Best Practice Guide
- Blue Gene/Q v1.1.1

Philippe Wautelet, IDRIS
Margareta Boiarcuic, IDRIS
Jean-Michel Dupays, IDRIS
Silvia Giuliani, CINECA
Massimiliano Guarrasi, CINECA
Giuseppa Muscianisi, CINECA
Maciej Cytowski, ICM

January 16th, 2014
# Table of Contents

1. Introduction .................................................................................................................. 5  
2. System architecture and configuration ........................................................................ 6  
   2.1. Building blocks ........................................................................................................ 6  
      2.1.1. Chip .................................................................................................................... 7  
      2.1.2. Compute card ...................................................................................................... 7  
      2.1.3. Node card ........................................................................................................... 7  
      2.1.4. Midplane .......................................................................................................... 7  
      2.1.5. I/O drawer ........................................................................................................ 7  
      2.1.6. Rack ................................................................................................................. 7  
      2.1.7. Login nodes ....................................................................................................... 8  
      2.1.8. Service nodes ................................................................................................. 8  
   2.2. Processor architecture ............................................................................................... 8  
   2.3. Memory architecture ............................................................................................... 8  
   2.4. Networks .................................................................................................................. 9  
      2.4.1. Five-dimensional torus ...................................................................................... 9  
      2.4.2. Functional and I/O network ............................................................................. 10  
      2.4.3. Service network .............................................................................................. 10  
   2.5. Operating system ..................................................................................................... 10  
   2.6. I/O subsystem .......................................................................................................... 10  
      2.6.1. I/O nodes ........................................................................................................ 11  
      2.6.2. GPFS servers ................................................................................................. 13  
3. System access ................................................................................................................. 14  
   3.1. How to get an account ............................................................................................ 14  
      3.1.1. FERMI .............................................................................................................. 14  
      3.1.2. JUQUEEN ....................................................................................................... 14  
   3.2. How to contact support .......................................................................................... 14  
      3.2.1. FERMI .............................................................................................................. 14  
      3.2.2. JUQUEEN ....................................................................................................... 15  
   3.3. How to reach the system ......................................................................................... 16  
      3.3.1. FERMI .............................................................................................................. 16  
      3.3.2. JUQUEEN ....................................................................................................... 16  
4. Production environment ................................................................................................. 17  
   4.1. Module environment ............................................................................................... 17  
      4.1.1. The software catalogue .................................................................................... 17  
      4.1.2. The module command .................................................................................... 18  
   4.2. Batch system .......................................................................................................... 18  
      4.2.1. Queue classes ................................................................................................... 19  
      4.2.2. LoadLeveler commands ............................................................................... 21  
      4.2.3. LoadLeveler script syntax ............................................................................ 21  
      4.2.4. Using multiple job steps ............................................................................... 26  
      4.2.5. MPMD jobs ..................................................................................................... 28  
      4.2.6. Sub-block jobs ............................................................................................... 29  
      4.2.7. Serial jobs ...................................................................................................... 32  
      4.2.8. Job monitoring ............................................................................................... 32  
   4.3. Accounting .............................................................................................................. 33  
      4.3.1. Quota schema ................................................................................................. 33  
      4.3.2. Job accounting ............................................................................................... 34  
      4.3.3. Querying quota information .......................................................................... 34  
   4.4. File systems ............................................................................................................ 35  
      4.4.1. Home, scratch, long time storage ................................................................ 35  
      4.4.2. Performance of file systems ......................................................................... 39  
      4.4.3. Backup policies .............................................................................................. 41  
      4.4.4. Archive your data ......................................................................................... 42  
5. Programming environment and basic porting ............................................................... 43  
   5.1. Compilation ............................................................................................................. 43
5.1.1. Available compilers ......................................................... 43
5.1.2. Compiler flags ........................................................... 44
5.2. Available libraries ............................................................ 45
  5.2.1. IBM ESSL (Engineering and Scientific Subroutine Library) ........ 45
  5.2.2. IBM MASS (Mathematical Acceleration SubSystem) .................. 46
  5.2.3. Other libraries ......................................................... 46
5.3. MPI .............................................................................. 47
  5.3.1. Compiling MPI applications ........................................... 47
  5.3.2. MPI extensions for Blue Gene/Q ...................................... 48
5.4. Hybrid MPI/OpenMP .......................................................... 50
  5.4.1. Compiler flags .......................................................... 50
  5.4.2. Running hybrid MPI/OpenMP applications ............................ 50
6. Debugging ....................................................................... 51
  6.1. Compiler flags ............................................................. 51
    6.1.1. Debugging options of the compiler ................................ 51
    6.1.2. Compiler flags for using debuggers ................................ 54
  6.2. Debuggers ..................................................................... 55
    6.2.1. STAT on JUQUEEN ...................................................... 55
    6.2.2. TotalView ................................................................ 56
    6.2.3. GDB ................................................................... 62
  6.3. Analysing core dump files .................................................. 64
    6.3.1. Core dump file analysis using addr2line .......................... 64
    6.3.2. TotalView ................................................................. 65
7. Performance analysis .............................................................. 66
  7.1. Performance analysis tools .................................................. 66
    7.1.1. gprof .............................................................. 66
    7.1.2. Performance Application Programming Interface (PAPI) ....... 69
    7.1.3. SCALASCA ......................................................... 70
    7.1.4. TAU ................................................................. 74
  7.2. Hints for interpreting results ................................................. 76
    7.2.1. Spotting load-imbalance ............................................. 76
8. Tuning and optimisation .......................................................... 76
  8.1. Single core optimisation ..................................................... 76
    8.1.1. Advanced and aggressive compiler flags ............................ 77
    8.1.2. Enabling vectorisation on the Blue Gene/Q architecture ....... 79
    8.1.3. Optimisation strategies .............................................. 80
  8.2. Advanced MPI usage .......................................................... 84
    8.2.1. Tuning and environment variables .................................. 84
    8.2.2. Mapping tasks on node topology ................................... 85
  8.3. Advanced OpenMP usage ..................................................... 88
    8.3.1. Thread affinity ......................................................... 88
  8.4. Memory optimisation .......................................................... 89
    8.4.1. Memory consumption ............................................... 89
    8.4.2. Memory fragmentation .............................................. 90
    8.4.3. Maximum memory available ....................................... 91
  8.5. Transactional memory ........................................................ 92
    8.5.1. Transactional memory support on the Blue Gene/Q ............. 92
    8.5.2. Environment variables and built-in functions ................... 93
    8.5.3. Usage recommendations ............................................ 93
  8.6. Thread level speculation ..................................................... 94
    8.6.1. Rules for committing data ......................................... 94
    8.6.2. Thread-level speculative execution and OpenMP ............... 94
  8.7. I/O optimisation ............................................................... 95
    8.7.1. General guidelines .................................................. 95
    8.7.2. MPI I/O .................................................................. 96
    8.7.3. HDF5 .................................................................. 97
    8.7.4. netCDF .................................................................. 97
    8.7.5. SIONlib ............................................................... 97
Further documentation
1. Introduction

This Best Practice Guide is intended to help users obtain the best productivity from the two PRACE Tier-0 Blue Gene/Q systems, FERMI and JUQUEEN. The information provided about these supercomputers will enable their users to achieve good performance of their applications. The guide covers a wide range of topics from a detailed description of the hardware to information about the basic production environment: how to login, the accounting procedure, information about porting and submitting jobs, and tools and strategies on how to analyse and improve the performance of applications.

The IBM Blue Gene/Q system JUQUEEN is hosted by the Forschungszentrum Jülich (FZJ) in Germany. With a peak performance of 5.9 Petaflop/s, JUQUEEN was the first HPC system in Europe to pass the 5 Petaflop/s barrier (quadrillions of calculations per second). It holds the 8th place in the TOP500 world-wide listing (November 2013).

Information about JUQUEEN is available online on the FZJ website [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html].

Figure 1. IBM Blue Gene/Q JUQUEEN hosted by Forschungszentrum Jülich.

The IBM Blue Gene/Q system FERMI is hosted by CINECA in Italy. It has 10 racks and a peak performance of 2.1 Petaflop/s. It holds the 15th place in the TOP500 world-wide listing (November 2013).

You can find extensive information about FERMI online on the CINECA website [http://www.hpc.cineca.it/content/documentation].
2. System architecture and configuration

IBM Blue Gene/Qs are massively parallel supercomputers composed of several building blocks. The two PRACE Tier-0 systems are JUQUEEN and FERMI.

JUQUEEN has 28 racks (7 rows of 4 racks) with a total of 28,672 nodes, 458,752 cores and 448 TiB of main memory.

FERMI is composed of 10 racks with a total of 10,240 nodes, 163,840 cores and 160 TiB of main memory.

This section describes the main hardware characteristics from the different building blocks to the file systems through the processor and memory architectures, the networks and the I/O subsystem.

More information can be found in the IBM Redbook "IBM System Blue Gene Solution: Blue Gene/Q Application Development". Detailed information is also presented in the "The IBM Blue Gene/Q System" and in the "The IBM Blue Gene/Q Interconnection Network and Message Unit" documents.

2.1. Building blocks

In this section, the structure of the Blue Gene/Q system is presented from the chip up to the whole system.
2.1.1. Chip

At the heart of the Blue Gene/Q, there is a PowerPC A2 chip running at 1.6 GHz with 16 cores for applications, one for the operating system and one as spare. Each chip has a theoretical peak performance of 204.8 Gflop/s.

2.1.2. Compute card

The compute card or compute node contains an 18-core chip with its 32 MiB shared L2 cache, 16 GiB of DDR3 memory and all the connectors for the network and the power supply.

2.1.3. Node card

A node card contains 32 compute cards. Part of the network is directly provided by the node card (the fifth dimension of the 5D torus is entirely contained in each node card).

2.1.4. Midplane

Node cards are grouped by 16 into a midplane. A midplane is the smallest building block to have a true 5D torus network. A midplane contains 512 compute cards for a total of 8,192 processing cores, 8 TiB of main memory and a peak performance of 104.9 Tflop/s.

2.1.5. I/O drawer

Each I/O drawer contains 8 I/O nodes. An I/O node has the same hardware as a compute card. Each I/O node provides both external connectivity and file system access to a group of compute cards (32, 64 or 128, depending on the configuration).

2.1.6. Rack

A rack is composed of 2 midplanes and 1, 2 or 4 I/O drawers.
2.1.7. Login nodes

Login nodes or front-end nodes are used to connect to the Blue Gene/Q system. They provide the only interactive service on the Blue Gene/Q. No interactive access is possible on the compute cards. Front-ends are used to cross-compile for the compute nodes and to submit batch jobs.

2.1.8. Service nodes

Service nodes are only accessible to system administrators. They are dedicated to system management tasks such as database and error reporting.

2.2. Processor architecture

Each compute node has a PowerPC A2 microprocessor chip. A chip contains 18 64-bit cores running at a frequency of 1.6 GHz. 16 of these cores are reserved for computing tasks, one is for the operating system and the last one is a spare. Each core has 32 KiB for its private L1 cache (16 KiB for data and 16 KiB for instructions). There is a 32 MiB L2 cache shared between all the cores. The chip also contains 2 DDR3 memory controllers and the network controllers.

Each core is able to execute up to 4 hardware threads. Each hardware thread can be used for a standard software thread (i.e. an OpenMP or pthread thread) and also for an MPI process. It is possible, therefore, to simultaneously run up to 64 MPI processes or software threads per chip or a combination of MPI processes and threads (for example, 4 MPI processes each with up to 16 OpenMP threads).

Each core can execute up to 8 double-precision floating-point operations during each cycle thanks to its 4-wide multiply-add vector unit. The peak performance of a core is 12.8 Gflop/s and of a chip (using 16 cores) is 204.8 Gflop/s. Two instructions can be executed at each processor cycle: one floating point operation plus one other (integer operation, load, store...). However, these two instructions cannot be executed by the same hardware thread during the same cycle.

**Tip**

To maximise the usage of the execution units, it is necessary to run more than one hardware thread per core. Executing several hardware threads per core will give performance gains most of the time. This is especially true if a thread is blocked while waiting for data from the memory whilst another one has enough data to continue to work. On most applications, running two hardware threads per core seems optimal (to test on your own application).

The clock speed of a compute core is relatively low (1.6 GHz) and also, therefore, its individual performance. This greatly reduces electrical consumption by each core while simultaneously multiplying the number of cores which can be used at the same time for a fixed quantity of energy (electric consumption is by order of frequency cube). The machine performance, therefore, comes from a large number of compute cores with low clock speed.

2.3. Memory architecture

The Blue Gene/Q is composed of a series of compute nodes. Inside each compute node, the memory is shared in a symmetric way (true SMP nodes). Between nodes, there is no sharing of the memory (distributed memory system).

Each compute node has 16 GiB of DDR3-1333 main memory. It is off-chip and is divided into two 8 GiB banks that can be accessed in parallel. It is shared between the 16 computing cores and the core reserved for the system. The maximum throughput is 40.7 GiB/s (2x20.3 GiB/s) and the latency is 350 processor cycles.

Each core has 32 256-bit registers for the QPU (4-wide double-precision floating point unit) that can contain 4 double-precision floats each.

There is a private first level (L1) cache for each core. The L1 cache is 16 KiB for data and 16 KiB for instructions. It is 8-way set-associative for data and 4-way set-associative for instructions with a cache-line size of 64 bytes. The throughput between a core and its L1 cache goes up to 48.8 GiB/s and the latency is 6 cycles.
Between the L1 and L2 caches, there is a prefetching unit (L1P) for each core. It is composed of 32 lines of 128 bytes. It is fully associative and works with cache lines of 128 bytes. Its latency is 24 processor cycles and the replacement policies are depth-stealing and round-robin. Its role is both to try to load data before it is needed by its core and to perform write combining. In the latter, several small writes are aggregated to generate just one larger write to the L2 switch. Prefetching is very important for performance because the access latencies to the L2 cache and to the main memory are high (82 cycles for the L2 and 350 for the main memory). If the L1P succeeds in loading data in advance, part or all of the access latencies will be hidden.

Two prefetch algorithms are available. By default, the L1P unit will detect linear streams. It can manage up to 16 streams simultaneously (using at least 2 128-byte lines per stream). A depth increase (use of additional 128-byte lines for the stream) will occur if data is not prefetched fast enough from a stream. In that case, it could use resources from another stream (depth stealing).

It is also possible to record an access pattern to the memory and to replay it later. List prefetch can be very useful for applications doing non-linear memory accesses but with a repetitive pattern. The application must be adapted to use this functionality.

