_evict_normal` writes all modified lines from the L1 data cache. For large memory regions, `rts_dcache_evict_normal` is more efficient than `rts_dcache_store`. No errors are defined.

4.4 rts_dcache_invalidate

```
#include <rts.h>

int rts_dcache_invalidate(
void *beg,
void *end);
```

`rts_dcache_invalidate` invalidates a region of memory in the L1 data cache using the **`dbci`** PowerPC 440 instruction.

The `beg` parameter is a pointer to the beginning of the memory region. This parameter must point to memory that is 32-byte aligned since the L1 cache line size is 32 bytes.

The `end` parameter is a pointer to the end of the memory region. The entire cache line containing end is invalidated. If end is not 32-byte aligned, data beyond `end` is also invalidated.

`rts_dcache_invalidate` returns 0 when successful. Otherwise it returns -1 and sets errno to the value:

- EFAULT if either the `beg` or `end` parameter is an invalid application address or if the `beg` parameter is not 32-byte aligned.

4.5 rts_dcache_store

```
#include <rts.h>

int rts_dcache_store(
void *beg,
void *end);
```

`rts_dcache_store` writes the modified lines from the L1 data cache for a region of memory using the **`dbcst`** PowerPC 440 instruction.

The `beg` parameter is a pointer to the beginning of the memory region. This parameter must point to memory that is 32-byte aligned, since the L1 cache line size is 32 bytes.

The `end` parameter is a pointer to the end of the memory region. The entire cache line containing `end` is written. If end is not 32-byte aligned, data beyond `end` is also written.

`rts_dcache_store` returns 0 when successful. Otherwise it returns -1 and sets errno to the value:

- EFAULT if either the `beg` or `end` parameter is an invalid application address or if the `beg` parameter is not 32-byte aligned.

4.6 rts_dcache_store_invalidate

```
#include <rts.h>

int rts_dcache_store_invalidate(
void *beg,
void *end);
```

`rts_dcache_store_invalidate` writes the modified lines from the L1 data cache and invalidates the lines for a region of memory using the **`dbcf`** PPC instruction.

The `beg` parameter is a pointer to the beginning of the memory region. This parameter must point to memory that is 32-byte aligned since the L1 cache line size is 32 bytes.

The `end` parameter is a pointer to the end of the memory region. The entire cache line containing `end` is written and invalidated. If `end` is not 32-byte aligned, data beyond `end` is also written and invalidated.

`rts_dcache_store_invalidate` returns 0 when successful. Otherwise it returns -1 and sets errno to the value:

- EFAULT if the `beg` or `end` parameter is an invalid application address or if the `beg` parameter is not 32-byte aligned.

4.7 rts_free_sram

```
#include <rts.h>

int rts_free_sram(void *ptr);
```

`rts_free_sram` frees SRAM storage that was previously allocated by `rts_malloc_sram`.

The `ptr` parameter specifies the storage to be freed. It must be an address returned by `rts_malloc_sram`.

`rts_free_sram` returns 0 when successful. Otherwise it returns -1 and sets errno to the value:

- EINVAL if the `ptr` parameter is not a valid SRAM pointer.

4.8 rts_get_dram_window

```
#include <rts.h>

int rts_get_dram_window(
unsigned int  requestedSize,
unsigned int  flags,
void**        location,
unsigned int* actualSize
);
```

`rts_get_dram_window` provides applications with a mechanism to request blocks of memory having one or more of the following L1 cache attributes:

- Write-through

  Memory with the write-through attribute is written through to the lower levels of the memory subsystem for store operations. If the memory also exists in the L1 data cache, it is written to the data cache and the cache line is marked as clean.
- Cache-inhibited

  Memory with the cache-inhibited attribute causes all load and store operations to access the data from lower levels of the memory subsystem.
- Store without allocate

  Memory with the store without allocate attribute bypasses the L1 data cache on a cache miss for a store operation and the data is stored directly to lower levels of the memory subsystem.
- Transient

  Memory with the transient attribute uses the transient region of the L1 data cache.

Applications must be linked differently in order to use this system call as explained in the following section.

The amount of memory available for memory block allocation is limited to 14 MB. Furthermore, memory blocks are allocated in 1 MB increments. Thus applications should not treat `rts_get_dram_window` as a fine-grained memory allocator like `malloc()`.

The `requestedSize` parameter specifies the size in bytes of the memory block required by the application.

The `flags` parameter specifies the storage attributes of the memory block. It is the logical OR of one or more of the following flags defined in rts.h:

- RTS_STORE_WITHOUT_ALLOCATE
- RTS_WRITE_THROUGH
- RTS_CACHE_INHIBITED
- RTS_TRANSIENT

Attention: We recommend that you do not combine RTS_CACHE_INHIBITED with RTS_STORE_WITHOUT_ALLOCATE, RTS_WRITE_THROUGH or RTS_TRANSIENT. Doing so results in an error.

The `location` parameter is the address of a pointer. Upon successful completion, `location` is set to the memory block's address.

The `actualSize` parameter is the address of the unsigned integer. Upon successful completion, `actualSize` is set to the memory block's actual size, which can be different from the requested size.

`rts_get_dram_window` returns 0 when successful. A memory block at least as large as `requestedSize` has been allocated and has the attributes requested by `flags`. The memory block's address and size are returned via the location and `actualSize` parameters, respectively.

`rts_get_dram_window` returns -1 when unsuccessful and sets errno to one of the following values:

- EFAULT if either the `location` or `actualSize` parameter is an invalid application address.
- EINVAL if conflicting cache attributes have been specified via the `flags` argument, or if RTS_TRANSIENT was specified and the transient region is not available.
- ENOMEM if insufficient storage is available to fulfill the request.

4.8.1 Linking applications to use rts_get_dram_window

To use `rts_get_dram_window`, applications must be linked in a special manner. Specifically, the executable origin must be moved from 0x00200000 (2 MB) to a higher address. This new origin must fall on a 1 MB boundary and must not be larger than 0x01000000 (16 MB). If you do not link your application in this manner, `rts_get_dram_window` returns an error and sets errno to ENOMEM.

To link an application in this manner:

1. Make a copy of the blrts-gnu/powerpc-bgl-blrts-gnu/lib/ldscripts/elf32ppcblrts.x file, which is in the Blue Gene/L toolchain image of your front end node.
2. Edit the copy, updating the `__executable_start` symbol.
3. Re-link your application, using the `-Xlinker --script=your-script` option of gcc.

The second step is achieved by updating the line that looks like this:

```
PROVIDE (__executable_start = 0x00200000); . = 0x00200000 + SIZEF_HEADERS;
```

You update it to something like this:

```
PROVIDE (__executable_start = 0x01000000); . = 0x01000000 + SIZEF_HEADERS;
```

4.9 rts_get_jobid

```
#include <rts.h>

int rts_get_jobid(void);
```

`rts_get_jobid` returns the job identifier assigned to the job by the control system. There are no defined errors.

4.10 rts_get_personality

```
#include <rts.h>
#include <bglpersonality.h>

int rts_get_personality(
BGLPersonality *dst,
unsigned int size);
```

`rts_get_personality` returns the node personality information which includes hardware configuration data, torus, and collective network configuration data, and processor set (pset) configuration data. See the bglpersonality.h header file for the details about the information in the personality.

The `dst` parameter is a pointer to a buffer for storing the BGLPersonality structure. Upon successful completion, the buffer pointed to by `dst` contains the personality information.

The `size` parameter is the size of the buffer pointed to by the `dst` parameter.

`rts_get_personality` returns 0 when successful. Otherwise it returns -1 and sets errno to one of the following values:

- EFAULT if the `dst` parameter is an invalid application address
- EINVAL if the `size` parameter is not the size of a BGLPersonality structure

4.11 rts_get_processor_id

```
int rts_get_processor_id(void);
```

`rts_get_processor_id` returns the processor identifier. When running in coprocessor mode, the main processor returns 0 and the coprocessor returns 1. When running in virtual node mode, one processor returns 0 and the other processor returns 1. No errors are defined.

4.12 rts_get_processor_version

```
int rts_get_processor_version(void);
```

`rts_get_processor_version` returns the version of the processor. The processor version is defined to be the contents of the PVR register in the PowerPC 440 processor. No errors are defined.

See the *PPC440x5 CPU Core User's Manual* at the following Web address for additional information about the PVR:

http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core

4.13 rts_get_scratchpad_window

```
int rts_get_scratchpad_window(
void **location,
unsigned int *size
);
```

`rts_get_scratchpad_window` returns the address and size of the scratchpad storage area, which is a cache-inhibited read/write region of DRAM. This area can be used for inter-processor communication between the two processors on a compute node.

The `location` parameter is the address of a pointer. Upon successful completion, the pointer contains the address of the scratchpad storage area.

The `size` parameter is the address of an unsigned integer. Upon successful completion, the unsigned integer contains the size of the scratchpad storage area.

`rts_get_scratchpad_window` does not allocate storage. It returns the address and size of the entire scratchpad area. Subsequent calls might return the same values.

Note: The MPI library uses the scratchpad storage area. Applications that use MPI should not use the scratchpad storage area.

rts_get_scratchpad_window returns 0 when successful. Otherwise it returns -1 and sets errno to the value:

- EFAULT if either the location or size parameter is an invalid application address.

4.14 rts_get_timebase

```
unsigned long long rts_get_timebase(void);
```

rts_get_timebase returns the contents of the PowerPC 440 Time Base registers. That is, it returns the concatenation of TBU and TBL registers. No errors are defined.

See *PPC440x5 CPU Core User's Manual* at the following Web address for additional information about Time Base:

http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core

4.15 rts_get_virtual_process_window

```
int rts_get_virtual_process_window(
void **location,
unsigned int *size
);
```

In virtual node mode, the other processor's memory is visible in the application's address space. rts_get_virtual_process_window returns the address and size of the other processor's memory. It is valid only in virtual node mode. The other processor's memory is read-only.

The location parameter is the address of a pointer. Upon successful completion, the pointer is set to the address of the other processor's memory.

The size parameter is the address of an unsigned integer. Upon successful completion, the unsigned integer is set to the size (in bytes) of the other processor's memory.

rts_get_virtual_process_window returns 0 when successful. Otherwise it returns -1 and sets errno to one of the following values:

- ENOSYS if the node is not in virtual node mode
- EFAULT if either the location or size parameter is an invalid application address

4.16 rts_interrupt_control

```
int rts_interrupt_control(
int action,
struct interrupt *intrinfo,
size_t size);
```

rts_interrupt_control allows an application to control interrupts for devices and exceptions.

The action parameter is one of the following values:

- INTERRUPT_SET_ENABLE to set an interrupt handler and enable an interrupt group
- INTERRUPT_ENABLE to enable a set of interrupts
- INTERRUPT_DISABLE to disable a set of interrupts
- INTERRUPT_CHANGE to enable and disable a set of interrupts for an interrupt group

The `intrinfo` parameter is a pointer to a buffer containing a struct interrupt, which has the following definition:

```
struct interrupt
{
   int group;
   unsigned int interrupts;
   unsigned int flags;
   intrhandler_t handler;
   unsigned int *enabled;
 };
```

The group field is one of the following values:

- INTERRUPT_COMMUNICATION to control interrupts for the communication libraries. This group is used by the MPI library and should not be used by applications
- INTERRUPT_PARITY to control exceptions for L1 data cache parity errors

The `interrupts` field is a *bit mask* of the set of interrupts to control with the specified action. Constants for the valid interrupts are defined in rts.h. For the INTERRUPT_SET_ENABLE and INTERRUPT_CHANGE actions, if a bit is on, the interrupt is enabled; if a bit is off, the interrupt is disabled. For the INTERRUPT_ENABLE action, if a bit is on, the interrupt is enabled; if a bit is off, the interrupt is not modified. For the INTERRUPT_DISABLE action, if a bit is on, the interrupt is disabled; if a bit is off, the interrupt is not modified.

The `flags` field controls how the specified handler is invoked. When the INTERRUPT_INVOKE_ENABLED flag is set, the handler is invoked with the specified interrupts in the group still enabled. When the INTERRUPT_NO_RETURN flag is set, the kernel assumes that the handler will not return after handling the interrupt.

The `handler` field is a function pointer to the handler function to be run when an interrupt or exception is delivered. By default, all interrupts in the group are disabled when the handler is invoked. The handler has the following prototype:

```
void handler(unsigned int, unsigned int *, void *);
```

The first parameter is a bit mask that identifies the interrupt that caused the handler to be invoked. The second parameter is the `enabled` pointer from the struct interrupt (see the following paragraph for details). The third parameter is a pointer to a buffer and which can be NULL. For the INTERRUPT_COMMUNICATION group, the third parameter is NULL. For the INTERRUPT_PARITY group, the third parameter points to a buffer that contains two unsigned integers. The first unsigned integer is the address of the instruction that was running when the parity error occurred. The second unsigned integer is the effective address of the parity error.

The `enabled` field is a pointer to an unsigned integer. Before the handler returns, it sets the unsigned integer to a bit mask; that is the set interrupts the kernel should re-enable before returning control to the application.

The `size` parameter is the size of the buffer pointed to by the `intrinfo` parameter.

rts_interrupt_control returns 0 when successful. Otherwise it returns -1 and sets errno to one of the following values:

- EFAULT if intrinfo is an invalid application address
- EINVAL if either action or size is an invalid value

See Appendix F, “L1 data cache parity error recovery: Sample code” on page 155, for an example of how this API can be used.

4.17 rts_malloc_sram

```
void *rts_malloc_sram(size_t size);
```

rts_malloc_sram allocates storage from SRAM, which is low-latency coherent memory that is shared by both PowerPC 440 processors on a compute node. It can be used as a fast communications buffer between the two processors. There is 8 KB of SRAM storage on a compute node. The MPI library uses at least 4 KB of SRAM storage.

The size parameter specifies the requested size of the storage to be allocated. The requested size is rounded to a multiple of 32 bytes.

rts_malloc_sram returns a pointer to the allocated storage when successful. Otherwise it returns -1 and sets errno to one of the following values:

- ENOMEM if SRAM storage is not available
- EINVAL if the size parameter is 0

4.18 rts_rankForCoordinates

```
int rts_rankForCoordinates(
      unsigned x,
      unsigned y,
      unsigned z,
      unsigned t,
      unsigned *logicalRank,
      unsigned *numProcs);
```

rts_rankForCoordinates returns the logical rank associated with the given physical coordinates. It also returns the number of processors participating in the current job.

The x, y, z, and t parameters specify the X, Y, Z, and T physical coordinates, respectively.

The logicalRank parameter is a pointer to an unsigned integer. Upon successful completion, the unsigned integer is set to the logical rank.

The numProcs parameter is a pointer to an unsigned integer. Upon successful completion, the unsigned integer is set to the number of processors participating in the current job.

rts_rankForCoordinates returns 0 when successful. Otherwise it returns -1 and sets errno to one of the following values:

- EFAULT if the logicalRank or numProcs parameter is an invalid application address
- EINVAL if x, y, z, or t parameters are invalid physical coordinates

4.19 rts_write_dcr

```
int rts_write_dcr(unsigned int num, unsigned int value);
```

rts_write_dcr sets the value of the specified device control register (DCR) or special purpose register (SPR).

When num is BGL_SPR_DVLIM, the value is the new setting for the data cache victim limit register, which configures the transient region and normal region of the L1 data cache. See *PPC440x5 CPU Core User's Manual*, SA14-2523, at the following Web address for additional information about the data cache victim limit register:

http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core

rts_write_dcr returns 0 when successful. Otherwise it returns -1 and sets errno to the following value:

- EINVAL if num is not a valid DCR or SPR

5

Developing applications with IBM XL compilers

The IBM XL family of optimizing compilers allows you to develop C, C++, and Fortran applications for Blue Gene/L. This family comprises the following products (referred to in this chapter as *Blue Gene XL compilers*):

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

The information presented in this document is specific to the Blue Gene/L 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.ibm.com/software/awdtools/fortran/xlfortran/library/`

This chapter discusses specific considerations for developing, compiling, and optimizing C/C++ and Fortran applications for the Blue Gene/L PowerPC 440d processor architecture and its Double Hummer floating-point unit (FPU).

5.1 Compiling and linking applications on Blue Gene/L

This section contains information about compiling and linking applications that will run on Blue Gene/L. For complete information about compiler and linker options, see the following documents available from the Web addresses that are provided:

- *XL Fortran User Guide*

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

- *XL C/C++ Compiler Reference*

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

You can also find these documents in the following directories:

- /opt/ibmcmp/xlf/bg/11.1/doc (Fortran)
- /opt/ibmcmp/vacpp/bg/9.0/doc (C++_)
- /opt/ibmcmp/vac/bg/9.0/doc (C)

5.2 Default compiler options

Compilations most commonly occur on the Front End Node. The resulting program can run on the Blue Gene/L system without manually copying the executable to the Service Node. See 6.1, "Running applications" on page 76, to learn how to run programs on Blue Gene/L.

The script or makefile that you use to invoke the compilers should set certain compiler options to the following defaults:

- `-qbgl`

  This setting marks the object file as stand-alone to run on Blue Gene/L.

- Architecture-specific options, which optimize processing for the Blue Gene/L 440d processor architecture:

  - `-qarch=440d`

    Generates parallel instructions for the 440d Double Hummer dual FPU. If you encounter problems with code generation, you can reset this option to `-qarch=440`. This generates code for a single FPU only, but it might give correct results if invalid code is generated by `-qarch=440d`.

  - `-qtune=440`

    Optimizes object code for the 440 family of processors.

  - `-qcache=level=1:type=i:size=32:line=32:assoc=64:cost=8`

    Specifies the L1 instruction cache configuration for the Blue Gene/L 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/L 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/L 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/L 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. The names are listed in Table 5-1.

Table 5-1 Scripts available in the bin directory for compiling and linking

Language	Script name or names
C	blrts_xlc
C++	blrts_xlc++
Fortran	blrts_xlf, blrts_xlf90, blrts_xlf95

Important: The Double Hummer FPU does not generate exceptions. Therefore, the `-qflttrap` option, which traps floating-point exceptions, is disabled by default. If you enable this option, `-qarch` is automatically reset to `-qarch=440`.

5.3 Unsupported options

The following compiler options are not supported by the Blue Gene/L hardware and should not be used:

- `-qsmp`: This option requires shared memory parallelism, which is not used by Blue Gene/L.
- `-q64`: Blue Gene/L uses a 32-bit architecture; you cannot compile in 64-bit mode.
- `-qaltivec`: The 440 processor does not support VMX instructions or vector data types.
- `-qpic`: This option controls the selection of TOC size for Position Independent Code.
- `-qmkshrobj`: This option creates a shared library object.
- `-shared`: This option specifies dynamic linking, which is not currently supported with Blue Gene/L.

5.4 Tuning your code for Blue Gene/L

The sections that follow describe strategies that you can use to best exploit the single-instruction-multiple-data (SIMD) capabilities of the Blue Gene/L 440d processor and the XL compilers' advanced instruction scheduling and register allocation algorithms.

5.5 Using the compiler optimization options

The `-O3` compiler option provides a high level of optimization and automatically sets other options that are especially useful on Blue Gene/L. The `-qhot=simd` option enables SIMD vectorization of loops. It is enabled by default if you use `-O4`, `-O5`, or `-qhot`.

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

- "Optimizing XL Fortran programs" in the *XL Fortran User Guide*

  `http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/`
- "Optimizing your applications" in the *XL C/C++ Programming Guide*

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

5.6 Structuring data in adjacent pairs

The Blue Gene/L 440d 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 speed up 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, the C code in Example 5-1 allows each pair of elements in a structure to be operated on in parallel.

Example 5-1 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 5.13, "Complex type manipulation functions" on page 55, for a description of the set of built-in functions that are available for Blue Gene/L. These functions are especially designed to efficiently manipulate complex-type data and include a function to convert non-complex data to complex types.

5.7 Using vectorizable basic blocks

The compiler schedules instructions most efficiently within *extended basic blocks*. These are code sequences which 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 needs to be generated, you can compile with the `-qnoeh` compiler option (automatically set with `-O3`).

In addition, the optimal basic blocks remove dependencies between computations, so that the compiler sees 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 5.9, “Removing possibilities for aliasing (C/C++)” on page 48, for additional strategies for removing data dependencies.

5.8 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`

The following options allow you to 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.

Important: 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 Fortran User Guide*

  http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/
- *XL C/C++ Compiler Reference*

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

5.9 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 5-2.

Example 5-2 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 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 5-3.

Example 5-3 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 the pragma to inform the compiler that two pointer variables do not point to the same memory address. The directive in Example 5-4 indicates to the compiler that the pointers-to-arrays of double `x` and `f` do not share memory.

Example 5-4 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.

5.10 Structure computations in batches of five or ten

Floating-point operations are pipelined in the 440 processor, so that one floating-point calculation is performed per cycle, with a latency of five cycles. Therefore, to keep the 440 processor's floating-point units busy, organize floating-point computations to perform step-wise operations in batches of five, for example, arrays of five elements and loops of five iterations. For the 440d, which has two FPUs, use batches of ten.

For example, with the 440d, at high optimization, the function in Example 5-5 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 5-5 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 5-6 shows "wrapping" the inlined, optimized `ten_reciprocal_roots` function, in Example 5-5, 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 5-6 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++;
    }
}
```

5.11 Checking for data alignment

The Blue Gene/L 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, which is 32-bytes. If they cross this boundary, the hardware generates an alignment trap. This trap might cause the program to crash or result in a severe performance penalty to be fixed at run-time by the kernel.

The compiler does not generate these parallel load and store instructions unless it is sure that 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.

Since data alignment functions are in fact considered "built-in" functions, they require you to include the builtins header file (builtins.h) into your application source.

The C/C++ prototype for the function is shown in Example 5-7.

Example 5-7 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 Fortran, 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 5-8 asserts that the variables *x* and *f* are aligned along 16-byte boundaries.

Example 5-8 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]);
}
```

Important: 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 5-9 checks to see whether the incoming values are correctly aligned. Then it calls the "aligned" (Example 5-8) or "unaligned" (Example 5-5) version of the function according to the result.

Example 5-9 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 5-9 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).

