IBM

Blue Gene/L: Application Development

- Explore the Blue Gene/L programming environment
- Learn how to run and debug MPI programs
- Understand checkpoint and restart, Bridge APIs, and more

Gary L. Mullen-Schultz

Redbooks

ibm.com/redbooks

International Technical Support Organization

Blue Gene/L: Application Development

July 2005

ZG24-6745-00

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

First Edition (July 2005)

This edition applies to Version 1, Release 1, Modification 0 of Blue Gene/L (product number 5733-BG1).

Contents

Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

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

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

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

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

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

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

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

Trademarks

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

@server®	DB2®	Redbooks (logo) ™
iSeries™	IBM®	Redbooks™
pSeries®	LoadLeveler®	Tracer™
AIX 5L™	PowerPC®	WebSphere®
AIX®	POWER™	

iThe following terms are trademarks of other companies:

Java, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation 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® Redbook is the second in a series of internal IBM publications written specifically for the Blue Gene/L supercomputer, which was developed by IBM in collaboration with Lawrence Livermore National Laboratory (LLNL). This redbook provides an overview of the application development environment for Blue Gene/L.

This redbook explains the instances where Blue Gene/L is unique in its programming environment. The book is divided into the following parts:

- Part 1, “MPI application information” on page 1
- Part 2, “System application information” on page 69
- Part 3, “Performance analysis” on page 93

Prior to reading this book, you must have a strong background in Message Passing Interface (MPI) programming.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, 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 redbook. Gary also focuses on Java™ and WebSphere®. He is a Sun™ Certified Java Programmer, Developer and Architect, and has three issued patents.

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

Charles Archer
Peter Bergner
Lynn Boger
Mike Brutman
Jay Bryant
Kathy Cebell
Jeff Chauvin
Roxanne Clarke
Darwin Dumonceaux
Mike Hjalmervik
Frank Ingram

Kerry Kaliszewski
Brant Knudson
Glenn Leckband
Dave Limpert
Chris Marroquin
Randall Massot
Curt Mathiowetz
Mark Megerian
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 Redbook dealing 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 Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

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

  ibm.com/redbooks

- Send your comments in an email to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. JLU Building 107-2
  3605 Highway 52N
  Rochester, Minnesota 55901-7829

Part 1

MPI application information

This part provide details about application programming interfaces (APIs) and other constructs that would be of interest to an application programmer writing a Message Passing Interface (MPI) application for the Blue Gene/L system. This part encompasses the following chapters:

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 MPI 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 important attributes of the MPI 1.2 standard that are supported by Blue Gene/L include:

- There is no one-sided communication. Only point-to-point is supported. That is, an MPI_Send operation must be matched in another node (rank) by an MPI_Recv.
- No spawning of other processes is allowed.
- The thread model supported is MPI_THREAD_SINGLE, which means that threads cannot be created.

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 (CNK), 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 the area used for communications. CNK 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*, SG24-6742.

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 may 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 may 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 your application runs well in Communication Coprocessor Mode.

1.3 Other considerations

It is important to understand that the operating system present on the Compute Node, the kernel (CNK), 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 Compute Node Kernel” on page 19.

1.3.1 Input/output

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 will result 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 20.

Standard input (stdin)

Standard input (stdin) is not supported on Blue Gene/L. If you need to pass input to your application, you need to do so by using 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 20.

1.3.2 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.”

1.4 Include and link files

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

1.4.1 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
BGLGi.h	
BGLML.h	
BGLMLLog.h	
BGLML_Lockbox.h	
BGLML_Message.h	
BGLMP_1P.h	
BGLMP_Adaptive.h	
BGLMP_Alltoall.h	
BGLMP_Eager.h	
BGLMP_RectAllreduce.h	
BGLMP_RectAllreduce1P.h	
BGLMP_RectBarrier.h	
BGLMP_RectBcast.h	
BGLMP_RectCommunicator.h	
BGLMP_RectReduce.h	
BGLMP_Rendezvous.h	
BGLMP_SimplePut.h	
BGLMP_VNMode.h	
IntHash.h	
MMCSranktable.h	
Queue.h	
attach_bgl.h	Provides data structures needed for debuggers to connect to **mpirun**.
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.
bgl_errors.h	
bgl_perfctr.h	Function header file for the universal performance counters.
bgl_perfctr_events.h	Event list header file for the universal performance counters.
bglaccessmac.h	

File name	Description
bgldcr.h	
bgldcrnames.h	
bgldcrprint.h	
bglddrdcr.h	
bglfpudcr.h	
bglgi.h	
bglic.h	
bglicdcr.h	
bglidochip.h	
bgll2dcr.h	
bgll3dcr.h	
bgllinkchip.h	
bgllinkchippersonality.h	
bgllockbox.h	
bgllockboxdcr.h	
bglmailbox.h	
bglmaldcr.h	
bglmccu.h	
bglmemmap.h	
bglpacket.h	
bglpersonality.h	
bglplbdcr.h	
bglsim_counters.h	
bglsp_lib.h	
bglsramdcr.h	
bgltestdcr.h	
bgltimebase.h	
bgltoruscapdcr.h	
bgltorusdcr.h	
bgltoruspacket.h	
bgltreecapdcr.h	
bgltreedcr.h	
bgltreepacket.h	
bgluicdcr.h	

File name	Description
bglupc.h	
bglupcdcr.h	
idoerrors.h	
idoproxy_lib.h	
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.
mpirun_common_rc.h	
mpirun_io.h	
mpirun_mtype.h	
mpirun_protocol.h	
rm_api.h	
rts.h	
sayMessage.h	
standalone.h	

1.4.2 Static link files

Link files for Blue Gene/L are in the /bgl/BlueLight/ppcfloor/bglsys/lib directory. Table 1-2 lists the static link files.

Table 1-2 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.
libbglmachine.a	
libbglsim_counters.440.a	
libbglsim_counters.rts.a	

File name	Description
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.
libdbbringup.a	
libdevices.440.a	
libdevices.rts.a	
libfmpich.rts.a	Contains the Fortran bindings for MPICH. Required for Fortran MPI applications.
libidoproxy.a	
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.
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	
libstandalone.440.a	
libstandalone.440sysbr.a	
modules	
ppc440_reset.lds	

1.5 Compilers overview

Two compiler families are supported on Blue Gene/L: GNU compilers and IBM XL compilers. You can find instructions for installation in *Blue Gene/L: System Administration*, ZG24-6744.

1.5.1 Programming environment overview

The following diagram provides a quick view into the software stack that supports the execution of Blue Gene/L applications.

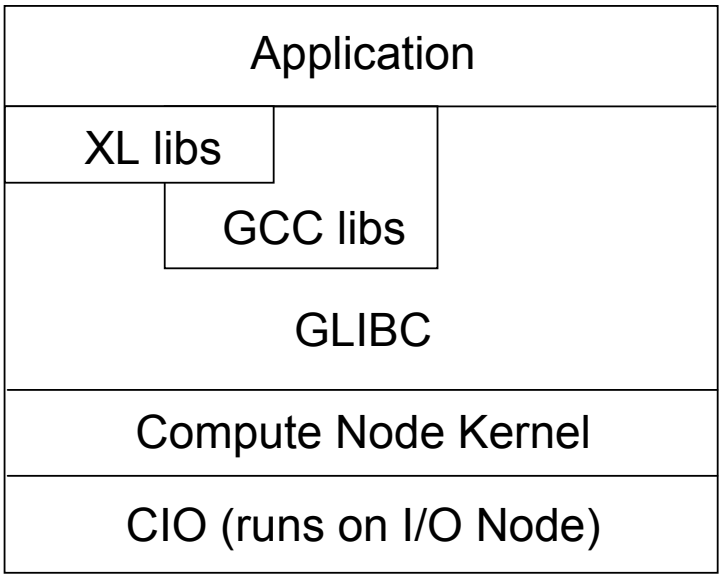

Figure 1-1 Software stack supporting execution of Blue Gene/L applications

1.5.2 GNU

The standard GNU version 3.2 C, C++ and Fortran77 compilers are modified during installation to support Blue Gene/L. Currently, version 2.2.5 of GLIBC is supported.

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

GNU runtime libraries

The following libraries are linked into your application automatically by the GNU linker when you create your application. We have separated them into the GCC libraries (Table 1-3) and GLIBC libraries (Table 1-4).

Table 1-3 GNU GCC libraries

File name	Description
libstdc++.a	GNU Standard C++ library
libgcc.a	GCC low-level run-time library
libg2c.a	G77 run-time library

Table 1-4 GNU GLIBC libraries

File name or names	Description
libc.a	GNU C library
libm.a	Math library
libieee.a	IEEE floating point library
libg.a	G++ runtime library
libcrypt.a	Cryptography library
libnss_dns.a, libnss_files.a, libresolv.a	NSS/Resolve libraries

1.5.3 IBM XL Compilers

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

- XL C/C++ Advanced Edition V7.0 for Linux
- XL Fortran Advanced Edition V9.1 for Linux

See Chapter 4, “Developing applications with IBM XL compilers” on page 25, for more information.

Note: Support for creating applications targeting both Linux on (IBM @server pSeries®), and Blue Gene/L is provided.

The XL C compilers are in the following directories:

- Blue Gene/L version

  /opt/ibmcmp/vac/7.0/bin/blrts_xlc
- Linux version

  /opt/ibmcmp/vac/7.0/bin/xlc

The XL C++ compilers are in these directories:

- Blue Gene/L version

  /opt/ibmcmp/vacpp/7.0/bin/blrts_xlc++
- Linux version

  /opt/ibmcmp/vacpp/7.0/bin/xlc++

The XL Fortran compilers are in the following locations:

- Blue Gene/L version

  /opt/ibmcmp/xlf/9.1/bin/blrts_xlf
- Linux version

  /opt/ibmcmp/xlf/9.1/bin/xlf

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.

MASS libraries

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

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

XL runtime libraries

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

Attention: The exception to this statement is for 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(), etc.
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
libmassv.a	IBM XL MASSV library: Vector intrinsic functions
ibxlomp_ser.a	IBM XL Open MP compatibility library

2

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* that both processors have access to.

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 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 Choosing modes

You choose 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*, ZG24-6744, for more information about the **mpirun** command.

3

System calls supported by Compute Node Kernel

This chapter discusses the system calls that are supported by the Compute Node Kernel (CNK). It is important to understand which functions can be called, and perhaps more importantly, which ones cannot be called, by your application running on 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 CNK 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 CNK is not intended to run system administration functions from the compute node.

Important: Since the CNK is a single-process operating system, no support is provided for an application to use multiple processes or threads. Calls to such functions as fork() return -1, with errno set to ENOSYS.

To achieve the best reliability, a small and simple kernel is a design goal. This enables a simpler checkpoint function. See Chapter 6, “Checkpoint and restart support” on page 61.

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.

Restriction: No shell utilities are supported in the CNK. Only the system calls listed in this document are supported in the CNK. For example, no utility commands provided by such shells as BASH or Bourne can be invoked by applications running in the CNK.

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

3.2.1 Return codes

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

3.2.2 List of supported system calls

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

Important: 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.

Table 3-1 List of supported system calls

System call	Category	Description
access	File I/O	Determine accessibility of a file
brk	Miscellaneous	Change data segment size
chdir	Directory	Change working directory
chmod	File I/O	Change mode of a file
chown	File I/O	Change owner and group of a file
close	File I/O	Close a file descriptor
connect	Sockets	Connect a socket
dup	File I/O	Duplicate an open descriptor
dup2	File I/O	Duplicate an open descriptor
exit	Miscellaneous	Terminate a process
fchmod	File I/O	Change mode of a file
fchown	File I/O	Change 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	Get file status
fstat64	File I/O	Get file status
fsync	File I/O	Synchronize changes to a file
ftruncate	File I/O	Truncate a file to a specified length
ftruncate64	File I/O	Truncate a file to a specified length
getcwd	Directory	Get the path name of the current working directory
getdents	Directory	Get directory entries
getdents64	Directory	Get directory entries
getgid	Process Info	Get the real group ID
getitimer	Time	Get the value of the interval timer
getpeername	Sockets	Get the name of the peer socket
getpid	Process Info	Get the process ID
getrusage	Process Info	Get information about resource utilization. All time reported is attributed to the user application, meaning that CNK time is included in the values returned.
getsockname	Sockets	Get the socket name

System call	Category	Description
gettimeofday	Time	Get the date and time
getuid	Process Info	Get the real user ID
kill	Signal	Send a `kill` command to the currently running process, for example, to yourself
lchown	File I/O	Change owner and group of a symbolic link
link	File I/O	Link to a file
lseek	File I/O	Move the read/write file offset
llseek	File I/O	Move the read/write file offset
lstat	File I/O	Get symbolic link status
lstat64	File I/O	Get symbolic link status
mkdir	Directory	Make a directory
rpen	File I/O	Open a file
read	File I/O	Read from a file
readlink	File I/O	Read the contents of a symbolic link
readv	File I/O	Read a vector
recv	Sockets	Receive a message from a connected socket
recvfrom	Sockets	Receive a message from a socket
rename	File I/O	Rename a file
rmdir	Directory	Remove a directory
send	Sockets	Send a message on a connected socket
sendto	Sockets	Send a message on a socket
setitimer	Time	Set value of interval timer. Only the following operations are supported: ► ITIMER_PROF ► ITIMER_REAL **Note**: An application can only set one active timer at a time.
signal	Signals	Signal management
sigreturn	Signals	Return from a signal handler
socket	Sockets	Open a socket
stat	File I/O	Get file status
stat64	File I/O	Get file status
symlink	File I/O	Make a symbolic link to a file
time	Time	Get time
times	Process Info	Get process times. All time reported is attributed to the user application, meaning that CNK time is included in the values returned.
truncate	File I/O	Truncate a file to a specified length

System call	Category	Description
truncate64	File I/O	Truncate a file to a specified length
umask	File I/O	Set and get the file mode creation mask
uname	Miscellaneous	Get the name of the current system, and other information (for example, version and release).
unlink	File I/O	Remove a directory entry
utime	File I/O	Set file access and modification times
write	File I/O	Write to a file
writev	File I/O	Write a vector

3.3 Unsupported calls

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

- Blue Gene/L does not support the use of the system() function. Therefore, for example, you can't use something like 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 sigaction(). It also doesn't support calls to signal(SIGTRAP, xl__trce) or signal(SIGNAL, xl__trbk).
- Calls to usleep() are not supported.

4

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 is comprised of the following products:

- XL C/C++ Advanced Edition V7.0 for Linux
- XL Fortran Advanced Edition V9.1 for Linux

The information presented in this document is Blue Gene/L specific. It does not include general XL compiler information. For complete documentation about these compilers, see the following Web pages:

- 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.

4.1 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 at the Web pages listed in the introduction to this chapter:

- *XL Fortran User Guide*
- *XL C/C++ Compiler Reference*

4.2 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 5.1, “Running applications” on page 54, to learn how to run programs on Blue Gene/L.