The L2 cache is shared by all the cores and is composed of 16 slices of 2 MiB for a total capacity of 32 MiB. Each slice is interconnected to each core through a crossbar switch. The maximum aggregated bandwidth is 390.6 GiB/s for the loads and 146.5 GiB/s for the stores. The cache lines contain 128 bytes and the latency is 82 processor cycles. The L2 has a 16-way associativity. It is in the L2 that the memory coherence between the cores is managed. The L2 is inclusive of the L1 caches and knows which cores have a copy of its cache lines. It is interconnected to the main memory via 2 memory controllers with each managing 8 slices of the L2 (16 MiB). The L2 also provides support for transactional memory and thread level speculation.

**Tip**

To obtain good performance, try to optimise for the L2 cache (L1 caches are very small) by reusing data without going to the main memory. L2 is much faster than main memory.

The BlueGene/Q memory provides also support for transactional memory (see Section 8.5) and for thread level speculation (see Section 8.6).

### 2.4. Networks

#### 2.4.1. Five-dimensional torus

The five-dimensional torus is the Blue Gene/Q internal network used for all MPI communications and also for all the I/O between the compute nodes and the I/O nodes.

It connects every compute node to 10 neighbours (2 in each dimension). Each link is bidirectional with a maximum throughput of 2 GiB/s per direction and is managed directly by the Blue Gene/Q chip. The total bandwidth per compute node is 40 GiB/s (20 GiB/s in and 20 GiB/s out). The hardware latency is around 600 ns for one hop (direct neighbour) and each new hop adds between 40 and 50 ns.

A true five-dimensional torus is only available to the application if the execution block is a multiple of a midplane (512 compute nodes) and if the application requests it. For smaller blocks, some dimensions are not a torus but a mesh because the extremities are not connected. For example, the network can be a 4D torus for 256 compute nodes (CN), a 3D torus for 128 CNs, and a 2D torus for 64 CNs; the other directions are a mesh.

The message unit has a DMA engine which allows the network operations to directly access the memory of the compute node without interrupting the cores. That way, communications can occur without disturbing the computation. Overlap of communications by computations is possible (by using non-blocking MPI messages, for example).

The network unit has logic which supports some floating point operations (add, min, max), some fixed point operations (signed and unsigned add, min and max) and bitwise AND, OR and XOR. This allows, for example, collective reduction operations in hardware without needing to interrupt the cores to do the maths.
2.4.2. Functional and I/O network

There is an 11th link on each compute node which can be used to connect an I/O node (ION) to a group of compute nodes. It has the same characteristics as the 5D torus links (bidirectional with a bandwidth of 2 GiB/s in each direction). Only one CN, for every n CN (32 for 1 rack of JUQUEEN and 128 for the other 27 racks of JUQUEEN, 64 for 2 racks of FERMI and 128 for the other 8 racks of FERMI), is connected to an ION. The CNs which are not directly connected to the ION must go through the one CN which is linked to it. Likewise, all the I/O must go through this ION and the one CN linked to it to reach the other CNs in the 5D torus. This link is the only connection to the outside of the Blue Gene/Q for the CNs and is therefore used for all the I/O operations (including the loading of the executable file at the beginning of a job).

Each ION is connected to the filesystems and to the external world via a PCIe based 10 Gb Ethernet or InfiniBand card.

2.4.3. Service network

Each node has a JTAG interface (1 Gb Ethernet). It is accessed and used by the service nodes (SN) to control, monitor and debug the system. The SNs also use them to provide run-time non-invasive reliability, availability, and serviceability (RAS) support.

2.5. Operating system

The compute nodes of FERMI and JUQUEEN run a lightweight, proprietary operating system called CNK (Compute Node Kernel). This Linux-like 64-bit OS is executed on the 17th core to prevent interference with the 16 computing cores. It is a minimal kernel and provides only basic services such as process and memory management, process debugging, network access, sockets, dynamic libraries, reliability, availability and serviceability (RAS) support. The CNK also provides most of the Linux system calls. (Caution: fork or system calls are not available; see the IBM Redbook "Blue Gene/Q Application Development" for the complete list of available calls.) CNK does not manage I/O directly but ships all requests in reading or writing to the I/O node.

CNK provides support for MPI, OpenMP and pthreads. Other parallel paradigms are also available (CHARM++, ARMCI, UPC...). Non-portable (IBM proprietary) low-level parallel libraries can also be used (PAMI and SPI).

The I/O nodes provide access to external devices. All I/O requests are routed through them. They run a patched Red Hat Enterprise Linux 6 optimised for the Blue Gene/Q. This supports several parallel filesystems: GPFS (used on FERMI and JUQUEEN), NFS, PVFS and Lustre.

The login nodes (front-end nodes) provide the working environment for the users with a Red Hat Enterprise Linux 6 full-featured operating system.

2.6. I/O subsystem

The I/O subsystems of FERMI and JUQUEEN are systems composed of different hardware layers with connectivity layers between them shown schematically in Figure 4. Within the Blue Gene/Q itself there are a number of I/O nodes. These constitute the I/O subsystem interface to the BlueGene/Q's computational resources. The I/O nodes act as shared file system clients. They do not contain any storage components themselves, nor do they connect to dedicated storage components that could be considered local to any particular I/O node. For all file systems to which users have read and write access, the IBM General Parallel File System (GPFS) technology is used (on these supercomputers).

The GPFS can be implemented in two different ways: The first way is used on FERMI, and also on some of the JUQUEEN file systems via a cluster of file servers that presents Network Storage Devices (NSD) to their clients, performs meta-data operations and is responsible for the integrity of the shared file system. NSDs are the distributed shareable storage units from which GPFS file systems are built. They use logical units (LUN) which are served by back-end storage of storage controllers and disk enclosures. The storage controllers use hardware RAID technology to pack several physical disks into an aggregate LUN which gives enhanced performance as well as better protection against failure than each of the individual physical disks involved could provide on its own.
Figure 4. The I/O subsystem.

The fibre channel connectivity layer, which connects the NSDs with the storage controllers, consists entirely of cabling. It contains no switching hardware; however, the file server network connectivity layer is implemented with switching technology.

On JUQUEEN, another way of implementation is used for some file systems. Instead of relying on NSD servers connected to storage controllers, the disks are directly connected to the GPFS Storage Servers (GSS), thereby removing an intermediate layer. It also allows the use of a declustered RAID approach which improves data integrity as well as reducing rebuild time; as a result, there is reduced impact on file system performance in case of a disk failure.

2.6.1. I/O nodes

The Blue Gene/Q I/O nodes are dedicated to I/O and do not permit running any tasks of the user's parallel application. Each I/O node is connected via two 10 Gigabit Ethernet adapters to a network through which it accesses the shared GPFS (see Figure 4). The compute nodes which run the application processes do not have a direct connection to this network. Nevertheless, a user process can simply use I/O related system calls like chdir(), open(), read(), write() or even socket(). "Under the hood", the CNK forwards all file system and socket related operations to an I/O node using the 5D torus network. This service is provided automatically and transparently by the CNK.

On the I/O nodes, the forwarded operations are handled by the Common I/O Service (CIOS), which subsequently directs them to the appropriate component of the GPFS client software. The typical path travelled by data which is sent to and returned by I/O functions on a compute node, is presented schematically in Figure 5.
Figure 5. I/O forwarding and function shipment in more detail. The black arrows show the typical path travelled by data associated with the function calls issued by an application process running on a compute node.

The assignment of a set of compute nodes to its dedicated I/O node in the tree topology is static. It is determined by the number of I/O nodes plugged into a node board and the position of the node boards with I/O nodes in a midplane.

**Important**

Asynchronous I/O is not supported on Blue Gene/Q and will cause runtime errors if it is used.

On FERMI, the 96 I/O nodes are distributed across the system in the following way:

- 8 racks have 4 I/O nodes per midplane (one I/O node for 128 compute nodes);
- 2 racks have 8 I/O nodes per midplane (one I/O node for 64 compute nodes).

On JUQUEEN, the 248 I/O nodes are distributed across the system in the following way:

- 27 racks have 4 I/O nodes per midplane (one I/O node for 128 compute nodes);
- 1 rack has 16 I/O nodes per midplane (one I/O node for 32 compute nodes).

The following table summarises the maximum and average I/O performance for a rack with 4 I/O nodes per midplane.

**Table 1. Theoretical maximum and average available bandwidths at various levels of aggregation of computational resources.**

<table>
<thead>
<tr>
<th>System Unit</th>
<th>Max. bandwidth</th>
<th>Avg. bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>midplane</td>
<td>8 GiB/s</td>
<td>5 - 8 GiB/s</td>
</tr>
<tr>
<td>1 I/O node and its 128 compute nodes</td>
<td>2 GiB/s</td>
<td>1 - 2 GiB/s</td>
</tr>
<tr>
<td>compute node</td>
<td>2 GiB/s</td>
<td>16 MiB/s</td>
</tr>
<tr>
<td>single core</td>
<td>2 GiB/s</td>
<td>1 MiB/s</td>
</tr>
</tbody>
</table>
2.6.2. GPFS servers

2.6.2.1. FERMI

On FERMI there are two different GPFS clusters:

- One with scratch and home areas
- The other for repository

Accordingly, there are two different storage server clusters on FERMI:

- 24 IBM x3650M4, each equipped with 2 Intel(R) Xeon(R) CPU E5-2650 @ 2.00 GHz (16 cores in total), NSD servers for $WORK (scratch area) and $HOME (home area) file systems.
- 4 IBM x3650M4 NSD servers for a separate GPFS cluster dedicated to a different class of storage ($REPO).

All 28 NSD servers are equipped with identical resources to access storage on one side of the server, and to communicate with their GPFS clients on the other side:

- 1 x 4x FDR10 Infiniband Host channel adapter port links to the storage controller
- 2 x 4x QDR Infiniband links to the functional network of the Blue Gene/Q

The maximum aggregate bandwidth of the cluster of 24 data servers on the Blue Gene/Q functional network is 96 GB/s. The maximum aggregate bandwidth of the server cluster on the side of the InfiniBand storage controllers is 107 GB/s.

The NSDs are attached to 6 storage controllers which provide a net capacity of 2.5 PB.

There are currently 3 $WORK file systems, each one composed of 1 SFA12K-40 connected to 8 NSDs. The maximum I/O bandwidth is 90 GB/s.

There is 1 $REPO file system built on 1 SFA10K-20 connected to 4 NSDs.

2.6.2.2. JUQUEEN

The GPFS clients on JUGENE's I/O nodes are served by the JUST (Juelich Storage) cluster.

The $HOME and $ARCH file systems are provided by JUST4. It consists of 20 IBM x3560 servers, each equipped with 8 cores. Four of these servers are exclusively dedicated to storing GPFS meta-data. The remaining 16 serve as data NSDs.

All servers are equipped with identical resources to access storage on one server side, and to communicate with their GPFS clients on the other side:

- 4 x 8 Gb/s Fibre Channel host adapter ports
- 4 x 10 Gb/s Ethernet links into the functional network of the Blue Gene/Q

The maximum aggregate bandwidth of the cluster of 16 data servers on the Blue Gene/Q functional network is 80 GB/s. The maximum aggregate bandwidth of the server cluster on the side of the fibre channel adapters that connect to the storage boxes is 64 GB/s.

The NSDs are attached to 22 storage controllers that provide a net capacity of 3.4 PB.

The $WORK file system is provided by JUST4-GSS. It is composed of 20 IBM GSS 24 building blocks, each with two 16-core servers. They are each equipped with 6 x 10 Gb/s Ethernet links into the functional network of the Blue Gene/Q. The maximum I/O bandwidth is 200 GB/s. Each GSS 24 has 232 SAS disks of 2 TB and 6 SSDs of 200 GB. The total net capacity is 7.4 PB.
3. System access

3.1. How to get an account

3.1.1. FERMI

You can access FERMI in several ways:

- **PRACE Projects** [http://www.prace-project.eu/HPC-access?lang=en]: Computing time allocation at the European level on the basis of a European research project. You can get more information by accessing the dedicated web site.

- **ISCRA Projects** [http://www.hpc.cineca.it/services/iscra]: Computing time allocation at the national level on the basis of a national research project. You can get more information by accessing the dedicated web page.

- **Agreements**: Several Italian research institutions have special agreements for computing provision. Ask the CINECA Help Desk [mailto:superc@cineca.it] for more information.

- **General users and industrial applications**: Send a request to the CINECA Help Desk [mailto:superc@cineca.it] to obtain an "on-demand" computing provision.

You will be given access credentials (user name and password) on FERMI.

3.1.2. JUQUEEN

Project applications for the supercomputer JUQUEEN may be submitted by any scientist qualified in his or her respective field of research. Computing resources are allocated on the basis of independent referees' reports. Apart from the scientific relevance of the project, an important criterion for the allocation of computing resources is that the project can make appropriate use of the computer by using a large number of processors in parallel in the computations.

Computing time periods are yearly - with the possibility of application twice per year - beginning May 1st and November 1st of each year.

European scientists from outside of Germany should apply for computing time on JUQUEEN via:

- **PRACE Projects** [http://www.prace-project.eu/HPC-access?lang=en]: Computing time allocation at European level on the basis of a research project. More information is available on the dedicated web site.

Aachen and Jülich scientists should apply via the JARA-HPC Partition.

- **Project application for JUQUEEN** [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/ComputingTime/computingTime_node.html].


Contact:

Dr. Walter Nadler [http://www.fz-juelich.de/SharedDocs/Personen/IAS/JSC/EN/staff/nadler_w.html], email [mailto:w.nadler@fz-juelich.de]

3.2. How to contact support

3.2.1. FERMI

The Help Desk is staffed during working days by competent experts with scientific and technical backgrounds.
The name of the consultant "on-duty" at a given time is shown in the "Help desk" image visible in the left-hand column of the main page on HPC Cineca web site [http://www.hpc.cineca.it/content/users].

Please send requests by e-mail to the CINECA help desk [mailto:superc@cineca.it].

The user support staff also manages a mailing list (HPC-news) for posting announcements, scheduled down times, software updates, problems... on the CINECA HPC computing resources. It is advised that HPC users subscribe to this mailing list.

You can subscribe (or unsubscribe) to HPC-news by sending an email from the address you want to use for the subscription. You can consult the mail archives on the archive web site [http://www.hpc.cineca.it/content/stay-tuned].

User guide, FAQ and focus documents are available on the HPC Cineca web site [http://www.hpc.cineca.it/content/documentation].