5.12 Using XL built-in floating-point functions for Blue Gene/L

The XL C/C++ and Fortran 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 Blue Gene/L, the XL compilers provide a set of built-in functions that are specifically optimized for the PowerPC 440d Double Hummer dual FPU. These built-in functions provide an almost one-to-one correspondence with the Double Hummer 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 *second* 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/L built-in functions perform the 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 5-1 illustrates how a parallel-multiply operation is performed.

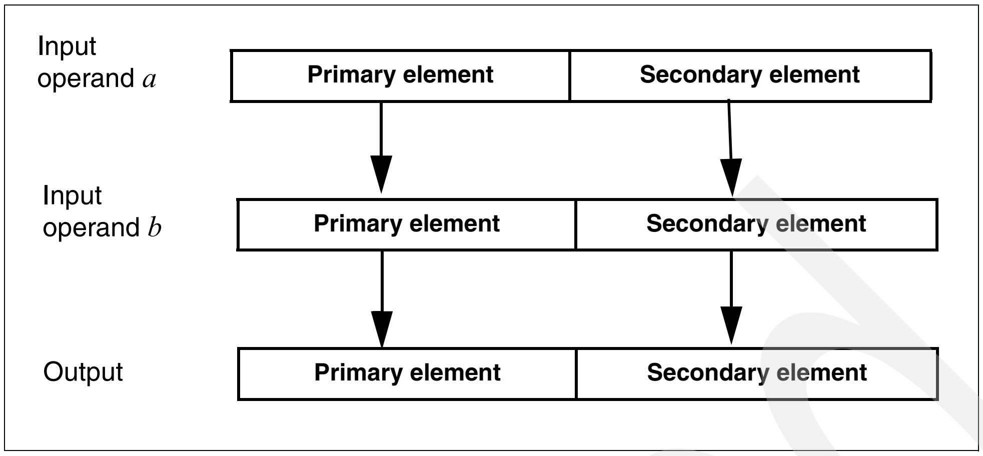


Figure 5-1 Parallel operations

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 5-2 illustrates how a cross-multiply operation is performed.

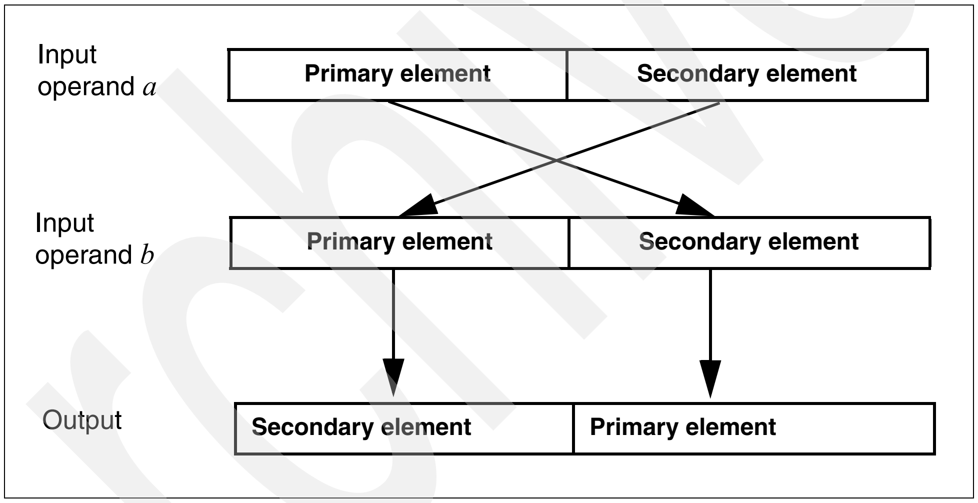


Figure 5-2 Cross operations

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 5-3 illustrates how a cross-primary-multiply operation is performed.

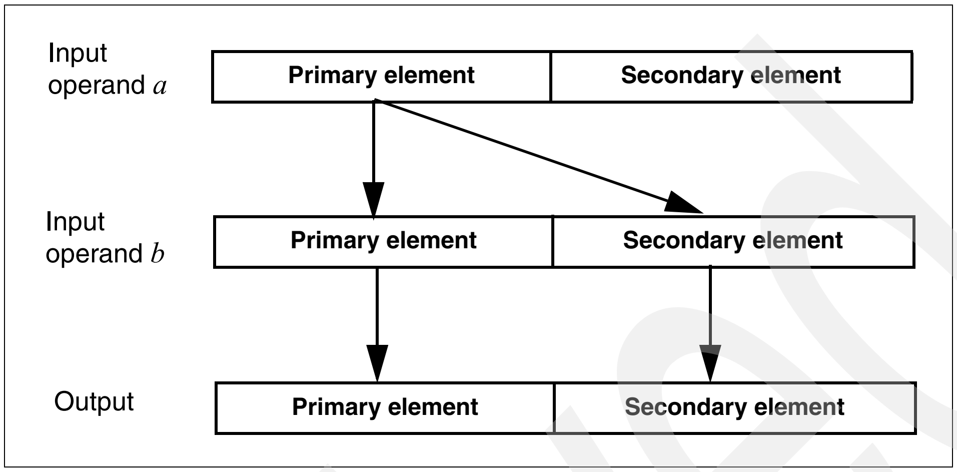


Figure 5-3 Copy-primary operations

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 5-4 illustrates how a cross-secondary-multiply operation is performed.

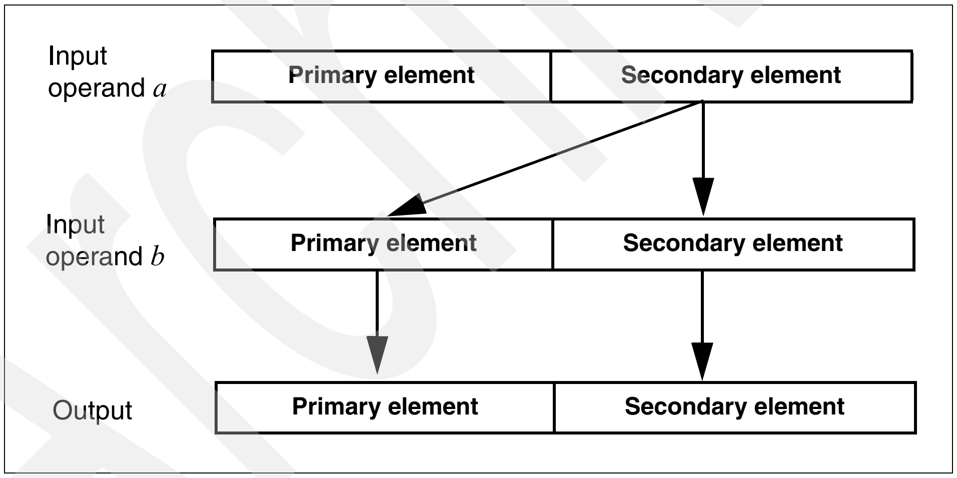


Figure 5-4 Copy-secondary operations

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 5-5 illustrates the result of a cross-copy-multiply operation.

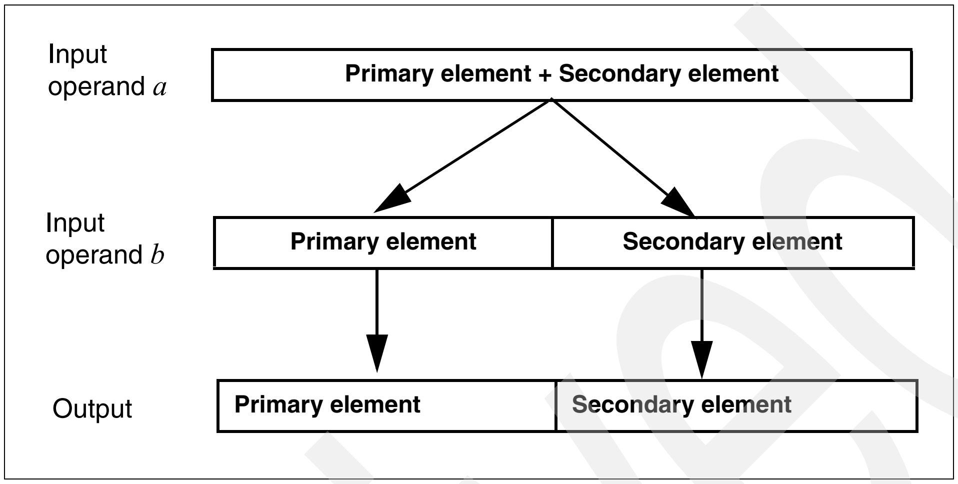


Figure 5-5 Cross-copy operations

The following paragraphs 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 need to include the header file builtins.h.

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

All of the built-in functions, with the exception of the complex type manipulation functions, require compilation under `-qarch=440d`. This is the default setting on Blue Gene/L.

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/L built-in functions, follow the guidelines provided in 5.4, "Tuning your code for Blue Gene/L" on page 45. Using the `alignx` built-in function (described in 5.11, "Checking for data alignment" on page 49), and specifying the `disjoint` pragma (described in 5.9, "Removing possibilities for aliasing (C/C++)" on page 48), are recommended for code that calls any of the built-in functions.

5.13 Complex type manipulation functions

These functions, listed in Table 5-2, are useful for efficiently manipulating complex data types. They allow you to automatically convert real floating-point data to complex types and to extract the real (primary) and imaginary (secondary) parts of complex values.

Table 5-2 Complex type manipulation functions

Function	**Convert dual reals to complex (single-precision): __cmplxf**
Purpose	Converts two single-precision real values to a single complex value. The real *a* is converted to the primary element of the return value, and the real *b* is converted to the secondary element of the return value.
Formula	primary(result) =a secondary(result) = b
C/C++ prototype	float _Complex __cmplxf (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): __cmplx**
Purpose	Converts two double-precision real values to a single complex value. The real *a* is converted to the primary element of the return value, and the real *b* is converted to the secondary element of the return value.
Formula	primary(result) =a secondary(result) = b
C/C++ prototype	double _Complex __cmplx (double a, double b); long double _Complex __cmplxl (long double a, long double b);[1]
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): __crealf**
Purpose	Extracts the primary part of a single-precision complex value *a*, and returns the result as a single real value.
Formula	result =primary(a)
C/C++ prototype	float __crealf (float _Complex a);
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): __creal, __creall**
Purpose	Extracts the primary part of a double-precision complex value *a*, and returns the result as a single real value.
Formula	result =primary(a)
C/C++ prototype	double __creal (double _Complex a); long double __creall (long double _Complex a);[1]

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): __cimagf**
Purpose	Extracts the secondary part of a single-precision complex value *a*, and returns the result as a single real value.
Formula	result =secondary(a)
C/C++ prototype	float __cimagf (float _Complex a);
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): __cimag, __cimagl**
Purpose	Extracts the imaginary part of a double-precision complex value *a*, and returns the result as a single real value.
Formula	result =secondary(a)
C/C++ prototype	double __cimag (double _Complex a); long double __cimagl (long double _Complex a);[1]
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)

1. 128-bit C/C++ long double types are not supported on Blue Gene/L. Long doubles are treated as regular double-precision longs.

5.14 Load and store functions

Table 5-3 lists and explains the various parallel load and store functions that are available.

Table 5-3 Load and store functions

Function	**Parallel load (single-precision): __lfps**
Purpose	Loads parallel single-precision values from the address of *a*, and converts the results to double-precision. The first word in *address(a)* is loaded into the primary element of the return value. The next word, at location *address(a)*+4, 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): __lfxs**
Purpose	Loads single-precision values that have been converted to double-precision, from the address of *a*. The first word in *address(a)* is loaded into the secondary element of the return value. The next word, at location *address(a)*+4, 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: __lfpd**
Purpose	Loads parallel values from the address of *a*. The first word in *address(a)* is loaded into the primary element of the return value. The next word, at location *address(a)*+8, 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: __lfxd**
Purpose	Loads values from the address of *a*. The first word in *address(a)* is loaded into the secondary element of the return value. The next word, at location *address(a)*+8, 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 *address(b)*. The primary element of *a* is converted to single-precision and stored as the first word in *address(b).* The secondary element of *a* is converted to single-precision and stored as the next word at location *address(b)*+4.

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 *address(b)*. The secondary element of *a* is converted to single-precision and stored as the first word in *address(b).* The primary element of *a* is converted to single-precision and stored as the next word at location *address(b)*+4.

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 *address(b)*. The primary element of *a* is stored as the first double word in *address(b).* The secondary element of *a* is stored as the next double word at location *address(b)*+8.

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: __stfxd**
Purpose	Stores values into *address(b)*. The secondary element of *a* is stored as the first double word in *address(b).* The primary element of *a* is stored as the next double word at location *address(b)*+8.

Formula	b[0] = secondary(a)
	b[1] = primary(a)

C/C++ prototype	void __stfxd (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	Parallel store as integer: __stfpiw
Purpose	Stores in parallel floating-point double-precision values into *b* as integer words. The lower-order 32 bits of the primary element of *a* are stored as the first integer word in *address(b).* The lower-order 32 bits of the secondary element of *a* are stored as the next integer word at location *address(b)*+4. This function is typically preceded by a call to the `__fpctiw` or `__fpctiwz` built-in functions, described in 5.16.1, "Unary functions" on page 60, which perform parallel conversion of dual floating-point values to integers.

Formula	b[0] = primary(a)
	b[1] = secondary(a)

C/C++ prototype	void __stfpiw (int * b, double _Complex a);
Fortran description	STOREFP(B, A) where B is of type INTEGER(4) A is of type COMPLEX(8) result is none

5.15 Move functions

Table 5-4 lists and explains the parallel move functions that are available.

Table 5-4 Move functions

Function	Cross move: __fxmr
Purpose	Swaps the values of the primary and secondary elements of operand *a*.

Formula	primary(result) = secondary(a)
	secondary(result) = primary(a)

C/C++ prototype	double _Complex __fxmr (double _Complex a);
Fortran description	FXMR(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

5.16 Arithmetic functions

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

5.16.1 Unary functions

Unary functions operate on a single input operand. These functions are listed in Table 5-5.

Table 5-5 Unary functions

Function	**Parallel convert to integer: __fpctiw**
Purpose	Converts in parallel the primary and secondary elements of operand *a* to 32-bit integers using the current rounding mode. After a call to this function, use the `__stfpiw` function to store the converted integers in parallel, as explained in 5.14, “Load and store functions” on page 56.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpctiw (double _Complex a);
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: __fpctiwz**
Purpose	Converts in parallel the primary and secondary elements of operand *a* to 32 bit integers and rounds the results to zero. After a call to this function, use the `__stfpiw` function to store the converted integers in parallel, as explained in 5.14, “Load and store functions” on page 56.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpctiwz(double _Complex a);
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: __fprsp**
Purpose	Rounds in parallel the primary and secondary elements of double-precision operand *a* to single precision.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fprsp (double _Complex a);
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 *a*.

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 *a*.

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 *a*.

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 *a*.

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: __fpnabs**
Purpose	Calculates in parallel the negative absolute values of the primary and secondary elements of operand *a*.

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)

5.16.2 Binary functions

Binary functions operate on two input operands. The functions are listed in Table 5-6.

Table 5-6 Binary functions

Function	**Parallel add: __fpadd**
Purpose	Adds in parallel the primary and secondary elements of operands *a* and *b*.

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: __fpsub**
Purpose	Subtracts in parallel the primary and secondary elements of operand *b* from the corresponding primary and secondary elements of operand *a*.

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: __fpmul**
Purpose	Multiples in parallel the values of primary and secondary elements of operands *a* and *b*.

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 *a* and the primary element of *b* is stored as the primary element of the return value. The product of the primary element of *a* and the secondary element of *b* is stored as the secondary element of the return value.

Formula	primary(result) = secondary(a) x primary(b)
	secondary(result) = primary(a) × 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 *a* and the primary element of *b* is stored as the primary element of the return value. The product of *a* and the secondary element of *b* 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)

5.16.3 Multiply-add functions

Multiply-add functions take three input operands, multiply the first two, and add or subtract the third. Table 5-7 lists these functions.

Table 5-7 Multiply-add functions

Function	**Parallel multiply-add: __fpmadd**
Purpose	The sum of the product of the primary elements of *a* and *b*, added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of the secondary elements of *a* and *b,* added to the secondary element of *c,* 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 __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: __fpnmadd**
Purpose	The sum of the product of the primary elements of *a* and *b,* added to the primary element of *c,* is negated and stored as the primary element of the return value. The sum of the product of the secondary elements of *a* and *b,* added to the secondary element of *c,* 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 __fpnmadd (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 *c,* subtracted from the product of the primary elements of *a* and *b,* is stored as the primary element of the return value. The difference of the secondary element of *c,* subtracted from the product of the secondary elements of *a* and *b,* 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: __fpnmsub**
Purpose	The difference of the primary element of *c,* subtracted from the product of the primary elements of *a* and *b,* is negated and stored as the primary element of the return value. The difference of the secondary element of *c,* subtracted from the product of the secondary elements of *a* and *b,* 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 __fpnmsub (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: __fxmadd**
Purpose	The sum of the product of the primary element of *a* and the secondary element of *b,* added to the primary element of *c,* is stored as the primary element of the return value. The sum of the product of the secondary element of *a* and the primary *b,* added to the secondary element of *c,* 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: __fxnmadd**
Purpose	The sum of the product of the primary element of *a* and the secondary element of *b*, added to the primary element of *c,* is negated and stored as the primary element of the return value. The sum of the product of the secondary element of *a* and the primary element of *b,* added to the secondary element of *c,* 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: __fxmsub**
Purpose	The difference of the primary element of *c,* subtracted from the product of the primary element of *a* and the secondary element of *b*, is stored as the primary element of the return secondary element of *a* and the primary element of *b* 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: __fxnmsub**
Purpose	The difference of the primary element of *c,* subtracted from the product of the primary element of *a* and the secondary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c,* subtracted from the product of the secondary element of *a* and the primary element of *b*, 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: __fxcpmadd, __fxcsmadd**
Purpose	Both of these functions can be used to achieve the same result. The sum of the product of *a* and the primary element of *b,* added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of *a* and the secondary element of *b,* added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) + primary(c)
	secondary(result) = a x 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: __fxcpnmadd, __fxcsnmadd**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b,* is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) + primary(c))
	secondary(result) = -(a x 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: __fxcpmsub, __fxcsmsub**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b,* is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) - primary(c)
	secondary(result) = a x secondary(b) - 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: __fxcpnmsub, __fxcsnmsub**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b,* is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) - primary(c))
	secondary(result) = -(a x secondary(b) - 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: __fxcpnpma, __fxcsnpma**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b,* is negated and stored as the primary element of the return value. The sum of the product of *a* and the secondary element of *b,* added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) - primary(c))
	secondary(result) = a x secondary(b) + 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: __fxcpnsma, __fxcsnsma**
Purpose	Both of these functions can be used to achieve the same result. The sum of the product of *a* and the primary element of *b,* added to the primary element of *c*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) + primary(c))
	secondary(result) = -(a x secondary(b) - 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: __fxcxma**
Purpose	The sum of the product of *a* and the secondary element of *b,* added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of *a* and the primary element of *b,* added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = a x secondary(b) + primary(c)
	secondary(result) = a x primary(b) +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: __fxcxnms**
Purpose	The difference of the primary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the primary element of *b,* is negated and stored as the primary secondary of the return value.

Formula	primary(result) = -(a × secondary(b) - primary(c))
	secondary(result) = -(a × primary(b) - 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: __fxcxnpma
Purpose	The difference of the primary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is stored as the primary element of the return value. The sum of the product of *a* and the primary element of *b,* added to the secondary element of *c*, 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: __fxcxnsma**
Purpose	The sum of the product of *a* and the secondary element of *b,* added to the primary element of *c*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the primary element of *b,* is stored as the secondary element of the return value.

Formula	primary(result) = a x secondary(b) + primary(c))
	secondary(result) = -(a x 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)

5.17 Select functions

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

Table 5-8 Select functions

Function	Parallel select: __fpsel
Purpose	The value of the primary element of *a* is compared to zero. If its value is equal to or greater than zero, the primary element of *c* is stored in the primary element of the return value. Otherwise the primary element of *b* is stored in the primary element of the return value. The value of the secondary element of *a* is compared to zero. If its value is equal to or greater than zero, the secondary element of *c* is stored in the secondary element of the return value. Otherwise, the secondary element of *b* 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: __fpsel
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)

5.18 Examples of built-in functions usage

The built-in functions described in this section are contained in the builtins header file (builtins.h), which is contained in the compiler's include directory (for example, "/opt/ibmcmp/vacpp/8.0/include").

The following definitions 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 5-10 for C/C++ and Example 5-11 for Fortran.

Example 5-10 Using built-in functions in C/C++

```
double _Complex padd(double *x, double *y)
{
double _Complex a,b,c;
/* note possiblity 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 5-11 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 5-12 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
```

```
!------------------------------------------------------
! 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))
```

```
          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
```

6

Running and debugging applications

This chapter explains how to run and debug applications on Blue Gene/L.

6.1 Running applications

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

Note: Throughout this section, we use a generic Secure Shell (SSH) client to access the various Blue Gene/L nodes.

6.1.1 mmcs_db_console

It is possible to run applications directly from **`mmcs_db_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 **`mmcs_db_console`**, it is necessary to first manually select and allocate a block. At this point, it is possible to run Blue Gene/L applications. The set of commands in Figure 6-1 from the **`mmcs_db_console`** window shows how to accomplish this. The name of the system used is `beta18sn`, the block used is `BETA18`, and the name of the program executed is `calc_pi`.

```
beta18sn:/ # cd /bgl/BlueLight/ppcfloor/bglsys/bin
beta18sn:/bgl/BlueLight/ppcfloor/bglsys/bin # source db2profile
beta18sn:/bgl/BlueLight/ppcfloor/bglsys/bin # ./mmcs_db_console
connecting to mmcs server
set_username root
OK
connected to mmcs server
connected to DB2
mmcs$ free BETA18
OK
mmcs$ allocate BETA18
OK
mmcs$ submitjob BETA18 /bgl/home/garymu/a.out /bgl/home/garymu/out
OK
jobId=11019
mmcs$
```

Figure 6-1 Commands to manually select and allocate a block and run a program

For more information about using **`mmcs_db_console`**, see *Blue Gene/L: System Administration*, SG24-7178.