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

- `-qbgl`: 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 floating-point unit (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 may 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.

There are scripts 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 4-1.

Table 4-1 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 `-qfltrap` option, which traps floating-point exceptions, is disabled by default. If you enable this option, `-qarch` is automatically reset to `-qarch=440`.

4.3 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.

4.4 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.

4.5 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 "Optimizing your applications" in the *XL C/C++ Programming Guide* and "Optimizing XL Fortran programs" in the *XL Fortran User's Guide* (refer to the Web links provided in the introduction to this chapter).

4.6 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 4-1 allows each pair of elements in a structure to be operated on in parallel.

Example 4-1 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 actually need to represent complex numbers. In fact, both elements are represented internally as two real values. See 4.13, “Complex type manipulation functions” on page 36, 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.

4.7 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 NaN (not-a-number) 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 4.9, “Removing possibilities for aliasing (C/C++)” on page 29, for additional strategies for removing data dependencies.)

4.8 Using inline functions

An inline function is expanded in any context in which it is called. This 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 use 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 *XL Fortran User Guide* and *XL C/C++ Compiler 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, see the *XL C/C++ Language Reference.*

4.9 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 following code example:

```
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 4-2.

Example 4-2 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 4-3 indicates to the compiler that the pointers-to-arrays of double `x` and `f` do not share memory.

Example 4-3 #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.

4.10 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; that is, 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 4-4 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 4-4 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 4-5 shows “wrapping” the inlined, optimized `ten_reciprocal_roots` function, in Example 4-4, 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 4-5 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++;
    }
}
```

4.11 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 may 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.

The C/C++ prototype for the function is:

```
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 4-6 asserts that the variables *x* and *f* are aligned along 16-byte boundaries.

Example 4-6 Using the __alignx built-in function

```
#include <math.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 4-7 checks to see whether the incoming values are correctly aligned. Then it calls the “aligned” (Example 4-6) or “unaligned” (Example 4-4) version of the function according to the result.

Example 4-7 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 4-7 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).

4.12 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 (available from the Web links listed at the beginning of this chapter):

- Appendix B: “Built-In Functions” in *XL C/C++ Compiler Reference*
- “Intrinsic Procedures” in *XL Fortran Language Reference*

In addition, on Blue Gene/L, the XL compilers provide a set of built-in functions that are specifically optimized for the PowerPC 440d’s 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 actually 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 4-1 illustrates how a parallel multiply operation is performed.

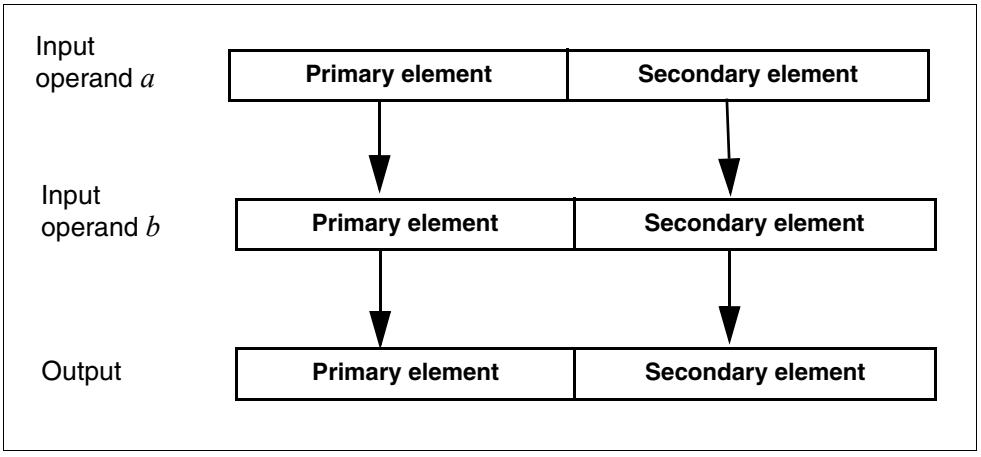

Figure 4-1 Parallel operations

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 4-2 illustrates how a cross multiply operation is performed.

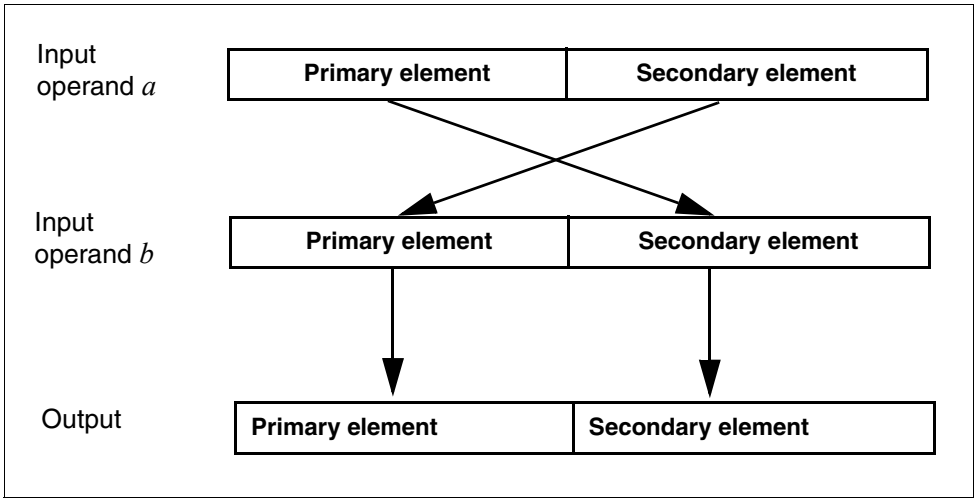

Figure 4-2 Cross operations

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 4-3 illustrates how a cross-primary multiply operation is performed.

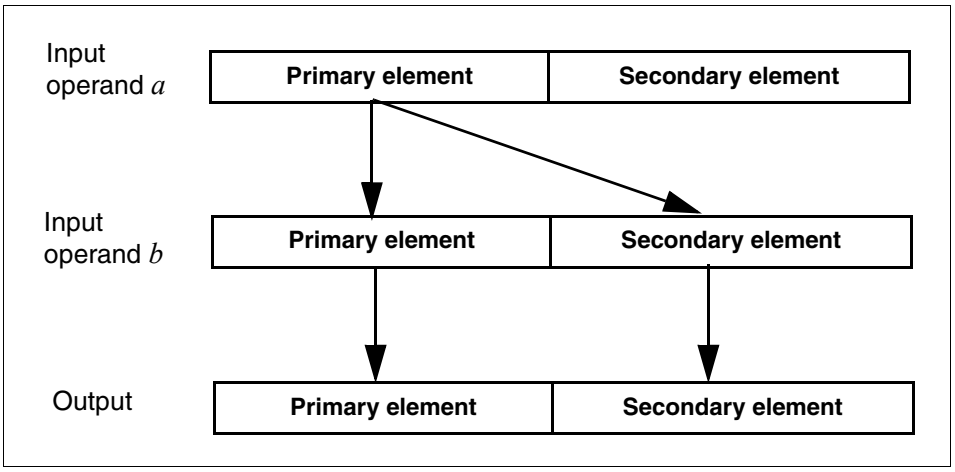

Figure 4-3 Copy-primary operations

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

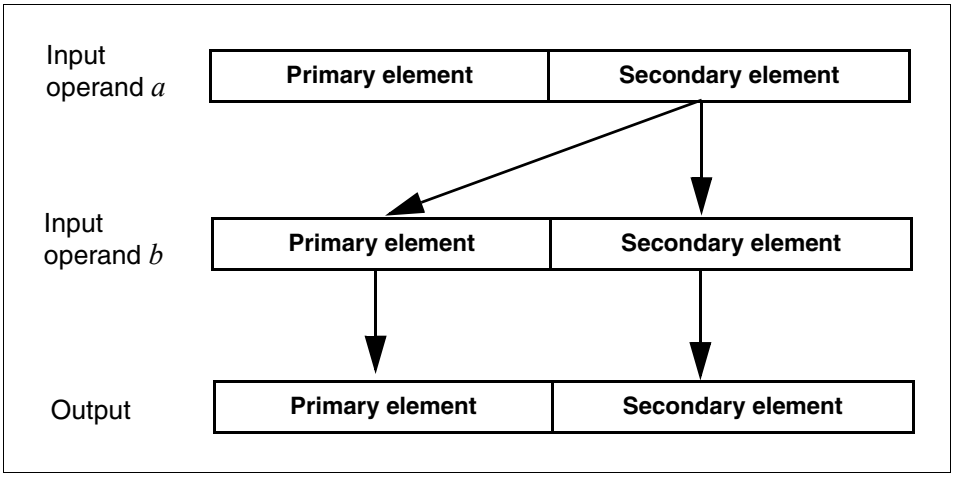

Figure 4-4 Copy-secondary operations

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 4-5 illustrates the result of a cross-copy multiply operation

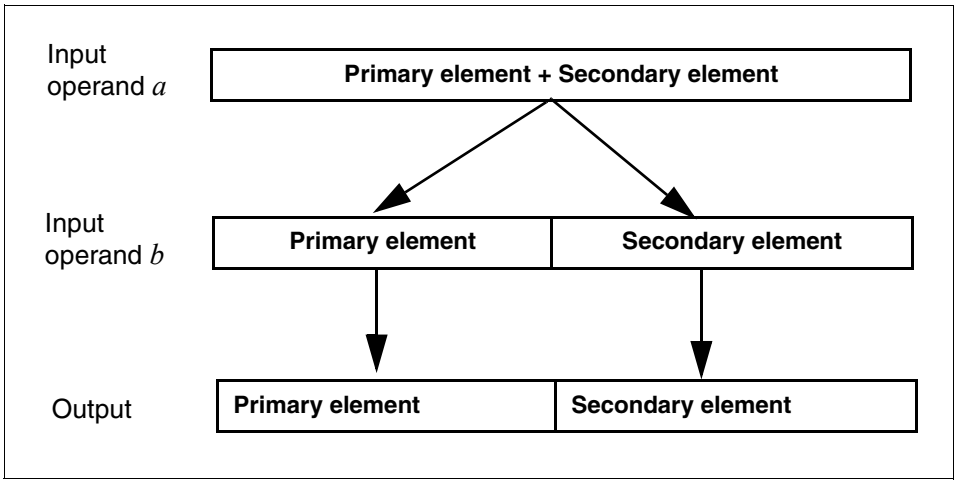

Figure 4-5 Cross-copy operations

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 4.4, “Tuning your code for Blue Gene/L” on page 27. Using the `alignx` built-in function (described in 4.11, “Checking for data alignment” on page 31), and specifying the `disjoint` pragma (described in 4.9, “Removing possibilities for aliasing (C/C++)” on page 29), are recommended for code that calls any of the built-in functions.

4.13 Complex type manipulation functions

These functions, listed in Table 4-2, 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 4-2 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.

4.14 Load and store functions

Table 4-3 lists and explains the various parallel load and store functions that are available.

Table 4-3 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 4.16.1, “Unary functions” on page 41, 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

4.15 Move functions

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

Table 4-4 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)

4.16 Arithmetic functions

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

4.16.1 Unary functions

Unary functions operate on a single input operand. These functions are listed in Table 4-5.

Table 4-5 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 4.14, “Load and store functions” on page 38.
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 4.14, “Load and store functions” on page 38.
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)

4.16.2 Binary functions

Binary functions operate on two input operands. The functions are listed in Table 4-6.

Table 4-6 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)

4.16.3 Multiply-add functions

Multiply-add functions take three input operands, multiply the first two, and add or subtract the third. Table 4-7 lists these functions.

Table 4-7 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 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 secondary element 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 the primary element of *a* and the secondary *b,* added to the secondary element of *c,* is stored as the secondary element of the return value.
Formula	primary(result) = secondary(a) × primary(b) + primary(c) secondary(result) = primary(a) × secondary(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 secondary element of *a* and the primary 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 primary element of *a* and the secondary 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) = -(secondary(a) × primary(b) + primary(c)) secondary(result) = -(primary(a) × secondary(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 secondary element of *a* and the primary element of *b*, is stored as the primary element of the return 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) × primary(b) - primary(c) secondary(result) = primary(a) × secondary(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 secondary element 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 the primary element of *a* and the secondary element of *b*, is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(secondary(a) × primary(b) - primary(c)) secondary(result) = -(primary(a) × secondary(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)

4.17 Select functions

Table 4-8 lists and explains the parallel select functions that are available.

Table 4-8 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);
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)

4.18 Examples of built-in functions usage

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 4-8 for C/C++ and Example 4-9 for Fortran.

Example 4-8 Using built-in functions: 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 4-9 Using built-in functions: 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
```

5

Running and debugging

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

5.1 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.

5.1.1 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 following set of commands 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$
```

For more information about using **mmcs_db_console**, see *Blue Gene/L: System Administration*, ZG24-6744.

5.1.2 mpirun

In the absence of a scheduling application, **mpirun** is the recommended way 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*, ZG24-6744.

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 5-1.

Example 5-1 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, it is necessary to add something like the following line to the end of the **mpirun** command:

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

This 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.

5.1.3 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 may 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*, ZG24-6744.

5.1.4 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 7, “Control system (Bridge) APIs” on page 71.

5.2 Debugging applications

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

5.2.1 GDB

GDB is the primary debugger of the GNU project. You can learn more about GDB at:

`http://www.gnu.org/software/gdb/gdb.html`

A great amount of documentation is available about the GDB debugger. 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 which allows the GDB debugger to work with applications running on Compute Nodes. Each running instance of GDB is associated with one, and only one, Compute Node. If you need to 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 5.2.2, “TotalView” on page 59), which makes it possible for a single debug instance to control multiple Compute Nodes simultaneously.

Prerequisite software

The GDB debugger should have been installed during the installation procedure defined in *Blue Gene/L: System Administration*, ZG24-6744. You can verify the installation by seeing if the /bgl/BlueLight/ppcfloor/ppc/linux-gnu/bin/powerpc-linux-gnu-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, etc.) to be included in the executable. In addition, do *not* use compiler optimization because it make 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/7.0/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 4, “Developing applications with IBM XL compilers” on page 25.

Important: Make sure that the text file containing 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 these steps to start debugging your application. For the sake of this example, let’s say that the program’s name is MyMPI.rts, and the source code file is MyMPI.c. We use a partition (block) called BETA18.

1. Open two separate console shells.
2. Go to the first shell window.
   a. Change (**cd**) to the directory containing 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 shown in Example 5-2 in the console.

Example 5-2 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 02:59:22> BE_MPI (Info) : No CWD specified ('-cwd' option)
<Jun 26 02:59:22> BE_MPI (Info) :   - it will be set to '/bgl'
<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 by either entering the rank of the program instance that you want to debug and pressing Enter, or by dumping the address or port of each node by typing `dump_proctable` and pressing 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/linux-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 hit in the second shell and you can use standard GDB commands to continue.