### 3.2.2. JUQUEEN

#### 3.2.2.1. Support team

The FZJ high-level support team provides help to the users in case of problems on the systems, such as porting of the application, parallelisation and performance issues as well as usage of the HPC system.

Tel. +49 2461 - 61 2828, e-mail [mailto:sc@fz-juelich.de]

#### 3.2.2.2. Dispatch

Our dispatch team is responsible for user administration such as creating user accounts on the HPC systems and supporting user access to the supercomputers. The dispatch is additionally responsible for database-accounts as well as external access to JuNet and WLAN, and is the registration authority for certificates.

Tel. +49 2461 - 61 5642, Fax. +49 2461 - 61 2810, email [mailto:dispatch.jsc@fz-juelich.de]

#### 3.2.2.3. Project advisors

Each of the more than 200 scientific research projects currently using the centre's supercomputers has an advisor assigned from our ParaTeam who, typically, is an experienced scientist in the respective research area. The ParaTeam is an interdisciplinary group of staff members from the divisions of Application Support, High-Performance Computing Systems, Computational Science, Mathematics and Education. Each team member individually participates in providing high-level user support. Users requesting help with implementing, porting or optimising their applications on our supercomputer systems are referred to their personal advisor. These advisors, for the most part, have a background in scientific computing and are backed by specialists in algorithms, parallel programming, programming languages, compilers, libraries, and tools.

The technical review of project proposals is also performed by the ParaTeam. The team is led by the High-Level User Support Group [http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Organisation/ApplicationSupport/as.html].

#### 3.2.2.4. Cross-sectional teams

There are three special task forces which deal with code scalability:

- The cross-sectional team "Mathematical Methods and Algorithms" provides know-how in numerical kernels and assists users in using these algorithms (see Mathematical Methods and Algorithms [http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Organisation/MathematicsandEducation/MathandEducation.html]);

- The cross-sectional team "Performance Analysis" offers several tools to analyse parallel programs and provides support in the usage of these tools (see Performance Analysis [http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Organisation/ApplicatinSupport/CST-PerformanceAnalysis.html]);

- The cross-sectional team "Application Optimisation" offers special support to users on program optimisation (see Application Optimisation [http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Organisation/ApplicationSupport/CST-ApplicationOptimization.html]).
All teams strongly collaborate with the simulation laboratories [http://www.fz-juelich.de/ias/jsc/EN/Expertise/SimLab/simlab.html].

3.3. How to reach the system

IBM Blue Gene/Qs use front-end nodes (also called login nodes) for interactive access and the submission of batch jobs.

3.3.1. FERMI

The host name of FERMI is:

login.fermi.cineca.it

The FERMI machine can be accessed with:

- SSH: The Secure Shell protocol allows data to be exchanged over a secure channel between two computers. SSH is typically used to log into a remote machine and execute commands (remote console), but it can also be used to run programs and transfer files. On Linux systems the SSH client is usually pre-loaded:

  ssh <userid>@login.fermi.cineca.it

  On Windows systems, you have to download and install it. You can get a free client from the OpenSSH website [http://www.openssh.com/windows.html].

  On FERMI, password-free login based on SSH key exchange is possible. For further details on SSH key generation, see the CINECA website [http://www.hpc.cineca.it/documenti/create_public_key].

- SCP (Secure Copy), SFTP (SSH File Transfer Protocol): These are part of the SSH protocol and can be used to transfer data from/to the systems:

  scp <local data> <username>@login.fermi.cineca.it:<destination_path>

  To copy recursively a directory, you can use the -r option.

- FTP (File Transfer Protocol): This can be used occasionally, if scp/sftp do not allow reasonable bandwidth to the specified system. FTP can be enabled on request, if needed, on any system.

- rsync is a utility software and network protocol (file transfer program) for Unix-like systems (with ports to Windows) which synchronises files and directories in different locations; it minimises data transfer by using delta encoding when appropriate. A detailed description is provided in a specific document [http://www.hpc.cineca.it/content/rsync-command#Fermi];

- GridFTP: A protocol which allows very efficient data transfers between different HPC platforms. A detailed description is provided in a specific document [http://www.hpc.cineca.it/content/data-transfer].

3.3.2. JUQUEEN

JUQUEEN has two different login nodes (juqueen1 and juqueen2) which may be addressed as:

juqueen fz-juelich.de
The two front-end nodes have an identical environment, but multiple sessions of the same user may reside on both nodes which must be taken into account when killing processes.

The JUQUEEN machine can be accessed with:

- **SSH**: The Secure Shell protocol allows data to be exchanged over a secure channel between two computers. SSH is typically used to log into a remote machine and execute commands (remote console), but it can also be used to run programs and transfer files. On Linux systems the SSH client is usually pre-loaded:

  ```
  ssh <userid>@juqueen.fz-juelich.de
  ```

  On Windows systems, you have to download and install it. You can get a free client from the OpenSSH website [http://www.openssh.com/windows.html](http://www.openssh.com/windows.html).

  On JUQUEEN, password free login based on SSH key exchange is required. For further details on SSH key generation see the JSC FAQ [http://www.fz-juelich.de/SharedDocs/FAQs/IAS/JSC/EN/Supercomputers/FAQ_generation_ssh_key.html](http://www.fz-juelich.de/SharedDocs/FAQs/IAS/JSC/EN/Supercomputers/FAQ_generation_ssh_key.html).

- **SCP (Secure Copy), SFTP (SSH File Transfer Protocol)**: These are part of the SSH protocol and can be used to transfer data from/to the systems.

  ```
  scp <local data> <userid>@juqueen.fz-juelich.de:<destination_path>
  ```

- **GSISCP** (high performance transfer). If your local SSH also is a High Performance SSH you may transfer your data using:

  ```
  gsiscp -oNoneSwitch=yes -oNoneEnabled=yes [-P <port>] <source> \[
  [<user>@]<host>:<target>
  ```

  For further details about gsiscp, see this webpage [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/UserInfo/DataTransfer.html](http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/UserInfo/DataTransfer.html).

- **BBCP**: Securely and quickly copy data from source to target.

  ```
  bbcp [ options ] <source> [<user>@]<host>:<target>
  ```

  For further details about bbcp, see this webpage [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/UserInfo/DataTransfer.html](http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/UserInfo/DataTransfer.html).

### 4. Production environment

Both FERMI and JUQUEEN production environments consist of scientific applications available from the Software catalogue, a batch scheduler for submitting jobs and the “module” environment and tools for data management.

#### 4.1. Module environment

##### 4.1.1. The software catalogue

FERMI and JUQUEEN offer a variety of third-party applications and community codes which are installed on their HPC systems. Most of the third-party software are installed using the software modules mechanism (see Section 4.1.2, “The module command”).
The available packages and their detailed descriptions can be viewed in the full catalogue, organised by discipline, on the CINECA web site (http://www.hpc.cineca.it/content/resources/) by selecting "software" and "Application Software for Science" for FERMI. They are also on the FZJ web site (http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/UserInfo/ModuleEnvironment.html).

On FERMI, if you do not see an application you are interested in, or if you have questions about software that is currently available, please contact the specialists of CINECA [mailto:superc@cineca.it].

### 4.1.2. The module command

A basic default environment is already set up by the system login configuration files but it does not include the application environment. Public applications described in our Software Catalog (see Section 4.1.1, “The software catalogue”) need to be initialised for the current shell session by means of the module command.

This means you should simply "load" and "unload" modules to control the environment needed by applications. Table 2, “Module commands.” contains the basic options of the module command:

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>module avail</td>
<td>Shows the available modules on the machine.</td>
</tr>
<tr>
<td>module list</td>
<td>Shows the modules currently loaded on the shell session.</td>
</tr>
<tr>
<td>module load &lt;appl&gt;</td>
<td>Loads the module &lt;appl&gt; in the current shell session, preparing the environment for the application.</td>
</tr>
<tr>
<td>module purge</td>
<td>Unloads all the loaded modules.</td>
</tr>
<tr>
<td>module unload &lt;appl&gt;</td>
<td>Removes loaded modules. It is allowed to remove more than one module per command invocation.</td>
</tr>
<tr>
<td>module swap &lt;appl_1&gt; &lt;appl_2&gt;</td>
<td>Unloads appl_1 and loads appl_2.</td>
</tr>
<tr>
<td>module show &lt;appl&gt;</td>
<td>Shows the actual changes that will take place in the environment when the module is loaded.</td>
</tr>
<tr>
<td>module help &lt;appl&gt;</td>
<td>Provides the user with some information about the module and the corresponding software package.</td>
</tr>
</tbody>
</table>

**Caution**

Remember, you also need to load the needed modules in batch scripts before using the related applications.

### 4.2. Batch system

Because the FERMI and JUQUEEN HPC systems are shared by many users, long production jobs should be submitted using a scheduler. This ensures that access to the resources is as fair as possible.

There are two different modes for using an HPC system:

- **interactive**: For data movement, archiving, code development, compilations, basic debugger usage; also for very short test runs and general interactive operations. A task in this class should not exceed 10 minutes CPU-time on FERMI and is free of charge on HPC systems with the current billing policy.

- **batch**: For the production runs. Users must prepare a shell script containing all the operations to be executed in batch mode after the requested resources are available and assigned to the job. The job then starts and executes on the compute nodes of the cluster. You must have a valid project which is active on the system to be able to run batch jobs. On FERMI, remember to put all your data, programs and scripts in the $WORK filesystem (referred to by using the environment variable $WORK) which is the only storage area accessible to execution nodes.
The scheduler (or queuing system) for both JUQUEEN and FERMI systems is LoadLeveler (LL). LoadLeveler is the native batch scheduler for IBM Blue Gene/Q machines. The scheduler is responsible for managing jobs on the machine by allocating blocks for the user on the compute nodes as well as returning job output and error files to the users.

In order to correctly use LL, remember that the Blue Gene/Q compute nodes consist of 16 cores and 16 GiB of RAM. It is important, therefore, to follow these guidelines:

- **Compute nodes**: A compute node is not sharable; that is, a compute node cannot run multiple jobs at the same time. A job can only ask for whole compute nodes (containing 16 cores each). Compute nodes are allocated in groups of fixed size called blocks. You can ask for an arbitrary number of compute nodes, but LL will allocate them for you in fixed blocks which may contain more nodes than you requested. Each job, in fact, must be connected with at least one I/O node. This means that there is a minimum block size of 32 compute nodes (512 cores) on JUQUEEN and of 64 compute nodes (1024 cores) on FERMI. Larger blocks are allocated in multiples of 64 with the bg_size keyword of LL (described below). A batch job is charged for the number of cores (nodes*16) allocated, multiplied by the wall clock time spent.

- **Elapsed Time**: This is the time limit after which the job is killed. It is set by the wall_clock_limit keyword in the LoadLeveler script and is important for determining the specific class used by the scheduler.

- **Memory**: You cannot ask for a particular quantity of memory in a Blue Gene/Q job. You can, however, access all the memory of the nodes in your block (16 GiB per node). If you need more memory, you have to allocate more compute nodes. Jobs with high memory requirements but with poor parallel scalability are not suitable for the Blue Gene/Q architecture. Running one MPI process (rank) per node, there will be approximately 16 GiB available to the application. Running n ranks per node, there will be approximately 16/n GiB available for the application. Valid values for the number of ranks are: 1, 2, 4, 8, 16, 32, 64.

To submit a batch job, an LL script file must be written with directives for the scheduler, followed by the commands to be executed. The script file should then be submitted using the `llsubmit` command.

**For FERMI**

A job submitted for batch execution runs on the compute nodes of the cluster. Due to CINECA policies, the $HOME filesystem is not mounted for batch jobs. Therefore, all data and executable files needed by a batch job must be copied in advance onto the $WORK disks. This is not necessary for "public" executable files, accessible via the `module` command.

**Note**

Please note that the minimum job size for Tier-0 projects is 128 nodes (2048 cores) on FERMI and 256 nodes (4096 cores) on JUQUEEN.

### 4.2.1. Queue classes

On FERMI, the current public classes are defined in Table 3, “Queue classes on FERMI.” Please note that this structure is not final and some changes could occur. In this case, users will be notified via the HPC-news mailing list.

**Table 3. Queue classes on FERMI.**

<table>
<thead>
<tr>
<th>Queue name</th>
<th>Compute Nodes</th>
<th>Max. wall time</th>
<th>Defined on</th>
<th>Notes</th>
<th>Priority</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>64</td>
<td>00:30:00</td>
<td>2 racks with 16 I/O nodes each</td>
<td>-</td>
<td>High</td>
<td>For test/debug jobs, short time and few Compute Nodes.</td>
</tr>
<tr>
<td>longdebug</td>
<td>64</td>
<td>24:00:00</td>
<td>2 racks with 16 I/O nodes each</td>
<td>-</td>
<td>Low</td>
<td>For test/debug jobs, long time and few Compute Nodes.</td>
</tr>
<tr>
<td>Queue name</td>
<td>Compute Nodes</td>
<td>Max. wall time</td>
<td>Defined on</td>
<td>Notes</td>
<td>Priority</td>
<td>Use</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>----------------</td>
<td>------------</td>
<td>-------</td>
<td>----------</td>
<td>-----</td>
</tr>
<tr>
<td>special</td>
<td>128 - 512 (max 1 midplane)</td>
<td>24:00:00</td>
<td>2 racks with 16 I/O nodes each</td>
<td>To be requested with @#class=special</td>
<td>Very low</td>
<td>For production jobs, long time and I/O intensive.</td>
</tr>
<tr>
<td>parallel</td>
<td>128 - 512 (max 1 midplane)</td>
<td>24:00:00</td>
<td>8 racks with 8 I/O nodes each</td>
<td>-</td>
<td>High</td>
<td>For production jobs, long time and high parallelism.</td>
</tr>
<tr>
<td>bigpar</td>
<td>1024 - 2048 (max 2 racks)</td>
<td>24:00:00</td>
<td>8 racks with 8 I/O nodes each</td>
<td>-</td>
<td>Very high</td>
<td>For production jobs, long time and high parallelism.</td>
</tr>
<tr>
<td>keyproject</td>
<td>1024 - 6144 (1 - 6 racks)</td>
<td>24:00:00</td>
<td>8 racks with 8 I/O nodes each</td>
<td>Make request to the Help Desk [<a href="mailto:superc@cineca.it">mailto:superc@cineca.it</a>]</td>
<td>High</td>
<td>For large production runs; normally available only for selected users.</td>
</tr>
<tr>
<td>serial</td>
<td>0</td>
<td>06:00:00</td>
<td>front-end nodes</td>
<td>To be requested with @#class=serial</td>
<td>Low</td>
<td>For jobs of post-processing, compilation, and for moving data between the different filesystems and the tape library.</td>
</tr>
</tbody>
</table>

On JUQUEEN, the public classes are defined in Table 4, “Queue classes on JUQUEEN.”.