6.1.2 mpirun

In the absence of a scheduling application, we recommend that you use **`mpirun`** to run Blue Gene/L applications. Users can access this application from the Front End Node, which provides better security protection than using **`mmcs_db_console`**. For more complete information about using `mpirun`, see *Blue Gene/L: System Administration*, SG24-7178.

With **`mpirun`**, you can select and allocate a block and run a Message Passing Interface (MPI) application, all in one step. You can do this as shown in Example 6-1.

Example 6-1 Using mpirun

```
beta18sn:/bgl/BlueLight/ppcfloor/bglsys/bin # source db2profile
beta18sn:/bgl/BlueLight/ppcfloor/bglsys/bin # export
BRIDGE_CONFIG_FILE=/bgl/BlueLight/ppcdriver/bglsys/bin/bridge.config
beta18sn:/bgl/BlueLight/ppcfloor/bglsys/bin # ./mpirun -partition BETA18 \
                                              -exe /bgl/home/garymu/calc_pi \
                                              -cwd /bgl/home/garymu/out \
                                              -verbose 1

<Jun 26 02:38:34> FE_MPI (Info) : Initializing MPIRUN
<Jun 26 02:38:34> FE_MPI (Info) : Scheduler interface library loaded
<Jun 26 02:38:36> BRIDGE (Info) : The machine serial number (alias) is BGL
<Jun 26 02:38:36> FE_MPI (Info) : Back-End invoked:
<Jun 26 02:38:36> FE_MPI (Info) :   - Service Node: beta18sn.rchland.ibm.com
<Jun 26 02:38:36> FE_MPI (Info) :   - Back-End pid: 28879 (on service node)
<Jun 26 02:38:36> FE_MPI (Info) : Preparing partition
<Jun 26 02:38:37> BE_MPI (Info) : Examining specified partition
<Jun 26 02:38:37> BE_MPI (Info) : Checking partition BETA18 initial state ...
<Jun 26 02:38:37> BE_MPI (Info) : Partition BETA18 initial state = READY ('I')
<Jun 26 02:38:37> BE_MPI (Info) : Checking partition owner...
<Jun 26 02:38:37> BE_MPI (Info) : Checking if the partition is busy ...
<Jun 26 02:38:38> BE_MPI (Info) : Checking partition size ...
<Jun 26 02:38:38> BE_MPI (Info) : Partition owner matches the current user
<Jun 26 02:38:38> BE_MPI (Info) : Done preparing partition
<Jun 26 02:38:38> FE_MPI (Info) : Adding job
<Jun 26 02:38:39> FE_MPI (Info) : Job added with the following id: 760498
<Jun 26 02:38:39> FE_MPI (Info) : Starting job 760498
<Jun 26 02:38:39> FE_MPI (Info) : IO listener thread successfully started. Id=1099197216
<Jun 26 02:38:40> FE_MPI (Info) : Waiting for job to terminate
<Jun 26 02:38:44> FE_MPI (Info) : IO listener thread - Got connection request
<Jun 26 02:38:45> FE_MPI (Info) : IO listener thread - Threads initialized
***** Job Output Here *****
<Jun 26 02:38:51> BE_MPI (Info) : Job 760498 switched to state TERMINATED ('T')
<Jun 26 02:38:51> BE_MPI (Info) : Job successfully terminated
<Jun 26 02:38:53> FE_MPI (Info) : BG/L job exit status = (0)
<Jun 26 02:38:53> FE_MPI (Info) : Job terminated normally
<Jun 26 02:38:54> BE_MPI (Info) : Starting cleanup sequence
<Jun 26 02:38:54> BE_MPI (Info) : BG/L Job alredy terminated / hasn't been added
<Jun 26 02:38:55> BE_MPI (Info) : Partition was supplied with READY ('I') initial state
<Jun 26 02:38:55> BE_MPI (Info) : No need to destroy the partition
<Jun 26 02:38:57> BE_MPI (Info) : ==   BE completed   ==
<Jun 26 02:38:57> FE_MPI (Info) : ==   FE completed   ==
<Jun 26 02:38:57> FE_MPI (Info) : == Exit status:   0 ==
```

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:

```
>/bgl/home/garymu/out/calc_pi.stdout 2>/bgl/home/garymu/out/calc_pi.stderr
```

This line sends standard output to a file called calc_pi.stdout and standard error to a file called calc_pi.stderr. Both files are in the /bgl/home/garymu/out directory.

6.1.3 LoadLeveler

At present, LoadLeveler support for Blue Gene/L is provided via a PRPQ. LoadLeveler is an IBM product that 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 might be of several types: desktop workstations available for batch jobs (usually when not in use by their owner), dedicated servers, and parallel machines.

Jobs are allocated to machines in the cluster by a scheduler. The allocation of the 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. At the same time, it is the job of LoadLeveler to maximize the efficiency of the cluster. It attempts to do this by maximizing the utilization of resources, while at the same time minimizing the job turnaround time experienced by users.

Some of the tasks that LoadLeveler can perform include:

- Choosing the next job to run
- Examining the job requirements
- Collecting available resources in its cluster
- Choosing the "best" machines for the job
- Dispatching the job to the selected machine
- Controlling running jobs

For more information about LoadLeveler support, see *Blue Gene/L: System Administration*, SG24-7178.

6.1.4 Other scheduler products

You can use custom scheduling applications to run applications on Blue Gene/L. You write custom "glue" code between the scheduler and Blue Gene/L using the Bridge application programming interfaces (APIs), which are described in Chapter 9, "Control system (Bridge) APIs" on page 93.

6.2 Debugging applications

This section discusses the debuggers that are supported by Blue Gene/L.

6.2.1 General debugging architecture

Four major pieces of code are involved when debugging applications on Blue Gene/L:

- The Compute Node Kernel

  This provides the low-level primitives needed to debug an application.
- The Control and I/O Daemon (CIOD) running on the I/O Nodes

  This provides control and communications to compute nodes.
- A "debug server" running on the I/O Nodes

  This is vendor-supplied code that interfaces with the CIOD.
- A debug client running on a Front End Node

  This is where the user will actually do their work interactively.

A debugger wanting to debug an application running on a Compute Node must interface to the Compute Node through an API implemented in CIOD. This debug code is started on the I/O Nodes by the control system and might 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 withBlue Gene/L. Many possible debug servers are possible.

A *debug client* is a piece of code running 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.

6.2.2 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 Blue Gene/L that allows the GDB to work with applications running on Compute Nodes. IBM provides a simple debug server called "gdbserver.440." Each running instance of GDB is associated with one, and only one, Compute Node. If you must debug an MPI application running on multiple compute nodes, and you need to (for example) view variables associated with more than one instance of the application, you run multiple instances of GDB. This is different from the more sophisticated support offered by the TotalView debugger (see 6.2.3, "TotalView" on page 82), which makes it possible for a single debug instance to control multiple Compute Nodes simultaneously.

Most people use GDB to debug local processes running on the same machine that they are running GDB on. GDB also has the ability to do remote debug via a GDB server on the remote machine. GDB on Blue Gene/L 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.440 implements the minimum number of primitives required by the GDB remote protocol specification. As is 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 need to run multiple GDB clients at the same time. Although you might need multiple GDB clients for multiple compute nodes, one gdbserver.440 on each I/O Node is all that is required. The Blue Gene/L control system manages that part.
- IBM does not ship a GDB client with Blue Gene/L. The user might use an existing GDB client to connect to the IBM supplied gdbserver.440. 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) 9 for PowerPC are known to work.
- To debug an application, the debug server must be started and running before you attempt to debug. An option on the `mpirun` command allows you to get the debug server running before your application does. If you do not use this option and you need to debug your application, you do not have a mechanism to start the debug server and thus have no way to debug your application.

- Gdbserver.440 is not aware of user specified MPI topologies. You are still able to debug your application, but the connection information given to you by `mpirun` for each MPI rank might be incorrect.

Prerequisite software

The GDB should have been installed during the installation procedure. You can verify the installation by seeing if the /bgl/BlueLight/ppcfloor/blrts-gnu/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 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 C that you want to debug, compile the application using an invocation similar to the following example:

```
/opt/ibmcmp/vac/bg/8.0/bin/blrts_xlc -g -O0 -qarch=440d -qtune=440 ...
```

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 Blue Gene/L, see Chapter 5, “Developing applications with IBM XL compilers” on page 43.

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 MyMPI.rts, and the source code file is MyMPI.c. We use a partition (block) called BETA18.

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 but it does not start running immediately.
2. The control system starts the specified debug server (gdbserver.440) on all of the I/O Nodes in the partition running your job.
3. The `mpirun` command pauses and gives 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 `mpirun`, and then the application starts running on the compute nodes.

The pause in step 3 allows you to connect GDB clients to compute nodes before the application runs, which is desirable if you need to 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.

1. Open two separate console shells.
2. Go to the first shell window.
   a. Change (**`cd`**) to the directory that contains your program executable.
   b. Start your application (in this case, MyMPI.rts) using **`mpirun`** with a command similar to the following example:

      ```
      /bgl/BlueLight/ppcfloor/bglsys/bin/mpirun -partition BETA18 -exe
      /bgl/home/garymu/MyMPI.rts -cwd /bgl/home/garymu/out/ -start_gdbserver
      /bgl/BlueLight/ppcfloor/ppc/dist/sbin/gdbserver.440 -verbose 1
      ```

   c. You should see messages in the console, like those shown in Example 6-2.

Example 6-2 Messages in the console

```
<Jun 26 02:59:18> FE_MPI (Info) : Initializing MPIRUN
<Jun 26 02:59:18> FE_MPI (Info) : Scheduler interface library loaded
<Jun 26 02:59:20> BRIDGE (Info) : The machine serial number (alias) is BGL
<Jun 26 02:59:20> FE_MPI (Info) : Back-End invoked:
<Jun 26 02:59:20> FE_MPI (Info) :   - Service Node: beta18sn.rchland.ibm.com
<Jun 26 02:59:20> FE_MPI (Info) :   - Back-End pid: 30502 (on service node)
<Jun 26 02:59:20> FE_MPI (Info) : Preparing partition
<Jun 26 02:59:21> BE_MPI (Info) : Examining specified partition
<Jun 26 02:59:21> BE_MPI (Info) : Checking partition BETA18 initial state ...
<Jun 26 02:59:21> BE_MPI (Info) : Partition BETA18 initial state = READY ('I')
<Jun 26 02:59:21> BE_MPI (Info) : Checking partition owner...
<Jun 26 02:59:21> BE_MPI (Info) : Checking if the partition is busy ...
<Jun 26 02:59:21> BE_MPI (Info) : Checking partition size ...
<Jun 26 02:59:21> BE_MPI (Info) : Partition owner matches the current user
<Jun 26 02:59:21> BE_MPI (Info) : Done preparing partition
<Jun 26 02:59:22> FE_MPI (Info) : Adding job
<Jun 26 02:59:23> FE_MPI (Info) : Job added with the following id: 760506
<Jun 26 02:59:23> FE_MPI (Info) : Loading BG/L job
<Jun 26 02:59:23> BE_MPI (Info) : Loading BG/L job 760506 ...
<Jun 26 02:59:23> BE_MPI (Info) : Job load command successful
<Jun 26 02:59:23> BE_MPI (Info) : Waiting for BG/L job (760506) to get to Loaded/Running
state ...
<Jun 26 02:59:33> BE_MPI (Info) : Job 760506 switched to state LOADED
<Jun 26 02:59:38> BE_MPI (Info) : Job loaded successfully
<Jun 26 02:59:39> FE_MPI (Info) : Starting debugger setup for job 760506
<Jun 26 02:59:39> FE_MPI (Info) : Setting debug info in the block record
<Jun 26 02:59:39> BE_MPI (Info) : Set debugger executable and arguments in block
description
<Jun 26 02:59:39> BE_MPI (Info) : Data set successfully
<Jun 26 02:59:40> FE_MPI (Info) : Query job 760506 to find MPI ranks for compute nodes
<Jun 26 02:59:40> FE_MPI (Info) : Getting proctable for the debugger
<Jun 26 02:59:41> BE_MPI (Info) : Query job completed - proctable is filled in
<Jun 26 02:59:41> FE_MPI (Info) : Starting debugger servers on I/O nodes for job 760506
<Jun 26 02:59:41> FE_MPI (Info) : Attaching debugger to the BG/L job
<Jun 26 02:59:42> BE_MPI (Info) : Debugger servers are now spawning
<Jun 26 02:59:42> 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 it's MPI rank and press [Enter].  To see
all of the compute nodes, type 'dump_proctable'.
>
```

d. Find the IP address and port of the Compute Node that you want to debug. You can do this in one of two ways. Either enter the rank of the program instance that you want to debug and press Enter, or dump the address or port of each node by typing `dump_proctable` and press Enter.

```
> 2
MPI Rank 2: Connect to 172.30.255.85:7302
> 4
MPI Rank 4: Connect to 172.30.255.85:7304
>
<Jun 26 03:01:07> FE_MPI (Info) : Debug setup is complete
<Jun 26 03:01:07> FE_MPI (Info) : Waiting for BG/L job to get to Loaded state
<Jun 26 03:01:08> BE_MPI (Info) : Waiting for BG/L job (760506) to get to
Loaded/Running state ...
<Jun 26 03:01:13> BE_MPI (Info) : Job loaded successfully
<Jun 26 03:01:13> FE_MPI (Info) : Beginning job 760506
<Jun 26 03:01:14> BE_MPI (Info) : Beginning BG/L job 760506 ...
<Jun 26 03:01:14> BE_MPI (Info) : Job begin command successful
<Jun 26 03:01:14> FE_MPI (Info) : Waiting for job to terminate
```

3. From the second shell, follow these steps:

   a. Change (**`cd`**) to the directory that contains your program executable.

   b. Type the following command, using the name of your own executable instead of MyMPI.rts:

   ```
   /bgl/BlueLight/ppcfloor/blrts-gnu/bin/gdb MyMPI.rts
   ```

   c. Enter the following command, using the address of the compute node that you want to debug and determined in step d:

   ```
   target remote ipaddr:port
   ```

4. You are now debugging the specified application on the configured compute node. 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.

6.2.3 TotalView

TotalView is a debugger product sold by Etnus, LLC. It is a completely separate product from Blue Gene/L. For sales and support information, refer to the following Web address:

http://www.etnus.com

7

Checkpoint and restart support

This chapter provides details about the checkpoint and restart support provided by Blue Gene/L.

7.1 Why use checkpoint and restart

Given the scale of the Blue Gene/L system, faults are expected to be the norm rather than the exception. This is unfortunately inevitable, given the vast number of individual hardware processors and other components involved in running the system.

Checkpoint and restart are among the primary techniques for fault recovery. A special user-level checkpoint library has been developed for Blue Gene/L applications. Using this library, application programs can take a checkpoint of their program state at appropriate stages and can be restarted later from their last successful checkpoint.

Why should you be interested in this support? Among the following examples, numerous scenarios indicate that use of this support is warranted:

- Your application is a long-running one. You do not want it to fail a long time into a run, losing all the calculations made up until the failure. Checkpoint and restart allow you to restart the application at the last checkpoint position, losing a much smaller slice of processing time.
- You are given access to a Blue Gene/L system for relatively small increments of time, and you know that your application run will take longer than your allotted amount of processing time. Checkpoint and restart allows you to execute your application to completion in distinct “chunks,” rather than in one continuous period of time.

These are just two of many reasons to use checkpoint and restart support in your Blue Gene/L applications.

7.2 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 `BGLCheckpointInit()` function at the beginning of the program, to initialize the checkpoint related data structures, and carry out an automated restart when needed. The application can then make calls to the `BGLCheckpoint()` function to store a snapshot of the program state in stable storage (files on a disk). The current model assumes that, when an application needs to take a checkpoint, all of the following points are true:

- All the processes of the application will make a call to `BGLCheckpoint()`.
- When a process makes a call to `BGLCheckpoint()`, there are no outstanding messages in the network or buffers; that is the `recv` corresponding to all the send calls have taken place.
- After a process has made a call to `BGLCheckpoint()`, other processes do not send messages to the process until their checkpoint is complete. Typically, applications are expected to place calls to `BGLCheckpoint()` 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 MPI buffers and the network.

`BGLCheckpoint()` might 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 corresponding 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 pertaining 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 BGL_CHKPT_RESTART_SEQNO and BGL_CHKPT_DIR_PATH set to the appropriate values. The `BGLCheckpointInit()` function checks for these environment variables and, if specified, restarts the application from the desired checkpoint.

7.2.1 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 five main file I/O functions: `open`, `close`, `read`, `write`, and `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 and 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.

7.2.2 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 (sighandler) registered for each signal (signum). 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 being checkpointed. This might make the checkpoint inconsistent. Therefore, the signals arriving while a checkpoint is under progress need to 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 7-1. 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 7-1 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

Signal name	Signal type	Action to be taken
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 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.

7.3 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

The following sections describe each of these components in detail.

7.3.1 Checkpoint library API

To ensure minimal overhead, the basic interface has been kept fairly simple. Ideally, a programmer needs to 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 BGL_CHKPT_RESTART_SEQNO 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 rest of this section describes the checkpoint API in detail.

void BGLCheckpointInit(char * ckptDirPath)

`BGLCheckpointInit` is a mandatory function that must be invoked at the beginning of the program. This function initializes the data structures of the checkpoint library. In addition, this function is used 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 7.4, "Directory and file naming conventions" on page 89.

int BGLCheckpoint()

`BGLCheckpoint` takes a snapshot of the program state at the instant it is called. All the processes of the application must make a call to `BGLCheckpoint` to take a consistent global checkpoint.

When a process makes a call to `BGLCheckpoint`, there should be no outstanding messages in the network or buffers. That is, the `recv` corresponding to all the send calls should have taken place. And after a process has made a call to `BGLCheckpoint`, other processes must not send messages to the process until their call to `BGLCheckpoint` is complete. Typically, applications are expected to place calls to `BGLCheckpoint` 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 `BGLCheckpointInit`. If `ckptDirPath` is NULL, then the checkpoint file location is decided by the storage rules mentioned in 7.4, "Directory and file naming conventions" on page 89.

void BGLCheckpointRestart(int restartSqNo)

`BGLCheckpointRestart` 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 `BGLCheckpointInit`. If `ckptDirPath` is NULL, then the checkpoint file location is decided by the storage rules given in 7.4, "Directory and file naming conventions" on page 89.

An application developer does not need to explicitly invoke this function. `BGLCheckpointInit` automatically invokes this function whenever an application is re-started. The environment variable BGL_CHKPT_RESTART_SEQNO is set to an appropriate value. If the restartSqNo, the environment variable BGL_CHKPT_RESTART_SEQNO, 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 BGLCheckpointExcludeRegion(void *addr, size_t len)

`BGLCheckpointExcludeRegion` marks the specified region (addr to addr + len - 1) to be excluded from the program state, while a checkpoint is being taken. The state corresponding to this region is not saved in the checkpoint file. Therefore, after restart the corresponding memory region in the application is not overwritten. This facility can be used to protect critical data that should not be restored at the time of restart such as BGLPersonality and checkpoint data structures. This call can also be used by the application programmer to exclude a scratch data structure that does not need to be saved at checkpoint time.

int BGLAtCheckpoint((void *) function(void *arg), void *arg)

`BGLAtCheckpoint` registers the functions to be called just before taking the checkpoint. This can be used by the user to take some action at the time of checkpoint. For example, this can be called to close all the communication state 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 being called.

int BGLAtRestart((void *) function (void *arg), void *arg)

`BGLAtRestart` 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 registered are called in the reverse order of their registration. This can be used to resume or reinitialize functions or data structures at the time of restart. For example, in the coprocessor mode, the coprocessor needs to be reinitialized at the time of restart. The argument `arg` is passed to the function that is being called.

int BGLAtContinue((void *) function (void *arg), void *arg)

`BGLAtContinue` registers the functions to be called when continuing after a checkpoint. This can be used to reinitialize or resume some functions or data structures which were closed or stopped 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.

7.4 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 `BGLCheckpointInit()`. If ckptDirPath is not specified (or is null), the directory is picked from the environment variable BGL_CHKPT_DIR_PATH. This environment variable might 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, Blue Gene/L 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 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.

7.5 Restart

A transparent restart mechanism is provided through the use of the `BGLCheckpointInit()` function and the BGL_CHKPT_RESTART_SEQNO environment variable. Upon startup, an application is expected to make a call to `BGLCheckpointInit()`. The `BGLCheckpointInit()` function initializes the checkpoint library data structures.

Moreover the `BGLCheckpointInit()` function checks for the environment variable BGL_CHKPT_RESTART_SEQNO. 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.

7.5.1 Determining latest consistent global checkpoint

Mere 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 might crash, while some others might complete. This might create consistent checkpoint files for some of the processes and inconsistent or non-existent checkpoint files for some 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 BGL_CHKPT_RESTART_SEQNO corresponds to a consistent global checkpoint. In case BGL_CHKPT_RESTART_SEQNO 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 BGL_CHKPT_RESTART_SEQNO does not correspond to a global consistent checkpoint.

7.5.2 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 BGL_CHKPT_ENABLED. The checkpoint functionality is invoked only if this environment variable is set to a value of “1.” Table 7-2 summarizes the checkpoint-related function calls.

Table 7-2 Checkpoint and restart APIs

Function name	Usage
BGLCheckpointInit(char *ckptDirPath)	Sets the checkpoint directory to ckptDirPath. Initializes the checkpoint library data structures. Carries out restart if environment variable BGL_CHKPT_RESTART_SEQNO is set.
BGLCheckpoint()	Takes a checkpoint. Stores the program state in the checkpoint directory.
BGLCheckpointRestart(int rstartSqNo)	Carries out an explicit restart from the specified sequence number.
BGLCheckpointExcludeRegion (void *addr, size_t len)	Excludes the specified region from the checkpoint state.

Table 7-3 summarizes the environment variables.

Table 7-3 Checkpoint and restart environment variables

Environment variables	Usage
BGL_CHKPT_ENABLED	Set (to 1) if checkpoints desired, else not specified.
BGL_CHKPT_DIR_PATH	Default path to keep checkpoint files.
BGL_CHKPT_RESTART_SEQNO	Set to a desired checkpoint sequence number from where 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:

- BGL_CHKPT_ENABLED=1
- BGL_CHKPT_DIR_PATH= checkpoint directory
- BGL_CHKPT_RESTART_SEQNO=0

A combination of BGL_CHKPT_ENABLED and BGL_CHKPT_RESTART_SEQNO (as in Table 7-3) 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 BGL_CHKPT_ENABLED variable.