5.2.2 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 go to:

http://www.etnus.com

6

Checkpoint and restart support

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

6.1 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 one of 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? There are numerous scenarios that indicate that use of this support is warranted. We highlight a few in the following list:

- Your application is a very long-running one. You don't 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.

6.2 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()` may 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.

6.2.1 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 function _libc_open. The checkpoint library intercepts this call and provides its own implementation of open that internally uses the function _libc_open.

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.

6.2.2 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 may 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 6-1. If the signal is to be delivered immediately, the memory state of the application may 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 6-1 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.

6.3 Checkpoint API

The checkpoint interface consists of:

- 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.

6.3.1 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 6.4, “Directory and file naming conventions” on page 67.

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 6.4, “Directory and file naming conventions” on page 67.

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 section 3.2.

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)

BGLAtContiue 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.

6.4 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 may 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.

6.5 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.

6.5.1 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 may crash while some others may complete. This may 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.

6.5.2 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 6-2 summarizes the checkpoint-related function calls.

Table 6-2 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 6-3 summarizes the environment variables.

Table 6-3 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 6-3) automatically signifies that after restart, further checkpoints are taken. If the developer wants to restart an application but disable further checkpoints, he simply needs to unset (remove altogether) the BGL_CHKPT_ENABLED variable.

Part 2

System application information

This part provides details that are of interest to someone who is writing an application to help control the Blue Gene/L system. An example of such an application is a custom program that is intended to allow for the scheduling and controlling of jobs that are running on Blue Gene/L using the Bridge application programming interfaces (APIs). The details are covered in Chapter 7, “Control system (Bridge) APIs” on page 71.

To learn specifically about working with Message Passing Interface (MPI) applications, see Part 1, “MPI application information” on page 1.

7

Control system (Bridge) APIs

This section 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.

7.1 API support overview

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

7.1.1 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) 8 for PowerPC is the only supported platform.

Languages supported

C and C++ are supported with the GNU gcc 3.2 level compilers. For more information and downloads, see:

http://gcc.gnu.org/

System files

Two main files are provided that are required to compile and link code to interface with the Bridge APIs:

- /bgl/BlueLight/ppcfloor/bglsys/include/rm_api.h
- /bgl/BlueLight/ppcfloor/bglsys/lib/bglbridge.a

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

7.1.2 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, there is a corresponding “free” function that 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 will result in memory leaks.

7.2 APIs

The following sections describe the APIs in detail.

7.2.1 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 on transient entities, such as jobs and partitions. These functions do not impose side-effects on the actual machine.

The `rm_get_BG`L 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 on 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 7-1.

The `rm_add_partition()` and `rm_add_job()` 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()` are marked with an asterisk (*) in Table 7-1.

Table 7-1 Specification for rm_get_data/rm_set_data function

Object	Set using rm_set_data()?	Specification	Resulting data type	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.
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.

Object	Set using rm_set_data()?	Specification	Resulting data type	Description
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. The state can be UP or DOWN.
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_BPPartState	rm_partition_state_t *	A pointer to an enum value indicating the state of the partition (for more about partition states, see Figure 7-2 on page 84).
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.
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 connected to that switch.
Switch		RM_SwitchState	rm_switch_state_t *	A pointer to an enum value indicating the state of the switch. The state can be UP or DOWN.
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.

Object	Set using rm_set_data()?	Specification	Resulting data type	Description
Switch	*	RM_SwitchConnNum	int *	The number of connections in the switch.
Switch	*	RM_SwitchFirst Connection	rm_connections_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_connections_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_WireFromPort	rm_element_t *	A pointer to an element associated with the wire source port.
Wire		RM_WirePortTo	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_WirePartState	rm_partition_state_t *	A pointer to an enum value indicating the state of the partition (for more about partition states, see Figure 7-2 on page 84).
Port		RM_PortComponentID	rm_component_id_t *	A pointer to the ID that identifies the base partition or the switch the port is part of.
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_PartListSize	int *	The number of partitions in the list.
Partition list		RM_PartListFirstPart	rm_element_t *	A pointer to the first partition in the retrieved list.
Partition list		RM_PartListNextPart	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.

Object	Set using rm_set_data()?	Specification	Resulting data type	Description
Partition		PM_PartitionState	rm_partition_state_t *	A pointer to an enum value indicating the state of the partition (for more about partition states, see Figure 7-2 on page 84).
Partition	*	RM_PartitionBPNum	int *	The number of base partitions in the partition.
Partition	*	RM_PartitionSwitchNum	int *	The number of switches in the partition.
Partition	*	RM_PartitionFirstBP	rm_element_t *	A pointer to the element associated with the first base partition in the list.
Partition	*	RM_PartitionNextBP	rm_element_t *	A pointer to the element associated with the next base partition in the list.
Partition	*	RM_PartitiontFirstSwitch	rm_element_t *	A pointer to the element associated with the first switch in the list.
Partition	*	RM_PartitiontNextSwitch	rm_element_t *	A pointer to the element associated with the next switch in the list.
Partition	*	RM_PartitionConnection	rm_connection_ type_t *	The connection type of the partition. Can be TORUS or MESH.
Partition	*	RM_PartitionUserName	char *	A pointer to a string containing the user name of the user who submitted the job.
Partition	*	RM_PartitionMloaderImg	char *	A pointer to a string containing the file name of the machine loader image.
Partition	*	RM_PartitionBlrtsImg	char *	A pointer to a string containing the file name of the compute node's kernel image.
Partition	*	RM_PartitionLinuxImg	char *	A pointer to a string containing the file name of the I/O node's Linux image.
Partition	*	RM_PartitionRam diskImg	char *	A pointer to a string containing the file name of the ramdisk image.
Partition	*	RM_PartitionDescription	char *	A pointer to a string containing a description of the partition.
Partition	*	RM_PartitionSmall	boolean *	A flag indicating whether this partition is a partition smaller than the base partition.
Partition	*	RM_PartitionPsets PerBP	int *	The number of used PSets per BP.

Object	Set using rm_set_data()?	Specification	Resulting data type	Description
Partition		RM_PartitionUsersNum	int *	The number of users of the partition.
Partition		RM_PartitionFirstUser	char *	A pointer to the partition's first user name.
Partition		RM_PartitionNextUser	char *	A pointer to the partition's next user name.
Partition		RM_PartitionOptions	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_JobListFirstJob	rm_element_t *	A pointer to the first job in the retrieved list.
Job list		RM_JobListNextJob	rm_element_t *	A pointer to the next job in the retrieved list.
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 7-1 on page 83).
Job	*	RM_JobExecutable	char *	A string with the job executable name.
Job	*	RM_JobUserName	char *	A pointer to a string containing the Unix user name of the user who submitted the job.
Job		RM_JobDBJobID	db_job_id_t *	A pointer to an integer containing the ID given to the job by the DB.
Job	*	RM_JobOutFile	char *	A pointer to a string containing the job output file name.
Job	*	RM_JobInFile	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.

Object	Set using rm_set_data()?	Specification	Resulting data type	Description
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.
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 COPROCESSOR or VIRTUAL.

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 BPs. The possible flags are contained in the rm_api.h include file, and listed in Table 7-2 for your convenience. The states are represented by the bits in Table 7-2.

  Table 7-2 Flags for partition states

Flag	Value
PARTITION_FREE_FLAG	0x01
PARTITION_CONFIGURING_FLAG	0x02
PARTITION_READY_FLAG	0x04
PARTITION_BUSY_FLAG	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 7-3. The states are represented by the bits in Table 7-3.

  Table 7-3 Flags for job states

Flag	Value
JOB_IDLE_FLAG	0x001
JOB_STARTING_FLAG	0x002
JOB_RUNNING_FLAG	0x004

Flag	Value
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, etc.). The specifications available when using `rm_get_data()` are listed in Table 7-1 on page 73 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, etc.). The specifications available when using `rm_set_data()` are listed in Table 7-1 on page 73 and marked with an *.

- `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 DB 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 be `rm_set_serial()`.

7.2.2 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);`

7.2.3 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_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);`

7.2.4 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 would be printed to and what the verbosity level would be. 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.

7.2.5 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 actually complete.

Before performing 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 7-3 on page 79.

- `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.

7.2.6 API to the MMCS partition manager

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

Before performing 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 7-2.

- `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.

7.2.7 State diagrams for jobs and partitions

Figure 7-1 illustrates the main states tat a job goes through during its life cycle.

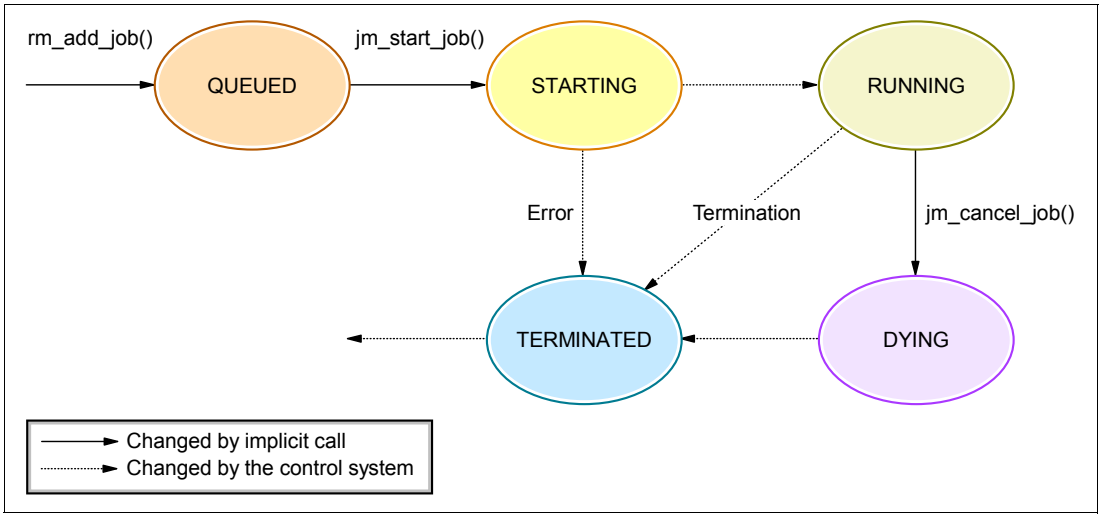

Figure 7-1 Job state diagram

Figure 7-2 describes the various partition states.

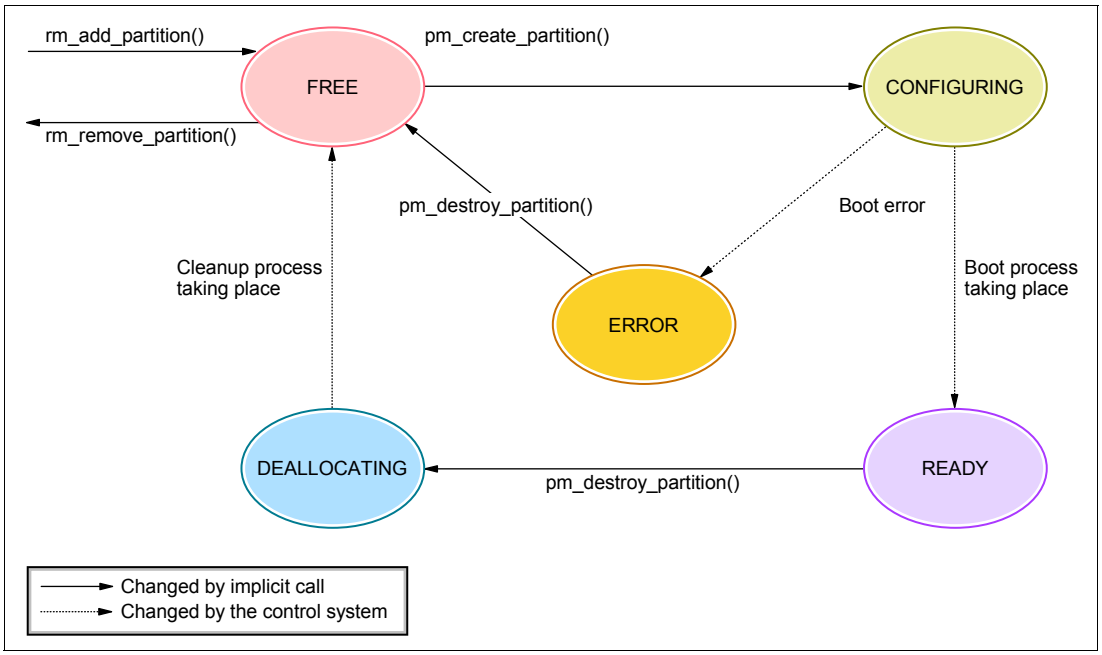

Figure 7-2 Partition state diagram

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

7.3 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 BPs 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 7-1 and Figure 7-2 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, etc.
- INCONSISTENT_DATA: The data retrieved from the control system is illegal or invalid.
- INTERNAL_ERROR: Errors that don't belong to any of the previously listed categories, such as a memory allocation problem or failures during the manipulation of internal XML files.

7.3.1 Return codes specification

The return codes for the various API functions are:

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

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

  Return codes:

  - STATUS_OK
  - CONNECTION_ERROR
  - INCONSISTENT_DATA: Possibly for one of the following reasons:
    - List of BPs is empty.
    - Wire list is empty and the number of BPs is greater than one.
    - Switch list is empty and the number of BPs is greater than one.
  - INTERNAL_ERROR

- `status_t rm_add_partition(rm_partition_t *p);`

  This function defines a partition in the control system.

  Return codes:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT: The data in the `rm_partition_t` structure is invalid.
    - No BP or switch list is supplied.
    - BP 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 BPs 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.

  Return codes:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID INPUT

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

- PARTITION_NOT_FOUND
- INCONSISTENT_DATA

  BP 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.

  Return codes:

  - 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_falg_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 BPs.

  The states are represented by the following bits:

  - PARTITION_FREE_FLAG 0x1
  - PARTITION_CONFIGURING_FLAG 0x2
  - PARTITION_READY_FLAG 0x4
  - PARTITION_BUSY_FLAG 0x8
  - PARTITION_DEALLOCATING_FLAG 0x10
  - PARTITION_ERROR_FLAG 0x20
  - PARTITION_ALL_FLAG 0xFF

  Return codes:

  - 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.

  Return codes:

  - 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 7-1 on page 83.

  - 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.

  Return codes:

  - 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 7-1 and Figure 7-2.
    - 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.

  Return codes:

  - 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 7-1 and Figure 7-2.
  - INTERNAL_ERROR

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

  Change the partition owner.

  Return codes:

  - 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);`

  Add a user to a partition.

  Return codes:

  - 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);`

Add a user to a partition.

Return codes:

- 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.