Table 4. Queue classes on JUQUEEN.

<table>
<thead>
<tr>
<th>Queue name</th>
<th>Compute Nodes</th>
<th>Max wall time</th>
<th>Priority</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch jobs</td>
<td>257 - 8192</td>
<td>24:00:00</td>
<td>High</td>
<td>For large production jobs, longer time and high parallelism.</td>
</tr>
<tr>
<td>Medium batch jobs</td>
<td>65 - 256</td>
<td>12:00:00</td>
<td>See below</td>
<td>For small size production runs.</td>
</tr>
<tr>
<td>Small batch jobs</td>
<td>32 - 64</td>
<td>00:30:00</td>
<td>See below</td>
<td>For testing jobs, short time and few compute nodes</td>
</tr>
<tr>
<td>serial</td>
<td>0</td>
<td>01:00:00</td>
<td></td>
<td>For jobs of post-processing, compilation, and for moving data between the different filesystems.</td>
</tr>
</tbody>
</table>

Note

On JUQUEEN:

- Jobs with 8,193 up to 28,672 nodes can only be run on demand.
- A job will always be charged for the full block allocated. The only possible block sizes that exist on JUQUEEN are 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 and 28672 compute nodes. For example, if you ask for 192 compute nodes, LoadLeveler will give you 256 compute nodes and you will be charged for these 256 compute nodes.
- One rack among the 28 racks has 32 I/O nodes (and 32 compute nodes for each I/O node). The 27 other racks have 8 I/O nodes (128 compute nodes for each I/O node). A job must use at least one I/O node. Therefore, the minimal number of compute nodes that can be reserved by a job is 32 (512 cores).
- The job class is automatically enforced by LoadLeveler depending on the size of a job. The scheduler prioritises the idle jobs regarding to the class priority, the job submission time, the user's quota, and
the number of jobs a user has already running. The scheduler runs the job with the highest priority if
the necessary resources are available. If the requested resources are not yet available, the scheduler
uses a backfilling strategy to run lower priority jobs with short duration on available resources as long
as doing so will not delay the starting of the highest priority job. The chance for a job to get scheduled
increases the smaller the wall clock limit is.

4.2.2. LoadLeveler commands

Here we provide a short list of the main commands of LoadLeveler:
llsubmit <script>: Submit the job described in the "script" file.
llq: Return information about all the jobs waiting and running in the LL queues.
llq -u $USER: Return information about your jobs only.
llq -l <jobID>: Return detailed information about the specific job (long format).
llq -s <jobID>: Return information about why the job remains queued.
llcancel <jobID>: Cancel a job from the queues; it is either waiting or running.
llstatus: Return information about the status of the machine.

4.2.3. LoadLeveler script syntax

In order to submit a job to the batch scheduler, you need to create a file (for example, job.cmd) which contains:

• LoadLeveler directives specifying how many resources (wallclock time and processors) you wish to allocate
to your job.

• Shell commands and programs you wish to execute. The file paths can be absolute or relative to the directory
from which you submit the job.

4.2.3.1. LoadLeveler directives

Note that if your job tries to use more memory or time than requested, it will be killed by LL. On the contrary,
if you request more resources (compute nodes or elapsed time) than needed, you will pay more (except for the
elapsed time) and/or you will wait longer before your job is taken into consideration. Design your applications in
such a way that they can be easily restarted and request the right amount of resources. The first part of the script
file contains directives for LL specifying the resources needed by the job. Some of them are general keywords
and others are Blue Gene/Q specific.

Useful directives not strictly related to BG/Q architecture are:

    # @ wall_clock_limit = <hh:mm:ss>

which sets the maximum elapsed time (<= 24:00:00) available to the job.

    # @ account_no = <myaccount>

which selects the account to be charged for the job. This keyword is mandatory on FERMI for the personal user
names on the system. On FERMI, the possible values for account_no are given by the saldo -b command.

    # @ input = <filename>

indicates the file used as standard input.

    # @ output = <filename>
indicates the file used as standard output.

    # @ error = <filename>

indicates the file used for standard error.

    # @ job_name = <name of my job>

sets the name of the job.

    # @ comment = <my comment>

adds a comment to the job.

    # @ initialdir = <dir>

selects the initial working directory during job execution (defaults to current working directory at the time the job was submitted).

    # @ notification = <value>

specifies when the user is sent emails about the job (valid options are: start, complete, error, always, never).

    # @ notify_user = <email addr>

specifies the address for email notification.

    # @ environment = <value>

specifies the initial environment variables set by LL when your job step starts (possible options are COPY_ALL: All variables are copied; $var: variable to be copied). Several variables can be specified; they have to be separated by a semi-column (;).

    # @ queue

concludes the LL directives.

The important Blue Gene/Q-specific keywords for a LoadLeveler batch job script are:

    # @ job_type = bluegene
which is necessary for running on FERMI and JUQUEEN compute nodes. To run a serial job on the front-end
nodes, the job_type keyword must be set to serial:

```
# @ job_type = serial
```

Another important keyword is

```
# @ bg_size = <no of compute nodes>
```

which defines block size in terms of compute nodes.

```
# @ bg_shape = <shape>
```

The bg_shape keyword specifies the shape of the block at the midplane level, not at the compute node level. For
example, a bg_shape value 1x2x2x2 means 1 midplane in the A direction, 2 in the B direction, 2 in the C direction
and 2 in D direction, for a total of 8 midplanes = 4096 compute nodes. bg_shape defines the logical dimensions of
your block. Please note that bg_size and bg_shape are mutually exclusive. For an efficient scheduling, LoadLeveler
may allocate a permutation of this shape on the system and will ensure the correct mapping of the MPI tasks.

```
# @ bg_rotate = <value>
```

This directive specifies if LL should consider all possible rotations of the given job shape when searching for
blocks (the possible options are TRUE (value by default) or FALSE).

On FERMI, you can ask for any block size (up to a maximum of 2048, with the exception of the keyproject class).
If it is larger than the minimum block size (64), it will be assigned in multiples of 128 to cover the requested size.
You will be informed at submit time of how many compute nodes will actually be allocated (to be implemented).
The total amount of CPU-hours that will be charged is the number of compute nodes multiplied by 16 (cores/
node) and by the consumed elapsed time.

Optionally, it is possible to specify the connection network used to link the allocated nodes:

```
# @ bg_connectivity = Mesh | Torus | Either
```

This directive can take different values: Torus, Mesh, Either, or a choice of the first four dimensions of the
5D Torus (for example, to have a torus in the 1st and 4th dimensions: Torus Mesh Mesh Torus). The 5th
dimension is not to be specified because it always has a size 2 torus. It is not possible to have a real 5D torus unless
there is an execution block of at least one midplane (512 compute nodes). With this number of compute nodes, it
is possible to request a torus in the 5 dimensions, but the waiting time can be longer. (With less than a midplane,
it is not possible to obtain a torus in all 5 dimensions.) Counting the 5th dimension (which is always included in a
torus), there can be a maximum of 2 dimensions for 64 nodes, 3 for 128 nodes, and 4 for 256 nodes. Mesh is by
default. Activation of the torus can have a significantly positive impact on the communication performances.

In general, users do not need to specify a class (queue) name: LoadLeveler chooses the appropriate class for the
user depending on the resources requested in terms of time and nodes. On FERMI, this is not true for the class
"archive" (for archiving data on cartridges) or the class "keyproject" (for exceptional long runs). These classes
must be selected by declaring them with:

```
# @ class = <class name>
```
4.2.3.2. The runjob command

User applications always need to be executed with the `runjob` command to run on the compute nodes. All the other commands in the LoadLeveler script run on the login nodes. Access to the compute nodes happens only through the `runjob` command.

The `runjob` command takes multiple arguments. Please type `runjob --h` to get a list of possible arguments. The syntax of this command is:

```
runjob [params]
runjob [params] : binary [arg1 arg2 ... argn]
```

The parameters of the `runjob` command can be set with either of the following:

- Environment variables
- Command-line options (higher priority than environment variables).

For a full description of the options, please type:

```
runjob --h
```

The most important parameters are shown in Table 5, “Runjob commands.” . Please note the use of "--" (double minus) to introduce all the options in the command line mode.

Table 5. Runjob commands.

<table>
<thead>
<tr>
<th>Command Line Options</th>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--exe &lt;exec&gt;</td>
<td>RUNJOB_EXE=&lt;exec&gt;</td>
<td>Specifies the full path to the executable (this argument can be also specified as the first argument after &quot;:&quot;).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>runjob --exe /home/user/a.out</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>runjob : /home/user/a.out</code></td>
</tr>
<tr>
<td>--args &lt;prg_args&gt;</td>
<td>RUNJOB_ARGS=&lt;prg_args&gt;</td>
<td>Passes args to the launched application on the compute node.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>runjob : a.out hello world</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>runjob --args hello --args world --exe a.out</code></td>
</tr>
<tr>
<td>--envs “ENVVAR=value”</td>
<td>RUNJOB_ENVS=&quot;ENVVAR=value&quot;</td>
<td>Sets the environment variable ENVVAR=value in the job environment on the compute nodes.</td>
</tr>
<tr>
<td>--exp-env ENVVAR</td>
<td>RUNJOB_EXP_ENV=ENVVAR</td>
<td>Exports the environment variable ENVVAR in the current environment to the job on the compute nodes.</td>
</tr>
<tr>
<td>--ranks-per-node</td>
<td></td>
<td>Specifies the number of processes per compute node. Valid values are: 1, 2, 4, 8, 16, 32, 64. Figure 6 shows the distribution of the cores on a Power A2 chip with 8 processes.</td>
</tr>
<tr>
<td>--np n</td>
<td>RUNJOB_NP=n</td>
<td>Number of processes in the job (&lt;=bg_size*ranks-per-node).</td>
</tr>
<tr>
<td>Command Line Options</td>
<td>Environment Variable</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>--mapping</td>
<td>RUNJOB_MAPPNG=mapfile</td>
<td>Permutation of ABCDET or a path to a mapping file containing coordinates for each rank.</td>
</tr>
<tr>
<td>--start_tool</td>
<td></td>
<td>Path to tool to start with the job (debuggers).</td>
</tr>
<tr>
<td>--tool_args</td>
<td></td>
<td>Arguments for the tool (debuggers).</td>
</tr>
</tbody>
</table>

Figure 6. Distribution of the cores on a Power A2 chip with 8 processes per node (runjob --ranks-per-node 8).

Following, you will find some typical job scripts. You can use these templates to write your own job scripts for running on the FERMI and JUQUEEN systems.

4.2.3.3. Example 1: Pure MPI batch job

```
#!/bin/bash
# @ job_type = bluegene
# @ job_name = bgsize.$(jobid)
# @ output = z.$(jobid).out
# @ error = z.$(jobid).err
# @ shell = /bin/bash
# @ wall_clock_limit = 02:00:00
# @ notification = always
# @ notify_user=<email addr>
# @ bg_size = 256
# @ account_no = <my_account_no> #Only on FERMI
# @ queue
runjob --ranks-per-node 64 --exe ./program.exe
```

This batch job asks for 256 compute nodes, corresponding to 256*16=4096 cores. With the runjob command, 64 ranks per node are required. This means that the parallelism of the program.exe code is 256*64=16,384 MPI processes, each of them with a per-process memory of 16 GiB/64=256 MiB.

4.2.3.4. Example 2: Pure MPI batch job
#!/bin/bash
# @ job_type = bluegene
# @ job_name = example_2
# @ comment = "BG/Q Job by Size"
# @ output = $(job_name).$(jobid).out
# @ error = $(job_name).$(jobid).out
# @ environment = COPY_ALL
# @ wall_clock_limit = 1:00:00
# @ notification = error
# @ notify_user = <email addr>
# @ bg_size = 1024
# @ bg_connectivity = TORUS
# @ account_no = <my_account_no> #Only on FERMI
# @ queue
runjob --ranks-per-node 16 --exe /gpfs/scratch/.../my_program

This job asks for 1024 nodes, 16 processes (ranks) per node, for a total of 16,384 MPI tasks. A TORUS connection is required.

4.2.3.5. Example 3: MPI + OpenMP hybrid job

#!/bin/bash
# @ job_type = bluegene
# @ job_name = example_3
# @ output = $(job_name).$(jobid).out
# @ error = $(job_name).$(jobid).out
# @ environment = COPY_ALL
# @ wall_clock_limit = 12:00:00
# @ bg_size = 1024
# @ bg_connectivity = TORUS
# @ account_no = <my_account_no> #Only on FERMI
# @ queue
module load scalapack
runjob --ranks-per-node 1 --envs OMP_NUM_THREADS=16 \ 
   --exe /path/myexe

This job requires 1024 nodes (a full Blue Gene/Q rack), 1 MPI process per node, and 16 threads per task (via the OMP_NUM_THREADS variable in runjob). When you use a hybrid MPI + OpenMP application, you need to set the environment variable OMP_NUM_THREADS on the compute nodes. This is done through the --envs argument of runjob (remember that the entire batch job script, except for the runjob line, is executed on the login nodes). Please note also that the ScalAPACK library is loaded using the module command.

4.2.4. Using multiple job steps

LoadLeveler scheduler allows stringing many jobs in a single multi-step job. In this case, each step has a name which is specified by the keyword:

    # @ step_name

and is terminated by the # @ queue statement. A given step is executed whenever the condition specified by the keyword:

    # @ dependency= ...
is satisfied. For example, setting:

```bash
# @ step_name = step01
# @ dependency = step00 == 0
```

means that step01 will wait for the step00 to complete and succeed (exit status=0). If you want to start the step01 after the step00 even if there was a failure (useful to retrieve restart files for example), you can write:

```bash
# @ dependency = step00 >= 0
```

It is necessary to specify the dependency directives if you want to insure that the different steps are executed consecutively, one after the other; without this directive, all the steps can begin independently of each other as soon as the resources are available.

Note that for each job step, you need to specify the keyword `# @ bg_size` since it is not inherited by subsequent steps. Otherwise, the default value `bg_size=64` will be assumed. All the other directives (except `# @ step_name`) are inherited from one step to the other if not redefined.