8

One-sided communications

This chapter provides information about the new support for one-sided communications on Blue Gene/L. Message Passing Interface (MPI) is by design a two-sided communications paradigm. By this, we mean that, for every “send” operation that is performed, a corresponding “receive” must be explicitly coded. It is impossible with MPI for a node to unilaterally get or put data from or to the memory of another node. One-sided communications makes this possible. However, it is possible to mix MPI and one-sided communications calls together in a single application.

There are two main ways to achieve one-sided communications on Blue Gene/L:

- Aggregate Remote Memory Copy Interface (ARMCI)
- Global Arrays (GA)

This chapter explains both of these methods and provides references to more detailed information.

8.1 Aggregate Remote Memory Copy Interface

The creator of ARMCI is the Pacific Northwest National Laboratory (PNNL). ARMCI is a popular library for one-sided communications. On Blue Gene/L, ARMCI serves as the underpinnings for the Global Array support described in “Global Arrays” on page 92.

Writing applications for Blue Gene/L that take advantage of ARMCI support is done exactly as with any other platform. Because of this fact, we do not delve into the specifics of how to write ARMCI applications. You can find further information, such as programming application programming interfaces (APIs) for ARMCI, on the Web at the following address:

`http://www.emsl.pnl.gov/docs/parsoft/armci/index.html`

In addition, the code that provides ARMCI support on Blue Gene/L will not be made available from IBM, but rather from the PNNL Web site. Users must download the source code from PNNL, build it (using the makefiles provided), and install it on their Service Node such that it is available to applications and it has access to the Blue Gene/L control code.

8.2 Global Arrays

Global Arrays are also a creation of PNNL. The implementation for Global Arrays was built on top of the ARMCI support.

You can find further information about Global Arrays on the Web at the following address:

`http://www.emsl.pnl.gov/docs/global/`

As with the ARMCI support, users must download the actual source code from PNNL and built is as described in the previous section.

8.3 BGLMPI_INTERRUPT environment variable

The new BGLMPI_INTERRUPT environment variable, described in “BGLMPI_INTERRUPT” on page 153, has no impact on one-sided communications. Even if an attempt is made to disable interrupts using this environment variable, they continue to be used if the application is written to use either ARMCI or GA.

Control system (Bridge) APIs

This chapter defines 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 is an example of an application that uses these APIs to manage partitions, jobs, and other similar aspects of the Blue Gene/L system. You can use these APIs to control Blue Gene/L job execution, as well as other similar administrative tasks, using any application that you choose.

9.1 API support overview

The following sections provide an overview of the support provided by the APIs.

9.1.1 Requirements

There are several requirements for writing programs to the Bridge APIs as explained in the following sections.

Operating system supported

Currently, SUSE Linux Enterprise Server (SLES) 9 for PowerPC is the only supported platform.

Configuring environment variables

Two files need to be created in order to set the necessary environment variables. These files tell the Bridge APIs the name of the database, the user ID and password to use to access the database, and other such important information.

This information is documented in the chapters about `mpirun` in the *Blue Gene/L: System Administration*, SG24-7178.

Languages supported

C and C++ are supported with the GNU gcc 3.4.3 level compilers. For more information and downloads, refer to the following Web address:

`http://gcc.gnu.org/`

Library files

Multiple system files are available for support of the Bridge APIs. The ones that you use depend on whether you are writing 32-bit or 64-bit code.

Important: You also need to link in the XML parser library (“expat”) and the DB2 CLI library (“db2”).

32-bit libraries

For 32-bit programs, there is one include file and multiple static link files:

- include/rm_api.h
- lib/bglbridge.a
- lib/libtableapi.a
- lib/libbgldb.a
- lib/libbglmachine.a
- lib/libsaymessage.a

64-bit libraries

For 64-bit programs, more options are available. The include file remains the same, include/rm_api.h.

However, for linking, there are two options: static and dynamic (shared). For static linking of 64-bit code, use the following files:

- lib64/libbglbridge_s.a
- lib64/libtableapi_s.a
- lib64/libbglmachine_s.a

- lib64/libbgldb_s.a
- lib64/libsaymessage_s.a

For dynamic linking of 64-bit code, link with these files:

- lib64/libbglbridge.so
- lib64/libtableapi.so
- lib64/libbglmachine.so
- lib64/libbgldb.so
- lib64/libsaymessage.so

These files should be available with the standard system installation procedure. They are contained in the bglcmcs.rpm file.

Makefile excerpt

Example 9-1 shows a possible excerpt from a makefile that you might want to create to help automate builds of your Blue Gene/L application. This excerpt allows the user to pass in the number of bits (32 or 64) to be used on the command line.

Example 9-1 Makefile excerpt to automate 32-bit and 64-bit development

```
...

CLINKER     = $(CXX)

ifndef BITS
  BITS=32
endif

LDFLAGS_32 = -L$(BGL_INSTALL)/bglsys/lib -lbglbridge -lbgldb -ltableapi -lbglmachine
-lexpat -lsaymessage -L/u/bgdb2cli/sqllib/lib -ldb2 -lpthread
LDFLAGS_64 = -L$(BGL_INSTALL)/bglsys/lib64 -lbglbridge_s -lbgldb_s -ltableapi_s
-lbglmachine_s -lexpat -lsaymessage_s -L/u/bgdb2cli/sqllib/lib64 -ldb2 -lpthread -m64
LDFLAGS_64so = -L$(BGL_INSTALL)/bglsys/lib64 -lbglbridge -lbgldb -ltableapi -lbglmachine
-lexpat -lsaymessage -L/u/bgdb2cli/sqllib/lib64 -ldb2 -lpthread -m64

LDFLAGS = ${LDFLAGS_${BITS}}

CXX_FLAGS_32 =
CXX_FLAGS_64 = -m64
CXX_FLAGS_64so = -m64

CXX_FLAGS = ${CXX_FLAGS_${BITS}}

...
```

9.1.2 General comments

All of the APIs that are used have some general considerations that apply to all calls. The following list highlights some of those common features.

- All the API calls return a “status_t” indicating either a 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.

- For adding information to the MMCS, use “new” functions as well as `rm_set_data()`. The “new” functions allocate memory for new data structures and the `rm_set_data()` 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_BGL(BGL **bgl)` is complemented by `rm_free_BGL(BGL *bgl)`.
- It is the responsibility of the caller to match the calls to the “get” and “new” allocators and to the corresponding “free” de-allocators. Not doing this results in memory leaks.

9.1.3 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 might cause the process to run out of memory. For a complete list of the APIs that require calls to free, see Table 9-1.

Avoiding invalid pointers

Some APIs return a pointer to an offset in a data structure (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` spec. For example, you might have “element” as 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.

The code snippet in Example 9-2 shows correct usage of the first and next API calls. Notice how memory is freed after the list is consumed.

Example 9-2 Correct usage of first and next API calls

```
status_t stat;
int list_size = 0;
rm_partition_list_t * bgl_part_list = NULL;
rm_element_t * bgl_part = NULL;

// Get all information on existing partitions
stat = rm_get_partitions_info(PARTITION_ALL_FLAG, &bgl_part_list);
if (stat != STATUS_OK) {
   // Do some error handling here...
   return;
}

// How much data (# of partitions) did we get back?
rm_get_data(bgl_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(bgl_part_list, RM_PartListFirstPart, bgl_part);
   }
   // Otherwise, use RM_PartListNextPart
   else {
      rm_get_data(bgl_part_list, RM_PartListNextPart, bgl_part);
   }
}

// Make sure we free the memory when finished
stat = rm_free_partition_list(bgl_part_list);
if (stat != STATUS_OK) {
   // Do some error handling here...
   return;
}
```

9.2 APIs

The following sections describe details about the APIs.

9.2.1 API to the MMCS Resource Manager

The Resource Manager API contains an `rm_get_BGL` function to retrieve updated configuration and status information about all the physical components of Blue Gene/L from the MMCS database. The Resource Manager API also includes a set of functions that add, remove, or modify information about transient entities, such as jobs and partitions. These functions do not impose side-effects on the actual machine.

The `rm_get_BGL` function supplies all the required information to allow partition allocation. 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 general structure called "BGL." 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 c-nodes.

There are additional "get" functions to retrieve information about the partitions and jobs entities. All the data retrieved using the "get" functions can be accessed using `rm_get_data()` with one of the specifications listed in Table 9-1.

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()`. The specifications that can be set using the `rm_set_data()` function are marked with an asterisk (*) in Table 9-1.

Important: Some specifications might be marked as "deprecated." A deprecated specification might be removed in future versions of Blue Gene.

Table 9-1 Specification for rm_get_data/rm_set_data function

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
BGL machine		RM_BPsize	rm_size3D_t *		The size of a base partition (in c-nodes) in each dimension.
BGL machine		RM_Msize	rm_size3D_t *		The size of the machine in base partition units.

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
BGL machine		RM_BPNum	int *		The number of base partitions in the machine.
BGL machine		RM_SwitchNum	int *		The number of switches in the machine.
BGL machine		RM_WireNum	int *		The number of wires in the machine.
BGL machine		RM_FirstBP	rm_element_t *		A pointer to the element associated with the first base partition in the list.
BGL machine		RM_NextBP	rm_element_t *		A pointer to the element associated with the next base partition in the list.
BGL machine		RM_FirstSwitch	rm_element_t *		The pointer to the element associated with the first switch in the list.
BGL machine		RM_NextSwitch	rm_element_t *		A pointer to the element associated with the next switch in the list.
BGL machine		RM_FirstWire	rm_element_t *		A pointer to the element associated with the first wire in the list.
BGL machine		RM_NextWire	rm_element_t *		A pointer to the element associated with the next wire in the list.
Base partition	*	RM_BPID	rm_bp_id_t *	*	A pointer to the identifier of base partition.
Base partition		RM_BPState	rm_BP_state_t *		A pointer to an enum value indicating the state of the base partition.
Base partition	*	RM_BPLoc	rm_location_t *		A pointer to a structure with the location of the base partition in the 3D machine.
Base partition		RM_BPPartID	pm_partition_id_t *	*	A pointer to the identifier of the partition with which the base partition is associated. If no partition is associated, null is returned.
Base partition		RM_BPPart State	rm_partition_state_t *		A pointer to an enum value indicating the state of the partition. (To learn more about partition states, see Figure 9-2 on page 111.)

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
Base partition		RM_BPSDB	boolean *		A flag indicating whether this base partition is being used by a small partition (smaller than a base partition).
Base partition		RM_BPSD	boolean*		A flag indicating whether this base partition is being divided into a small (free) partition.
Base partition		RM_BPCompute NodeMemory	rm_BP_computenode _memory_t *		A pointer to an enum value indicating the compute node memory size for the base partition.
NodeCard	*	RM_Node CardID	rm_nodecard_id_t*	*	A pointer to an identifier of node card.
NodeCard		RM_NodeCard Quarter	rm_quarter_t*		A pointer to an enum value indicating the quarter in the Base Partition that this node card is part of.
NodeCard		RM_Node CardState	rm_nodecard_state_t		A pointer to an enum value indicating the state of the node card.
NodeCard		RM_Node CardIONodes	int*		The number of IO nodes on the node card (can be 0, 2, or 4).
NodeCard		RM_Node CardPartID	pm_partition_id_t*	*	A pointer to the ID of the partition with which the node card is associated. If no partition is associated, Null is returned.
NodeCard		RM_Node CardPartState	rm_partition_state_t*		A pointer to an enum value indicating the state of the partition. (For the states of partition, see Figure 9-2 on page 111.)
NodeCard List		RM_Node CardListSize	int*		The number of node cards in the list.
NodeCard List		RM_Node CardListFirst	rm_element_t*		A pointer to the first node card in the retrieved list.
NodeCard List		RM_Node CardListNext	rm_element_t*		A pointer to the next node card in the retrieved list.
Switch	*	RM_SwitchID	rm_switch_id_t *	*	A pointer to the identifier of the switch.
Switch		RM_SwitchBPID	rm_bp_id_t *	*	A pointer to the identifier of the base partition that is connected to that switch.

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
Switch		RM_SwitchState	rm_switch_state_t *		A pointer to an enum value indicating the state of the switch.
Switch		RM_SwitchDim	rm_dimension_t *		A pointer to an enum representing one of these values: RM_DIM_X, RM_DIM_Y, RM_DIM_Z.
Switch	*	RM_Switch ConnNum	int *		The number of connections in the switch.
Switch	*	RM_SwitchFirst Connection	rm_connection_t *		A pointer to the first connection in the switch connection list. A connection is a pair of ports connected internally in the switch.
Switch	*	RM_SwitchNext Connection	rm_connection_t *		A pointer to the element associated with the next connection in the list.
Wire		RM_WireID	rm_wire_id_t *	*	A pointer to the identifier of the wire.
Wire		RM_WireState	rm_wire_state_t *		A pointer to an enum value indicating the state of the wire. The state can be UP or DOWN.
Wire		RM_WireFrom Port	rm_element_t *		A pointer to an element associated with the wire source port.
Wire		RM_WireToPort	rm_element_t *		A pointer to an element associated with the wire destination port.
Wire		RM_WirePartID	pm_partition_id_t *	*	A pointer to the ID of the partition with which the wire is associated. If no partition is associated, null is returned.
Wire		RM_WirePart State	rm_partition_state_t *		A pointer to an enum value indicating the state of the partition. (To learn more about partition states, see Figure 9-2 on page 111.)
Port		RM_Port ComponentID	rm_component_id_t *	*	A pointer to the ID that identifies the base partition or the switch the port is part of.

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
Port		RM_PortID	rm_port_id_t *		A pointer to an enum value indicating the port ID. The port ID can be one of these enum values: plus_x minus_x, plus_y, minus_y, plus_z minus_z for base partitions and s0..S5 for switches.
Partition list		RM_PartList Size	int *		The number of partitions in the list.
Partition list		RM_PartList FirstPart	rm_element_t *		A pointer to the first partition in the retrieved list.
Partition list		RM_PartList NextPart	rm_element_t *		A pointer to the next partition in the retrieved list.
Partition		RM_PartitionID	pm_partition_id_t *	*	A pointer to the ID of the partition.
Partition		RM_Partition State	rm_partition_state_t *		A pointer to an enum value indicating the state of the partition (to learn more about partition states, see Figure 9-2 on page 111).
Partition	*	RM_Partition BPNum	int *		The number of base partitions in the partition.
Partition	*	RM_Partition SwitchNum	int *		The number of switches in the partition.
Partition	*	RM_Partition NodeCardNum	int *		The number of node cards in the partition.
Partition	*	RM_Partition FirstNodecard	rm_nodecard_t *		A pointer to the first node card in the partition.
Partition	*	RM_Partition NextNodecard	rm_nodecard_t *		A pointer to the next node card in the partition.
Partition	*	RM_Partition FirstBP	rm_element_t *		A pointer to the element associated with the first base partition in the list.
Partition	*	RM_Partition NextBP	rm_element_t *		A pointer to the element associated with the next base partition in the list.
Partition	*	RM_Partitiont FirstSwitch	rm_element_t *		A pointer to the element associated with the first switch in the list.
Partition	*	RM_Partitiont NextSwitch	rm_element_t *		A pointer to the element associated with the next switch in the list.

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
Partition	*	RM_Partition Connection	rm_connection_ type_t *		The connection type of the partition. Can be TORUS or MESH.
Partition	*	RM_Partition UserName	char *	*	A pointer to a string containing the user name of the user who submitted the job.
Partition	*	RM_Partition MloaderImg	char *	*	A pointer to a string containing the file name of the machine loader image.
Partition	*	RM_Partition BlrtsImg	char *	*	A pointer to a string containing the file name of the compute node's kernel image.
Partition	*	RM_Partition LinuxImg	char *	*	A pointer to a string containing the file name of the I/O Node's Linux image.
Partition	*	RM_Partition RamdiskImg	char *	*	A pointer to a string containing the file name of the ramdisk image.
Partition	*	RM_Partition Description	char *	*	A pointer to a string containing a description of the partition.
Partition	*	RM_Partition Small	boolean *		A flag indicating whether this partition is a partition smaller than the base partition.
Partition	*	RM_Partition PsetsPerBP	int *		The number of used processor sets (psets) per base partition.
Partition		RM_Partition UsersNum	int *		The number of users of the partition.
Partition		RM_Partition FirstUser	char *	*	A pointer to the partition's first user name.
Partition		RM_Partition NextUser	char *	*	A pointer to the partition's next user name.
Partition		RM_Partition Options	char *	*	A pointer to a string containing the kernel debug option.
Job list		RM_JobListSize	int *		The size of the job list.
Job list		RM_JobListFirst Job	rm_element_t *		A pointer to the first job in the retrieved list.
Job list		RM_JobListNext Job	rm_element_t *		A pointer to the next job in the retrieved list.

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
Job	*	RM_JobID	rm_job_id_t *	*	A pointer to the job ID. This must be unique across all jobs on the system; if not, return code JOB_ALREADY_DEFINED is returned.
Job	*	RM_JobPartitionID	pm_partition_id_t *	*	A pointer to the partition ID assigned for the job.
Job		RM_JobState	rm_job_state_t *		A pointer to an enum value indicating the state of the job. (For more about job states, see Figure 9-1 on page 111.)
Job	*	RM_Job Executable	char *	*	A string with the job executable name.
Job	*	RM_JobUser Name	char *	*	A pointer to a string that contains the user name of the user who submitted the job.
Job		RM_JobDB JobID	db_job_id_t *		A pointer to an integer containing the ID given to the job by the database.
Job	*	RM_JobOutFile	char *	*	A pointer to a string containing the job output file name.
Job	*	RM_JobInFile (deprecated)	char *	*	A pointer to a string containing the job input file name.
Job	*	RM_JobErrFile	char *	*	A pointer to a string containing the job error file name.
Job	*	RM_JobOutDir	char *	*	A pointer to a string containing the job output directory. This directory contains the output files if a full path is not given.
Job		RM_JobErrText	char *	*	A pointer to a string containing the error text returned from the control daemons.
Job	*	RM_JobArgs	char *	*	A pointer to a string containing the arguments for the executable.
Job	*	RM_JobEnvs	char *	*	A pointer to a string containing the environment parameter needed for the job.

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
Job		RM_JobInHist	bool *		Indicates whether the job was retrieved from the history table.
Job	*	RM_JobMode	rm_job_mode_t *		A pointer to an enum value indicating the node mode of the partition the values. This can be RM_COPROCESSOR_MODE or RM_VIRTUAL_NODE_MODE.
Job	*	RM_JobStrace	rm_job_strace_t*		A pointer to an integer containing the system call trace indicator for compute nodes.
Job	*	RM_JobStdinInfo	rm_job_stdin_info_t*		A pointer to an integer containing the stdin information.
Job	*	RM_JobStdoutInfo	rm_job_stdout_info_t*		A pointer to an integer containing the stdout information.
Job	*	RM_JobStderrInfo	rm_job_stderr_info_t*		A pointer to an integer containing the stderr information.
Job		RM_JobStartTime	char *	*	A pointer to a string containing the job start time with format of 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 spec RM_JobInHist can be used to determine if a job has completed or not. If the job is an active job then the value returned is meaningless.
Job		RM_JobEndTime	char *	*	A pointer to a string containing the job end time with format of yyyy-mm-dd-hh.mm.ss.nnnnnn. Data is only valid for completed jobs. The rm_get_data spec RM_JobInHist can be used to determine if a job has completed or not. If the job is an active job then the value returned is meaningless.

Object	Set using rm_set_data()?	Specification	Resulting data type	Need to call free function?	Description
Job		RM_JobRun Time	rm_job_runtime_t *		A pointer to the job run time in seconds. Data is only valid for completed jobs. The rm_get_data spec RM_JobInHist can be used to determine if a job has completed or not. If the job is an active job then the value returned is meaningless.
Job		RM_Job ComputeNodes Used	rm_job_compute nodes_used_t *		A pointer to the number of compute nodes used by the job. Data is only valid for completed jobs. The rm_get_data spec RM_JobInHist can be used to determine if a job has completed or not. If the job is an active job then the value returned is meaningless.
Job		RM_JobExit Status	m_job_exitstatus_t *		A pointer to the job exit status. Data is only valid for completed jobs. The rm_get_data spec RM_JobInHist can be used to determine if a job has completed or not. If the job is an active job then the value returned is meaningless.

The APIs other than `rm_get_data` and `rm_set_data` are explained in the following list.

- `status_t rm_get_BGL(rm_BGL_t **bgl);`

  This function retrieves a snapshot of the Blue Gene/L machine, held in the BGL data structure.

- `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.

- `status_t rm_modify_partition(pm_partition_id_t,enum rm_modify_op, const void *value);`

  This function makes it possible to change a set of fields in an already existing partition. Only partitions in a FREE state can be modified. The fields that can be modified are owner, description, kernel options, and images.

- `status_t rm_get_partition(pm_partition_id_t pid, rm_partition_t**p);`

  This function retrieves a partition, according to its ID.

- `status_t rm_set_part_owner(pm_partition_id_t pid, const char *);`

  This function sets the new owner to the partition. Changing the partition's owner can be done only to partition in a FREE state.

- `status_t rm_add_part_user (pm_partition_id_t pid, const char *);`

  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.

- `status_t rm_remove_part_user(pm_partition_id_t pid, const char *);`

  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 INITIALIZE state.

- `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 with a specific state, as defined by the flag value (set of bits). For the set of all possible flags, see the rm_api.h include file.

- `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 with a specific state, as defined by the flag value (set of bits). This function returns the partition information without their base partitions. The possible flags are contained in the rm_api.h include file and listed in Table 9-2 for your convenience. The states are represented by the bits in Table 9-2.

Table 9-2 Flags for partition states

Flag	Value
PARTITION_FREE_FLAG	0x01
PARTITION_CONFIGURING_FLAG	0x02
PARTITION_READY_FLAG	0x04
PARTITION_BUSY_FLAG (deprecated)	0x08
PARTITION_DEALLOCATING_FLAG	0x10
PARTITION_ERROR_FLAG	0x20
PARTITION_ALL_FLAG	0xFF

- `status_t rm_remove_partition(pm_partition_id_t pid);`

  This function removes the specified partition record from MMCS.