Return codes:

- 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 7-1.
- INTERNAL_ERROR

► `status_t pm_destroy_partition (pm_partition_id_t pid);`

This function requests destruction of a partition.

Return codes:

- 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 7-1.
- INTERNAL_ERROR

- `status_t rm_add_job(db_job_id_t *job);`

  This function defines a job in the control system.

  Return codes:

  - 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”.

  Return codes:

  - 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).

  Return codes:

  - 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.

  Return codes:

  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job’s state prevents its removal. See Figure 7-2.
  - INTERNAL_ERROR

- `status_t jm_start_job(db_job_id_t jid);`

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

  Return codes:

  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

- The job's state prevents its execution. See Figure 7-2.
  - INTERNAL_ERROR
- `status_t jm_cancel_job(db_job_id_t jid);`

  This function requests cancellation of a job.

  Return codes:
  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job's state prevents it from being canceled. See Figure 7-2.
  - INTERNAL_ERROR
- `status_t jm_signal_job(db_job_id_t jid, rm_signal_t signal);`

  This function signals a job.

  Return codes:
  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job's state prevents it from being signaled.
  - 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, BP, wire, switch, connections, port).

  Return codes:
  - 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.

  Return codes:
  - 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.

  Return codes:
  - STATUS_OK
  - INTERNAL_ERROR

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

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

  Return codes:

  - STATUS_OK
  - INTERNAL_ERROR

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

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

  Return codes:

  - STATUS_OK
  - INTERNAL_ERROR

Part 3

Performance analysis

Several tools and techniques are available to analyze system and application performance on Blue Gene/L. This part examines some of these items in the following chapters:

8

Performance guidelines and tools

This chapter describes the process of using tools to analyze system and application performance.

8.1 Tooling overview

A variety of tools are available to help understand your application's performance when running on Blue Gene/L. Some of these tools are written by IBM, others are written by independent software vendors (ISVs), and still others are open source efforts.

Some tools have been ported to Blue Gene/L, and more are moving over every month. However, almost all tools work against the pSeries systems. At times it is advantageous to run your application on pSeries and profile it there if the particular tool you are interested in using does not yet support Blue Gene/L.

This section first discusses IBM's main tool suite. Then, it examines the tools that you can use on pSeries to help understand Blue Gene/L performance. Finally, it looks at tools that run natively with applications on Blue Gene/L.

8.1.1 IBM High Performance Computing Toolkit

The Advanced Computing Technology Center (ACTC), part of IBM Research in Yorktown Heights, New York, conducts research on the performance behavior of scientific and technical computing applications. Its role in IBM is to provide strategic technical direction for the research and development of server platforms to advance the state of the art in high performance computing offerings and solutions for IBM Clients in computationally intensive industries. Such industries include automotive, aerospace, petroleum, meteorology, and life science.

IBM offers the IBM High Performance Computing Toolkit, a suite of performance-related tools and libraries to assist in application tuning. This toolkit is an integrated environment for performance analysis of sequential and parallel applications using the Message Passing Interface (MPI) and OpenMP paradigms. It provides a common framework for IBM's mid-range server offerings, including IBM @server pSeries and iSeries™ servers and Blue Gene/L systems, on both AIX® and Linux.

8.2 General performance testing

IBM recommends testing an application on a pSeries system before running it on a Blue Gene/L system if possible. Use a memory size per compute node that is compatible with the Blue Gene/L architecture. For more information, see 1.2, "Memory considerations" on page 4. This approach makes it possible to check both memory utilization and performance issues. Both pSeries and the Blue Gene/L supercomputer use IBM XL compilers, which aids portability between the two systems.

8.2.1 Overview of the tools that are available on pSeries

For the best performance, it is good practice to obtain a performance profile for your application. IBM is porting its comprehensive performance analysis tools, the High Performance Computing Toolkit, to the Blue Gene/L supercomputer. In the meantime, we recommend that you perform profiling on a similar system, such as pSeries. Most computational performance issues are the same on Blue Gene/L as on other reduced instruction set computer (RISC) processors, so this method usually identifies the main issues.

For parallel performance, several MPI profiling tools are available, including the ones listed in the following sections.

IBM High Performance Computing Toolkit

The IBM High Performance Computing Toolkit is the foundation for all performance tools for Blue Gene/L and other IBM @server systems. The tools provide source code traceback of the performance data to help the user quickly identify any bottlenecks in the code. The toolkit includes low-overhead measurement of time spent in MPI routines for applications written in any mixture of Fortran, C, and C++.

The tools include Xprofiler, MPI_tracer, MPI_Profiler, and PeekPerf. The toolkit provides a text summary and an optional graphical display.

Paraver

Paraver is a graphical user interface (GUI)-based performance visualization and analysis tool that you can use to analyze parallel programs. It lets you obtain detailed information from raw performance traces. To learn more about Paraver, go to:

http://www.cepba.upc.es/paraver/

MPE/jumpshot

MPICH2 has extensions for profiling MPI applications, and the MPE extensions have been ported to Blue Gene/L. For more information, see the following Web site:

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

8.2.2 Overview of tools ported to Blue Gene/L

The following tools have been ported to the Blue Gene/L platform.

- KOJAK

  Kit for Objective Judgement and Knowledge (KOJAK)-based detection of performance bottlenecks

  http://www.fz-juelich.de/zam/kojak/

- TAU

  Tuning and Analysis Utilities

  http://www.cs.uoregon.edu/research/paracomp/tau/tautools/

8.3 Message passing performance

Measuring the performance of message passing (MPI) in an application can quickly help identify trouble areas. MPI Tracer™ and Profiler consist of a set of libraries that collect profiling and tracing data for MPI programs. Performance metrics, such as the time used by MPI function calls and message sizes, are reported.

These tools are available from the IBM ACTC. For more information about these and other tools that this organization provides, see:

http://www.research.ibm.com/actc/

8.3.1 MPI Tracer and Profiler

MPI Tracer and Profiler consists of a set of libraries that collect profiling and tracing data for MPI programs. Performance metrics, such as the time used by MPI function calls and message sizes, are reported.

MPI tracer works with the visualization tools PeekPerf and PeekView to better help users identify performance bottlenecks. *PeekPerf* maps performance metrics back to the source codes. *PeekView* gives a visual representation of the overall computation and communication pattern of the system.

MPI Profiler captures summary data for MPI calls. By this we mean that it does not show you the specifics of an individual call to, for example MPI_Send, but rather the combined data for all calls made to that routine during the profile period. See Figure 8-1 for an example.

```
---------------------------------------------------------------
MPI Routine            #calls     avg. bytes      time(sec)
---------------------------------------------------------------
MPI_Comm_size               3            0.0         0.000
MPI_Comm_rank           12994            0.0         0.016
MPI_Send                19575        11166.9        13.490
MPI_Isend              910791         5804.2         9.216
MPI_Recv               138173         2767.9        73.835
MPI_Irecv              784936        15891.6         2.407
MPI_Sendrecv           894809          352.0        88.705
MPI_Wait              1537375            0.0       288.049
MPI_Waitall             44042            0.0        25.312
MPI_Bcast                 464        41936.8         3.272
MPI_Barrier              1312            0.0        34.206
MPI_Gather                 68        16399.1         2.680
MPI_Scatter                 6        17237.3         0.532
---------------------------------------------------------------
total communication time = 770.424 seconds.
total elapsed time       = 1168.662 seconds.
user cpu time            = 1160.960 seconds.
system time              = 0.620 seconds.
maximum memory size      = 68364 KBytes.

To check load balance :  grep "total comm" mpi_profile.*
```

Figure 8-1 MPI Profiler summary data

Important: It is vital that you call `MPI_Finalize` in your application for the profiling function to correctly gather data.

No changes to your source code are required to use the MPI Profiler function. However, you must compile using the debug (`-g`) flag.

8.4 CPU performance

The CPU performance tools are from the IBM ACTC. For more information about these and other tools they provide, see

http://www.research.ibm.com/actc/

8.4.1 Hardware performance monitor

Hardware performance counter monitor module provides comprehensive reports of events that are critical to performance on IBM systems. In addition to the usual timing information, the hardware performance monitor (HPM) can gather critical hardware performance metrics. These may include the number of misses on all cache levels, the number of floating point instructions executed, and the number of instruction loads that cause TLB misses, that help the algorithm designer or programmer identify and eliminate performance bottlenecks.

8.4.2 Xprofiler

Xprofiler is among a set of CPU profiling tools, such as grof, pprof, pprof, and tprof, that are provided on AIX. You can use them to profile both serial and parallel applications.

Xprofiler uses procedure-profiling information to construct a graphical display of the functions within an application. It provides quick access to the profiled data and helps users identify the functions that are the most CPU-intensive. With the GUI, it is easy to find the application's performance-critical areas.

8.5 I/O performance

Understanding input/output (I/O) performance is just as important as understanding application and CPU performance issues.

8.5.1 Modular I/O

Modular I/O (MIO) is not yet officially supported on Blue Gene/L. MIO addresses the need of application-level optimization for I/O. For I/O-intensive applications, the MIO libraries provide a means to analyze the I/O behavior of applications and tune I/O at the application level for optimal performance. For example, when an application exhibits the I/O pattern of sequential reading of large files, MIO detects the behavior and invokes its asynchronous prefetching module to prefetch user data.

Tests with the AIX JFS file system demonstrates significant improvement over system throughput when using MIO.

8.6 Visualization and analysis

The PeekPer tools is from the IBM ACTC. For more information about this and other tools that they provide, see:

http://www.research.ibm.com/actc/

8.6.1 PeekPerf

PeekPerf visualizes the performance trace information generated by the performance analysis tools. PeekPerf also maps the collected performance data back to the source code, which makes it easier for users to find bottlenecks and points for optimizations. PeekPerf is available on several UNIX® derivations (AIX, Linux) and Microsoft® Windows®.

8.7 MASS and MASSV libraries

The MASS and MASSV libraries consist of a set of mathematical functions for C, C++, and Fortran-language applications that are tuned for specific POWER™ architectures. You can learn more about these libraries at:

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

Both scalar (libmass.a) and vector (libmassv.a) intrinsic routines are tuned for the Blue Gene/L computer. In many situations, using these libraries has been shown to result in significant code performance improvement.

Such routines as sin, cos, exp, log, and so forth from these libraries are significantly faster than the standard routines from GNU libm.a. For example, a `sqrt()` call costs about 106 cycles with libm.a, about 46 cycles for libmass.a, and 8 to 10 cycles per evaluation for a vector of `sqrt()` calls in libmassv.a. To link with libmass.a, include the following option on the link line:

```
-Wl,--allow-multiple-definition.
```

9

Performance counters and PAPI

This chapter provides details about the performance counters feature of Blue Gene/L. This is a Blue Gene/L-specific function that allows a user to determine how many times a certain, configurable event occurs during the course of an application run.

We also provide information about Performance Application Programming Interface (PAPI) support that is provided on Blue Gene/L. For more information about PAPI in general, see:

http://icl.cs.utk.edu/papi/index.html

9.1 Introduction to the performance counter interface

The Blue Gene/L performance counter (`bgl_perfctr`) interface to the Compute Node Kernel (CNK) provides a uniform application programming interface (API) for all universal performance counter (UPC) and floating point unit (FPU) counters. All counters are programmed and accessed using the same calling sequence. Mnemonics and descriptive information for all available events are provided by the API. The interface handles counter register allocation and provides the user a full specification of possible events to counter mappings.

The PAPI implementation described in this document exposes the `bgl_perfctr` to the user through the PAPI standard interface. PAPI is a well established de facto standard for user-level hardware counter access and is available on all major computational platforms in use today.

9.2 bgl_perfctr library API

The `bgl_perfctr` interface is a user-level API that provides access to the UPC and FPU counters. The `bgl_perfctr` presents the user with a set of 52 virtual 64-bit counters that map to the underlying hardware counters. The first 48 counters map to the UPC counters on the chip, while the last four counters map to the two counters in each of the FPU units.

The user instantiates counters by requesting to register a certain event. All possible events are available as mnemonics. Given a request to register an event, the library interface locates an available hardware counter capable of registering the particular event. If this search is successful, the event is registered as an event pending to be added. If there is no available hardware counter for the event, an error code is returned to the user.

The counters pending to be added are invoked through the user initiating a call to `bgl_perfctr_commit()`. At this point, all pending changes to the counter setup are performed, and the counter map is updated. A call to `bgl_perfctr_revoke()` clears all pending changes and leaves the hardware counters untouched.

The virtual counters in the `bgl_perfctr` interface are updated from the actual hardware counters by calling `bgl_perfctr_update()` directly. Also, calling any of the functions `bgl_perfctr_copy_counters()`, `bgl_perfctr_copy_state()`, or `bgl_perfctr_get_counters()` implicitly call `bgl_perfctr_update()`. The virtual counter update reads all active hardware counters and updates the corresponding virtual counter with the number of counts aggregated since the latest read. The configured UPC counters are read through the memory map interface while FPU counters are read through DCR access.

At library initialization, which is explicitly made by the user, the user can set up the library to periodically call `bgl_perfctr_update()` by means of a periodic timer interrupt. This interrupt occurs with an interval of approximately six seconds (on a 700 MHz system), which guards against any 32-bit counter overflowing more than once between updates to the virtual counters. By default, this interval timer is set up after synchronization between all nodes in the partition. This reduces the impact on a parallel running application from the periodic virtual counter updates.

9.2.1 API details

A list of the first 100 defined event mnemonics is provided in Table 9-1 to illustrate the naming scheme.