### 4.2.4.1. Exemple 4: Pure MPI multistep job

```bash
#!/bin/bash
# @ job_type = bluegene
# @ job_name = DNS
# @ output = $(job_name).$(step_name).$(jobid).out
# @ error = $(job_name).$(step_name).$(jobid).err
# @ wall_clock_limit = 1:00:00  # WCL for each step
# @ account_no = <my_account_no> #Only on FERMI
#
#************* STEP SECTION ************
# @ step_name=step_0
# @ bg_size = 256
# @ queue
#
# @ step_name=step_1
# @ bg_size= 64
# @ dependency = step_0 == 0
# @ queue

case $LOADL_STEP_NAME in
    step_0)
        set -e
        runjob --ranks-per-node 16 : ./your_exe input1a >output1a
        runjob --ranks-per-node 16 : ./your_exe input1b >output1b
    ;;
    step_1)
        runjob --ranks-per-node 64 : ./your_exe input2 >output2
    ;;
esac
```

This is an example of a multi-step job script. The first step requires 256 nodes with 16 MPI tasks per node. The second step runs only after the first step has successfully completed and requires 64 nodes with 64 MPI tasks for each node.
As further explanation, the submission of this job actually creates two sub-jobs (or steps), each having the same shell script but different directives. In order for each of the sub-jobs to execute different commands, it is necessary to make a jump statement for each step by using the sub-job name (stored in the variable LOADL_STEP_NAME). This is done by using a case. The usage is simple; you just need to follow the example. Don’t forget to add ;; (double semi-colons) to separate each list of commands and esac at the end.

Note the usage in step_0 of the command set -e. This permits interrupting the step while it is running as soon as a command sends back a return code different than zero. In the same way, taking into account the dependency relationship #@ dependency = step_0 == 0, step_1 will not be executed unless all of the previous commands (step_0) have been executed correctly. If the command set -e is not used, the dependency test of the step being run is done on the return code of the last command of the preceding step.

It should also be noted that at the starting up of each step, the directory by default is the one where the job was submitted; it is here that the output files will be written. Furthermore, it is indispensable to specify a different output file for each step. If not, the output of the last step crushes the preceding outputs (see the output line in the submission file).

4.2.5. MPMD jobs

The MPMD (Multiple Program Multiple Data) execution model is supported on Blue Gene/Qs. In this execution model, different executables are started within one block. They communicate with each other using MPI.

This execution model is implemented via mapfiles supplied by the user. A mapfile is a plain ASCII file that contains two sections. The first one specifies the executables to be used for MPMD together with the MPI ranks to be used for this executable. It can contain multiple entries for multiple executables. The second section specifies where (i.e. on which cores) the corresponding ranks should run in the format A B C D E T, where A to E are the physical (torus) coordinates of the cores and T is the coordinate of the core within one node. For further information about mapfiles, see Section 8.2.2.

Let’s take three executables prog1.x, prog2.x, and prog3.x that should be executed on one block. The first executable should run with 4 MPI tasks per node on one node, the second one with 4 tasks per node on another node, and the third one with 4 tasks per node on 2 nodes. So, in total 4 nodes will be used and an example job command file could look as follows (although only 4 nodes are used #@bg_size = 32 is set since this is the smallest block you can allocate on JUQUEEN; for FERMI, the minimum is #@bg_size = 64):

```
# @job_name = MPMD
# @comment = "MPMD-test"
# @environment = COPY_ALL
# @job_type = bluegene
# @bg_size = 32
# @wall_clock_limit = 00:30:00
# @queue

runjob --ranks-per-node 4 --np 16 --mapping mpmd_mapfile : ./prog1.x
```

The user needs to specify where to run which executable in the mapfile mpmd_mapfile. This mapfile has the following structure:

```
#mpmdbegin 0-3
#mpmdcmd prog1.x
#mpmdend

#mpmdbegin 4-7
#mpmdcmd prog2.x
#mpmdend

#mpmdbegin 8-15
```
This means prog1.x is executed with tasks 0 to 3 on the first 4 cores specified in the mapfile, prog2.x is executed with tasks 4 to 7 on the next 4 cores specified, and prog3.x is executed on the last 8 cores specified using tasks 8 to 15.

Important hints:

- It is not possible to share nodes between executables, i.e. on one node, only one executable can run.

- The number of ranks specified in the runjob command (--ranks-per-node) must be the largest number of ranks per node needed. Suppose the first two executables from the example above should again run with 4 tasks per node but the last one should run with 8 tasks per node instead of 4, then --ranks-per-node 8 must be set.

- The memory available to an MPI task depends on the number of ranks per node specified in the runjob command (--ranks-per-node option). If you specify --ranks-per-node m (i.e., using m tasks per node), each MPI task has 16/m GiB of main memory available regardless of the number of MPI tasks which are actually started on a node. For example, if you use --ranks-per-node 4 as in the above example and you start the first executable with only 1 task on the first node, this task will get only 4 GiB of main memory available, although no other task will run on that node.

- The executable specified in the runjob command is a dummy argument, i.e. it is ignored and the executables specified in the mapfile are used. However, the executable argument of the runjob command is mandatory.

4.2.6. Sub-block jobs

On FERMI, it is possible to launch multiple runs in the minimum allocatable block of 64 compute nodes and in the blocks of 128 nodes. The sub-blocking technique enables you to submit jobs with bg_size=64 in which 2, 4, 8, 16, 32 or 64 simulations are simultaneously running, each occupying 32, 16, 8, 4, 2 or 1 compute node(s) respectively, and submit jobs with bg_size=128 in which 2, 4, 8, 16, 32, 64 or 128 simulations are simultaneously running, each occupying 64, 32, 16, 8, 4, 2 or 1 compute node(s) respectively. Most of the environment variables required to set up the sub-blocks are set in specific files that need to be executed in the job script. These files are available after having loaded the subblock module:

```
module load subblock
```

The user just needs to modify the environment variables in the "User Section" of the following job script templates (see below).
4.2.6.1. Exemple 5: Pure MPI sub-block jobs

In this example, the \textit{bg\_size}=64 block is divided into 2 sub-blocks where two different inputs (pep1 and pep2) are simulated by using the same software, Gromacs, and the same options.

```bash
#!/bin/bash
#@ job_name = ggvvia.$(jobid)
#@ output = gmxrun.$(jobid).out
#@ error = gmxrun.$(jobid).err
#@ shell = /bin/bash
#@ job_type = bluegene
#@ wall_clock_limit = 1:00:00
#@ notification = always
#@ notify_user= <valid email address>
#@ account_no = <account no>
#@ bg\_size = 64
#@ queue

#############################################################
### USER SECTION
### Please modify only the following variables
#############################################################
### Dimension of bg\_size, the same set in the LoadLeveler keyword
export N_BGSIZE=64

### No. of sub-blocks you want. For bg\_size 64, you can choose between 2, 4, 8, 16, 32, 64
export N_SUBBLOCK=2

### No. of MPI tasks in each sub-block
export NPROC=512

### No. of MPI tasks in each node
export RANK_PER_NODE=16

### module load <your applications>
module load gromacs/4.5.5
mdrun=$(which mdrun\_bgq)
export WDR=$PWD
inptpr1="$WDR/pep1.tpr"
inptpr2="$WDR/pep2.tpr"
export EXE\_1="$mdrun -s $inptpr1 <options>"
export EXE\_2="$mdrun -s $inptpr2 <options>"
export EXECUTABLES="$EXE\_1 $EXE\_2"

#############################################################

echo "work dir: " $WDR
echo "executable: " $EXECUTABLES

module load subblock
source \
$\{SUBBLOCK\_HOME\}/bgsize\$_{N\_BGSIZE}/npart\$_{N\_SUBBLOCK}.txt
```

In the following example, a \textit{bg\_size}=64 block is divided into 4 sub-blocks where different executable files run (executable\_1, executable\_2, executable\_3, executable\_4). The output files (out\_1, out\_2, out\_3, out\_4) of the four runs will be stored in four different directories, i.e. dir\_1, dir\_2, dir\_3 and dir\_4.

```bash
#!/bin/bash
# @ job_name = sub_block.$(jobid)
# @ output = $(job_name).out
# @ error = $(job_name).err
# @ environment = COPY_ALL
# @ job_type = bluegene
# @ wall_clock_limit = 00:05:00
# @ bg_size = 128
# @ account_no = <account no>
# @ notification = always
# @ notify_user = >valid email address>
# @ queue

### USER SECTION
### Please modify only the following variables

### Dimension of bg\_size, the same set in the LoadLeveler keyword
export N_BGSIZE=128

### No. of required sub-blocks. For N_BGSIZE=128 you can choose between 2, 4, 8, 16, 32, 64, 128.
export N_SUBBLOCK=4

### No. of MPI tasks in each sub-block
export NPROC=512

### No. of MPI tasks in each node
export RANK_PER_NODE=16

### module load <your application>
export WDR=$PWD
export EXE_1=$WDR/executable_1.exe
export EXE_2=$WDR/executable_2.exe
export EXE_3=$WDR/executable_3.exe
export EXE_4=$WDR/executable_4.exe
export EXECUTABLES="$EXE_1,$EXE_2,$EXE_3,$EXE_4"

n_exe () { echo "$EXECUTABLES" | awk -F',' '{print $1}'; }

# work dir: " $WDR
```
echo "executable: " $EXECUTABLES

module load subblock
source \n${SUBBLOCK_HOME}/bgsize_${N_BGSIZE}/npart_${N_SUBBLOCK}.txt

for i in `seq 1 $N_SUBBLOCK`; do
    if [ ! -d $WDR/dir_$i ]; then
        mkdir dir_$i
        cd dir_$i
    else
        cd dir_$i
    fi
    echo $(n_exe $i)
    runjob --block $BLOCK_ALLOCATED --corner $(n_cor $i) \n    --shape $SHAPE_SB --np $NPROC --ranks-per-node \n    $RANK_PER_NODE : $(n_exe $i) > out_$i &
    cd ..
done
wait

4.2.7. Serial jobs

It is also possible to run serial jobs on the front-end nodes using a special class (queue) called serial. Please remember that this class is usable only for post processing activity or to copy your data. An example of the use of this special class is shown below:

#!/bin/bash
# @ job_type = serial
# @ job_name = serial.$(jobid)
# @ output = $(job_name).out
# @ error = $(job_name).err
# @ wall_clock_limit = 0:25:00
# @ class = serial
# @ queue
./executable < input >> output

Note
Remember to use the serial compiler for the front end to generate the executable.

4.2.8. Job monitoring

4.2.8.1. Job monitoring on FERMI

On FERMI, the bgtop application gives some information about the load of the FERMI queues using a graphical tool. It is available in the superc module. Usage:

$ module load superc
$ bgtop

4.2.8.2. Job monitoring on JUQUEEN

On JUQUEEN, the llqx application shows detailed information about all the jobs. It is a FZJ script.
There is also a graphical tool: llview is a client-server based application which allows monitoring of the utilisation of Blue Gene/Q systems. It is developed at the Jülich Supercomputing Centre. It gives a quick and compact summary of different information like the usage of nodes/processors and the resources required for running and waiting jobs. In order to use llview, the display needs to be exported correctly (login with ssh -X). Then the tool is invoked by

```
$ llview &
```

A typical display of llview is shown in Figure 7. Graphical elements of llview are a node display, a usage bar (which gives a direct view of the job granularity, a list of running jobs and a list of waiting jobs) and a graph chart displaying the number of jobs in the different queues.

**Figure 7.** llview client displaying the status of JUQUEEN.

### 4.3. Accounting

#### 4.3.1. Quota schema

##### 4.3.1.1. JUQUEEN

Job accounting is done via a central database at JSC and information about all the jobs is gathered once a day around midnight, based on information obtained from LoadLeveler. Users get information about their current quota status, or job consumption, by using the command:

```
q_cpuquota <options>
```

Usually, user groups are granted a CPU quota on a monthly basis. Jobs run at a normal priority unless 3 monthly quotas (previous, current and next month quotas) have been consumed. At the beginning of each month, the quota of the following month is added. The remaining quota of the earlier months is no longer available, so "saving of CPU time" is not possible.

Project accounts will not, however, show a CPU-quota balance below -101% at the beginning of every month. This results in the possibility for projects which have used the systems during times of low demand and in doing
so have strongly overdrawn their quota, to reach - within a reasonable time - a state in which they can again submit jobs with regular priority. The waived quota will be listed in the monthly overview.

Smaller quotas (e.g. \( \leq 1 \) rack month) are granted as a fixed quota over the whole allowance period. Users will be informed by mail if they run out of CPU quota or if a new quota is assigned.

4.3.1.2. FERMI

At CINECA, job accounting is also done via a central database and all the information about the jobs is gathered once a day around midnight, based on information obtained from LoadLeveler.

The account indicates the grant or resource allocation which can be used for your batch jobs. Usually, a “budget” is associated with an account and shows how many resources (computing hours) can be used by that account. You can list all the accounts attached to your usernames, together with the "budget" and the consumed resources, with the command:

```
saldo -b
```

One single username can use multiple accounts and one single account can be used by multiple usernames (possibly on multiple platforms), all competing for the same budget.

4.3.2. Job accounting

Currently, the following rules are used to define job accounting:

- Users will be charged for wallclock time, i.e. for the time that the nodes are occupied.
- LoadLeveler jobs will be charged for the complete time they are in execution and for the number of nodes they have occupied or reserved, regardless of whether the `runjob` command within the job is actively processing or if the job script is doing other things.
- Currently, but only on JUQUEEN, after exhausting the monthly or fixed quota, jobs may still be submitted with a reduced wallclock time and executed at a lower priority, when no other jobs are waiting. However, they will still be included in the job accounting.

4.3.3. Querying quota information

4.3.3.1. JUQUEEN

To obtain information about the current quota status or the CPU consumption of jobs from the central database, use `q_cpuquota <options>`: shows values of granted and used CPU quota.

Some useful options are reported in Table 6, “Options for CPU quota.”:

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-?</td>
<td>Usage information</td>
</tr>
<tr>
<td>-h juqueen</td>
<td>For JUQUEEN only</td>
</tr>
<tr>
<td>-j &lt;jobstepid&gt;</td>
<td>For a single job.</td>
</tr>
<tr>
<td>-t &lt;time&gt;</td>
<td>For all jobs in the specified time period (e.g. <code>-t 01.09.2013-23.09.2013</code>).</td>
</tr>
<tr>
<td>-d &lt;number&gt;</td>
<td>For last number of days (positive integer).</td>
</tr>
</tbody>
</table>

4.3.3.2. FERMI

On FERMI, to obtain information about your current quota status, you can use the `saldo <options>` command:
saldo is a command that returns, on the standard output, data about resource usage and reference budgets for usernames and accounts. It accepts the options reported in Table 7, “Options for saldo command.”:

Table 7. Options for saldo command.