- `status_t rm_assign_job(pm_partition_id_t pid , 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 specific partition, it can be assigned later by calling `rm_assign_job()`.

- `status_t rm_release_partition(pm_partition_id_t pid);`

  This function is the opposite of `rm_assign_job()`, because it releases the partition from all jobs. Only jobs that are in an IDLE state have their partition reference removed.

- `status_t rm_set_partition_debuginfo(partid, tv_server_exe, tv_server_args);`

  This function sets the debug info for the block.

- `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.

- `status_t rm_get_job(db_job_id_t jid, rm_job_t **job);`

  This function retrieves the specified job object.

- `status_t rm_get_jobs(rm_job_state_flag_t flag_t, rm_job_list_t **jobs);`

  This functions returns a list of jobs with a specific state or states, as defined by the flag value (set of bits). The set of all possible flags are contained in the rm_api.h include file, and are listed in Table 9-3. The states are represented by the bits in Table 9-3.

  Table 9-3 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

- `status_t rm_query_job(jobid, **MPIR_Proctable, *MPIR_proctable_size);`

  This function fills the `MPIR_Proctable` with information about the specified job.

- `status_t rm_remove_job(db_job_id_t jid);`

  This function removes the specified job record from MMCS.

- `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` (BGL, base partition, wire, switch, connection, port, and so on). The specifications available when using `rm_get_data()` are listed in Table 9-1 on page 97 and are grouped by the querying object.

- `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` (BGL, base partition, wire, switch, connection, port, and so on). The specifications available when using `rm_set_data()` are listed in Table 9-1 on page 97 and marked with an asterisk.

- `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.
- `status_t rm_get_serial(rm_serial_t *serial);`

  This function gets the machine serial number that was set previously by `rm_set_serial()`.
- `status_t rm_get_nodecards(rm_bp_id_t bpid, rm_nodecard_list_t **nc_list);`

  This functions returns all node cards in the specified base partition.

9.2.2 Resource Manager Memory Allocators API

The following APIs are used to allocate memory that is used with other API calls.

- `status_t rm_new_partition(rm_partition_t**partition);`
- `status_t rm_new_job(rm_job_t **job);`
- `status_t rm_new_BP(rm_BP_t **bp);`
- `status_t rm_new_switch(rm_switch_t **switch);`
- `status_t rm_new_nodecard(rm_nodecard_t ** nc);`

9.2.3 Resource Manager Memory Deallocators API

The following APIs are used to deallocate memory that was allocated with the APIs listed in the previous section.

- `status_t rm_free_partition(rm_partition_t*partition);`
- `status_t rm_free_job(rm_job_t *job);`
- `status_t rm_free_BP(rm_BP_t *bp);`
- `status_t rm_free_nodecard(rm_nodecard_t *nc);`
- `status_t rm_free_switch(rm_switch_t *wire);`
- `status_t rm_free_BGL(rm_BGL _t*bgl);`
- `status_t rm_free_partition_list(rm_partition_list_t *part_list);`
- `status_t rm_free_job_list(rm_job_list_t *job_list);`
- `status_t rm_free_nodecard_list(rm_nodecard_list_t *nc_list);`

9.2.4 Messaging API

This section describes 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, for example, MPIRUN. Each message is built using the following format:

```
<Timestamp> Component (Message type): Message text
```

Here is an example:

```
<Mar 9 04:24:30> BRIDGE (Debug): rm_get_BGL()- Completed Successfully
```

There are six types of messages:

- 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

There are also five verbosity levels to which the messaging APIs can be configured. These levels define the following 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.

The control system (Bridge) uses only debug messages, so by default, only error and warning messages are issued by the Bridge functions. To get basic debug messages, set the verbosity level to 2. To obtain more debug information, the level should be 3 or 4.

- `void sayPlainMessage(FILE * stream, char * format, ... );`

  This is a thread-safe version of `fprintf( )`. The message is always printed, regardless of the verbosity level that was set by the `setSayMessageParams( )` API.

- `void setSayMessageParams(FILE * stream, unsigned int level);`

  This function configures the `sayMessage( )` and `sayCatMessage( )` messaging APIs. It defines where the messages are printed to and defines the verbosity level. By default, if this function is not called, the messages are printed to `stderr`, and the verbosity level is set to 0 (only errors and warnings).

- `void sayMessage(char * component, message_type_t m_type, char * curr_func, char * format, ...)`

  This function prints a formatted message to the stream that was defined by the `setSayMessageParams( )` function based on the message type and verbosity level.

- `void sayCatMessage(char * current_func, cat_message_type_t  cat_message);`

  This function is used to print error message of a specific type. The message types that are defined by `cat_message_type_t` are:

  – CAT_BP_WO_WIRES: Base partitions cannot exist without wires.

  – CAT_MEM: Operation failed due to a memory allocation error.

  – CAT_PARSE_XML: Error parsing XML file.

  – CAT_RET_CODE: Unrecognized return code from internal function.

  – CAT_COMM: A communication problem occurred while attempting to connect to the database.

  – CAT_DB_ACCESS: An error occurred while attempting to access the database.

  – CAT_XML_ACCESS: Could not access (create or read) the XML file.

  – CAT_DATA_NOT_FOUND: Data record or records are not found.

  – CAT_SEQUENCE_ERR: A sequence error occurred.

  – CAT_BAD_ID: A bad ID was used for the call.

  – CAT_DUP_DATA: Attempt to insert duplicate record.

  – CAT_BGL_INFO: Failed to retrieve information about Blue Gene/L.

  – CAT_BAD_INPUT: Illegal input field used for the call.

  – CAT_FREE_ERR: An error occurred while trying to free object.

  – CAT_GENERAL_ERR: General error.

9.2.5 API to the MMCS job manager

The first three APIs (`jm_start_job`, `jm_signal_job`, and `jm_cancel_job`) are asynchronous. This means that control returns to your application before the operation requested is complete.

Before you perform additional operations on the job, check to make sure it is in a valid state by using the `rm_get_jobs()` API together with the flags for job states as listed in Table 9-3 on page 107.

- `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.

- `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.

- `status_t jm_cancel_job(db_job_id_t jid);`

  This function sends a request to cancel the job identified by the `jid` parameter.

- `status_t jm_load_job(jobid);`

  This function sets the job state to LOAD.

- `status_t jm_attach_job(jobid);`

  This function initiates the spawn of TotalView servers to a LOADED job.

- `status_t jm_debug_job(jobid);`

  This function initiates the spawn of TotalView servers to a RUNNING job.

- `status_t jm_begin_job(jobid);`

  This function begins a job which is already loaded.

9.2.6 API to the MMCS partition manager

These APIs are asynchronous. This means that control returns to your application before the operation requested is complete.

Before you perform additional operations on the partition, check to make sure it is in a valid state by using the `rm_get_partitions_info()` together with the flags for partition states as listed in Table 9-2.

- `status_t pm_create_partition(pm_partition_id_t pid);`

  This function gets a partition ID, creates (wires) the partition, and updates the resulting status in the database.

- `status_t pm_destroy_partition(pm_partition_id_t pid);`

  This function destroys (unwires) an existing partition and updates the database accordingly.

9.2.7 State diagrams for jobs and partitions

Figure 9-1 illustrates the main states that a job goes through during its life cycle.

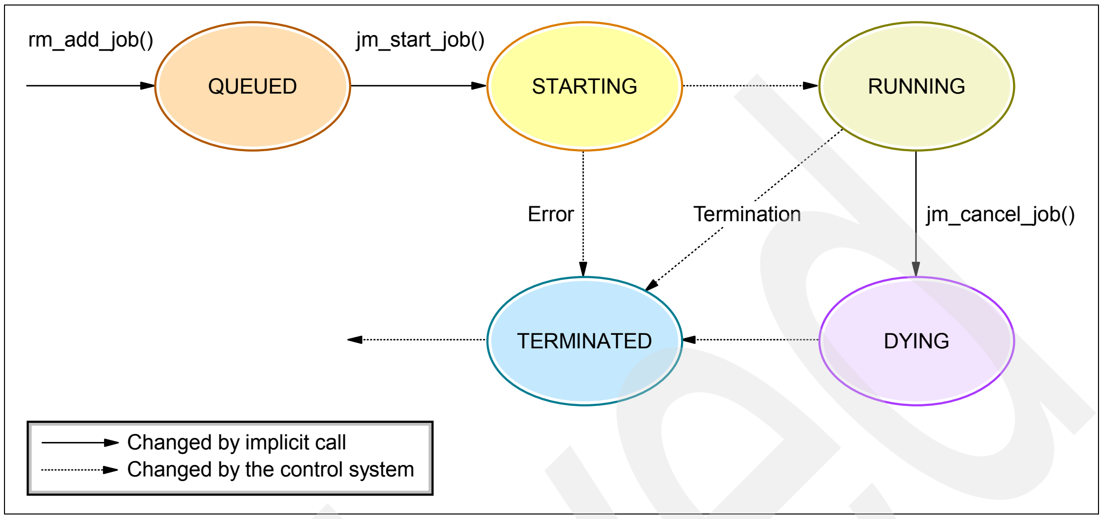


Figure 9-1 Job state diagram

Figure 9-2 describes the various partition states.

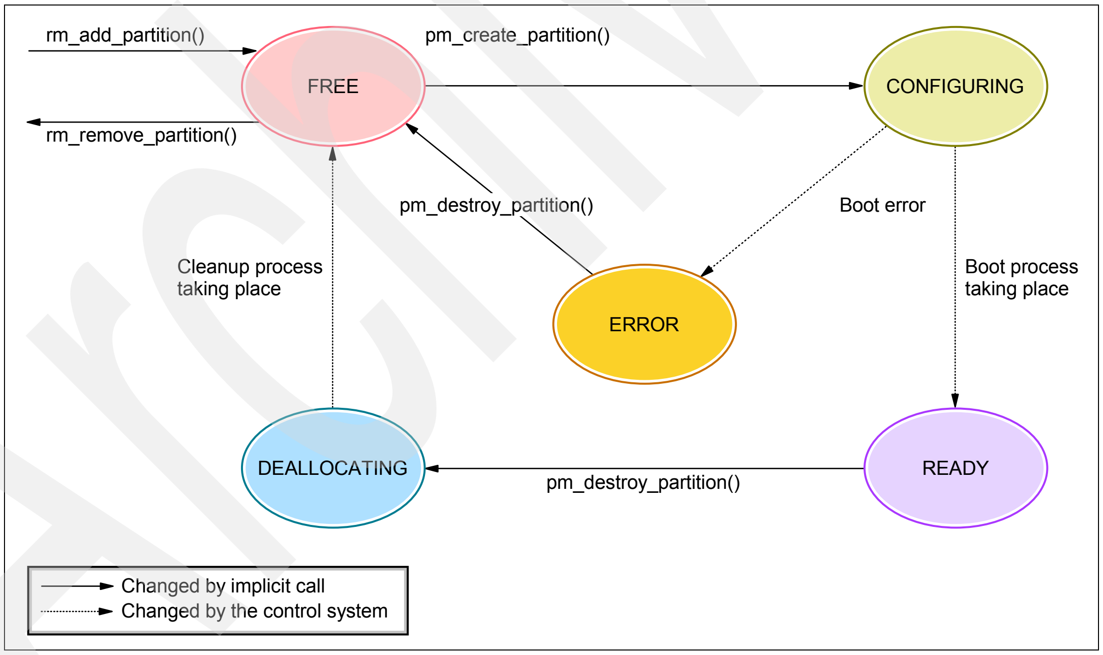


Figure 9-2 Partition state diagram

Base partitions, wires, switches and node cards have two states, UP and DOWN, which reflect the physical state of these components.

9.3 Control system API return codes

When a failure occurs, an API invocation returns an error code. This error code helps apply automatic corrective actions within the job scheduling system. 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 non-automatic 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 return codes of the Control System API are:

- STATUS_OK: 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.
- JOB_ALREADY_DEFINED: A job with the same name already exists.
- 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.
- INCOMPATIBLE_STATE: The state of the partition or job prohibits the specific action (see Figure 9-1 and Figure 9-2 for state diagrams).
- 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. This is due to missing required data, illegal data, and so on
- INCONSISTENT_DATA: The data retrieved from the control system is illegal or invalid.
- INTERNAL_ERROR: Errors that do not belong to any of the previously listed categories, such as a memory allocation problem or failures during the manipulation of internal XML files.

9.3.1 Return codes specification

The various API functions have the following return codes:

- `status_t rm_get_BGL(rm_BGL_t **bgl);`

  This function retrieves a snapshot of the Blue Gene/L machine.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – INCONSISTENT_DATA: Possibly for one of the following reasons:
    - 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_add_partition(rm_partition_t *p);`

  This function defines a partition in the control system.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – INVALID_INPUT: The data in the `rm_partition_t` structure is invalid.

    • No base partition or switch list is supplied.
    • Base partition or switches do not construct a legal partition.
    • No boot images or boot image name is too long.
    • No user or 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_get_partition(pm_partition_id_t pid, rm_partition_t **p);`

  This function retrieves a partition according to its ID.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – INVALID_INPUT

    "pid" is null or the length exceeds the control system limitations (configuration parameter).

  – PARTITION_NOT_FOUND
  – INCONSISTENT_DATA

    Base partition or switch list of the partition is empty.

  – INTERNAL_ERROR

- `status_t rm_get_partitions(rm_partition_state_flag_t flag, rm_partition_list_t * part_list);`

  This function returns a list of partitions with a specific state, defined by the flag value.

  The return codes are:

  – 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 with a specific state, as defined by the flag value (set of bits). This function returns the partitions information without their base partitions.

  The states are represented by the following bits:

  – PARTITION_FREE_FLAG 0x1
  – PARTITION_CONFIGURING_FLAG 0x2
  – PARTITION_READY_FLAG 0x4

  - PARTITION_BUSY_FLAG 0x8 (deprecated)
  - PARTITION_DEALLOCATING_FLAG 0x10
  - PARTITION_ERROR_FLAG 0x20
  - PARTITION_ALL_FLAG 0xFF

  The return codes are:

  - STATUS_OK
  - CONNECTION_ERROR
  - INCONSISTENT_DATA

    At least one of the partitions has an empty base partition list.

  - INTERNAL_ERROR

- `status_t rm_remove_partition (pm_partition_id_t pid);`

  This function removes a partition from the control system.

  The return codes are:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT

    “pid” is null or the length exceeds the control system limitations (configuration parameter).

  - PARTITION_NOT_FOUND
  - INCOMPATIBLE_STATE

    The partition’s current state forbids its removal. See Figure 9-1 on page 111.

  - INTERNAL_ERROR

- `status_t rm_assign_job (pm_partition_id_t pid, db_job_id_t jid);`

  This function assigns (associates) a job with a given partition.

  The return codes are:

  - STATUS_OK
  - CONNECTION_ERROR
  - PARTITION_NOT_FOUND
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE
    - The current state of the partition or the job prevents this assignment. See Figure 9-1 on page 111 and Figure 9-2 on page 111.
    - Partition and job owner do not match.
  - INVALID_INPUT

    “pid” is null or the length exceeds the control system limitations (configuration parameter).

  - INTERNAL_ERROR

- `status_t rm_release_partition (pm_partition_id_t pid);`

  This function disassociates all jobs with the given partition.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – INVALID_INPUT

    “pid” is null or the length exceeds the control system limitations (configuration parameter).

  – PARTITION_NOT_FOUND
  – INCOMPATIBLE_STATE

    The current state of some of the jobs assigned to the partition prevents this release. See Figure 9-1 on page 111 and Figure 9-2 on page 111.

  – INTERNAL_ERROR

- `status_t rm_set_part_owner(pm_partition_id_t pid, const char *owner);`

  This function changes the partition owner.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – INVALID_INPUT

    - “pid” is null or the length exceeds the control system limitations (configuration parameter).
    - “owner” in null or the length exceeds the control system limitations.

  – INTERNAL_ERROR

- `status_t rm_add_part_user(pm_partition_id_t pid, const char *user);`

  This function adds a user to a partition.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – INVALID_INPUT

    - “pid” is null or the length exceeds the control system limitations (configuration parameter).
    - “user” in null or the length exceeds the control system limitations.
    - “user” already defined as the partition’s user.

  – INTERNAL_ERROR

- `status_t rm_remove_part_user(pm_partition_id_t pid, const char *user);`

  This function adds a user to a partition.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR

  - INVALID_INPUT
    - “pid” is null or the length exceeds the control system limitations (configuration parameter).
    - “user” in null or the length exceeds the control system limitations.
    - “user” already defined as the partition’s user.
  - INTERNAL_ERROR
- `status_t pm_create_partition (pm_partition_id_t pid);`

  This function requests creation of a partition.

  The return codes are:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT

    “pid” is null or the length exceeds control system limitations (configuration parameter).
  - PARTITION_NOT_FOUND
  - INCOMPATIBLE_STATE

    The current state of the partition prohibits its creation. See Figure 9-1 on page 111.
  - INTERNAL_ERROR

    No owner was set for the partition.
- `status_t pm_destroy_partition (pm_partition_id_t pid);`

  This function requests destruction of a partition.

  The return codes are:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT

    “pid” is null or the length exceeds the control system limitations (configuration parameter).
  - PARTITION_NOT_FOUND
  - INCOMPATIBLE_STATE

    The state of the partition prohibits its destruction. See Figure 9-1 on page 111.
  - INTERNAL_ERROR
- `status_t rm_add_job(db_job_id_t *job);`

  This function defines a job in the control system.

  The return codes are:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT: Data in the `rm_job_t` structure is invalid.
    - No job name or job name is too long.
    - No user name or user name is too long.
    - No executable or executable name too long.
    - Output or error file name is too long.
  - JOB_ALREADY_DEFINED

    A job with the same name already exists.
  - INTERNAL_ERROR

- `status_t rm_get_job(db_job_id_t jid, rm_job_t **job);`

  This function retrieves a job by its ID, “jid”.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – JOB_NOT_FOUND
  – INTERNAL_ERROR

- `status_t rm_get_jobs(rm_job_state_flag_t flag, rm_job_list_t *jobs);`

  This function returns a list of jobs with a specific state (defined by the “flag” value).

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – INTERNAL_ERROR

- `status_t rm_remove_job(db_job_id_t jid);`

  This function removes a specific job from the control system.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – JOB_NOT_FOUND
  – INCOMPATIBLE_STATE

    The job’s state prevents its removal. See Figure 9-2.

  – INTERNAL_ERROR

- `status_t jm_start_job(db_job_id_t jid);`

  This function requests the start of execution for a specific job.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – JOB_NOT_FOUND
  – INCOMPATIBLE_STATE

    The job’s state prevents its execution. See Figure 9-2 on page 111.

  – INTERNAL_ERROR

- `status_t jm_cancel_job(db_job_id_t jid);`

  This function requests cancellation of a job.

  The return codes are:

  – STATUS_OK
  – CONNECTION_ERROR
  – JOB_NOT_FOUND
  – INCOMPATIBLE_STATE

    The job’s state prevents it from being canceled. See Figure 9-2 on page 111.

  – INTERNAL_ERROR

- `status_t jm_signal_job(db_job_id_t jid, rm_signal_t signal);`

  This function signals a job.

  The return codes are:

  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job's state prevents it from being signaled.
  - INTERNAL_ERROR
- `status_t rm_get_nodecards(rm_bp_id_t bpid,rm_nodecard_list_t **nc_list);`

  This function returns a list of all the node cards in the specified Base Partition.

  The return codes are:

  - STATUS_OK
  - CONNECTION_ERROR
  - INCONSISTENT_DATA

    The Base Partition was not found.
  - INTERNAL_ERROR
- `status_t rm_get_data(rm_element_t *rme, enum RMSpecification spec, void * result);`

  `status_t rm_set_data(rm_element_t *rme, enum RMSpecification spec, void * result);`

  These two auxiliary functions access the requested field in an `rm_element_t` structure (BGL, base partition, wire, switch, connections, port, node card).

  The return codes are:

  - 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 the following API calls.

  The return codes are:

  - STATUS_OK
  - INVALID_INPUT
    - The machine serial number "serial" is null.
    - The machine serial number is too long.
- `status_t rm_set_serial(rm_serial_t *serial);`

  This function retrieves the machine serial used with the APIs.

  The return codes are:

  - STATUS_OK
  - INTERNAL_ERROR

- `status_t rm_new_< partition, job, BP, switch, nodecard>`

  This auxiliary function allocates memory for an `rm_element` object.

  The return codes are:

  – STATUS_OK
  – INTERNAL_ERROR

- `status_t rm_free_<BGL, partition, job, BP, switch, partition_list, job_list, nodecard>`

  This auxiliary function frees the memory allocated by the `rm_new..` or `rm_get..` APIs.

  The return codes are:

  – STATUS_OK
  – INTERNAL_ERROR

9.4 Small partition allocation

The base allocation unit in Blue Gene/L 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 comprises a fraction of a base partition. Small partitions can come in two sizes: one fourth (1/4) of a base partition and one sixteenth (1/16). Base partitions are divided into four quarters, and each quarter is then divided into four node cards. This is illustrated in Figure 9-3.

Small partitions can be created from one node card (one sixteenth of a node card) or one quarter of a base partition (four node cards).

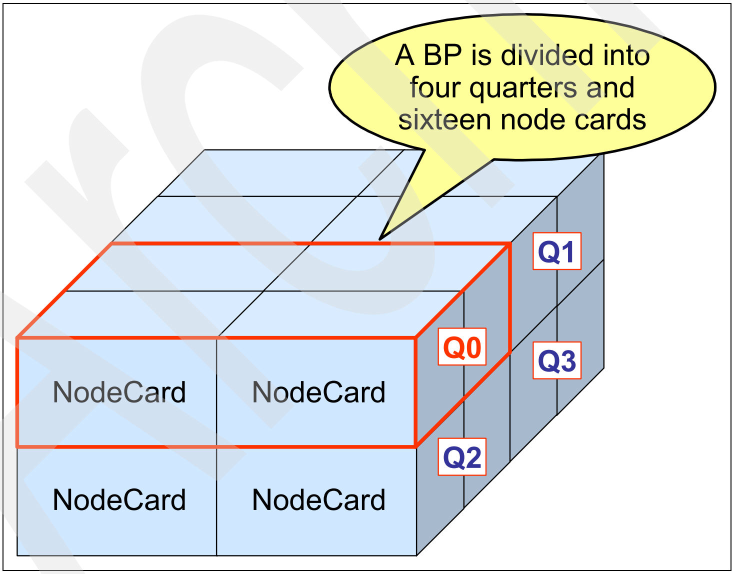


Figure 9-3 Dividing a base partition

9.4.1 Base partition definitions

You need to understand the following definitions when working with small partitions.