Table 9-1 Example event mnemonics (event numbers)

BGL_FPU_ARITH_ADD_SUBTRACT	BGL_FPU_ARITH_MULT_DIV
BGL_FPU_ARITH_OEDIPUS_OP	BGL_FPU_ARITH_TRINARY_OP
BGL_FPU_LDST_DBL_LD	BGL_FPU_LDST_DBL_ST
BGL_FPU_LDST_QUAD_LD	BGL_FPU_LDST_QUAD_ST
BGL_2NDFPU_ARITH_ADD_SUBTRACT	BGL_2NDFPU_ARITH_MULT_DIV
BGL_2NDFPU_ARITH_OEDIPUS_OP	BGL_2NDFPU_ARITH_TRINARY_OP
BGL_2NDFPU_LDST_DBL_LD	BGL_2NDFPU_LDST_DBL_ST
BGL_2NDFPU_LDST_QUAD_LD	BGL_2NDFPU_LDST_QUAD_ST
BGL_UPC_L3_CACHE_HIT	BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR
BGL_UPC_L3_CACHE_MISS_DATA_WILL_BE_REQED_DDR	BGL_UPC_L3_EDRAM_ACCESS_CYCLE
BGL_UPC_L3_EDRAM_RFR_CYCLE	BGL_UPC_L3_LINE_STARTS_EVICT_LINE_NUM_PRESSURE
BGL_UPC_L3_MISS_DIR_SET_DISBL	BGL_UPC_L3_MISS_NO_WAY_SET_AVAIL
BGL_UPC_L3_MISS_REQUIRING_CASTOUT	BGL_UPC_L3_MISS_REQUIRING_REFILL_NO_WR_ALLOC
BGL_UPC_L3_MSHNDLR_TOOK_REQ	BGL_UPC_L3_MSHNDLR_TOOK_REQ_PLB_RDQ
BGL_UPC_L3_MSHNDLR_TOOK_REQ_RDQO	BGL_UPC_L3_MSHNDLR_TOOK_REQ_RDQ1
BGL_UPC_L3_MSHNDLR_TOOK_REQ_WRBUF	BGL_UPC_L3_PAGE_CLOSE
BGL_UPC_L3_PAGE_OPEN	BGL_UPC_L3_PLB_WRQ_DEP_DBUF
BGL_UPC_L3_PLB_WRQ_DEP_DBUF_HIT	BGL_UPC_L3_PREF_REINS_PULL_OUT_NEXT_LINE
BGL_UPC_L3_PREF_REQ_ACC_BY_PREF_UNIT	BGL_UPC_L3_RD_BURST_1O24B_LINE_RD
BGL_UPC_L3_RD_EDR__ALL_KINDS_OF_RD	BGL_UPC_L3_RD_MODIFY_WR_CYCLE_EDR
BGL_UPC_L3_REQ_TKN_CACHE_INHIB_RD_REQ	BGL_UPC_L3_REQ_TKN_CACHE_INHIB_WR
BGL_UPC_L3_REQ_TKN_NEEDS_CASTOUT	BGL_UPC_L3_REQ_TKN_NEEDS_REFILL
BGL_UPC_L3_WRBUF_LINE_ALLOC	BGL_UPC_L3_WRQO_DEP_DBUF
BGL_UPC_L3_WRQO_DEP_DBUF_HIT	BGL_UPC_L3_WRQ1_DEP_DBUF
BGL_UPC_L3_WRQ1_DEP_DBUF_HIT	BGL_UPC_L3_WR_EDRAM__INCLUDING_RMW
BGL_UPC_PUO_DCURD_1_RD_PEND	BGL_UPC_PUO_DCURD_2_RD_PEND
BGL_UPC_PUO_DCURD_3_RD_PEND	BGL_UPC_PUO_DCURD_BLIND_REQ
BGL_UPC_PUO_DCURD_COHERENCY_STALL_WAR	BGL_UPC_PUO_DCURD_L3_REQ
BGL_UPC_PUO_DCURD_L3_REQ_PEND	BGL_UPC_PUO_DCURD_LINK_REQ
BGL_UPC_PUO_DCURD_LINK_REQ_PEND	BGL_UPC_PUO_DCURD_LOCK_REQ
BGL_UPC_PUO_DCURD_LOCK_REQ_PEND	BGL_UPC_PUO_DCURD_PLB_REQ
BGL_UPC_PUO_DCURD_PLB_REQ_PEND	BGL_UPC_PUO_DCURD_RD_REQ

BGL_FPU_ARITH_ADD_SUBTRACT	BGL_FPU_ARITH_MULT_DIV
BGL_UPC_PU0_DCURD_SRAM_REQ	BGL_UPC_PU0_DCURD_SRAM_REQ_PEND
BGL_UPC_PU0_DCURD_WAIT_L3	BGL_UPC_PU0_DCURD_WAIT_LINK
BGL_UPC_PU0_DCURD_WAIT_LOCK	BGL_UPC_PU0_DCURD_WAIT_PLB
BGL_UPC_PU0_DCURD_WAIT_SRAM	BGL_UPC_PU0_PREF_FILTER_HIT
BGL_UPC_PU0_PREF_PREF_PEND	BGL_UPC_PU0_PREF_REQ_VALID
BGL_UPC_PU0_PREF_SELF_HIT	BGL_UPC_PU0_PREF_SNOOP_HIT_OTHER
BGL_UPC_PU0_PREF_SNOOP_HIT_PLB	BGL_UPC_PU0_PREF_SNOOP_HIT_SAME
BGL_UPC_PU0_PREF_STREAM_HIT	BGL_UPC_PU1_DCURD_1_RD_PEND
BGL_UPC_PU1_DCURD_2_RD_PEND	BGL_UPC_PU1_DCURD_3_RD_PEND
BGL_UPC_PU1_DCURD_BLIND_REQ	BGL_UPC_PU1_DCURD_COHERENCY_STALL_WAR
BGL_UPC_PU1_DCURD_L3_REQ	BGL_UPC_PU1_DCURD_L3_REQ_PEND
BGL_UPC_PU1_DCURD_LINK_REQ	BGL_UPC_PU1_DCURD_LINK_REQ_PEND
BGL_UPC_PU1_DCURD_LOCK_REQ	BGL_UPC_PU1_DCURD_LOCK_REQ_PEND
BGL_UPC_PU1_DCURD_PLB_REQ	BGL_UPC_PU1_DCURD_PLB_REQ_PEND
BGL_UPC_PU1_DCURD_RD_REQ	BGL_UPC_PU1_DCURD_SRAM_REQ
BGL_UPC_PU1_DCURD_SRAM_REQ_PEND	BGL_UPC_PU1_DCURD_WAIT_L3
BGL_UPC_PU1_DCURD_WAIT_LINK	BGL_UPC_PU1_DCURD_WAIT_LOCK
BGL_UPC_PU1_DCURD_WAIT_PLB	BGL_UPC_PU1_DCURD_WAIT_SRAM
BGL_UPC_PU1_PREF_FILTER_HIT	BGL_UPC_PU1_PREF_PREF_PEND
BGL_UPC_PU1_PREF_REQ_VALID	BGL_UPC_PU1_PREF_SELF_HIT

These mnemonics define an enumerated data type that is used to identify the events in `bgl_perfctr`. A full event descriptor is a structure with two components: the event mnemonic and the edge or state to monitor.

Example 9-1 shows the definition of the event descriptor together with an example of its use.

Example 9-1 bgl_perfctr event descriptor (from bgl_perfctr_events.h)

```
typedef struct BGL_PERFCTR_event {
  BGL_PERFCTR_event_num_t num;
  unsigned int edge;
} BGL_PERFCTR_event_t;

#define BGL_PERFCTR_UPC_EDGE_HI   0x0
#define BGL_PERFCTR_UPC_EDGE_RISE 0x1
#define BGL_PERFCTR_UPC_EDGE_FALL 0x2
#define BGL_PERFCTR_UPC_EDGE_LOW  0x3

Example of use of the event descriptor

BGL_PERFCTR_event_t fpu_example={ BGL_FPU_ARITH_MULT_DIV, 0};
BGL_PERFCTR_event_t upc_example={ BGL_UPC_PU0_DCURD_3_RD_PEND,
                                  BGL_PERFCTR_UPC_EDGE_HI};
```

UPC events need to specify both an event type and an edge type to be complete. For FPU events, the edge selector is not used and should always be set to zero. In Example 9-2, two examples of events are shown. The first event counts, multiplies, and divides in FPU0 while the second event counts the duration in UPC cycles (CLOCKx2 cycles) where there were three outstanding read requests in CPU core 0. All durations are in the unit of UPC cycles, which is equal to two CPU cycles.

The internal data structures of the counter control substrate are instantiated at the time of application launch.

The hardware counter is a shared resource on the Blue Gene/L compute node. For this reason, any event programmed for a counter on one of the cores is also seen on the second core of the node. This behavior is present in virtual node mode as well as in coprocessor mode.

The major part of the internal data structure consists of a control structure. This structure contains the complete state of the virtual counters and is shown in Example 9-2.

Example 9-2 bgl_perfctr structure

```
typedef struct bgl_perfctr_control {
  /* Bit pattern (one bit per counter register) */
  unsigned long long in_use;
  /* Bit pattern (one bit per counter control register */
  unsigned long long modified;
  unsigned long long virtual[BGL_PERFCTR_NUM_COUNTERS];
  unsigned int ctrl[BGL_PERFCTR_NUM_CTRL];
  unsigned int last[BGL_PERFCTR_NUM_COUNTERS];
  int nmapped;
  bgl_perfctr_control_map_t map[BGL_PERFCTR_NUM_COUNTERS];
  volatile unsigned long long last_updated;
} bgl_perfctr_control_t;

struct bgl_perfctr_control_map {
  BGL_PERFCTR_event_t event;
  int counter_register;
  int cntrl_register;
  int ref_count;
  int new_count;
} bgl_perfctr_control_map_t;
```

The 52 available counters are internally enumerated from 0 to 51. The structure sets the corresponding *bit* (starting to count from the least significant bit) in the `in_use` component for each counter in use. Counters with pending changes are marked in the `modified` bit map. After a `bgl_perfctr_commit()` or `bgl_perfctr_revoke()`, this bit map is reset to 0.

The latest value recorded in the virtual counters is available in the `virtual` component. Virtual counter k is the virtualization of physical hardware counter k. The `ctrl` component contains the value for the different control registers for the counters. The `last` component is the content of the physical counter register at the latest read. The `last_updated` component is updated with the current value of the time base register at the onset and completion of each virtual counter update.

At each point in time, there are N counting registers in use. To facilitate a simple read out of counter values, there is an array of length N in the `bgl_perfctr` control structure. This array shows the use of each counter. The map is sorted in ascending order according to the `bgl_perfctr` event descriptor enumerator. The number of active events, N, is stored in the

component nmapped. The virtual and physical counter map[k].counter_register, where k<nmapped, is counting the event described by map[k].event.

The complete bgl_perfctr API consists of the following 14 functions.

- bgl_perfctr_init: Initializes the library. This function is equivalent to bgl_perfctr_init_synch(BGL_PERFCTR_MODE_LOCAL). On success, zero is returned. On failure a negative value is returned.
- bgl_perfctr_init_synch: An alternative initialization routine that allows the user to control the amount of synchronization between the tasks in Blue Gene/L. Possible values are:
  - BGL_PERFCTR_MODE_LOCAL, which provides no synchronization and no counter overflow protection
  - BGL_PERFCTR_MODE_ASYNC, which starts a local timer that initiates counter reads at approximately every 6 seconds to prevent counter overflow
  - BGL_PERFCTR_MODE_SYNC, which also provides overflow protection using the local timer

  BGL_PERFCTR_MODE_SYNC differs from BGL_PERFCTR_MODE_ASYNC in that the previous mode starts the timers after a global barrier to allow for synchronous counter updates across the application. The return value indicates the synchronization mode accomplished. This is equal to or lower than the supplied mode.
- bgl_perfctr_shutdown: Stops local timed interrupts on the local core; if there is no core using the counters, clears the internal state and stops all counters.
- bgl_perfctr_add_event: Attempts to schedule an event to be added to the running set of counters.
- bgl_perfctr_remove_event: Attempts to schedule an event to be removed from the running set of counters.
- int bgl_perfctr_commit: Commits all pending changes to the running set of counters.
- int bgl_perfctr_revoke: Removes all pending changes and restores the internal state of the library to the running set of counters.
- int bgl_perfctr_update: Updates the virtual counters with the current value of the hardware counters.
- int bgl_perfctr_copy_counters: Updates the virtual counters with the current value of the counters, and provides a copy of the virtual counter values in the supplied buffer.
- int bgl_perfctr_copy_hwstate: Updates the virtual counters with the current value of the counters, and provides a copy of the complete internal state of the library in the supplied buffer. This dump includes the information of all configured counters as well as the value of the virtual counters after the update.
- int bgl_perfctr_dump_state: Dumps the complete state of the library to a provided file handle. This function is mainly intended for debugging code that uses the bgl_perfctr interface.
- bgl_perfctr_control_t* bgl_perfctr_hwstate: Gets a pointer to the internal state of the bgl_perfctr interface.
- int bgl_perfctr_get_counters: Takes the lock on the internal virtual counters and updates the virtual counters with the current value of the hardware counters. The function returns without releasing the lock.
- int bgl_perfctr_release_counters: Releases the lock taken by bgl_perfctr_get_counters().

The end user is typically not interested in accessing the content of the control registers in the `bgl_perfctr` control structure, but the information is available. For asymmetric counters where read and write bit patterns are not the same, `bgl_perfctr` uses the write pattern. That is, any time `bgl_perfctr` reads a counter control register state from the hardware, it is translated into its corresponding write bit-order in the library layer.

9.2.2 Ways to access the counters

Because the counters are a shared resource, you must use care when accessing the virtual counters. Under normal conditions, the use of the library interface is straightforward. When multiple agents are involved in accessing the counter, substrate application code must take this into account or results may appear confusing.

Since the virtual counters may be updated by either of the cores and by interval timer controlled interrupts, the value of the virtual counters may change between a user induced counter update and a subsequent access to the memory location of the virtual counter. Depending on the degree of control users of the library need on this behavior, you can use any of the calling sequences in the following sections.

Counter update and copy-out

A call to the `bgl_perfctr_copy_counters()`function updates the internal virtual counters and copies their updated values to the user-provided memory buffer. The update and copy are made within a lock of the virtual counters to guarantee coherence.

Counter update and immediate access

In cases where the user knows that no other agent will access the counters in between an initiated virtual counter update and a read-out of the counter values, or if such updates have negligible influence on the results, he can use the `bgl_perfctr_update()` function. After the update, the user can read the current values of the virtual counters from the SRAM memory region. The memory address of the virtual counters are given by `bgl_perfctr_hwstat()->virtual`.

There should be only a short code path between the call to the update function and the readout of the counters. Further updates to the counters may occur if user code on the other core executes the update function or if the timed update feature sets in. With a short code path, such updates produce low amount of update increments to the virtual counters.

Counter update and lock

Advanced users who want complete control of the behavior of the library between the counter update and counter readout without taking the overhead of the `bgl_pefctr_copy_counters()` function can use the acquire-and-lock function provided in `bgl_perfctr_get_counters()`. This call acquires a lock of the virtual counters and then updates their content with the current value of the hardware counters.

While the lock is held, timed interrupt updates of the counters from any core are automatically disabled. Also access to the virtual counters from the other CPU core is blocked. Application code can read the content of the virtual counter content as explained in the previous section. It is essential that the lock of the virtual counters is released by the `bgl_perfctr_release_counters()` function.

9.2.3 Available counter events

`bgl_perfctr` provides a static array, `BGL_PERFCTR_event_table[]` (Example 9-3), with one entry per hardware event on the Blue Gene/L compute node.