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Print the help page.</td>
</tr>
<tr>
<td>-b</td>
<td>Print budgets, validity ranges, consumed resources both on the local cluster and on all clusters, percentage for accounts enabled for given usernames.</td>
</tr>
<tr>
<td>-r</td>
<td>Require a resource usage report.</td>
</tr>
<tr>
<td>-s &lt;yyyyymm&gt;</td>
<td>Starting date for the resource usage report (–r option is needed).</td>
</tr>
</tbody>
</table>

4.4. File systems

This section presents the file systems available on FERMI and JUQUEEN and also their performances.

4.4.1. Home, scratch, long time storage

4.4.1.1. FERMI

The organisation of the FERMI file systems is being renovated and, therefore, the architectural solution which is presented in this guide has not yet been fully implemented (as of November 2013).

The storage spaces will have the following properties:

- *temporary* (data are automatically deleted after a given period)
- *permanent* (data are never automatically deleted)

Furthermore, they can be:

- *local* (accessible only from a given system)
- *shared* (shared among all the connected systems)

Finally, they can be:

- *user-specific* (read and write permissions only for the owner)
- *project-specific* (read and write permissions for the project collaborators)

The available data spaces are defined through the following predefined environment variables: `$HOME`, `$WORK`.

4.4.1.1.1. HOME (permanent/backuped, local, user specific data space)

This is a local directory where you are placed after the login procedure. It is usually where system and user applications store their parameter files (hidden in dot-files and in dot-directories such as `.nwchemrc`, `.ssh...`) and where users keep initialisation files specific for the system (.cshrc, .profile,...).

At present, this space is quite large (50 GB) and is designed to also store programs and small personal data. Files are never deleted from this area. Moreover, they are guaranteed by daily backups. If you delete or accidentally overwrite a file, you can ask the CINECA Help Desk [mailto:super.c@cinca.it] to restore it.

The home quota and consumption can be listed by:
4.4.1.1.2. WORK (temporary/not backed up, local, project-specific data space)

This is a local temporary project-specific storage. The retention time is the project duration.

This space is designed for hosting large temporary data files since it is characterised by the high bandwidth of a parallel file system and for submitting jobs via batch. It behaves very well when I/O is performed accessing large blocks of data but it is not well suited for small frequent I/O operations. Parallel systems have a lot of distributed memory: Use it!

$WORK$ has the quota requested by the project. If no quota was explicitly required, it is set to a default of 500 GB.

The $WORK$ quota and consumption can be listed by:

cindata -h

4.4.1.1.3. Long time storage: TAPE

Job output results that have to be kept for some time may very well become too large to be kept on the $HOME$ or $WORK$ file system. Basically, all files that are to be kept but will not be actively used for some time can be moved to the archive. Currently, there are no archive file systems. Whenever a user wants to archive his data, he has to explicitly move his data to the archive server using the cart* commands.

A user can acquire as many “volumes” as he wants (they can be regarded as virtual CDs) and store data on them. When he does not need his archived data anymore, he can delete single files or even whole volumes. The most useful cart commands are:

- cart_new: create a new volume
- cart_dir: show all volumes and files
- cart_put: save a file in a volume
- cart_get: read a file in a volume
- cart_del: delete a file from a volume or a full volume

For more details, refer to the commands short help (-? option).

The cart commands can be run in interactive mode (from the login nodes) or in batch mode if more than 10 minutes are required. In this case, a user must use the “serial” batch queue. This is the only class allowed to access the cartridge system.

In order to access this archive server, the user has to be authorised by the user support staff [mailto:supere@cineca.it].

4.4.1.1.4. Long-time storage: REPO

This is a data repository service implemented through iRODS, the Integrated Rule-Oriented Data System, for the management of long-lasting data.

The aim of this service is to store and maintain scientific data sets for a long time. It allows a user to safely back up data and at the same time manage it through a variety of clients, such as web browsers, graphical desktops and command line interfaces.

Each project hosted at CINECA can request a storage space of REPO type (ask the CINECA Help Desk [mailto:supere@cineca.it]). If the request is approved, the requested space is allocated and made available via the
Data Repository service. Then each member of the project can have access to the service, using the same credentials as his/her account at CINECA.

There are three different access channels:

- iRODS commands, also available for windows
- gridftp clients, such as globus-url-copy [http://www.globus.org/toolkit/docs/5.2/5.2.4/gridftp/user/#gridftp-pUser] or GlobusOnline [https://www.globusonline.org/] [BETA]
- web interface [https://dataweb.hpc.cineca.it/login/login] [BETA]

On FERMI, you can use iRODS commands by typing:

```bash
module load irods
```

### 4.4.1.2. JUQUEEN

All user file systems on JUQUEEN are provided via GPFS from the fileserver JUST [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/OnlineStorage/JUST/JUST_node.html].

The storage locations assigned to each user in the JUQUEEN environment are encapsulated with help of shell environment variables.

Several file systems are available. They are summarised in Table 8, “Available file systems on JUQUEEN.”

<table>
<thead>
<tr>
<th>File System</th>
<th>DESCRIPTION</th>
<th>JUQUEEN ACCESS</th>
</tr>
</thead>
</table>
| $HOME       | Full path to the user's home directory inside GPFS.  
  For source code, binaries, libraries and applications. | Front-end nodes + I/O nodes |
| $WORK       | Full path to the user's scratch directory inside GPFS.  
  Temporary storage location for applications with large size and I/O demands. | Front-end nodes + I/O nodes |
| $DATA       | Full path to limited available user's project data directory inside GPFS.  
  Large projects in collaboration with JSC must apply for needed space explicitly. | Front-end nodes + I/O nodes |
| $ARCH       | Full path to users archive directory inside GPFS.  
  Storage for all files not in use for a long time. | Front-end nodes only |
| /usr/local  | Software repository (usage via $PATH, /usr/local, $LD_LIBRARY_PATH,...). | Front-end nodes + I/O nodes |

All variables will be set during the login process by /etc/profile. It is highly recommended to always access files with the help of these variables.
File system resources are controlled by quota policy for each user and group.

- **$HOME:**
  
The disk space is limited to 6 TB (soft), 7 TB (hard).
  
The number of files is limited to 2 million (soft), 2.2 million (hard).

- **$WORK:**
  
The disk space is limited to 20 TB (soft), 21 TB (hard).
  
The number of files is limited to 4 million (soft), 4.4 million (hard).

- **$ARCH:**
  
  No hard disk space limits exist. If more than 100 TB are requested, please contact the supercomputing support at JSC [mailto:sc@fz-juelich.de] to discuss optimal data processing particularly concerning the end of the project. Furthermore, for some projects there may be special guidelines.
  
The number of files is limited to 2 million (soft), 2.2 million (hard).

### 4.4.1.2.1. File size limit

Although the file size limit is set to unlimited (`ulimit -f`), the maximum file size cannot exceed the GPFS group quota limit for the corresponding file system. The actual limits can be listed by `q_dataquota`.

### 4.4.1.2.2. Recommendations

- **$HOME**
  
  acts as a repository for source codes, binaries, libraries and applications with small size and I/O demands.

- **$WORK**
  
  acts as a temporary storage location. If the application is able to handle large files, `$WORK` is the right file system to place them in. Cleanup Procedure:

  - *Normal files older than 90 days* (modification date and access date) will be purged automatically. For performance reasons, access date is not automatically set by the system but can be set explicitly by the user with `touch -a <filename>`.

  - *Empty directories* (due among other things to deletion of old files) will be deleted after 3 days. This applies also to trees of empty directories which will be deleted recursively from bottom to top in one step.

- **$ARCH**

  For the `$ARCH` file system, it is recommended to use tar files with a maximum size of 1 TB. This is because of the time needed for reading/writing data from/to tape. Firstly, all data in `$ARCH` must be backed up. This will take 10 h for 1 TB. Next, the data will be migrated to tape which will take 3 h per TB. In addition, please keep in mind that a recall of the data will require approximately the same amount of time.

### 4.4.1.2.3. List data quota and data usage

Members of a group/project can display the total limits, quotas and usage by each user of the group in a group special file (`/homex/group/usage.quota`) which is updated every two hours during the daytime shift (see time stamp at the top of the file). Since the end of January 2013, the unit of measure has been set to GB instead of KB for easy reading. This causes the displayed values to always be rounded up to the next GB value. If less then 1 GB is used (e.g. 256 KB or 128 MB), 1 GB will always be displayed.

more $HOME/../../usage.quota
This file can also be listed in a short and long format by the command

```
q_dataquota [-l]
```

The short format will display the group quota limits and group data usage for each file system, followed by the usage of the user herself/himself. The long listing includes the data usage of all users of the group in descending order.

**Note**

- Although no quota limits for a group may be listed for the $WORK file system, quotas are set! Counting quotas will start with the first file created by a user of the group.

- If the message Cannot exceed the user or group quota is displayed when writing data to a file, the sum of used and *in_doubt* blocks has exceeded the hard limit. (Not only the used blocks are taken into account!)

- The grace column reports the status of the quota:
  - *none*: No quota exceeded
  - *xdays*: Remaining grace period to clean up after the soft limit is exceeded
  - *expired*

### 4.4.1.2.4. List group data usage in real time

A prompt update of the group's data usage and limits can be displayed with:

```
mmlsquota -g <group> \n   [ <FS_without_leading_/> | -C just.fz-juelich.de ]
```

The output for the specified file system or all file systems of the JUST storage cluster will show the usage summary of the specified group (not the members) in KByte units by default. For better reading, a unit of measure can be specified or GPFS can select the one that best fits. To do so, specify the option (with GPFS 3.5.x):

```
--block-size {M|G|T|auto}
```

### 4.4.2. Performance of file systems

#### 4.4.2.1. FERMI

##### 4.4.2.1.1. HOME and WORK file systems

Each FERMI node has access to local node (node specific) file systems as well as several shared file systems. The local node file systems are not of practical interest to users and are best considered “read only” for users. The performance, capacity, and other characteristics of the local node file systems are not described by this guide.

The file systems that are of interest to the user are the two read/write data spaces. These two file systems can be addressed via the predefined environment variables:

- *$HOME*:
  - Home directory
Permanent, backed up space

- \$WORK:
  Data area and scratch space

Temporary space

You are strongly encouraged to use these environment variables instead of full paths to refer to data in your scripts and codes.

The characteristics concerning the performance of the HOME and WORK file systems are presented later in Table 9, “Read/Write file systems on FERMI.”

### Table 9. Read/Write file systems on FERMI.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Size</th>
<th>Number of NSD/LUN</th>
<th>GPFS block size</th>
</tr>
</thead>
<tbody>
<tr>
<td>/fermi/home</td>
<td>Contains the home directories of the users. In this file system a “quota” of 50 GB is defined for each user.</td>
<td>85 TiB</td>
<td>4</td>
<td>1 MiB</td>
</tr>
<tr>
<td>/gpfs/work</td>
<td>Large file system optimised for large size blocks of data.</td>
<td>2.5 PiB</td>
<td>180</td>
<td>8 MiB</td>
</tr>
</tbody>
</table>

The GPFS block sizes of the file systems are also included here because they are a very important factor in tuning the I/O of an application. To obtain the best performance from the GPFS file system, the I/O request size should be a multiple of the block size and address block-aligned sections of a file.

#### 4.4.2.1.2. System services

A few small shared GPFS file systems are used exclusively for system services and for providing additional software and thus are “read only” (/cineca, /prod). For details see Table 10, “System services on FERMI.”

### Table 10. System services on FERMI.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Dimension</th>
<th>Number of NSD/LUN</th>
<th>GPFS block size</th>
</tr>
</thead>
<tbody>
<tr>
<td>/fermi/cineca</td>
<td>Contains the files related to system management: configurations, accounting, binary packages for the operating system, control scripts.</td>
<td>1 TB</td>
<td>4</td>
<td>1 MiB</td>
</tr>
<tr>
<td>/fermi/prod</td>
<td>Linked to /cineca/prod. Contains the application software for the end users.</td>
<td>1 TB</td>
<td>4</td>
<td>1 MiB</td>
</tr>
</tbody>
</table>

#### 4.4.2.1.3. Long time storage

The *Tape* file system is mounted on a STK SL8500 system. The STK SL8500 is a new generation robotic magnetic tape library dedicated to medium/long-term storage. Its main feature is the independent movement of the 4 robotic
arms providing a higher availability and shorter response times to mount/unmount requests. It provides direct
user access by means of specialized software (Tivoli/TSM) running on highly available servers for scientific and
enterprise applications and services. The main features of this long term storage are:

- Model: StorageTek StreamLine 8500
- Features: Optical label recognition, Robotic cartridge loader
- Magnetic Media: STK 9940B 200 GB (600 GB compressed)
- Read/Write Units: 6*9940B Fibre Channel
- Connectivity: Switched Fabric SAN, Local FC HUB
- Capacity: 2.7 PB total (max. density media), 4500 cartridge slots
- Performance: Up to 500 mount operation per hour

4.4.2.2. JUQUEEN

The performance of GPFS file systems is summarised in Table 11.

**Table 11. File system performance.**

<table>
<thead>
<tr>
<th>File system</th>
<th>File system bandwidth</th>
<th>GPFS block size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOME</td>
<td>8.5 GB/s</td>
<td>1 MiB</td>
</tr>
<tr>
<td>DATA</td>
<td>200 GB/s</td>
<td>2 MiB</td>
</tr>
<tr>
<td>WORK</td>
<td>200 GB/s</td>
<td>4 MiB</td>
</tr>
<tr>
<td>ARCH</td>
<td>2.8 GB/s</td>
<td>2 MiB</td>
</tr>
</tbody>
</table>

The peak performance for applications on JUQUEEN using the fast scratch file system ($WORK) at the standard
GPFS cluster and the dedicated data file system on the GSS infrastructure can be up to 200 GB/sec.

The GPFS block sizes of the file systems are also included here because they are a very important factor in tuning
the I/O of an application. To get the best performance out of the GPFS file system the I/O request size should be
a multiple of the block size and address block-aligned sections of a file.

4.4.3. Backup policies

4.4.3.1. FERMI

The $HOME filesystem is guaranteed by daily backups.