- **Subdivided**: A base partition is considered subdivided if at least one partition is defined for a subset of its node cards. That is, either a single node card partition exists, or a four-node card ("quarter") partition exists.
- **Subdivided busy**: A base partition is considered subdivided busy if at least one of the partitions, which is defined for the subset of its node cards, is both busy and not in an error state.

A base partition that is subdivided (not busy) can still be booted as part of another partition as a whole, regardless of the partitions defined for its subcomponents. A base partition that is subdivided busy cannot be booted as a whole.

A base partition can have sub-midplane (small) partitions (32 or 128 compute nodes) and full midplane blocks (512 compute nodes) defined for it in the database. If the base partition has small blocks defined, then they do not have to be in use, and a full block can use the actual midplane. In this case, the block name that is using the base partition is returned on the `RM_BPPartID` specification.

For small blocks, you can have multiple blocks using the same base partition. This is the subdivided busy (SDB) example. In this situation, what is returned on the `RM_BPPartID` specification is meaningless. You can use the `RM_BPSDB` specification to determine if the base partition is subdivided busy (small block in use). If the subdivided flag (`RM_BPSDB` specification) is true, you do not have enough information to know whether the block returned is relevant. Only the subdivided busy flag (RM_BPSDB spec) can help determine that.

9.4.2 API examples

In this section, we provide example API calls for several common situations.

Detailed information about the node cards comprising a given base partition can be obtained by calling:

```
rm_get_nodecards(rm_bp_id_t bpid, rm_nodecard_list_t **nc_list);
```

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. The list is traversed using the `rm_get_data` function as shown in the following examples:

```
rm_get_data(nc_list, RM_NodeCardListFirst,rm_nodecard_t **nc);
rm_get_data(nc_list, RM_NodeCardListNext,rm_nodecard_t **nc);
```

Information about each node card can retrieved by calling `rm_get_data` with the `nodecard` object as shown in the following examples:

```
Get the id of a node card:
rm_nodecard_id_t ncid;
rm_get_data(nc, RM_NodeCardID, &ncid);

Get the quarter of a node card:
rm_quarter_t quarter;
rm_get_data(nc, RM_NodeCardQuarter, &quarter);

Get the state of a node card:
rm_nodecard_state_t  state;
rm_get_data(nc, RM_NodeCardState, &state);
```

```
Get the number of IO nodes on a node card:
int ios;
rm_get_data(nc, RM_NodeCardIONodes, &ios);

Get the partition ID with which the node card is associated:
pm_partition_id_t partid;
rm_get_data(nc, RM_NodeCardPartID, &partid);

Get the partition state:
pm_partition_state partstate;
rm_get_data(nc, RM_NodeCardPartState, &partstate);
```

9.4.3 Allocating a new small partition

Example 9-3 is pseudo code that shows how to allocate a new small partition.

Example 9-3 Allocating a new small partition

```
bool isSmall = true;

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, BGL_MLOADER_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, “R010”);
rm_set_data(newpart, RM_PartitionFirstBP, BP);

// Add the node card(s) comprising the partition
ncNum = 4; // The number of node cards can be only 1 or 4
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);
```

9.4.4 Querying a small partition

Example 9-4 is pseudo code that shows how to query a small partition for its node cards.

Example 9-4 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 IO 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
   }
}
```

10

mpirun APIs

This chapter defines a list of application programming interfaces (APIs) that allow interaction with the **mpirun** program. These APIs are used by applications, such as external schedulers, that want to programatically invoke jobs via **mpirun**.

10.1 API support overview

The following sections provide an overview of the support provided by the APIs.

10.1.1 Requirements

There are several requirements for writing programs to the **mpirun** APIs as explained in the following sections.

Operating system supported

Currently, SUSE Linux Enterprise Server (SLES) 9 for PowerPC is the only supported platform.

Languages supported

C and C++ are supported with the GNU gcc 3.4.3 level compilers. For more information and downloads, refer to the following Web address:

`http://gcc.gnu.org/`

Include files

The include file is include/sched_api.h.

Library files

Support for both 32-bit and 64-bit dynamic libraries is provided.

32-bit libraries

The 32-bit dynamic library file called by mpirun must be called libsched_if.so.

64-bit libraries

The 64-bit dynamic library file called by mpirun must be called libsched_if64.so.

10.2 APIs

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 (libsched_if64.so if you are linked to the 64-bit version of the binaries). **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 and the job to run. Each field corresponds to one of the command-line parameters or environment variables.

mpirun complements the information retrieved by `get_parameters()` with values from its command-line parameters and environment variables. It gives precedence to the information 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 or libsched_if64.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 might 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 about the library implementation and data structures in the sched_api.h header file.

10.2.1 Supported APIs

The following list shows the APIs that 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`** will proceed to terminate. Some external schedulers use this technique to prevent stand-alone **`mpirun`** from being used. If the plug-in provider wants **`mpirun`** processing to continue, then they need to 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.

11

Dynamic partition allocator APIs

This chapter defines a list of application programming interfaces (APIs) into the Midplane Management Control System (MMCS) dynamic partition allocator. The intention of the APIs is to provide an easy-to-use interface for the dynamic creation of blocks. The allocate partition API inspects the current state of the Blue Gene/L machine and attempts to create a partition based on available resources. If no resources are available that match the block requirements, then the block is not created. It is expected that any job scheduler using the partition allocator does so from a centralized process to avoid conflicts in finding free resources to build the partition.

These APIs are thread safe. Only 64-bit static and shared libraries are being provided; 32-bit interfaces are not supported.

See Chapter 9, “Control system (Bridge) APIs” on page 93, for details about the Bridge APIs.

11.1 API support overview

The following sections provide an overview of the support provided by the APIs.

11.1.1 Requirements

The following sections explain the requirements for writing programs to the dynamic partition allocator APIs.

Operating system supported

Currently, SUSE Linux Enterprise Server (SLES) 9 for PowerPC is the only supported platform.

Languages supported

C and C++ are supported with the GNU gcc 3.4.3 level compilers. For more information and downloads, refer to the following Web address:

`http://gcc.gnu.org/`

Include files

The include file is include/allocator_api.h.

Library files

Only 64-bit static and shared libraries are provided.

64-bit static libraries

For static linking of 64-bit code, use the following files:

- liblbglbridge_s
- libbgldb_s
- libtableapi_s
- libbglmachine_s
- libsaymessage_s
- libbglallocator_s

Note: All of these libraries can be found in the /usr/lib64 directory.

The following libraries must also be linked in:

- libexpat
- libdb2

64-bit dynamic (shared) libraries

For dynamic (shared) linking, use:

- liblbglbridge
- libbgldb
- libtableapi
- libbglmachine
- libsaymessage
- libbglallocator

Note: All of these libraries can be found in the /usr/lib64 directory.

The following libraries must also be linked in:

- libexpat
- libdb2

11.2 API details

This section contains details about the APIs and return codes for dynamic partition allocation.

11.2.1 APIs

The following APIs are used for dynamic partition allocation. These APIs are all thread safe.

- `BGALLOC_STATUS rm_init_allocator( char * caller_desc, 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” will cause any partitions created by `rm_allocate_partition()` to have “ABC job scheduler” as the block description.

  The caller can also optionally specify a drain list file name that contains the base partitions (midplanes) that will be excluded from list of resources to consider when building the partition. If NULL is passed in for the drain list file name, the API will look for input in the `/etc/allocator_drain.lst` file. If that file does not exist, no resources will be excluded. If an invalid file name is passed in, the call will fail.

  For example, a drain list file with the following content would exclude base partitions R000, R001, and R010 when allocating resources for a block:

  ```
  R000
  R001
  R010
  ```

  Note that 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 simply ignored—no error is produced.

- `BGALLOC_STATUS rm_allocate_partition(rm_size_t size, rm_connection_type_t conn, rm_size3D_t shape, rm_job_mode_t mode, rm_psetsPerBP_t psetsPerBP, const char * user_name, const char * caller_desc, rm_resource_t * resources, allocate_type atype, char ** partition_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/L machine resources. If resources are available that match the requirements, then 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, then the allocation will be based on the shape value only.

  The caller to `rm_allocate_partition()` must specify the allocate type. The options for this are:

  – ALL_RESOURCES: Allocate from all resources.
  – FROM_RESOURCES: Allocate based on a set of resources.
  – EXCLUDE_RESOUCES: Exclude specific resources in the allocation decision making.

  If ALL_RESOURCES is selected, then NULL should be passed on the resources input parameter. If including or excluding resources, then the resources structure should be passed as an input parameter. If a drain list is being used, then those base partitions in the list are always excluded by `rm_allocate_partition()` no matter what other resources are included/excluded by the allocate type feature.

Important: The returned char * value for partition_id should be freed by caller when it is no longer needed to avoid memory leaks.

11.2.2 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:

- BGALLOC_OK: Invocation completed successfully.
- BGALLOC_ILLEGAL_INPUT: The input to the API invocation is invalid. This 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.

11.3 Sample program

The following sample program shows how to allocate a partition from resources on base partition R001.

Example 11-1 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() {
    //set lowest level of verbosity
    setSayMessageParams(stderr, MESSAGE_DEBUG1);

    rm_size3D_t shape;
    rm_resource_t* resource = new rm_resource_t[2];
    rm_connection_type_t conn = RM_MESH;
    shape.X = 0;
    shape.Y = 0;
    shape.Z = 0;

    //create resource struct
    resource[0] = new rm_resource_struct;
    resource[0]->bp  = "R001";
    //add node card names
    resource[0]->ncs = new char*[17];
    resource[0]->ncs[0] = "J102";
    resource[0]->ncs[1] = "J104";
    resource[0]->ncs[2] = "J106";
    resource[0]->ncs[3] = "J108";
    resource[0]->ncs[4] = "J111";
    resource[0]->ncs[5] = "J113";
    resource[0]->ncs[6] = "J115";
    resource[0]->ncs[7] = "J117";
    resource[0]->ncs[8] = "J203";
    resource[0]->ncs[9] = "J205";
    resource[0]->ncs[10] = "J207";
    resource[0]->ncs[11] = "J209";
    resource[0]->ncs[12] = "J210";
    resource[0]->ncs[13] = "J212";
    resource[0]->ncs[14] = "J214";
    resource[0]->ncs[15] = "J216";
    resource[0]->ncs[16] = NULL;

    //null terminate resource array
    resource[1] = NULL;

    char** partition_id;
    *partition_id = "NULL"; //default value

    BGALLOC_STATUS alloc_rc;
    alloc_rc = rm_init_allocator("test", NULL);
    alloc_rc = rm_allocate_partition(256, conn, shape, RM_COPROCESSOR_MODE, 0, "user1",
"New block description", resource, FROM_RESOURCES, partition_id);
    if (alloc_rc == BGALLOC_OK) {
        cout << "successfully allocated block: " << *partition_id << endl;
    } else {
        cerr << "could not allocate block: " << *partition_id << 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 {
            cerr << "internal error" << endl;
        }
    }
}
```

12

High Throughput Computing on Blue Gene/L

The original focus of the Blue Gene project was to create a high performance computer with a small footprint that consumed a relatively small amount of power. The model of running parallel applications (typically using Message Passing Interface (MPI)) on Blue Gene/L is known as *High Performance Computing* (HPC). Recent research has shown that Blue Gene/L also provides a good platform for *High Throughput Computing* (HTC). This chapter discusses HTC on Blue Gene/L.

12.1 Applications: MPI or HTC

As previously stated, the Blue Gene architecture targeted MPI applications for optimal execution. These applications are characterized as Single Instruction Multiple Data (SIMD) with synchronized communication and execution. The tasks in an MPI program are cooperating to solve a single problem. Because of this close cooperation, a failure in a single node, software or hardware, requires the termination of all nodes.

HTC applications have different characteristics. The code that executes on each node is independent of work that is being done on another node; communication between nodes is not required. At any specific time, each node is solving its own problem. As a result, a failure on a single node, software or hardware, does not necessitate the termination of all nodes.

Initially, in order to run these applications on a Blue Gene supercomputer, a port to the MPI programming model was required. This was done successfully for several applications, but was not an optimal solution in some cases. Some MPI applications may benefit by being ported to the HTC model. In particular, some embarrassingly parallel MPI applications may be good candidates for HTC mode because they do not require communication between the nodes. Also the failure of one node does not invalidate the work being done on other nodes.

A key advantage of the MPI model is a reduction of extraneous booking by the application. An MPI program is coded to handle data distribution, minimize I/O by having all I/O done by one node (typically rank 0), and distribute the work to the other MPI ranks. When running in HTC mode, the application data needs to be manually split up and distributed to the nodes. This porting effort may be justified to achieve better application reliability and throughput than could be achieved with an MPI model.

12.2 HTC mode

In HTC mode, the compute nodes in a partition are running independent programs that do not communicate with each other. A launcher program (running on a compute node) requests work from a dispatcher that is running on a remote system that is connected to the functional network. Based on information provided by the dispatcher, the launcher program spawns a worker program that performs some task.

When running in HTC mode, there are two basic operational differences over the default HPC mode. First, after the worker program completes, the launcher program is reloaded so it can handle additional work requests. Second, if a compute node encounters a soft error, such as a parity error, the entire partition is not terminated. Rather, the control system attempts to reset the single compute node while other compute nodes continue to operate. The control system polls hardware on a regular interval looking for compute nodes in a reset state. If a failed node is discovered and the node failure is not due to a network hardware error, a software reboot is attempted to recover the compute node.

On a remote system that is connected to the functional network, which could be a Service Node or a Front End Node, there is a dispatcher that manages a queue of jobs to be run. There is a client/server relationship between the launcher program and the dispatcher program. After the launcher program is started on a compute node, it connects back to the dispatcher and indicates that it is ready to receive work requests. When the dispatcher has a job for the launcher, it responds to the work request by sending the launcher program the job related information, such as the name of the worker program executable, arguments, and environment variables. The launcher program then spawns off the worker program. When the worker program completes, the launcher is reloaded and repeats the process of requesting work from the dispatcher.

Launcher program

In the default HPC mode, when a program ends on the compute node, the Compute Node Kernel sends a message to the I/O node that reports how the node ended. The message indicates if the program ended normally or by a signal and the exit value or signal number, respectively. The I/O node then forwards the message to the control system. When the control system has received messages for all of the compute nodes, it then ends the job.

In HTC mode, the Compute Node Kernel handles a program ending differently depending on the program that ended. The Compute Node Kernel records the path of the program that is first submitted with the job, which is the launcher program. When a program other than the launcher program (or the worker program) ends, the Compute Node Kernel records the exit status of the worker program, and then reloads and restarts the launcher program (Figure 12-1). If the worker program ended by a signal, the Compute Node Kernel generates an event to record the signal number that ended the program. If the worker program ended normally, no information is logged. The launcher program can retrieve the exit status of the worker program using a Compute Node Kernel system call.

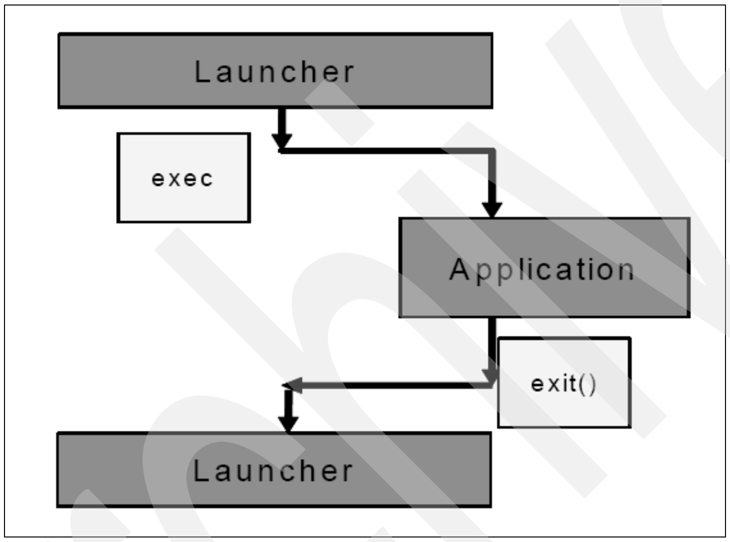


Figure 12-1 Launcher run sequence

Since no message is sent to the control system indicating that a program ended, the job continues running. The effect is to have a continually running program on the compute nodes.

To reduce the load on the file system, the launcher program is cached in memory on the I/O node. When the Compute Node Kernel requests to reload the launcher program, it does not need to be read from the file system but can be sent directly to the compute node from memory. Since the launcher is typically a small executable, it does not require much additional memory to cache it. When the launcher program ends, the Compute Node Kernel reports that a program ended to the control system as it does for HPC mode. This allows the launcher program to cleanly end on the compute node and for the control system to end the job.

12.3 Template for an asynchronous task dispatch subsystem

This section gives a high level overview of one possible method to implement an asynchronous task dispatch subsystem. Figure 12-2 shows the relationship between the client, dispatcher program, and the launcher program in our example.

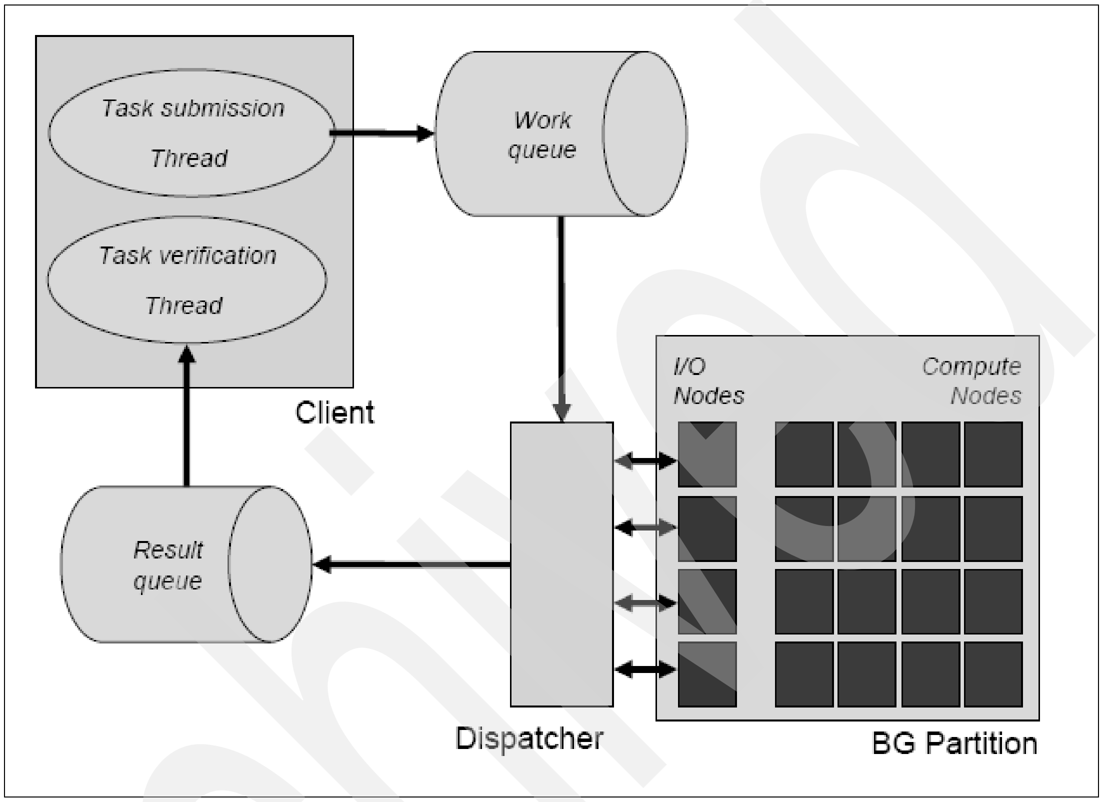


Figure 12-2 Asynchronous task dispatch subsystem model

Clients

The client implements a task submission thread that publishes task submission messages onto a work queue. It also implements a task verification thread that listens for task completion messages. When the dispatch system informs the client that the task has terminated, and optionally supplies an exit status, the client is then responsible for resolving task completion status and for taking actions, including relaunching tasks. Keep in mind that the client is ultimately responsible for ensuring successful completion of the job.

Message queue

The design is based on publishing and subscribing to message queues. Clients publish task submission messages onto a single work queue. Dispatcher programs subscribe to the work queue and process task submission messages. Clients subscribe to task completion messages.

Messages consist of text data that is comprised of the work to be performed, a job identifier, a task identifier, and a message type. Job identifiers are generated by the client process and are required to be globally unique. Task identifiers are unique within the job session. The message type field for task submission messages is used by the dispatcher to distinguish work of high priority versus work of normal priority. Responsibility for reliable message delivery belongs to the message queueing system.

Dispatcher program

The dispatcher program first pulls a task submission message off the work queue. Then it waits on a socket for a launcher connection and reads the launcher ID from the socket. It writes the task into the socket, and the association between task and launcher is stored in a table. The table stores the last task dispatched to the launcher program. This connection is an indication that the last task has completed and the task completion message can be published back to the client. Figure 12-3 shows the entire cycle of a job submitted in HTC mode.