Example 9-3 Event information table BGL_PERFCTR_event_table

```
BGL_PERFCTR_event_encoding {
  unsigned int group;       /* Which counter group to use */
  unsigned int counter;     /* Which counter {A,B,C} to use */
  unsigned int code;        /* Which hw-counter code */
} BGL_PERFCTR_event_encoding_t;

typedef const struct BGL_PERFCTR_event_descr  {
  BGL_PERFCTR_event_num_t event_num;
  int num_encodings;
  u_int64_t mapping;
  BGL_PERFCTR_event_encoding_t encoding[BGL_PERFCTR_MAX_ENCODINGS];
  const char *event_name;
  const char *event_descr;
} BGL_PERFCTR_event_descr_t;

BGL_PERFCTR_event_descr_t
  BGL_PERFCTR_event_table[BGL_PERFCTR_NUMEVENTS];
```

This table is indexed using a C enumerated type and the event number. It can be used to learn all details about the event. For each event, the num_encodings field denotes in how many different locations of the hardware the event can be located. For each such location, the encoding[] field lists the counter group, the counter number within the group, and the actual code used to program the event in that location.

The event table also provides fields for the mnemonic name of the event and a description of the event to facilitate event number to descriptive string translations. This table provides easy and accurate access to information about possible counter allocations and event descriptions. It does not need to be used by the user.

9.2.4 Correct API usage

The bgl_perfctr library and its API are an abstraction of the underlying hardware. In such, it shares some of the properties of the physical counters. This becomes important when used by advanced users in a multithreaded fashion. Predictable behavior is the result when the recommendations listed in the following sections are honored.

Using the second CPU

Calls to the bgl_perfctr library can be made from either CPU on the compute node. The library does the necessary locking internally to guarantee coherency of the virtual counters with the hardware counters.

Calls that modify counter control register content can be used on either CPU core. `bgl_perfctr_add_event()`, `bgl_perfctr_remove_event()`, `bgl_perfctr_revoke()` work transparently by the internal use of the reference count in the library. Thus, if the same event is added by both cores, this results in the reference count of that event to be two. The event starts counting at the first time the `bgl_perfctr_commit()` function is called after the event is added. The event does not disappear from the configured counters until the reference count drops to zero and a subsequent commit operation is performed by any core.

Library initialization is either a local or global operation, depending on the mode selected. Initializing the user level counters using the `bgl_perfctr_init()` function is equivalent to `bgl_perfctr_init_synch` with an argument of BGL_PERFCTR_MODE_LOCAL. In this mode as well as in the other modes, the virtual counter structure is a shared resource between the CPU cores on the compute node. In local mode, it is the responsibility of the user to make sure that calls to `bgl_perfctr_update()` are performed frequently enough to ensure that the 32-bit hardware counters do not collect more than 2^{32}–1 events in-between calls. `Bgl_perfctr_update()` can be called directly, but also indirectly using the `bgl_perfctr_copy_counters()` and `bgl_perfctr_copy_state()` functions.

Automatic prevention of counter overflow can be achieved by providing the argument BGL_PERFCTR_MODE_ASYNC or BGL_PERFCTR_MODE_SYNC to `bgl_perfctr_init_synch()`. In this mode, a user level timed interrupt is installed that executes a virtual counter update within the passing of 2^{32} CPU cycles.

The two modes differ in their global synchronization behavior. The synchronous mode executes a global barrier using the global barrier network together with local synchronization within the node using the CPU lockbox. The asynchronous mode does not perform this synchronization before starting the interval timer interrupts. A safety timeout of five seconds is used in the global barrier to safe-guard for the cases when the global barrier is not available, for example, when not all nodes on a partition have user code loaded.

The core synchronization on the local node is performed on all nodes that have two user applications loaded. This means that virtual node mode can use the synchronous mode successfully in all cases where there is at least one process running on each node. Any nodes with two processes on them take appropriate action to guarantee synchronization within the chip in parallel to the internode synchronization.

Counter start, stop, and reset

In `bgl_perfctr`, there is no explicit start, stop, or reset of a counter. The underlying hardware counter starts incrementing at the moment the control word is written into the counter group control register. Start, stop, and reset of counters is accomplished by means of the update function or functions calls that have an update of the virtual counters as a side effect. This function call establishes a baseline for the virtual counters to which later returned values from the same function can be compared.

The PAPI library, which is implemented using `bgl_perfctr`, provides an API with full start, stop, and reset functionality.

Locking semantics of bgl_perfctr

The `bgl_perfctr` interface uses two locks internally to guarantee a coherent view of the counter state. One lock protects updates of the control data of the library, while the other lock is exclusively used to protect the virtual counters against incoherent updates. These two locks are allocated from the set of 64 user-level locks that are available to user code on Blue Gene/L.

Updates of the virtual counters can take place without acquiring a lock of the control structure. Likewise, in most cases, modifications to the counter control registers can take place independently of acquiring a lock of the virtual counters.

The interval timer controlled update of the virtual counters use the virtual counter lock as explained in the following words. When the interrupt handler is called, it attempts to get hold of the counter lock. If locking is successful, it updates the counters and releases the lock. If the handler fails in acquiring the lock, it is because user level code, or an interrupt handler on

the other CPU core, is performing an update. In this case, this instance of the handler immediately exits since no further virtual counter update is necessary.

9.3 PAPI implementation

The PAPI implementation on Blue Gene/L is based on the 2.3.4 release available at the time when the work was initiated. The PAPI library consists of two parts: the common library API and a substrate interface. The substrate interface (often called *substrate*) contains all the platform-specific code in a PAPI implementation, while the main code is identical among all platform implementations. This particular port of PAPI to the Blue Gene/L CNK conforms to this with a few minor modifications as detailed in 9.3.3, “Modifications to PAPI” on page 112.

9.3.1 linux-bgl PAPI substrate

The PAPI substrate for the Blue Gene/L CNK is located in a subdirectory of the PAPI distribution named **`linux-bgl`**. The substrate is built in top of `bgl_perfctr` and uses this API for all hardware counter manipulation. The substrate enables a fully functional PAPI v2 library, including overlapping counters.

Due to lack of operating system support and the nature of the intended use of the Blue Gene/L machine, the `PAPI_overflow()` function is unimplemented. Also a call to this function returns PAPI_ESBSTR according to library convention.

There is no notion of virtual CPU time in the CNK. For this reason, both `PAPI_get_real_cyc()` and `PAPI_get_virt_cyc()` are mapped to the CPU time base register. By the same reason, `PAPI_get_real_usec()` and `PAPI_get_virt_usec()` report the same amount of elapsed time.

9.3.2 PAPI event mapping for Blue Gene/L

The Blue Gene/L substrate for PAPI includes a default mapping of standard PAPI events to available counters in the Blue Gene/L hardware counter infrastructure. Due to the nature of the application-specific integrated circuit (ASIC) design of Blue Gene/L, many events available on commodity machines are not available on this platform. This typically includes events that are only detectable inside the PPC cores of the ASIC. Examples of such events are L1 cache events, branch prediction events, and instruction counts. The ASIC design of Blue Gene/L makes available to the user a complete new set of events that relate to states in the network controllers on the chip.

Through the PAPI native event mechanism, any event available in the UPC or FPU counters can be programmed and controlled through PAPI. A native event is handled in the same way as the PAPI predefined events and passed through the same API calls. The difference is that instead of passing a PAPI predefined event name, a bit pattern corresponding to the event code and, where applicable, an edge detection mask is used. This is shown in Example 9-4.

Example 9-4 PAPI native event format for Blue Gene/L

```
#include "papi.h"
include "bgl_perfctr.h"

int eventFPU, eventUPC;

/*   Code initializing PAPI not shown here   */
   . . .

/*   Encode a BG/L native event for PAPI   */
eventFPU= BGL_2NDFPU_TRINARY_OP & 0x3FF;
eventUPC= BGL_UPC_L3_PAGE_OPEN & 0x3FF |
            BGL_PERFCTR_UPC_EDGE_RISE << 10;

retval=PAPI_add_event(&evset,eventFPU);
retval=PAPI_add_event(&evset,eventUPC);
```

To simplify the usage of some of the communication-related events and to encourage the usage of these counters, the standard PAPI event mapping has been expanded with several new presets designed for Blue Gene/L. Example 9-5 shows the full set of new events.

Example 9-5 New PAPI nonstandard predefined events on Blue Gene/L

```
PAPI_BGL_OED        Oedipus operations
The event is a convenience name for:
    {BGL_FPU_ARITH_OEDIPUS_OP,0,0}

PAPI_BGL_TS_32B     Torus 32B chunks sent
The event is the sum of the following 6 events:
    {BGL_UPC_TS_XM_32B_CHUNKS,BGL_PERFCTR_UPC_EDGE_RISE,0}
       and similarly for _XP_, _YM_, _YP_, _ZM_ and _ZP_

PAPI_BGL_TS_FULL    Torus no token UPC cycles
The event is the sum of the following 6 events:
    {BGL_UPC_TS_XM_LINK_AVAIL_NO_VCDO_VCD_VCBN_TOKENS,
        BGL_PERFCTR_UPC_EDGE_HI,0},
       and similarly for _XP_, _YM_, _YP_, _ZM_ and _ZP_

PAPI_BGL_TR_DPKT    Tree 256 byte packets
The event is the sum of the following 6 events:
    {BGL_UPC_TR_SNDR_0_VC0_DPKTS_SENT,
        BGL_PERFCTR_UPC_EDGE_RISE,0},
    {BGL_UPC_TR_SNDR_0_VC1_DPKTS_SENT,
        BGL_PERFCTR_UPC_EDGE_RISE,0},
       and similarly for SNDR_1_ and SNDR_2_

PAPI_BGL_TR_FULL  UPC cycles (CLOCKx2) tree rcv is full
The event is the sum of the following 6 events:
    {BGL_UPC_TR_RCV_0_VC0_FULL,BGL_PERFCTR_UPC_EDGE_HI,0},
    {BGL_UPC_TR_RCV_0_VC1_FULL,BGL_PERFCTR_UPC_EDGE_HI,0},
       and similarly for RCV_1_ and RCV_2_
```

The communication events are designed to provide easy aggregated counts of the traffic occurring at each node. The PAPI_BGL_TS_32B event counts the number of all 32 byte data chunks that have been sent from the node. This includes traffic injected at the node and traffic cutting through the network controller. The same holds true for the PAPI_BGL_TR_DPKT event that reports tree network traffic.

For the two duration count events defined, PAPI_BGL_TS_FULL and PAPI_BGL_TR_FULL, the count at each UPC cycle is effectively multiplied by the number of channels experiencing the condition. That is, if both the X-minus and the Y-plus FIFOs experience the condition of no to-kens available, both contribute with one count each UPC clock cycle (every second CPU cycle) until sufficient token acknowledgements are received.

9.3.3 Modifications to PAPI

The standard PAPI distribution, excluding the Blue Gene/L specific substrate, is unchanged from the official release version but for the following modifications.

- A set of new predefined events were added to the existing set of events. These events are:
  - PAPI_BGL_OED (Oedipus operations in FPU0)
  - PAPI_BGL_TS_32B (No. 32 byte packets sent on torus network)
  - PAPI_BGL_TS_FULL (No. UPC cycles × torus links with no available tokens)
  - PAPI_BGL_TR_DPKT (No. packets sent on the tree network)
  - PAPI_BGL_TR_FULL (UPC cycles × No. full tree receivers)
- The semantics of `PAPI_library_init()` is changed from the standard distribution. In Blue Gene/L, `PAPI_library_init()` is a synchronizing call that should be executed by all processes on the partition. It uses the global barrier with a pre-set time-out to initiate the periodic timers that prevent counter overflows. This assures that these interrupts are localized in time over the set of allocated nodes. In virtual node mode, this means that `PAPI_library_init` should be called by all processes, including the processes running on CPU1 on each node.
- When `PAPI_library_init()` is called on a partition where not all nodes are participating in the call, a global barrier time-out occurs and no global synchronization is achieved.

9.4 Examples of using HPM libraries for Blue Gene/L

This section provides some examples of how the Hardware Performance Monitor (HPM) on Blue Gene/L.

9.4.1 PAPI library usage examples

Example 9-6 shows an example program using the PAPI library API. This examples illustrates the configuration of five counters into an event set as well as, start, stop, read and reset of this event set. Measurements are taken over the `fpmaddv` subroutine, which is a naïve implementation of a FMA-like operation on three input vectors and one output vector using the Blue Gene/L specific FPMA operation.

In the experiment, five counters are set up. The counters used are the time base register and the four floating point unit registers. The order of the events when printed out is:

1. PAPI_TOT_CYC
2. BGL_FPU_ARITH_OEDIPUS_OP
3. BGL_2NDFPU_ARITH_OEDIPUS_OP
4. BGL_FPU_LDST_QUAD_LD
5. BGL_2NDFPU_LDST_QUAD_LD

The counters are started, some load operations are performed, and then the vectorized FMA routine is called. After this, the counters are read, but left running. Before repeating the call to the FMA routine, the running counters are reset to zero, without stopping or re-starting. The FMA routine is called and the counters are stopped.

To illustrate the effect of using both floating point units, the code is run both in coprocessor mode and virtual node mode. As expected, the registered number of counts is zero in the second floating point unit when run in co-processor mode. In virtual node, mode counts are registered in both units, since both units are active. This illustrates the property of the counters that the hardware counters are a shared resource between the two processes on the node in virtual node mode. The example also illustrates that the library interface itself resolves the multiple accesses to the hardware as well as the virtualized counters. Although both processes create an event set and add counters to it, the library recognizes that the same hardware counter can be reused. Similarly, when a process releases a counter, the underlying hardware counter may remain allocated, if it is used by the other processor.