The backup procedure runs daily and a maximum of three different copies of the same file are preserved. Older
versions are kept for 1 month. The last version of deleted files is kept for 2 months (60 days), then permanently
removed from the backup archive.

For example, the file myfile.txt was modified on January the 1st, 2nd, and 3rd. On January the 4th it is
possible to recover only the last three versions (backups):

- Jan 2nd (i.e. backup of Jan 1st)
- Jan 3rd
- Jan 4th

If following January 4th the file is not modified, then after 30 days, only the version Jan "4th" will be retained.
If on January the 6th, for example, the file is deleted, it will be possible to recover it for the next 60 days: After
that period it will no longer be possible to recover that file.
The standard backup policies have been presented. If needed, a different procedure is possible in special cases. Please contact the HPC support [mailto:superc@cineca.it].

4.4.3.2. JUQUEEN

A daily incremental backup service is in place on the HOME, creating backups of all new and recently modified files. The backup service will skip files that are open and in the process of being modified by other processes while the backup service is running. But a file that has resided unmodified for at least 24 hours on a home file system will have a backup copy made.

4.4.4. Archive your data

4.4.4.1. Using cart commands on FERMI

On the FERMI system, you can archive a large amount of data by using the cart commands. The first time you use this command, you need a special authorisation; please contact the CINECA HPC support [mailto:superc@cineca.it].

You can acquire as many "volumes" as you want (imagine them like virtual CDs) and then store data on them. When you do not need your archived data anymore, you can delete single files or even a whole volume. The cart commands are described in Table 12, "Cart command accepted."

Table 12. Cart command accepted.

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>cart_new <code>&lt;vol_name&gt;</code></td>
<td>Creates a new volume <code>&lt;vol_name&gt;</code>.</td>
</tr>
<tr>
<td>cart_dir</td>
<td>Shows all the defined volumes from the user.</td>
</tr>
<tr>
<td>cart_dir <code>&lt;vol_name&gt;</code></td>
<td>Shows the files stored on the volume <code>&lt;vol_name&gt;</code>.</td>
</tr>
<tr>
<td>cart_put <code>&lt;vol_name&gt;</code> <code>&lt;file&gt;</code></td>
<td>Saves <code>&lt;file&gt;</code> in the volume <code>&lt;vol_name&gt;</code>.</td>
</tr>
<tr>
<td>cart_get <code>&lt;vol_name&gt;</code> <code>&lt;file&gt;</code></td>
<td>Retrieves <code>&lt;file&gt;</code> from the volume <code>&lt;vol_name&gt;</code>.</td>
</tr>
<tr>
<td>cart_del <code>&lt;vol_name&gt;</code> <code>&lt;file&gt;</code></td>
<td>Deletes <code>&lt;file&gt;</code> from the volume <code>&lt;vol_name&gt;</code>.</td>
</tr>
<tr>
<td>cart_del <code>&lt;vol_name&gt;</code></td>
<td>Deletes the volume <code>&lt;vol_name&gt;</code> (the volume is empty).</td>
</tr>
<tr>
<td>cart_acl -u <code>&lt;user&gt;</code> <code>&lt;vol_name&gt;</code></td>
<td>Grants access to a given volume for a specific user.</td>
</tr>
<tr>
<td>cart_dir_access -u <code>&lt;owner&gt;</code> <code>{&lt;vol_name&gt;}</code></td>
<td>Shows the volumes/files of a different owner (cart_acl is needed).</td>
</tr>
<tr>
<td>cart_get_access -u <code>&lt;owner&gt;</code> <code>&lt;vol_name&gt;</code> <code>&lt;file&gt;</code></td>
<td>Retrieves <code>&lt;file&gt;</code> from a volume of a different owner (cart_acl is needed).</td>
</tr>
</tbody>
</table>

All the cart commands accept the `-?` flag for a short help.

Since it is more efficient to store a few large files than many small ones, we suggest using the tar command to aggregate many small files if needed.

```
> tar cvf archive.tar
> cart_put my_volume archive.tar
```

Be careful, it is not possible to create sub-volumes or sub-directories within a volume.

The cart commands can be run in interactive mode (from the login nodes) or in batch mode if more than 10 minutes are required. In the latter case, you must use the archive batch queue which is the only class allowed to access the cartridge system.
ARCH is intended as an archive facility. Job output results that have to be kept for some time (at most for the lifetime of the user's project) may very well become too large to be kept in the HOME file system. Basically, all files that need to be kept but will not be actively used for some time can be moved to the archive. Currently, there are three archive file systems which are identical in terms of capacity and performance. Automated migration policies are in place on all archive file systems. Therefore, files that have resided on an archive file system for a period of time will have been migrated to tape. A daily incremental backup service is also in place: If a file is migrated to tape, an independent backup copy exists and resides on a different tape.

Limits are enforced on a per group basis for the number of I-nodes. The per-group I-node limit is 2 million. There is no automatic enforcement of a data limit on archive file systems. PRACE projects are expected to respect a data limit of 20 TB.

**Important**

Users with sets of many small files are urged to organise such files into sets that are put into a single container file. Then you put only the container file in the archive. This significantly reduces the number of I-nodes within the archive file systems and thus will have a very beneficial effect on the overall performance of many routines that handle meta-data. Furthermore, it takes about 2 minutes per file to retrieve data from tape. Thus, trying to retrieve several thousand separate files (not in a container) from the archive is simply impossible.

Optionally, the container file can also be compressed. The size of the (compressed) container file should not exceed 1 TB because of considerations concerning the efficiency of the tape back-end. The tar command is the most appropriate tool to create such container files. It also has options to list files in a tar archive, extract files, etc. Please consult the online tar(1) manual page for further information.

### 5. Programming environment and basic porting

#### 5.1. Compilation

The Blue Gene/Q programming environment consists of two sets of compilers: The GNU Compiler Collection (GCC) for the Blue Gene architecture and the IBM XL compiler suite. The system supports cross-compilation and the compilers run on the front-end nodes.

**5.1.1. Available compilers**

**Important on FERMI**

On FERMI, to cross-compile with the XL or GNU compilers, load respectively the module:

```
$ module load bgq-xl
$ module load bgq-gnu
```

Different compilers are necessary to generate executable files for both types of nodes: front-end nodes and compute nodes.

For the compute nodes, the IBM XL C/C++ and XL Fortran families of compilers should be the first choice for the compilation because of their specific optimisation possibilities targeted to the Blue Gene/Q architecture. Each of the compilers from the XL family and the corresponding MPI wrappers have a thread-safe version (the name ends in "_r"). They should be used for any multi-threaded applications. The available XL compilers and compiler wrappers are summarised in Table 13.
Table 13. XL family of compilers - compiler invocation for the compute nodes.

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler invocation</th>
<th>Compiler invocation - MPI wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>bgxlc, bgc89, bgc99, bgcc</td>
<td>mpixlc, mpixlc_r</td>
</tr>
<tr>
<td>C++</td>
<td>bgxlc++, bgxlc</td>
<td>mpixlcxx, mpixlcxx_r</td>
</tr>
<tr>
<td>Fortran</td>
<td>bgxlf, bgxlf90, bgxlf95, bgxlf2003, bgf77, bgf90, bgf95, bgf2003</td>
<td>mpixlf77, mpixlf90, mpixlf95, mpixlf2003, mpixlf77_r, mpixlf90_r, mpixlf95_r, mpixlf2003_r</td>
</tr>
</tbody>
</table>

A collection of GNU compilers for C, C++ and Fortran is also available. These compilers do not support some features of the Blue Gene/Q architecture.

Table 14. GNU compiler collection for Blue Gene/Q - compiler invocation for the compute nodes.

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler invocation (in the directory /bgsys/drivers/ppcfloor/gnu-linux/bin/)</th>
<th>Compiler invocation on FERMI - MPI wrapper</th>
<th>Compiler invocation on JUQUEEN - MPI wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>powerpc64-bgq-linux-cc</td>
<td>mpicc</td>
<td>mpiqcc</td>
</tr>
<tr>
<td>C++</td>
<td>powerpc64-bgq-linux-g++</td>
<td>mpicxx</td>
<td>mpiq++</td>
</tr>
<tr>
<td>Fortran</td>
<td>powerpc64-bgq-linux-gfortran</td>
<td>mpif[77</td>
<td>90]</td>
</tr>
</tbody>
</table>

In addition, the corresponding compilers for the front-end nodes are available (XL and GNU compilers). Table 15 gives an overview of the invocation of the XL and GNU compilers for the front-end nodes.

Table 15. Compiler invocation for the front-end nodes. The commands for the XL and GNU family of compilers are shown.

<table>
<thead>
<tr>
<th>Language</th>
<th>XL compiler family</th>
<th>GNU compiler family</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>xlc</td>
<td>gcc</td>
</tr>
<tr>
<td>C++</td>
<td>xlc++, xlC</td>
<td>g++</td>
</tr>
<tr>
<td>Fortran</td>
<td>xlf, xlf90, xlf95, xlf2003</td>
<td>gfortran</td>
</tr>
</tbody>
</table>

5.1.2. Compiler flags

We recommend beginning with -O2 and increasing the level of optimisation stepwise according to Table 16. Always check that the numerical results are still correct when using more aggressive optimisation flags. For OpenMP codes, please add -qsmp=omp -qthreaded.

All the necessary flags to debug an application are presented in the Section 6.1.

All the optimisation flags are described in the Section 8.1.1.

Table 16. Compiler flags for XL compilers in the order of increasing optimisation potential.

<table>
<thead>
<tr>
<th>Optimisation level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O2</td>
<td>Basic optimisation</td>
</tr>
<tr>
<td>-O3 -qstrict</td>
<td>More aggressive optimisations are performed, no impact on accuracy</td>
</tr>
<tr>
<td>-O3 -qhot</td>
<td>Aggressive optimisations which may impact the accuracy (high order transformations)</td>
</tr>
<tr>
<td>Optimisation level</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>-O4</td>
<td>Interprocedural optimisation at compile time</td>
</tr>
<tr>
<td>-O5</td>
<td>Interprocedural optimisation at link time, whole-program analysis</td>
</tr>
</tbody>
</table>

## 5.2. Available libraries

### 5.2.1. IBM ESSL (Engineering and Scientific Subroutine Library)

The vendor optimised numerical library on Blue Gene/Q is ESSL (Engineering and Scientific Subroutine Library) provided by IBM. ESSL is a collection of high performance mathematical subroutines providing a wide range of functions for many common scientific and engineering applications. The mathematical subroutines are divided into nine computational areas:

- Linear algebra subprograms
- Matrix operations
- Linear algebraic equations
- Eigensystem analysis
- Fourier transforms, convolutions, correlations and related computations
- Sorting and searching
- Interpolation
- Numerical quadrature
- Random number generation

The ESSL Blue Gene SMP library provides thread-safe versions of the ESSL subroutines. Some of these subroutines also have a multithreaded version. These multithreaded versions support the shared memory parallel processing programming model. The number of threads employed by the ESSL SMP library is set by the environment variable OMP_NUM_THREADS. (More information in IBM documentation [http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.essl.doc/esslbooks.html])

### 5.2.1.1. Usage of ESSL on JUQUEEN

- Compiling and linking a Fortran program with the sequential version of ESSL on JUQUEEN looks as follows:
  
  
  $ mpixlf90_r call_essl.f90 -L/bgsys/local/lib -lesslbg

- Compiling and linking a Fortran program with the multithreaded version of ESSL on JUQUEEN looks as follows:

  
  $ mpixlf90_r -qsmp call_essl.f90 -L/bgsys/local/lib -lesslsmpbg

  To compile SMP code, you must specify the -qsmp compiler option.

- Compiling and linking a C program with ESSL looks as follows:

  
  $ mpixlc_r -c -I/opt/ibmmath/essl/5.1/include call_essl.c
  $ mpixlf90_r call_essl.o -L/bgsys/local/lib -lesslbg

  In this case Fortran linking must be used.
5.2.1.2. Usage of ESSL on FERMI

The usage of ESSL on FERMI is identical to JUQUEEN but you should load the ESSL module and add the environment variables at the compiling and/or linking phases. For example, compiling and linking a Fortran program with the sequential version of ESSL on FERMI looks as follows:

```
$ module load essl
$ module load bgq-xl
$ mpixlf90_r call_essl.f90 -L$ESSL_LIB -lesslbg
```

Note that the name and the value of ESSL_LIB are obtained by the suboptions show or display of module:

```
$ module show essl
...
setenv ESSL_HOME /opt/ibmmath/essl/5.1/
setenv ESSL_LIB /opt/ibmmath/essl/5.1/lib64
setenv ESSL_INC /opt/ibmmath/essl/5.1/include
setenv ESSL_INCLUDE /opt/ibmmath/essl/5.1/include
prepend-path LIBPATH /opt/ibmmath/essl/5.1/lib64 :
prepend-path LD_LIBRARY_PATH /opt/ibmmath/essl/5.1/lib64 :
```

You will find more detailed instructions on how to link the library with the help of the module:

```
$ module help
    essl ... Example of usage:
```

```
$ module load essl
$ module load bgq-xl
$ mpicc_r -o a.out foo.c -L$ESSL_LIB -lesslbg -lesslsmbpbg
$ mpixlf90_r -o a.out foo.f90 -L$ESSL_LIB -lesslbg -lesslsmbpbg
```

5.2.2. IBM MASS (Mathematical Acceleration SubSytem)

The IBM MASS (Mathematical Acceleration SubSytem) libraries consist of a library of scalar routines (`libmass.a`) and a set of vector routines (`libmassv.a`). The mathematical functions contained in both scalar and vector libraries are automatically called at certain levels of optimisation (replacing the mathematical intrinsic functions). But you can also call them explicitly in your programs (more information in IBM documentation [http://www-01.ibm.com/software/awdtools/mass/bgl/mass-bgl.html]).

The usage of MASS on JUQUEEN or FERMI is similar to ESSL usage on these platforms (see below).