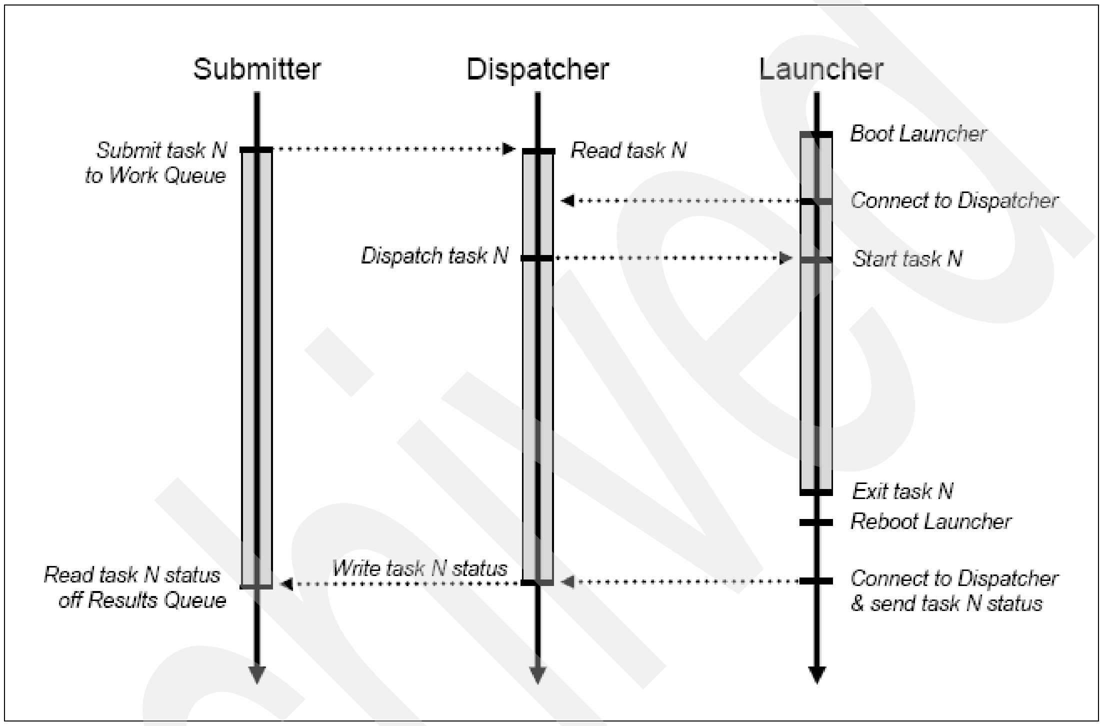


Figure 12-3 HTC job cycle

The intention of this design is to optimize the launcher program. The dispatcher program spends little time between connect and dispatch, so latency volatility is mainly due to the waiting time for dispatcher program connections. After rebooting, the launcher program connects to the dispatcher program and passes the completion information back to the dispatcher program. To assist task status resolution, the Compute Node Kernel stores the exit status of the last running process in a buffer. After the launcher program restarts, the contents of this buffer can be written to the dispatcher and stored in the task completion message.

Launcher program

The launcher program is intentionally kept simple. Arguments to the launcher program describe a socket connection to the dispatcher. When the launcher program starts, it connects to this socket, writes its identity into the socket, and waits for a task message. Upon receipt of the task message, the launcher parses the message and calls the execve system call to execute the task. When the task exits (for any reason), the Compute Node Kernel restarts the launcher program again. The launcher program is not a container for the application. Therefore, regardless of what happens to the application, the launcher program will not fail to restart.

Tip: Basic sample programs are available for download in the Blue Gene/L Knowledge Base. To access the Knowledge Base, you must sign in using your IBM ID and password. Search on either HTC or document number 442342092. You can access the Knowledge Base at the following address:

`http://www-304.ibm.com/jct01004c/systems/support/bluegene/index.html`

12.4 I/O considerations

Running in HTC mode changes the I/O patterns. When a program reads and writes to the file system, it is typically done with small buffer sizes. While the administrator can configure the buffer size, the most common sizes are 256K or 512K. When running in HTC mode, loading the worker program requires reading the complete executable into memory and sending it to a compute node. An executable is at least several megabytes and can be many megabytes. To achieve the fastest I/O throughput, low compute node to I/O node ratio is the best.

For more information about allocating blocks in HTC mode and submitting jobs, see *Blue Gene/L: System Administration*, SG24-7178.

A

Statement of completion

IBM considers installation to be complete when the following activities have taken place:

- The Blue Gene/L rack or racks have been physically placed in position.
- The cabling is complete, including power, Ethernet, and torus cables.
- The Blue Gene/L racks can be powered on.
- All hardware is displayed in the Navigator and is available.

B

Electromagnetic compatibility

This chapter provides important electromagnetic compatibility information about Blue Gene/L in various geographic countries or regions around the world.

European Union Electromagnetic Compatibility Directive	This product is in conformity with the protection requirements of EU Council Directive 89/336/EEC on the approximation of the laws of the Member States relating to electromagnetic compatibility. IBM cannot accept responsibility for any failure to satisfy the protection requirements resulting from a non-recommended modification of the product, including the fitting of non-IBM option cards.
Canada	This Class A digital apparatus complies with Canadian ICES-003. Cet appareil numérique de la classe A est conform à la norme NMB-003 du Canada
European Union - Class A	**Attention:** This is a Class A product. In a domestic environment, this product may cause radio interference in which case the user may be required to take adequate measures.
	This product has been tested and found to comply with the limits for Class A Information Technology Equipment according to European Standard EN 55022. The limits for Class A equipment were derived for commercial and industrial environments to provide reasonable protection against interference with licensed communication equipment. Properly shielded and grounded cables and connectors must be used in order to reduce the potential for causing interference to radio and TV communications and to other electrical or electronic equipment. IBM cannot accept responsibility for any interference caused by using other than recommended cables and connectors.
Japan - VCCI Class A	この装置は、情報処理装置等電波障害自主規制協議会（ＶＣＣＩ）の基準に基づくクラスＡ情報技術装置です。この装置を家庭環境で使用すると電波妨害を引き起こすことがあります。この場合には使用者が適切な対策を講ずるよう要求されることがあります。

Korean	이 기기는 업무용으로 전자파적합등록을 한 기기이오니 판매자 또는 사용자는 이 점을 주의하시기 바라며 , 만약 잘못 판매 또는 구입하였을 때에는 가정용으로 교환하시기 바랍니다 .
United States - FCC class A	**Federal Communications Commission (FCC) Statement:** This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense. Properly shielded and grounded cables and connectors must be used in order to meet FCC emission limits. IBM is not responsible for any radio or television interference caused by using other than recommended cables and connectors or by unauthorized changes or modifications to this equipment. Unauthorized changes or modifications could void the user's authority to operate the equipment. This device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

C

Blue Gene/L safety considerations

This appendix describes important safety considerations that you must follow when installing and using the Blue Gene/L system.

Important safety notices

Here are some important general comments about the Blue Gene/L system regarding safety.

- **CAUTION**
  This equipment must be installed by trained service personnel in a restricted access location as defined by the NEC (U.S. National Electric Code) and IEC 60950, The Standard for Safety of Information Technology Equipment. (C033)
- **CAUTION**
  The doors and covers to the product are to be closed at all times except for service by trained service personnel. All covers must be replaced and doors locked at the conclusion of the service operation. (C013)
- **CAUTION**
  Servicing of this product or unit is to be performed by trained service personnel only. (C032)
- **CAUTION**
  This product is equipped with a 4 wire (three-phase and ground) power cable. Use this power cable with a properly grounded electrical outlet to avoid electrical shock. (C019)
- **DANGER**
  To prevent a possible electric shock from touching two surfaces with different protective ground (earth), use one hand, when possible, to connect or disconnect signal cables. (D001)
- **DANGER**
  An electrical outlet that is not correctly wired could place hazardous voltage on the metal parts of the system or the devices that attach to the system. It is the responsibility of the customer to ensure that the outlet is correctly wired and grounded to prevent an electrical shock. (D004)
- **CAUTION**
  Ensure the building power circuit breakers are turned off *before* you connect the power cord(s) to the building power. (C023)

This system relies on branch circuit protection in the building installation for protection against short circuits and earth faults. All protection should comply with local and national electrical codes.

The client’s room emergency power off (EPO) can disconnect power for the entire system (including Front End Node and Service Nodes). The unplugging of the power plug from the mains power receptacle provides a means to remove power from each individual rack. The system power supply circuit breakers can remove power from an individual rack, but they do not remove power to the input terminal blocks.

Blue Gene/L is designed for restricted access locations.

- Only specifically trained personnel should be granted access to the system.
- Access should be controlled via key lock (located on the front and back covers) and only granted by the authority responsible for the installation location.

Stability and weight

Service personnel working on or around this equipment should be aware of the following guidelines:

- Total system weight is between 1000 and 1650 pounds (lb.). Exercise caution when transporting or moving the system, when repositioning the system, or when working on or around the system.
- The system has four full swivel casters for mobility. For maximum stability, the system should only be pushed or rolled in a front to back or back to front direction except during final positioning.
- Exercise caution when moving or rolling the system around raised floor cutouts and other obstructions.
- Ensure all four leveling feet are lowered after final positioning to prevent system from rolling on its casters.
- Plenums and end caps weigh approximately 115 lb. each.

  CAUTION
  The weight of this part or unit is between 32 and 55 kg (70.5 and 121.2 lb.). It takes three persons to lift this part or unit. (C010)

- Bulk power modules (BPM) weigh approximately 16 lb. each and are positioned at a height of six feet when installed in the system (overhead). Ensure proper handling methods and or equipment are used when removing or replacing a BPM.

  CAUTION
  This part or unit is heavy, but has a weight of less than 18 kg (39.7 lb.). Use care when lifting, removing, or installing this part or unit. (C008)

- Front and back covers weigh approximately 33 lb. each.

  CAUTION
  This part or unit is heavy, but has a weight of less than 18 kg (39.7 lb.). Use care when lifting, removing, or installing this part or unit. (C008)

Circuit breakers

The circuit breaker switch located on the front of the systems bulk power enclosure is used to shut down power to the system but does not remove power to the ac terminal blocks where the mains power connects to the bulk power enclosure. To remove all power from the system, disconnect the power cord plug from the main power source (receptacle).

Ac terminal blocks

The system operates on 208V 3P 100A power source.

Ensure all wiring is securely connected to the terminal block and the terminal block shield is securely in place prior to connecting the power cord plug to the mains power source.

DANGER
High voltage present. (L004)

Line cord retention

Ensure proper tightening of the ac line cord strain relief prior to securing the ac terminal block shield.

Bulk power module bay

Limit any action to BPM removal or replacement only.

Hazardous voltage and energy are present in the bulk power enclosure (BPE) through the BPM bay (48 V dc, hazardous energy, 208 V 3P power).

Do not access, probe, or attempt to fix anything beyond the front BPM opening.

DANGER
High voltage present. (L004)

Cover access

In general, hazardous energy may be present when the front or back system cover is opened.

CAUTION
High energy present. (L005)

Fan assembly/cards

Hazardous energy may be present (48 V dc, 2.5 V dc, 1.5 V dc hazardous energy) on cards and midplane.

Do not reach beyond the front of the opening for the fans or for the Service, Node or Link cards.

CAUTION
High energy present. (L005)

D

Restriction of Hazardous Substances Directive (RoHS) compliance

IBM now has a version of Blue Gene/L that is Restriction of Hazardous Substances Directive (RoHS) compliant. Customers can obtain Blue Gene/L systems that comply with the Directive.

RoHS applies to newly built systems after 01 July 2006. Any system sold in the European Union (EU) after 01 July 2006, must be RoHS compliant. However, non-compliant system upgrades and field spares can be used to service or upgrade an existing machine that was put on the market in the EU prior to 01 July 2006.

Each European Union member state will adopt its own enforcement and implementation policies using the directive as a guide. Therefore, there could be as many different versions of the law as there are states in the EU.

IBM will continue to sell systems outside the EU that are non-compliant with RoHS until the inventory of parts is exhausted. Eventually, all Blue Gene/L systems sold will be RoHS compliant.

E

Environment variables

This appendix documents several environment variable that the end user can change to affect the runtime characteristics of the system. This includes subsystems such as the Message Passing Interface (MPI). Usually changes are made in an attempt to improve performance, although on occasion the goal is to modify functional attributes of the application.

Setting environment variables

The easiest and most convenient way to set environment variables is to pass them in on the command line when running the **mpirun** script. For example, if you want to set environment variable “XYZ” to value “ABC,” you can call **mpirun** as follows:

```
mpirun -env "XYZ=ABC" -partition R03 -exe /home/garymu/cpi.rts -cwd /home/garymu/out/
```

Multiple environment variables can be passed by separating them by a space, for example:

```
mpirun -env "XYZ=ABC DEF=123" -partition R03 -exe /home/garymu/cpi.rts -cwd
/home/garymu/out/
```

There are other ways to pass environment variables with **mpirun**. For more information, see *Blue Gene/L: System Administration*, SG24-7178.

BGLMPI_COLLECTIVE_DISABLE

The BGLMPI_COLLECTIVE_DISABLE variable makes it possible to specify whether the optimized collective routines are used, or whether the MPICH code is employed. You usually turn on this variable if unexpected application errors occur that seem to be related to collective operations. Disabling the optimized algorithms and forcing usage of the “safe” MPICH routines might help to determine where the problem lies.

To disable the optimized collective operations, set the BGLMPI_COLLECTIVE_DISABLE environment variable to a value of “1”, for example:

```
mpirun -env "BGLMPI_COLLECTIVE_DISABLE=1" ...
```

Make sure that you remove this environment variable after you solve the problem to ensure optimal performance for your application.

BGLMPI_EAGER, BGLMPI_RVZ, and BGLMPI_RZV

BGLMPI_EAGER, BGLMPI_RVZ, and BGLMPI_RZV are all treated exactly the same by Blue Gene/L. From this point forward, we use only BGLMPI_EAGER to refer to any of the three names.

This variable can be set to an integer that specifies a number of bytes. This value specifies the size of message (in bytes) above which the *rendezvous protocol* is used. Currently, the default value for this is 1000 bytes. Any message that is less than or equal to 1000 bytes is sent by using the *eager protocol*. Messages that are 1001 bytes or greater are sent using the rendezvous protocol.

The eager protocol involves sending the data immediately to the destination, in a more asynchronous manner. With the rendezvous protocol, data is only sent to the destination upon request. In general, the eager protocol is faster but can result in more problems, such as memory issues and link contention.

To better understand the difference between these two protocols, refer to the following Web address about MPI performance topics:

http://www.llnl.gov/computing/tutorials/mpi_performance/

BGL_APP_L1_WRITE_THROUGH

The L1 data cache supports both write-back and write-through modes. In write-back mode, store operations are not necessarily propagated beyond the L1 data cache immediately. Modified data might be kept in the cache and transferred to lower levels of the memory subsystem only when a cache line is evicted. In write-through mode, data is written through to lower levels of the memory subsystem for store operations. If the data also exists in the L1 data cache, it is written to the data cache, and the cache line is marked as clean.

The application can choose write-back or write-through mode for all of application memory. The default is write-back mode. In write-through mode, L1 data cache parity errors can be handled because all cache lines in the cache are clean. But there is additional overhead in using write-through mode that impacts application performance. Users are encouraged to benchmark their applications with various memory configurations to find the right balance between performance and the ability to recover from L1 data cache parity errors.

Note: These performance characteristics might change based on environmental changes such as a larger or smaller partition, the values of other performance-related settings, and so on.

To configure write-through, mode you must set the BGL_APP_L1_WRITE_THROUGH environment variable to 1. The following example shows one way to do this:

```
mpirun -env "BGL_APP_L1_WRITE_THROUGH=1"
```

BGL_APP_L1_WRITEBACK_RECOVERY

For the Compute Node Kernel to recover L1 data cache parity errors when running in write-back mode, additional information needs to be collected from the memory subsystem. The overhead to collect the additional information can impact application performance. The default is not recover L1 data cache parity errors when running in write-back mode. When a parity error occurs, the job is terminated.

When write-back recovery is turned on and the parity error occurs on a clean cache line, the Compute Node Kernel flushes any dirty cache lines from the L1 data cache, invalidates the data cache, and allows the data to be reloaded from lower levels of the memory subsystem. If the parity error occurs on a dirty cache line, the Compute Node Kernel flushes any other dirty cache lines in the L1 data cache, invalidates the data cache, and transfers control to an optional application supplied handler. The handler can try to recalculate the data and resume. If the handler is not specified or the handler cannot recover from the parity error, the job is terminated.

Note: These performance characteristics might change based on environmental changes such as a larger or smaller partition, the values of other performance-related settings, and so on.

To configure write-back recovery, you must set the BGL_APP_L1_WRITEBACK_RECOVERY environment variable to 1. The following example shows one way to do this:

```
mpirun -env "BGL_APP_L1_WRITEBACK_RECOVERY=1"
```

BGL_APP_L1_SWOA

The L1 data cache supports both store with allocate (SWA) and store without allocate (SWOA) modes. In SWA mode, a store operation allocates a line in the L1 data cache when there is a cache miss for a store operation. In SWOA mode, a store operation bypasses the L1 data cache on a cache miss and the data is stored directly to lower levels of the memory subsystem.

The application can choose SWA or SWOA for all of application memory. The default is SWO mode. Some applications can see improved performance with SWOA mode based on how they access data in memory. Users are encouraged to benchmark their applications with various memory configurations to determine which provide the best performance.

Note: These performance characteristics might change based on environmental changes such as a larger or smaller partition, the values of other performance-related settings, and so on.

To configure SWOA mode, you must set the BGL_APP_L1_SWOA environment variable to 1. The following example shows one way to do this:

```
mpirun -env "BGL_APP_L1_SWOA=1"
```

BGLMPI_COPRO_SENDS

The BGLMPI_COPRO_SENDS environment variable controls MPI coprocessor sends. Prior to V1R3, when running in coprocessor mode, responsibility for only message receives was handled by the communications coprocessor; sends were still managed by the core running the user application. By turning the environment variable BGLMPI_COPRO_SENDS on (to a value of 1), the communications subsystem is instructed to hand off responsibility for both the receiving and sending of messages to the communications coprocessor. Depending on the computational and communications mix of the application in question, enabling this function might result in improved overall performance.

Note: This new functionality applies only when the application is being executed in coprocessor mode. If the application is executing in virtual node mode, the setting of environment variable BGLMPI_COPRO_SENDS is ignored.

It is difficult to provide concrete guidelines that specify when turning this environment variable on will result in improved performance. Your best option is to try running your application in both modes and see which performs better.

To configure coprocessor sends, you must set the BGLMPI_COPRO_SENDS environment variable to 1. The following example shows one way to do this:

```
mpirun -env "BGLMPI_COPRO_SENDS=1"
```

BGLMPI_INTERRUPT

The BGLMPI_INTERRUPT environment variable controls interrupt-driven communications. This variable has four options:

Y Turn on both send and receive interrupts.

N Turn off both send and receive interrupts.

S Turn on only send interrupts (for example, `MPI_Send()` can interrupt to copy data to the network).

R Turn on only receive interrupts (for example, data coming in can generate an interrupt to copy the data from the network).

To enable one-sided communication (described in Chapter 8, "One-sided communications" on page 91), interrupt-driven communications support was added to Blue Gene/L. Turning the BGLMPI_INTERRUPT environment variable on allows the MPI communications subsystem to decide to use interrupts if it determines that it might be a faster mechanism, even when executing MPI-only applications.

A possible scenario where this might occur is if there are a "chain" of nodes where the first sends a non-blocking message to the second (using **`isend`**), does some computational work, and then does a wait. The second receives the message from the first node using **`irecv`**, does an **`isend`** to a third node, performs some computational work, and then waits. This then continues for many nodes. Without interrupts, we are likely to see a "stair step" pattern form. With interrupts, it is possible that much more of this activity can be done concurrently.

Note: This setting does not disable one-sided communications mechanisms such as ARMCI and global arrays. These functions require interrupt-driven communication, and interrupts are used even if this environment variable is configured to disable the support.

Again, it is difficult to provide concrete guidelines that specify when turning this environment variable on will result in improved performance. Your best option is to try running your application in both modes and see which performs better.

To configure the desired interrupt mode, you must set the BGLMPI_INTERRUPT environment variable to one of four values. The following example shows one way to do this:

```
mpirun -env "BGLMPI_INTERRUPT=Y"
```

F

L1 data cache parity error recovery: Sample code

This appendix contains sample code that shows how to register a callback that is invoked in the case of a data cache parity error and how to simulate such an error. For more information about this support, see 3.3.3, “Balancing performance and parity error recovery” on page 28.