Example 9-6 PAPI example code

```
#include <stdio.h>
#include <stdlib.h>
#include "papi.h"
#include "bgl_perfctr_events.h"


#define N 8
#define NCOUNTS 5

int main(int argc, char* argv[]) {
  double v1[N], v2[N], v3[N], r1[N], r2[N];
  double a=1.01,b=1.02,c=1.03,t=0.0,t2=0.0;
  int i, rank;
  int perr, ev_set;
  int encoding;
  long_long counts[NCOUNTS];

#include "bglpersonality.h"
#include "rts.h"

  if(PAPI_VER_CURRENT!=
    (perr=PAPI_library_init(PAPI_VER_CURRENT)))
    printf("\nPAPI_library_init failed. %s\n",PAPI_strerror(perr));

 {
   BGLPersonality me;
   rts_get_personality(&me,sizeof(me));
   if(me.xCoord != 0 ) goto fine;
   if(me.yCoord != 0 ) goto fine;
   if(me.zCoord != 0 ) goto fine;
 }
  for(i=0;i<N;i++) {
    v1[i]=1.01+0.01*i;
    v2[i]=2.01+0.01*i;
    v3[i]=3.01+0.01*i;
    r1[i]=v1[i]*v2[i]+v3[i];
  }
if((perr=PAPI_create_eventset(&ev_set)))
    printf("\nPAPI_create_eventset failed. %s\n",PAPI_strerror(perr));

  /*
  encoding=( BGL_FPU_ARITH_MULT_DIV & 0x3FF );
  encoding=( BGL_FPU_ARITH_ADD_SUBTRACT & 0x3FF );
  encoding=( BGL_FPU_ARITH_TRINARY_OP & 0x3FF );
  */

  if((perr=PAPI_add_event(&ev_set,PAPI_TOT_CYC)))
```

```
    printf("PAPI_add_event failed. %s\n",PAPI_strerror(perr));

  encoding=( BGL_FPU_ARITH_OEDIPUS_OP & 0x3FF );
  if((perr=PAPI_add_event(&ev_set,encoding)))
    printf("\nPAPI_add_event failed. %s\n",PAPI_strerror(perr));

  encoding=( BGL_2NDFPU_ARITH_OEDIPUS_OP & 0x3FF );
  if((perr=PAPI_add_event(&ev_set,encoding)))
    printf("\nPAPI_add_event failed. %s\n",PAPI_strerror(perr));

  encoding=( BGL_FPU_LDST_QUAD_LD & 0x3FF );
  if((perr=PAPI_add_event(&ev_set,encoding)))
    printf("\nPAPI_add_event failed. %s\n",PAPI_strerror(perr));

  encoding=( BGL_2NDFPU_LDST_QUAD_LD & 0x3FF );
  if((perr=PAPI_add_event(&ev_set,encoding)))
    printf("\nPAPI_add_event failed. %s\n",PAPI_strerror(perr));

printf("\nAssigning a vector of length %1d and computing "
    "A()=B()*C()+D().\n",N);
if((perr=PAPI_start(ev_set)))
    printf("\nPAPI_start_event failed. %s\n",PAPI_strerror(perr));

  for(i=0;i<N;i++) r2[i]=-1.001;
  fpmaddv(N,v1,v2,v3,r2);

  if((perr=PAPI_read(ev_set,counts)))
    printf("PAPI_read failed. %s\n",PAPI_strerror(perr));

  printf("Counts registered: ");
  for(i=0;i<NCOUNTS;i++) printf("  %12llu",counts[i]);
  printf("\n");

for(i=0;i<N;i++) {
    printf(" %g * %g + % g = %g  (%g)\n",
      v1[i],v2[i],v3[i],r2[i],r1[i]);
  }

  for(i=0;i<N;i++) r2[i]=-1.001;

  printf("\nResetting the running counter and com-puting "
    "A(1:%1d)=B()*C()+D().\n",N);

  if((perr=PAPI_reset(ev_set)))
    printf("\nPAPI_reset failed. %s\n",PAPI_strerror(perr));

  fpmaddv(N,v1,v2,v3,r2);

if((perr=PAPI_stop(ev_set,counts)))
printf("PAPI_stop failed. %s\n",PAPI_strerror(perr));

  for(i=0;i<N;i++) {
    printf(" %g * %g + % g = %g  (%g)\n",
      v1[i],v2[i],v3[i],r2[i],v1[i]*v2[i]+v3[i]);
  }

  printf("Testing to read stopped counters\n");
  if((perr=PAPI_read(ev_set,counts)))
    printf("PAPI_read failed. %s\n",PAPI_strerror(perr));
```

```
  printf("Counts registered: ");
  for(i=0;i<NCOUNTS;i++) printf("  %12llu",counts[i]);
  printf("\n");

 fine:
  PAPI_shutdown();
  return 0
```

When looking at the output generated by the program, when executed in co-processor mode (Example 9-7), there are no surprises. When run in virtual node mode (Example 9-8) the output has been compressed somewhat to make the results fit onto one page. In the virtual node mode case, the two processes (and the two cores) are running with no synchronization between the cores after the initial synchronization at PAPI library initialization. At each core, vectors of length eight are processed. This is the reason for detecting four double operations on the local FPU.

The experiment illustrates that the counter reads are naturally only synchronized with the local program activity in the local core, unless specifically programmed to do so. In the illustrated output, process 0 and process 32, which ran on the same node with process 0 on core 0, apparently did not execute the first section of the test example simultaneously. That's because no counts were generated in the non-local FPU during the execution of the local floating point activity. The reason behind this is the serialization introduced by printouts to stdout from the processes. In the second part of the experiment, core1 did its local counter reset and reads so that it saw the events generated in FPU0.

Example 9-7 shows sample output from the application shown in Example 9-6, running in coprocessor mode.

Example 9-7 Running the PAPI example code in coprocessor mode

```
program is loading...ok
program is running

stdout[0]:
stdout[0]: Assigning a vector of length 8 and computing A()=B()*C()+D().
stdout[0]: Counts registered:          9572           4           0         140
0
stdout[0]:  1.01 * 2.01 +  3.01 = 5.0401  (5.0401)
stdout[0]:  1.02 * 2.02 +  3.02 = 5.0804  (5.0804)
stdout[0]:  1.03 * 2.03 +  3.03 = 5.1209  (5.1209)
stdout[0]:  1.04 * 2.04 +  3.04 = 5.1616  (5.1616)
stdout[0]:  1.05 * 2.05 +  3.05 = 5.2025  (5.2025)
stdout[0]:  1.06 * 2.06 +  3.06 = 5.2436  (5.2436)
stdout[0]:  1.07 * 2.07 +  3.07 = 5.2849  (5.2849)
stdout[0]:  1.08 * 2.08 +  3.08 = 5.3264  (5.3264)
stdout[0]:
stdout[0]: Resetting the running counter and computing A(1:8)=B()*C()+D().
stdout[0]:  1.01 * 2.01 +  3.01 = 5.0401  (5.0401)
stdout[0]:  1.02 * 2.02 +  3.02 = 5.0804  (5.0804)
stdout[0]:  1.03 * 2.03 +  3.03 = 5.1209  (5.1209)
stdout[0]:  1.04 * 2.04 +  3.04 = 5.1616  (5.1616)
stdout[0]:  1.05 * 2.05 +  3.05 = 5.2025  (5.2025)
stdout[0]:  1.06 * 2.06 +  3.06 = 5.2436  (5.2436)
stdout[0]:  1.07 * 2.07 +  3.07 = 5.2849  (5.2849)
stdout[0]:  1.08 * 2.08 +  3.08 = 5.3264  (5.3264)
stdout[0]: Testing to read stopped counters
```

```
stdout[0]: Counts registered:          8486           4           0          140
0
Checking status

program terminated successfully
```

Example 9-8 shows output from the same application, but this time running in virtual node mode.

Example 9-8 Running the PAPI example in virtual node mode

```
program is running

stdout[32]:
stdout[0]:
stdout[32]: Assigning a vector of length 8 and computing A()=B()*C()+D().
stdout[0]: Assigning a vector of length 8 and computing A()=B()*C()+D().
stdout[32]: Counts registered:          9776           0           4           0
140
stdout[0]: Counts registered:          9664           4           0          140
0
stdout[32]:  1.01 * 2.01 +  3.01 = 5.0401  (5.0401)
stdout[0]:  1.01 * 2.01 +  3.01 = 5.0401  (5.0401)

          …
stdout[32]:  1.08 * 2.08 +  3.08 = 5.3264  (5.3264)
stdout[0]:  1.08 * 2.08 +  3.08 = 5.3264  (5.3264)
stdout[32]:
stdout[0]:
stdout[32]: Resetting the running counter and computing A(1:8)=B()*C()+D().
stdout[0]: Resetting the running counter and computing A(1:8)=B()*C()+D().
stdout[32]:  1.01 * 2.01 +  3.01 = 5.0401  (5.0401)
stdout[0]:  1.01 * 2.01 +  3.01 = 5.0401  (5.0401)

                  …
stdout[32]:  1.08 * 2.08 +  3.08 = 5.3264  (5.3264)
stdout[0]:  1.08 * 2.08 +  3.08 = 5.3264  (5.3264)
stdout[32]: Testing to read stopped counters
stdout[0]: Testing to read stopped counters
stdout[32]: Counts registered:          8474           4           4          188
140
stdout[0]: Counts registered:          9638           4           0          140
128
Checking status

program terminated successfully
```

A second test example is included here as well. In this example, a similar code (Example 9-9) is used, but a much larger number of counts is generated. This example illustrates the transparent 32-bit overflow protection in the performance counter API. In contrast to the previous example, the computation routine used here utilizes a standard FMA instruction and not the Blue Gene/L-specific FPMA instruction.

The counters used in this experiment are the time base register and the four floating point unit registers. The order of the events when printed out is:

1. PAPI_TOT_CYC
2. BGL_FPU_ARITH_TRINARY_OP
3. BGL_2NDFPU_ARITH_TRINARY_OP

4. BGL_FPU_LDST_QUAD_LD
5. BGL_2NDFPU_LDST_QUAD_LD

The experiment is set up to perform 4.4·109 trinary operations, which exceeds 232 as shown in the generated output in co-processor mode (Example 9-10 on page 118) as well as in virtual node mode (Example 9-11 on page 119). The output illustrates that the library correctly protects against 32-bit convolution errors.

Example 9-9 PAPI example code exercising 32-bit overflow protection

```
#include <stdio.h>
#include <stdlib.h>
#include "papi.h"
#include "bgl_perfctr_events.h"

#undef FPMA  // Use the FPM version of the com-putation

#define N 4000000
#define NITER 1100
#define NCOUNTS 5

int main(int argc, char* argv[]) {
  double v1[N], v2[N], v3[N], r1[N], r2[N];
  double a=1.01,b=1.02,c=1.03,t=0.0,t2=0.0;
  int i, rank, iter;
  int perr, ev_set;
  int encoding;
  long_long counts[NCOUNTS];

#include "bglpersonality.h"
#include "rts.h"

  if(PAPI_VER_CURRENT!=(perr=PAPI_library_init(PAPI_VER_CURRENT)))
    printf("PAPI_library_init failed. %s\n",PAPI_strerror(perr));

 {
   BGLPersonality me;
   rts_get_personality(&me,sizeof(me));
   if(me.xCoord != 0 ) goto fine;
   if(me.yCoord != 0 ) goto fine;
   if(me.zCoord != 0 ) goto fine;
for(i=0;i<N;i++) {
    v1[i]=1.01+0.01*i;
    v2[i]=2.01+0.01*i;
    v3[i]=3.01+0.01*i;
    r1[i]=v1[i]*v2[i]+v3[i];
  }

  for(i=0;i<N;i++) r2[i]=-1.001;


  if((perr=PAPI_create_eventset(&ev_set)))
    printf("PAPI_create_eventset failed. %s\n",PAPI_strerror(perr));

  if((perr=PAPI_add_event(&ev_set,PAPI_TOT_CYC)))
    printf("PAPI_add_event failed. %s\n",PAPI_strerror(perr));

  encoding=( BGL_FPU_ARITH_TRINARY_OP & 0x3FF );
  if((perr=PAPI_add_event(&ev_set,encoding)))
    printf("\nPAPI_add_event failed. %s\n",PAPI_strerror(perr));
```

```
  encoding=( BGL_2NDFPU_ARITH_TRINARY_OP & 0x3FF );
  if((perr=PAPI_add_event(&ev_set,encoding)))
    printf("\nPAPI_add_event failed. %s\n",PAPI_strerror(perr));

  encoding=( BGL_FPU_LDST_DBL_LD & 0x3FF );
  if((perr=PAPI_add_event(&ev_set,encoding)))
    printf("\nPAPI_add_event failed. %s\n",PAPI_strerror(perr));

  encoding=( BGL_2NDFPU_LDST_DBL_LD & 0x3FF );
  if((perr=PAPI_add_event(&ev_set,encoding)))
    printf("\nPAPI_add_event failed. %s\n",PAPI_strerror(perr));
if((perr=PAPI_start(ev_set)))
    printf("PAPI_start_event failed. %s\n",PAPI_strerror(perr));


  printf("\n\nPerforming %d iterations of vector operations for\n"
    "a total of %lld (0x%llx) number of FMAs\n",
    NITER,((long long)NITER)*N,((long long)NITER)*N);

  for(iter=0;iter<NITER;iter++) {

    if(iter%100==0)
      printf("\t----  Iteration %4.4d of %4.4d ----\n",iter,NITER);

#ifdef FPMA
    fpmaddv(N,v1,v2,v3,r2);
#else
    fmaddv(N,v1,v2,v3,r2);
#endif

  }

  if((perr=PAPI_stop(ev_set,counts)))
    printf("PAPI_stop failed. %s\n",PAPI_strerror(perr));

  printf("Counts registered: ");
  for(i=0;i<NCOUNTS;i++) printf("  %12llu",counts[i]);
  printf("\n");

 fine:
  PAPI_shutdown();
  return 0;
}
```

Example 9-10 Running the PAPI overflowing example code in coprocessor mode

```
program is running

stdout[0]:
stdout[0]:
stdout[0]: Performing 1100 iterations of vector operations for
stdout[0]: a total of 4400000000 (0x10642ac00) number of FMAs
stdout[0]: Time base: 915546797451
stdout[0]:      ----  Iteration 0000 of 1100 ----
stdout[0]:      ----  Iteration 0100 of 1100 ----
stdout[0]:      ----  Iteration 0200 of 1100 ----
stdout[0]:      ----  Iteration 0300 of 1100 ----
```

```
stdout[0]:      ----  Iteration 0400 of 1100 ----
stdout[0]:      ----  Iteration 0500 of 1100 ----
stdout[0]:      ----  Iteration 0600 of 1100 ----
stdout[0]:      ----  Iteration 0700 of 1100 ----
stdout[0]:      ----  Iteration 0800 of 1100 ----
stdout[0]:      ----  Iteration 0900 of 1100 ----
stdout[0]:      ----  Iteration 1000 of 1100 ----
stdout[0]: Counts registered:   85820449687   4400000000           0  13200000232
0
Checking status

program terminated successfully
```

Example 9-11 Running the PAPI overflowing example code in virtual node mode

```
program is running

stdout[0]:
stdout[32]:
stdout[0]:
stdout[32]:
stdout[0]: Performing 1100 iterations of vector operations for
stdout[32]: Performing 1100 iterations of vector operations for
stdout[0]: a total of 4400000000 (0x10642ac00) number of FMAs
stdout[32]: a total of 4400000000 (0x10642ac00) number of FMAs
stdout[0]:      ----  Iteration 0000 of 1100 ----
stdout[32]:     ----  Iteration 0000 of 1100 ----
stdout[0]:      ----  Iteration 0100 of 1100 ----
stdout[32]:     ----  Iteration 0100 of 1100 ----
                  …
stdout[32]:     ----  Iteration 1000 of 1100 ----
stdout[0]:      ----  Iteration 1000 of 1100 ----
stdout[32]: Counts registered:  109898564159   4400000000   4400000000  13200000137
13200000174
stdout[0]: Counts registered:  109898570635   4400000000   4400000000  13200000246
13200000235
Checking status

program terminated successfully
```

9.4.2 bgl_perfctr usage example

Example 9-12 illustrates the usage of the lower-level substrate. To make visible the behavior of the internals of the library to different function calls, heavy use of the `bgl_perfctr_dump_state()` function is used. In normal operation, this function is not used, but it is helpful to illustrate the changes in the internal state of the control structure.