Example: F-1 Sample code: Data cache parity error callback

```
/* -------------------------------------------------------------  */
/* Product(s):                                                    */
/* 5733-BG1                                                       */
/*                                                                */
/* (C)Copyright IBM Corp. 2004, 2006                              */
/* 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                             */
/* -------------------------------------------------------------  */

static char COPYRIGHT[] = "Licensed Materials - Property  of IBM, "
"5733-BG1 (C) COPYRIGHT 2004, 2004 "
"All Rights Reserved. US Government Users restricted Rights - "
"Use, Duplication or Disclosure restricted by GSA ADP Schedule "
"Contract with IBM Corp.";

/* IMPORTANT *  IMPORTANT * IMPORTANT * IMPORTANT * IMPORTANT *      */
/* You must turn on the BGL_APP_L1_WRITEBACK_RECOVERY environment    */
/* variable for your parity error handler to be invoked.  If you     */
/* register a handler without setting this environment variable, the */
/* handler will successfully register but will never be invoked.  The*/
/* environment variable is not enabled by default because clean cache*/
/* line recovery does take time to perform, and thus has performance */
/* implications that many users will not want to incur.              */
/* If the two following conditions are true, the handler shall       */
/* attempt recovery and terminate the application otherwise:         */
/* 1. The cache line containing the parity error contains only data  */
/* structures owned by the application (eg, not runtime libraries,   */
/* MPI, ESSL, etc).                                                   */
/* 2. The instruction being executed at the time of the parity error */
/* was in the application code (eg, not runtime libraries, MPI,      */
/* ESSL, etc).                                                        */

/* In order to make it possible for the handler to determine         */
/* these conditions, the kernel will pass another parameters to the  */
/* handler that will be the address of the instruction that was      */
/* being executed when the parity error was detected.  Another       */
/* parameter will indicate the EA/VA of the cache line which caused  */
/* the parity error.                                                  */
//
/* Note the EA will be of the start of the cache line which          */
/* contained the error - not the exact address.                       */

/* Only the node on which the parity error occurred will be notified */
/* of the error.  The handler will only be called on that node and   */
/* notified of the parity error.                                      */

/* You must compile this application at optimization level 0 so that */
/* important lines of code do not get optimized out by the compiler. */
/* IMPORTANT *  IMPORTANT * IMPORTANT * IMPORTANT * IMPORTANT *      */


/* The goal of this program is to show how to use and test the new   */
/* functionality in V1R3M0 to set a handler to be called in the event*/
/* of a parity error.  The function we register in this example is   */
/* called "parityhandler()," which simply maintains a count of the   */
/* number of times it has been invoked.  The                          */
/* main function of this application registers the "parityhandler()" */
/* function with the Blue Gene/L CNK, and then simulates             */
/* a parity error using the "rts_change_parity()" system function.   */
/* We then try to read the memory in question; if no handler was     */
/* registered, the application would fail.  Since we have a handler  */
/* in place, we are able to continue.                                 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <rts.h>
#include <string.h>
#include <setjmp.h>


// A buffer of 256 integers (1024 bytes) to be used to simulate
// the parity error in data
union
{
  unsigned int  iarray[256];
} glob __attribute__ ((aligned ( 32)));

// Global variable to keep track of how many times handler
// has been invoked
unsigned int paritycount;

// Required as "dummy" out parameter for interrupt structure;
// not used for parity handlers, but must be passed in
```

```
unsigned int enable_parm;

// Structure to get the data and instruction addresses involved
// in given parity error
struct handler_parms {
    // Note that the ea is *not* the exact address of the
    // byte that caused the parity error.
    // It is the base address of the cache line (which is
    // 32 bytes long) where the parity error occurred.
    unsigned parity_ea;  // effective data address - address
                         // that caused parity error

    // Note that this instruction address may not be the
    // exact address of the instruction that caused
    // the parity error, as it may have been caused by someone
    // else trying to "steal" the cache line.
    unsigned parity_iar; // instruction that was executing - directly
                         // or indirectly caused parity error
  } ;

// Address used to get back to code after handler gets invoked
jmp_buf env;


/* This is the handler we register below.  It will get called in the */
/* event of a parity error. Note that the first two parameters of    */
/* this function are not used for parity error handlers, and thus    */
/* are ignored.  All this handler does is                            */
/* 1. Bump the counter.                                              */
/* 2. Print out the effective data address and instruction address   */
/* involved in the parity error (see handler_parms structure above). */
/* 3. Jump back to application executing when parity error occurred. */
/*                                                                   */
/* IMPORTANT *  IMPORTANT * IMPORTANT * IMPORTANT * IMPORTANT *      */
/* You cannot simply return from a handler.  Executing a return call */
/* with the expectation of getting back to the application that      */
/* caused the handler to be invoked will result in application       */
/* termination.  You must use techniques such as longjmp() and       */
/* setjmp() (not return) to get back to the original application.    */
/* IMPORTANT *  IMPORTANT * IMPORTANT * IMPORTANT * IMPORTANT *      */
void parityhandler(unsigned int interrupts, unsigned int *enabled, void *parms)
{
   paritycount += 1;  // bump to keep track of count
   // cast parms to handler variable
   handler_parms * handler_input =  (handler_parms *) parms;
   printf("In user parity handler EA =%x IAR =  %x\n",
handler_input->parity_ea,handler_input->parity_iar );
   longjmp(env,11);
   return;
}


/* This is the main application.  As described above, it registers   */
/* our handler ("parityhandler()") and then simulates a parity error.*/
/* The handler should get invoked, which allows application recovery.*/
/* Otherwise the application would be terminated.                    */
int main(int argc, char **argv)
{
   // Structure used to register handler.  In rts.h
   struct interrupt intrinfo;
```

```
// BG/L personality - in rts.h
BGLPersonality pers;
rts_get_personality(&pers, sizeof(pers));  // Get personality

// We will use the 47th integer to simulate the parity error
const int INT_TO_TOUCH = 46;

int rc;  // return code

// Get x, y and Z coordinates of this application to find out what node
// we are running on.
const unsigned x = pers.xCoord, xsize = pers.xSize;
const unsigned y = pers.yCoord, ysize = pers.ySize;
const unsigned z = pers.zCoord, zsize = pers.zSize;

// Only execute this code on one compute node
if (x == 1 && y == 0 && z == 0)
   {
     // Set return address.  This is where control will come
     // back to after handler is invoked.
if(setjmp(env) == 0) {
        // This is hit when setjmp() is first called to
        // set the return point
}
else {
       // This else is called when a return is actually made
       // from a longjmp called in the handler itself
  printf(" longjump target \n");
  return 12;
}

     //  Set up interrupt structure to register our handler
intrinfo.group = INTERRUPT_PARITY; // set type of handler
intrinfo.interrupts = 0;
intrinfo.flags = 0;
intrinfo.handler = parityhandler;  // this is the address of our
                                        // handler, defined above
intrinfo.enabled = &enable_parm;   // out parm, not used in parity
                                        // but needs to be set

// Set and enable an interrupt handler for the parity group
rc = rts_interrupt_control(INTERRUPT_SET_ENABLE, &intrinfo, sizeof(intrinfo));
if (rc == 0) {
       // Success
  printf("rts_interrupt_control(SET_ENABLE)=ok\n");
}
else {
       // Failure registering handler - exit with error return code
  printf("rts_interrupt_control(SET_ENABLE)=error, rc=%d errno=%d\n", rc, errno);
  return 1; // return any non-zero on failure
}

     // Initialize buffer to zeros
     int i;
     for (i = 0; i < 256; i++) {
        glob.iarray[i] = 0;
     }

printf("Modify integer at = %8.8x\n", &glob.iarray[INT_TO_TOUCH]);
```

```
   // Set our testing integer to all ones
   glob.iarray[INT_TO_TOUCH] = 0xFFFF;

   // Simulate parity error
   rts_change_parity(RTS_LINE_DATA_DIRTY, &glob.iarray[INT_TO_TOUCH]);
   int sum = 0;

   // this should cause handler to be invoked
   sum = glob.iarray[INT_TO_TOUCH];
        printf("It this got printed, our handler was *not* invoked: rc=%d\n", rc);
        return sum;  // We return sum so that compiler doesn't
                     // optimize out the memory touch above...
      }
   // Not rank 0
   else
     {
       return 0;
     }
}
```

Glossary

32b executable Executable binaries (user applications) with 32b (4B) virtual memory addressing. Note that this is independent of the number of bytes (4 or 8) used for floating-point number representation and arithmetic.

32b floating-point arithmetic Executable binaries (user applications) with 32b (4B) floating-point number representation and arithmetic. Note that this is independent of the number of bytes (4 or 8) used for memory reference addressing.

32b virtual memory addressing All virtual memory addresses in a user application are 32b (4B) integers. Note that this is independent of the type of floating-point number representation and arithmetic.

64b executable Executable binaries (user applications) with 64b (8B) virtual memory addressing. Note that this is independent of the number of bytes (4 or 8) used for floating-point number representation and arithmetic. Also, all user applications should be compiled, loaded with subcontractor-supplied libraries, and executed with 64b virtual memory addressing by default.

64b floating-point arithmetic Executable binaries (user applications) with 64b (8B) floating-point number representation and arithmetic. Note that this is independent of the number of bytes (4 or 8) used for memory reference addressing.

64b virtual memory addressing All virtual memory addresses in a user application are 64b (8B) integers. Note that this is independent of the type of floating-point number representation and arithmetic. Also all user applications should be compiled, loaded with subcontractor-supplied libraries, and executed with 64b virtual memory addressing by default.

Advanced Simulation and Computing Program (ASCI) Administered by Department of Energy (DOE)/National Nuclear Security Agency (NNSA).

API See *application programming interface*.

application programming interface (API) Defines the syntax and semantics for invoking services from within an executing application. All APIs shall be available to both Fortran and C programs, although implementation issues, such as whether the Fortran routines are simply wrappers for calling C routines, are up to the supplier.

Application Specific Integrated Circuit (ASIC) Includes two 32-bit PowerPC cores (the 440) that was developed by IBM for embedded applications.

ASCI See *Advanced Simulation and Computing Program*.

ASIC See *Application Specific Integrated Circuit*.

BGL See *Blue Gene/L*.

BGL8K The Phase 1 build of Blue Gene/L, which contains 8192 Compute Nodes (CN), 128 I/O Nodes, one-eighth of the I/O subsystem and the all of the Front End Nodes.

BGL Compute ASIC (BLC) This high-function Blue Gene/L ASCI is the basis of the Compute Nodes and I/O Nodes.

BGL Link (BLL) ASIC This high-function Blue Gene/L ASCI is responsible for redriving communication signals between midplanes and is used to repartition Blue Gene/L.

bit (b) A single, indivisible binary unit of electronic information.

BLC See *BGL Compute ASIC*.

BLL BGL Link.

Blue Gene/L (BGL) The name given to the collection of Compute Nodes, I/O Nodes, Front End Nodes (FEN), file systems, and interconnecting networks that is the subject of this statement of work.

byte (B) A collection of eight bits.

central processing unit (CPU) or processor A VLSI chip that constitutes the computational core (integer, floating point, and branch units), registers, and memory interface (virtual memory translation, TLB and bus controller).

cluster A set of nodes connected via a scalable network technology.

Cluster Monitoring and Control System (CMCS)

Cluster Wide File System (CWFS) The file system that is visible from every node in the system with scalable performance.

CMCS Cluster Monitoring and Control System.

CMN See *Control and Management Network*.

CN See *Compute Node*.

compute card One of the field replaceable units (FRUs) of Blue Gene/L. Contains two complete Compute Nodes, and is plugged into a node card.

Compute Node (CN) The element of Blue Gene/L that supplies the primary computational resource for execution of a user application.

Control and Management Network (CMN) Provides a command and control path to Blue Gene/L for functions such as health status monitoring, repartitioning, and booting.

Core Subcontractor delivered hardware and software. The Blue Gene/L Core consists of the Blue Gene/L Compute Main Section, Front End Node, Service Node (SN), and a control and management Ethernet.

CPU See *central processing unit*.

current standard (as applied to system software and tools) Applies when an API is not “frozen” on a particular version of a standard, but shall be upgraded automatically by the subcontractor as new specifications are released. For example, *MPI version 2.0* refers to the standard in effect at the time of writing this document, while *current version of MPI* refers to further versions that take effect during the lifetime of this contract.

CWFS See *Cluster Wide File System*.

DDR See *Double Data Rate*.

Double Data Rate (DDR) A technique for doubling the switching rate of a circuit by triggering on both the rising edge and falling edge of a clock signal.

EDRAM See *enhanced dynamic random access memory*.

enhanced dynamic random access memory (EDRAM) Dynamic random access memory that includes a small amount of static random access memory (SRAM) inside a larger amount of DRAM. Performance is enhanced by organizing so that many memory accesses are to the faster SRAM.

ETH The ETH is a high-function Blue Gene/L ASIC that is responsible for Ethernet-to-JTAG conversion and other control functions.

Federated Gigabit-Ethernet Switch (FGES) Connects the I/O Nodes of Blue Gene/L to external resources, such as the FEN and the CWFS.

FEN See *Front End Node*.

FGES See *Federated Gigabit-Ethernet Switch*.

Field Replaceable Unit (FRU)

Floating Point Operation (FLOP or OP) Plural is FLOPS or OPS.

FLOP or OP See *Floating Point Operation*.

FLOP/s or OP/s Floating Point Operation per second.

Front End Node (FEN) Is responsible, in part, for interactive access to Blue Gene/L.

FRU Field Replaceable Unit.

fully supported (as applied to system software and tools) Refers to product-quality implementation, documented and maintained by the HPC machine supplier or an affiliated software supplier.

GFLOP/s, GOP/s, gigaFLOP/s A billion (109 = 1 000 000 000) 64-bit floating point operations per second.

gibibyte (GiB) A billion base 2 bytes. This is typically used in terms of RAM and is 230 (or 1 073 741 824) bytes. For a complete description of SI units for prefixing binary multiples, see:
http://physics.nist.gov/cuu/Units/binary.html

gigabyte (GB) A billion base 10 bytes. This is typically used in every context except for RAM size and is 109 (or 1 000 000 000) bytes.

host complex Includes the Front End Node and Service Node.

HSN Hot Spare Node.

Internet Protocol (IP) The method by which data is sent from one computer to another on the Internet.

IP Internet Protocol.

job A cluster wide abstraction similar to a POSIX session, with certain characteristics and attributes. Commands shall be available to manipulate a job as a single entity (including kill, modify, query characteristics, and query state).

input/output (I/O) Describes any operation, program, or device that transfers data to or from a computer.

I/O card One of the FRUs of Blue Gene/L. An I/O card contains two complete I/O Nodes and is plugged into a node card.

I/O Node (ION) Are responsible, in part, for providing I/O services to Compute Nodes.

International Business Machines Corporation (IBM)

ION See *I/O Node.*

limited availability Represents an intermediate operational level of major computing systems at LLNL. Limited availability is characterized by system access limited to a select set of users, with reduced system functionality.

LINPACK A collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems.

Linux A free UNIX®-like operating system originally created by Linus Torvalds with the assistance of developers around the world. Developed under the GNU General Public License, the source code for Linux is freely available to everyone.

Mean Time Between Failure (MTBF) A measurement of the expected reliability of the system or component. The MTBF figure can be developed as the result of intensive testing, based on actual product experience, or predicted by analyzing known factors. See:
`http://www.t-cubed.com/faq_mtbf.htm`

mebibyte (MiB) A million base 2 bytes. This is typically used in terms of Random Access Memory and is 220 (or 1 048 576) bytes. For a complete description of SI units for prefixing binary multiples, see:
`http://physics.nist.gov/cuu/Units/binary.html`

megabyte (MB) A million base 10 bytes. This is typically used in every context except for RAM size and is 106 (or 1 000 000) bytes.

Message Passing Interface (MPI)

MFLOP/s, MOP/s, or megaFLOP/s A million (106 = 1 000 000) 64-bit floating point operations per second.

midplane An intermediate packaging component of Blue Gene/L. Multiple node cards plug into a midplane to form the basic scalable unit of Blue Gene/L.

MPI See *Message Passing Interface*.

MPICH2 MPICH is an implementation of the MPI standard available from Argonne National Laboratory.

MTBF See *Mean Time Between Failure*.

node Operates under a single instance of an operating-system image and is an independent operating-system partition.

node card An intermediate packaging component of Blue Gene/L. FRUs (compute cards and I/O cards) are plugged into a node card. Multiple node cards plug into a midplane to form the basic scalable unit of Blue Gene/L.

OCF See *Open Computing Facility*.

Open Computing Facility (OCF) The unclassified partition of Livermore Computing, the main scientific computing complex at LLNL.

OpenMP A portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications.

peak rate The maximum number of 64-bit floating point instructions (add, subtract, multiply or divide) per second that can conceivably be retired by the system. For RISC CPUs, the peak rate is calculated as the maximum number of floating point instructions retired per clock times the clock rate.

PTRACE A facility that allows a parent process to control the execution of a child process. Its primary use is for the implementation of breakpoint debugging.

published (as applied to APIs) Refers to the situation where an API is not required to be consistent across platforms. A "published" API refers to the fact that the API shall be documented and supported, although it by a subcontractor or platform specific.

Purple ASCI Purple is the fourth generation of ASCI platforms.

RAID See *redundant array of independent disks.*

RAM See *random access memory.*

random access memory (RAM) Computer memory in which any storage location can be accessed directly.

RAS See *reliability, availability, and serviceability.*

redundant array of independent disks (RAID) A collection of two or more disk physical drives that present to the host an image of one or more logical disk drives. In the event of a single physical device failure, the data can be read or regenerated from the other disk drives in the array due to data redundancy.

reliability, availability, and serviceability (RAS) Include those aspects of hardware and software design and development, solution design and delivery, manufacturing quality, technical support service and other services which contribute to assuring that the IBM offering will be available when the client wants to use it; that it will reliably perform the job; that if failures do occur, they will be nondisruptive and be repaired rapidly and that after repair the user might resume operations with a minimum of inconvenience.

SAN See *storage area network*.

scalable A system attribute that increases in performance or size as some function of the peak rating of the system. The scaling regime of interest is at least within the range of 1 teraflop/s to 60.0 (and possibly to 120.0) teraflop/s peak rate.

SDRAM See *synchronous, dynamic random access memory*.

Service Node Is responsible, in part, for management and control of Blue Gene/L.

service representative On-site hardware expert who performs hardware maintenance with DOE Q-clearance.

single-point control (as applied to tool interfaces) The ability to control or acquire information about all processes or PEs using a single command or operation.

Single Program Multiple Data (SPMD) A programming model wherein multiple instances of a single program operate on multiple data.

SMFS See *System Management File System*.

SMP See *symmetric multiprocessor*.

SNL See *Sandia National Laboratories*.

SOW See *Statement of Work*.

SPMD See *Single Program Multiple Data*.

sPPM This is a benchmark that solves a 3D gas dynamics problem on a uniform Cartesian mesh, using a simplified version of the Piecewise Parabolic Method (PPM) code.

SRAM static random access memory.

standard (as applied to APIs) Where an API is required to be consistent across platforms, the reference standard is named as part of the capability. The implementation shall include all routines defined by that standard, even if some simply result in no-ops on a given platform.

Statement of Work (SOW) This document is a statement of work. A document prepared by a Project Manager (PM) as a response to a Request for Service from a client. The project SOW is the technical solution proposal, and it should describe the deliverables and identify all Global Services risks and impacts, infrastructure investments, capacity, cost elements, assumptions and dependencies.

static random access memory (SRAM)

storage area network (SAN) A high-speed subnetwork of storage devices.

symmetric multiprocessor (SMP) A computing node in which multiple functional units operate under the control of a single operating-system image.

synchronous, dynamic random access memory (SDRAM) A type of dynamic random access memory (DRAM) with features that make it faster than standard DRAM.

System Management File System (SMFS) Provides a single, central location for administrative information about Blue Gene/L.

TCP/IP See *Transmission Control Protocol/Internet Protocol*.

tebibyte (TiB) A trillion bytes base 2 bytes. This is typically used in terms of Random Access Memory and is 240 (or 1099511627776) bytes. For a complete description of SI units for prefixing binary multiples, see:
`http://physics.nist.gov/cuu/Units/binary.html`

terabyte (TB) A trillion base 10 bytes. This is typically used in every context except for Random Access Memory size and is 1012 (or 1000000000000) bytes.

teraflop/s (TFLOP/s) A trillion (1012 = 1000000000000) 64-bit floating point operations per second.

tori The plural form of the word *torus*.

torus network Each processor is directly connected to six other processors: two in the “X” dimension, two in the “Y” dimension, and two in the “Z” dimension. One of the easiest ways to picture a torus is to think of a 3-D “cube” of processors, where every processor on an edge has “wraparound” connections to link to other similar edge processors.

TotalView A parallel debugger from Etnus LLC, Natick, MA.

Transmission Control Protocol/Internet Protocol (TCP/IP) The suite of communications protocols used to connect hosts on the Internet.

Tri-Lab Includes Los Alamos National Laboratory, Lawrence Livermore National Laboratory, and Sandia National Laboratories.

UMT2000 The UMT benchmark is a 3D, deterministic, multigroup, photon transport code for unstructured meshes.

Unified Parallel C (UPC) A programming language with parallel extensions to ANSI C. For an example, see:
`http://upc.gwu.edu/`

University Alliances Members of the Academic Strategic Alliances Program (ASAP) of ASCI, academic institutions engaged in accelerating simulation science.

UPC See *Unified Parallel C*.

XXX-compatible (as applied to system software and tool definitions) Requires that a capability be compatible, at the interface level, with the referenced standard, although the lower-level implementation details will differ substantially. For example, *NFSv4-compatible* means that the distributed file system shall be capable of handling standard NFSv4 requests, but need not conform to NFSv4 implementation specifics.

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 publications

For information about ordering these publications, see "How to get IBM Redbooks publications" on page 168. Some of the documents referenced here might be available only in softcopy.

- *Blue Gene/L: Hardware Overview and Planning*, SG24-6796
- *Blue Gene/L: System Administration*, SG24-7178
- *Linux Clustering with CSM and GPFS*, SG24-6601
- *Workload Management with LoadLeveler*, SG24-6038

Other publications

These publications are also relevant as further information sources:

- *General Parallel File System (GPFS) for Clusters: Concepts, Planning, and Installation,* GA22-7968
- *IBM General Information Manual, Installation Manual-Physical Planning,* GC22-7072
- *LoadLeveler for AIX 5L and Linux V3.2 Using and Administering,* SA22-7881
- *PPC440x5 CPU Core User's Manual*

  `http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core`
- *Exploiting the Dual Floating Point Units in Blue Gene/L*

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

Online resources

These Web sites and URLs are also relevant as further information sources:

- MPI Performance Topics

  `http://www.llnl.gov/computing/tutorials/mpi_performance/`
- MPI-2 Reference

  `http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm`
- Etnus TotalView

  `http://www.etnus.com/`
- GDB: The GNU Project Debugger

  `http://www.gnu.org/software/gdb/`
- SUSE Linux Enterprise Server

  `http://www.novell.com/products/linuxenterpriseserver/`

- XL C/C++

  http://www.ibm.com/software/awdtools/xlcpp/library/
- XL Fortran

  http://www.ibm.com/software/awdtools/fortran/xlfortran/library/

How to get IBM Redbooks publications

You can search for, view, or download IBM Redbooks, Redpapers, Technotes, and draft publications, and Additional materials, as well as order hardcopy IBM Redbooks pubications, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

IBM Redbooks

IBM System Blue Gene Solution: Application Development

(0.2”spine)
0.17”<->0.473”
90<->249 pages

IBM System Blue Gene Solution: Application Development

Understand the Blue Gene/L programming environment

Learn how to run and debug MPI programs

Learn about checkpoint/restart, Bridge APIs, and more

This IBM Redbooks publication is one in a series of IBM publications written specifically for the IBM System Blue Gene Solution, Blue Gene/L, which was developed by IBM in collaboration with Lawrence Livermore National Laboratory (LLNL). It provides an overview of the application development environment for Blue Gene/L.

This book explains the instances where Blue Gene/L is unique in its programming environment. It does not delve into great depth about the technologies that are commonly used in the supercomputing industry, such as Message Passing Interface (MPI) and Aggregate Remote Memory Copy Interface (ARMCI). References are provided in those instances so you can find more information if desired.

Prior to reading this book, you must have a strong background in MPI programming.

SG24-7179-04 ISBN 0738489395