The code in Example 9-12 performs the following operations at the time of each counter state dump.

1. Initializing the library
2. Scheduling an event for addition
3. Scheduling a second event for addition
4. Committing pending configuration changes
5. Scheduling an event for removal
6. Revoking pending changes

7. Updating a virtual counter to establish a counter baseline
8. Updating a second virtual counter to see how many counts were aggregated
9. Updating a third virtual counter to increment the virtual counters with events since the last update

Example 9-12 bgl_perfctr example code

```
#include <stdio.h>
#include <stdlib.h>
#include "bgl_perfctr.h"
#include "bgl_perfctr_events.h"
#define EV1 BGL_UPC_L3_CACHE_HIT
#define EV2 BGL_UPC_L3_CACHE_MISS_DATA_WILL_BE_REQED_DDR
//#define EV1 BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR
//#define EV2 BGL_UPC_L3_EDRAM_ACCESS_CYCLE

int main() {

  bgl_perfctr_control_t *hwctrs;
  BGL_PERFCTR_event_t ev;
  int i,n,err,rank;
  int *memarea;

#include "bglpersonality.h"
#include "rts.h"
  {
    BGLPersonality me;
    rts_get_personality(&me,sizeof(me));
    if(me.xCoord != 0 ) goto fine;
    if(me.yCoord != 0 ) goto fine;
    if(me.zCoord != 0 ) goto fine;
  }

  if(bgl_perfctr_init())
    abort();

  bgl_perfctr_dump_state(stdout);
ev.edge=0x1;
  ev.num=EV1;
  err=bgl_perfctr_add_event(ev);
  if(err) {
    printf("Add event line %d failed.\n",__LINE__-2);
    exit(1);
  } else printf("One event added. %s\n",BGL_PERFCTR_event_table[EV1].event_name);

  bgl_perfctr_dump_state(stdout);

  ev.num=EV2;
  err=bgl_perfctr_add_event(ev);
  if(err) {
    printf("Add event line %d failed.\n",__LINE__-2);
    exit(1);
  } else printf("One more event added. %s\n",BGL_PERFCTR_event_table[EV2].event_name);

  bgl_perfctr_dump_state(stdout);

  err=bgl_perfctr_commit();
  if(err) {
    printf("Commit %d failed.\n",__LINE__-2);
    exit(1);
```

```
  } else printf("Commit successful.\n");

  bgl_perfctr_dump_state(stdout);

  ev.num=EV1;
  err=bgl_perfctr_remove_event(ev);
  if(err) {
    printf("Remove %d failed.\n",__LINE__-2);
    exit(1);
  } else printf("Remove successful.\n");

  bgl_perfctr_dump_state(stdout);
err=bgl_perfctr_revoke();
  if(err) {
    printf("Commit %d failed.\n",__LINE__-2);
    exit(1);
  } else printf("Commit successful.\n");

  bgl_perfctr_dump_state(stdout);

  printf("\n\n----------------------\n\n");

  printf("\n bgl_perfctr_update \n");
  bgl_perfctr_update();
  bgl_perfctr_dump_state(stdout);

  n=1024*1024;
  memarea=(int *) malloc(1024*1024*sizeof(int));
  for(i=0;i<n;i++)
    memarea[i]=n-1;

  printf("\n bgl_perfctr_update again after loop\n");
  bgl_perfctr_update();
  bgl_perfctr_dump_state(stdout);

  for(i=0;i<n;i++)
    memarea[i]-=1;

  printf("\n bgl_perfctr_update again after loop\n");
  bgl_perfctr_update();
  bgl_perfctr_dump_state(stdout);

  if(bgl_perfctr_shutdown())
    abort();

fine:

  return 0;
```

Example 9-13 shows the output from running the program in Example 9-12.

Example 9-13 Running the bgl_perfctr example code

```
program is running

stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   0 defined events. in_use=0x00000000 modified=0x00000000
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]: Id     Event    H/W     CtrlReg  RefCount NewCount
stdout[0]: Current cached values in the active counters
stdout[0]:             Last              Virtual
stdout[0]: One event added. BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR
stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   1 defined events. in_use=0x00000000 modified=0x00000010
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
```

```
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00081000 M     0     0   -  |      1     1   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]:  Id    Event   H/W    CtrlReg  RefCount NewCount
stdout[0]:   0:    17      13       4       0        1
(BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR)
stdout[0]: Current cached values in the active counters
stdout[0]:             Last               Virtual
stdout[0]: One more event added. BGL_UPC_PUO_DCURD_WAIT_L3
stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   2 defined events. in_use=0x00000000 modified=0x00000050
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00081000 M     0     0   -  |      1     1   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00005000 M     0     0   -  |      0     5   -  |      0     0  -
c-mode=0
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
```

```
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]: Id    Event   H/W    CtrlReg  RefCount NewCount
stdout[0]:  0:    17      13       4        0        1
(BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR)
stdout[0]:  1:    66      19       6        0        1 (BGL_UPC_PUO_DCURD_WAIT_L3)
stdout[0]: Current cached values in the active counters
stdout[0]:             Last               Virtual
stdout[0]: Commit successful.
stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   2 defined events. in_use=0x00082000 modified=0x00000000
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00081000 -     0     0   -  |      1     1   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00005000 -     0     0   -  |      0     5   -  |      0     0  -
c-mode=0
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
```

```
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]: Id    Event   H/W    CtrlReg  RefCount NewCount
stdout[0]:   0:    17      13       4        1        1
(BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR)
stdout[0]:   1:     66       19        6        1        1 (BGL_UPC_PUO_DCURD_WAIT_L3)
stdout[0]: Current cached values in the active counters
stdout[0]:             Last               Virtual
stdout[0]:  13:            0                   0
stdout[0]:  19:            0                   0
stdout[0]: Remove successful.
stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   2 defined events. in_use=0x00082000 modified=0x00000000
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00081000 -     0     0   -  |      1     1   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00005000 -     0     0   -  |      0     5   -  |      0     0  -
c-mode=0
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]: Id    Event   H/W    CtrlReg  RefCount NewCount
stdout[0]:  0:    17      13       4        1        0
(BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR)
stdout[0]:  1:    66      19       6        1        1 (BGL_UPC_PUO_DCURD_WAIT_L3)
stdout[0]: Current cached values in the active counters
stdout[0]:             Last               Virtual
stdout[0]:  13:            0                   0
stdout[0]:  19:            0                   0
```

```
stdout[0]: Revoke successful.
stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   2 defined events. in_use=0x00082000 modified=0x00000000
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00081000 -     0     0   -  |      1     1   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00005000 -     0     0   -  |      0     5   -  |      0     0  -
c-mode=0
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]: Id  Event   H/W    CtrlReg  RefCount NewCount
stdout[0]:  0:   17     13       4       1       1
(BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR)
stdout[0]:  1:   66     19       6       1       1 (BGL_UPC_PUO_DCURD_WAIT_L3)
stdout[0]: Current cached values in the active counters
stdout[0]:             Last               Virtual
stdout[0]:  13:            0                    0
stdout[0]:  19:            0                    0
stdout[0]:
stdout[0]:  bgl_perfctr_update
stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   2 defined events. in_use=0x00082000 modified=0x00000000
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
```

```
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00081000 -     0     0   -  |      1     1   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00005000 -     0     0   -  |      0     5   -  |      0     0  -
c-mode=0
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]: Id   Event   H/W    CtrlReg  RefCount NewCount
stdout[0]:  0:   17      13       4        1        1
(BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR)
stdout[0]:  1:   66      19       6        1        1 (BGL_UPC_PUO_DCURD_WAIT_L3)
stdout[0]: Current cached values in the active counters
stdout[0]:             Last               Virtual
stdout[0]:  13:         8318                  8318
stdout[0]:  19:       231293                231293
stdout[0]:
stdout[0]:  bgl_perfctr_update again after loop
stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   2 defined events. in_use=0x00082000 modified=0x00000000
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00081000 -     0     0   -  |      1     1   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00005000 -     0     0   -  |      0     5   -  |      0     0  -
c-mode=0
```

```
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]: Id   Event    H/W    CtrlReg  RefCount NewCount
stdout[0]:  0:    17     13       4        1        1
(BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR)
stdout[0]:  1:    66     19       6        1        1 (BGL_UPC_PUO_DCURD_WAIT_L3)
stdout[0]: Current cached values in the active counters
stdout[0]:             Last               Virtual
stdout[0]:  13:       133235              133235
stdout[0]:  19:      1727334             1727334
stdout[0]:
stdout[0]:  bgl_perfctr_update again after loop
stdout[0]: -------- bgl_perfctr_dump_state -------
stdout[0]:   2 defined events. in_use=0x00082000 modified=0x00000000
stdout[0]:  Id      code   - Interpretation
stdout[0]:      UPC events  A: edge  code IRQ | B: edge  code IRQ | C: edge code IRQ
stdout[0]:   0: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   1: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   2: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   3: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   4: 0x00081000 -     0     0   -  |      1     1   -  |      0     0  -
c-mode=0
stdout[0]:   5: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   6: 0x00005000 -     0     0   -  |      0     5   -  |      0     0  -
c-mode=0
stdout[0]:   7: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   8: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:   9: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  10: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  11: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
```

```
stdout[0]:  12: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  13: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  14: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:  15: 0x00000000 -     0     0   -  |      0     0   -  |      0     0  -
c-mode=0
stdout[0]:      FPU Hummer  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  16: 0x00000000 -           0   0  |            0   0
stdout[0]: FPU Hummer CPU2  ARITH:   Act Code | LD/ST:   Act Code
stdout[0]:  17: 0x00000000 -           0   0  |            0   0
stdout[0]: Id    Event   H/W    CtrlReg  RefCount NewCount
stdout[0]:  0:    17      13       4        1        1
(BGL_UPC_L3_CACHE_MISS_DATA_ALRDY_WAY_DDR)
stdout[0]:  1:    66      19       6        1        1 (BGL_UPC_PUO_DCURD_WAIT_L3)
stdout[0]: Current cached values in the active counters
stdout[0]:             Last               Virtual
stdout[0]:  13:       170127              170127
stdout[0]:  19:      2064954             2064954


Checking status

program terminated successfully
```

9.5 Conclusions

This chapter detailed the implementation of the user APIs to access and control hardware performance counters on Blue Gene/L. The APIs consist of two libraries: bgl_perfctr and PAPI.

Bgl_perfctr is a low-level abstraction that unifies the behavior of the different counter sources into a single abstraction and which takes care of 64-bit virtualization and automatic overflow protection of virtual event counters. The bgl_perfctr is intended to reflect the hardware implementation of performance counters in a user-friendly way, without hiding the details of this hardware implementation API. Examples in 9.4, “Examples of using HPM libraries for Blue Gene/L” on page 112, illustrate the virtualization to 64-bit counters and the 32-bit overflow protection.

PAPI is a higher-level abstraction which aims to make hardware counter access uniform between different computer platforms using different CPU architectures and from different vendors. This chapter presented specific details about PAPI when implemented on Blue Gene/L including newly introduced PAPI preset events for Blue Gene/L and minor changes to library behavior that are pertinent to Blue Gene/L. In 9.4, “Examples of using HPM libraries for Blue Gene/L” on page 112, shows demonstrations of start, stop, read and reset of 64-bit virtual counters as well as the ability to correctly register events in excess of 2^{32}.

A

Statement of completion

IBM considers installation and integration services complete when the following activities have taken place:

- Service Node powers on and off and reports the system status.
- Rack and system diagnostic runs have completed.
- The ability of the Front End Node to submit the Linpack application to a target 512 Compute Node partition has been demonstrated.
- Linpack has run on a maximum system partition.
- The ability to submit multiple Linpack jobs to multiple partitions simultaneously has been demonstrated.
- The ability to route Ethernet traffic to a destination TCP/IP has been demonstrated.

B

Electromagnetic compatibility

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。

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 mains 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

MPI environment variables

This appendix documents several environment variable that the end user can change to affect the runtime characteristics of Message Passing Interface (MPI) for the application being executed. Usually this is done 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 these variables is by passing 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*, ZG24-6744.

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 may 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, see the following page from the Argonne National Laboratory Web site:

http://www-unix.mcs.anl.gov/mpi/mpich/papers/mpicharticle/node23.html

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 (PPC) 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.

FEN See *Front End Node*.

FGES See *Federated Gigabit-Ethernet Switch*.

Front End Node (FEN) Is responsible, in part, for interactive access to Blue Gene/L.

Federated Gigabit-Ethernet Switch (FGES) Connects the I/O Nodes of Blue Gene/L to external resources, such as the FEN and the CWFS.

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.

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.

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.

GFLOP/s, GOP/s, gigaFLOP/s A billion (109 = 1 000 000 000) 64-bit floating point operations per second.

host complex Includes the Front End Node and Service Node.

Hot Spare Node (HSN)

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 1048576) 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 1000000) bytes.

Message Passing Interface (MPI)

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.

MFLOP/s, MOP/s, or megaFLOP/s A million (106 = 1000000) 64-bit floating point operations per second.

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 may 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.

University Alliances Members of the Academic Strategic Alliances Program (ASAP) of ASCI, academic institutions engaged in accelerating simulation science.

Unified Parallel C (UPC) A programming language with parallel extensions to ANSI C. For an example, see: http://upc.gwu.edu/

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 redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 148. Note that some of the documents referenced here may be available in softcopy only.

- *Workload Management with LoadLeveler*, SG24-6038
- *Linux Clustering with CSM and GPFS*, SG24-6601
- *Blue Gene/L: Hardware Overview and Planning*, SG24-6742
- *Blue Gene/L: Hardware Installation and Serviceability*, ZG24-6743 (available by August 2005)
- *Blue Gene/L: System Administration*, ZG24-6744 (available by August 2005)

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

Online resources

These Web sites and URLs are also relevant as further information sources:

- 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/`

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

D

E

F

G

H

I

K

L

M

P

Q

R

S

T

U

V

X

(0.2”spine)
0.17”<->0.473”
90<->249 pages

Blue Gene/L: Application Development

Redbooks

Explore the Blue Gene/L programming environment

Learn how to run and debug MPI programs

Understand checkpoint and restart, Bridge APIs, and more

This IBM Redbook is the second in a series of internal IBM publications written specifically for the Blue Gene/L supercomputer, which was developed by IBM in collaboration with Lawrence Livermore National Laboratory (LLNL). This redbook provides an overview of the application development environment for Blue Gene/L.

This redbook explains the instances where Blue Gene/L is unique in its programming environment. The book is divided into the following parts:

Prior to reading this book, you must have a strong background in Message Passing Interface (MPI) programming.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

ZG24-6745-00 ISBN 0738491160