5.2.2.1. Usage of MASS on JUQUEEN

```
$ mpixlf90_r call_mass.f90 -L/opt/ibmcmp/xlmass/bg/7.3/bglib64 \ -lmass -lmassv
```

5.2.2.2. Usage of MASS on FERMI

```
$ module load mass
$ module load bgq-xl
$ mpixlf90_r call_mass.f90 -L$MASS_LIB -lmass -lmassv
```

5.2.3. Other libraries

JSC and CINECA offer a huge variety of third-party applications and community codes that are installed on its HPC systems. Most of the third-party software is installed using the software modules mechanism. The libraries are frequently updated with new software so it would be better to refer to the specific web pages as follows:
• Documentation of software installed on JUQUEEN [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/_node.html]

• Documentation of software installed on FERMI [http://www.hpc.cineca.it/content/application-software-science]

5.3. MPI

5.3.1. Compiling MPI applications

The MPI implementation for JUQUEEN and FERMI is based on the MPICH2 implementation of the Mathematics and Computer Science Division (MCS) at Argonne National Laboratory. It supports the MPI 2.2 standard except for the process creation and management functions.

In order to compile your MPI applications, please use the corresponding MPI compiler wrapper (either for the GNU or the XL suite of compilers) described in Section 5.1.1. For example, to compile your C code with MPI support, you can use (IBM XL compiler):

```
mpixlc -o application.x routine.c
```

For further information about compilers and compiler flags please see Section 5.1.1 and Section 8.1.1 of this guide.

There are six versions of the MPI libraries and of their scripts (see Table 17). By default, the `xl` version is used on JUQUEEN and the `xl.ndebug` version is used on FERMI.

Table 17. Available MPI libraries and scripts.

<table>
<thead>
<tr>
<th>Library/Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>A version of the libraries that was compiled with the GNU Compiler Collection (GCC) and uses fine-grained locking in MPICH. These libraries also have error checking and assertions enabled.</td>
</tr>
<tr>
<td>gcc.legacy</td>
<td>A version of the libraries that was compiled with the GNU Compiler Collection (GCC) and uses coarse-grain lock in MPICH. These libraries also have error checking and assertions enabled and can provide slightly better latency in single-threaded codes, such as those that do not call MPI_Init_thread(...MPI_THREAD_MULTIPLE).</td>
</tr>
<tr>
<td>xl</td>
<td>A version of the libraries with MPICH compiled with the XL compilers and PAMI compiled with the GNU compilers. This version has fine-grained locking and all error checking and asserts enabled. These libraries can provide a performance improvement over the gcc libraries. This is the default version for JUQUEEN.</td>
</tr>
<tr>
<td>xl.legacy</td>
<td>A version of the libraries with MPICH compiled with the XL compilers and PAMI compiled with the GNU compilers. This version has the coarse-grained MPICH lock and all error checking and assertions are enabled. These libraries can provide a performance improvement over the gcc.legacy libraries for single-threaded applications.</td>
</tr>
<tr>
<td>xl.ndebug</td>
<td>A version of the libraries with MPICH compiled with the XL compilers and PAMI compiled with the GNU compilers. This version has fine-grained MPICH locking. Error checking and assertions are disabled. This setting can provide a substantial performance improvement when an application functions satisfactorily. Do not use this library version for initial porting and application development. This is the default version for FERMI.</td>
</tr>
<tr>
<td>xl.legacy.ndebug</td>
<td>A version of the libraries with MPICH compiled with the XL compilers and PAMI compiled with the GNU compilers. This version has the coarse-grained MPICH lock. Error checking and asserts are disabled. This can provide a substantial performance improvement when an application functions satisfactorily. Do not use this library version for initial porting and application development. This library version can provide a performance improvement over the xl.ndebug library version for single-threaded applications.</td>
</tr>
</tbody>
</table>
5.3.2. MPI extensions for Blue Gene/Q

IBM provides extensions to MPICH2 in order to ease the use of the Blue Gene/Q hardware. These extensions start with MPIX instead of MPI. A C/C++ interface is available for all extensions. A Fortran interface is available for a subset. In order to use the extensions, please include:

```c
#include <mpix.h>, for a C/C++ program
```

or

```c
include <mpif.h>, for a Fortran program
```

and compile your program with the usual MPI compiler wrapper (mpixlc_r, mpixlcxx_r, ... for C/C++, or mpixlf90_r, mpixlf95_r, ... for Fortran).

The supported MPIX functions are listed below:

- **MPIX_Cart_comm_create**
  
  **Fortran:** MPIX_CART_COMM_CREATE(INTEGER cart_comm, INTEGER ierr)
  
  **C/C++:** int MPIX_Cart_comm_create(MPI_Comm *cart_comm);

  Creates a Cartesian communicator that mimics the exact hardware on which it is run.

- **MPIX_Comm_rank2global**
  
  **Fortran:** MPIX_COMM_RANK2GLOBAL(INTEGER comm, INTEGER crank, INTEGER grank, INTEGER ierr)
  
  **C/C++:** int MPIX_Comm_rank2global(MPI_Comm comm, int crank, int *grank);

  Determines the rank in MPI_COMM_WORLD of the process associated with the crank rank in the comm communicator.

- **MPIX_Comm_update**
  
  **C/C++:** int MPIX_Comm_update(MPI_Comm comm, int optimise);

  Optimise/deoptimise a communicator by adding/stripping platform specific optimisations (i.e. class route support for efficient broadcast/reductions).

- **MPIX_Dump_stacks**
  
  **C/C++:** void MPIX_Dump_stacks();

  Prints the current system stack, the output is directed to stderr. The first frame (this function) is discarded to make the trace look nicer.

- **MPIX_Get_last_algorithm_name**
  
  **C/C++:** int MPIX_Get_last_algorithm_name(MPI_Comm comm, char *protocol, int length);

  Returns the most recently used collective protocol name. The maximum length of the protocol string is '100'.

- **MPIX_Hardware**
  
  **C/C++:** int MPIX_Hardware(MPIX_Hardware_t *hw);

  Returns information about the hardware the application is running on.
• MPIX_IO_distance

Fortran: MPIX_IO_DISTANCE(INTEGER io_distance)

C/C++: int MPIX_IO_distance();

Returns the distance to the associated I/O node in number of hops.

• MPIX_IO_link_id

Fortran: MPIX_IO_LINK_ID(INTEGER io_link_id)

C/C++: int MPIX_IO_link_id();

Returns the ID of the link to the associated I/O node.

• MPIX_IO_node_id

Fortran: MPIX_IO_NODE_ID(INTEGER io_node_id)

C/C++: int MPIX_IO_node_id();

Returns the ID of the associated I/O node.

• MPIX_Progress_quiesce

C/C++: int MPIX_Progress_quiesce(double timeout);

Wait for network to quiesce.

• MPIX_Pset_diff_comm_create

Fortran: MPIX_PSET_DIFF_COMM_CREATE(INTEGER pset_comm_diff, INTEGER ierr)

C/C++: int MPIX_Pset_diff_comm_create(MPI_Comm *pset_comm_diff);

Returns a communicator which contains only MPI ranks which run on nodes belonging to different I/O Bridge Nodes.

• MPIX_Pset_diff_comm_create_from_parent

Fortran: MPIX_PSET_DIFF_COMM_CREATE_FROM_PARENT(INTEGER parent_comm, INTEGER pset_comm_diff, INTEGER ierr)

C/C++: int MPIX_Pset_diff_comm_create_from_parent(MPI_Comm parent_comm, MPI_Comm *pset_comm_diff);

Returns a communicator which contains only MPI ranks which run on nodes belonging to different I/O Bridge Nodes.

• MPIX_Pset io_node (deprecated)

C/C++: void MPIX_Pset_io_node(int *io_node_route_id, int *distance_to_io_node);

Returns information about the I/O node associated with the local compute node.

• MPIX_Pset_same_comm_create

Fortran: MPIX_PSET_SAME_COMM_CREATE(INTEGER pset_comm_same, INTEGER ierr)

C/C++: int MPIX_Pset_same_comm_create(MPI_Comm *pset_comm_same);
Returns a communicator which contains only MPI ranks which run on nodes belonging to the same I/O Bridge Node.

- MPIX_Pset_same_comm_create_from_parent
  
  **Fortran:** MPIX_PSET_SAME_COMM_CREATE_FROM_PARENT(INTEGER parent_comm, INTEGER pset_comm_same, INTEGER ierr)
  
  **C/C++:** int MPIX_Pset_same_comm_create_from_parent(MPI_Comm parent_comm, MPI_Comm *pset_comm_same);

  Returns a communicator which contains only MPI ranks which run on nodes belonging to the same I/O Bridge Node.

- MPIX_Rank2torus
  
  **C/C++:** int MPIX_Rank2torus(int rank, int *coords);

  Converts an MPI rank into physical coordinates (ABCDE coordinated plus core ID T).

- MPIX_Torus_ndims
  
  **C/C++:** int MPIX_Torus_ndims(int *numdim);

  Determines the number of physical hardware dimensions. This does not include the coordinate T for the core.

- MPIX_Torus2rank
  
  **C/C++:** int MPIX_Torus2rank(int *coords, int *rank);

  Converts a set of coordinates (physical+core/thread) to an MPI rank.

### 5.4. Hybrid MPI/OpenMP

The Blue Gene/Q systems support shared-memory parallelism on single nodes. OpenMP is supported in the IBM extensible language (XL) and the GNU GCC compilers. When using either XL compilers or the GNU compilers, OpenMP can be used with MPI.

The IBM XL compilers provide support for OpenMP v3.1. The GNU compilers provide support for OpenMP v3.0. pthreads are also available on Blue Gene/Q.

Remember that all MPI applications that are multithreaded must initialise the MPI environment with a call to MPI_Init_thread and with a multithreading level of at least MPI_THREAD_FUNNELED. A multithreaded application which initialises MPI with a call to MPI_Init is incorrect and could crash in strange ways. On Blue Gene/Q, MPI_THREAD_MULTIPLE is fully supported.

#### 5.4.1. Compiler flags

For OpenMP applications, it is recommended to use the thread-safe version of the compilers by adding an "_r" to the name, e.g. bgxIC_r or mpixlf95_r, and add the -qsmp=omp -qnosave compiler flags.

#### 5.4.2. Running hybrid MPI/OpenMP applications

The OMP_NUM_THREADS environment variable sets the number of threads that a program will use when it runs. It has to be set in the runjob command. The syntax is as follows:

```bash
runjob --envs OMP_NUM_THREADS=num <other runjob parameters>
```

where `num` is the maximum number of threads that can be used. The number of threads cannot be more than the maximum number of available hardware threads (that is, 64 hardware threads per compute node). It is not
mandatory to choose the number of threads and, in case of omission, the value is set at the maximum number of threads possible.

6. Debugging

If an application aborts unexpectedly, it is useful to monitor the execution of the application in more detail in order to check which branches of the code are actually executed, what the actual values of variables are, which part of the memory is used, etc. One way to do this debugging is to insert print statements in the code to obtain the desired information. However, this is tedious (each time a print or write statement is added, the source needs to be recompiled and rerun). Furthermore, when the code is modified, the runtime conditions change and may influence the behaviour of the application. Therefore, this way of debugging is not recommended.

Instead, it is better to use another approach. First, compilers offer the possibility of checking for certain errors not only during compilation but also during execution. To take advantage of these functionalities, special compiler flags have to be used. They will be described in detail in the next section. It is recommended to begin this way, because this method is quite easy and does not require any additional software.

However, only some errors can be detected by using compiler flags. Therefore, if the use of the compiler flag approach is not effective, the use of a debugger is necessary. A debugger is a powerful tool that can be used to analyse the execution of a running application. Usually, the corresponding application needs to be recompiled using appropriate compiler flags and is then executed under the control of the debugger. This recompilation is often necessary to provide more information to the debugger.

6.1. Compiler flags

6.1.1. Debugging options of the compiler

In this section, useful debugging options for the XL compilers are listed and explained. Simply add them to the compile command you usually use for your application. The information is taken from the manual pages of the XL compilers. For further information about compiler flags, just type man bgxlf or man bgxlc.

6.1.1.1. Language independent option

-00

With this option, all optimisations performed by the compiler are switched off. Sometimes, errors can occur due to overly aggressive compiler optimisations (rounding of floating point numbers, rearrangement of loops and/or operations, etc). If you encounter problems that might be connected to such issues (for example, wrong or inaccurate numeric results), try this option and check whether the problem persists. If it does not, then moderately increase the optimisation level. See Section 5.1.2 for further details.

-qcheck[=<suboptions_list>]

For Fortran, this option is identical to the -C option (see list of flags for Fortran codes below). For C/C++ codes, this option enables different runtime checks depending on the suboptions_list specified (colon-separated list, see below for suboptions) and raises a runtime exception (SIGTRAP signal) if a violation is encountered.

all

Enables all suboptions.

bounds

Performs runtime checking of addresses when subscripting within an object of known size.

divzero

Performs runtime checking of integer division. A trap will occur if an attempt is made to divide by zero.
nullptr
Performs runtime checking of addresses contained in pointer variables used to reference storage.

-qflttrap[=<suboptions_list>]
Generates instructions to detect and trap runtime floating-point exceptions. <suboptions_list> is a colon-separated list of one or more of the following suboptions:

- enable
- imprecise
Only checks for the specified exceptions on subprogram entry and exit.

- inexact
Detects floating-point inexact exceptions.

- invalid
Detects floating-point invalid operation exceptions.

- nanq
Generates code to detect and trap NaNQ (Quiet Not-a-Number) exceptions handled or generated by floating-point operations.

- overflow
Detects floating-point overflow.

- underflow
Detects floating-point underflow.

- qpxstore
Detects and traps on not-a-number (NaN) or infinity values in Quad Processing eXtension (QPX) vectors.

- zerodivide
Detects floating-point division by zero.

-qhalt=<sev>
 Stops the compiler after the first compilation phase if the severity level of errors detected equals or exceeds the specified level <sev>. The severity levels in increasing order of severity are:

- i = informational messages
- l = language-level messages (Fortran only)
- w = warning messages
- e = error messages
- s = severe error messages
- u = unrecoverable error messages (Fortran only)

-qinitauto=[<hex_value>]
Initialises each byte or word of storage for automatic variables to the specified hexadecimal value `<hex_value>`. This generates extra code and should only be used for error determination. If you specify the `-qinitauto` option without a `<hex_value>`, the compiler initialises the value of each byte of automatic storage to zero.

```
-qlanglvl=<suboptions_list>
```

Determines which language standard (or superset, or subset of a standard) to consult for nonconformance. It identifies nonconforming source code and also options that allow such nonconformances.

For Fortran, the following suboptions are recognised: 77std, 90std, 90pure, 90ext, 95std, 95pure, 2003std, 2003pure, 2008std, 2008pure and extended. For C, the suboptions are: classic, extended, saa, saa12, stdc89, stdc99, extc89, extc99. For C++, there are many suboptions. See the compiler manual for their list.

### 6.1.1.2. Fortran options

The following options can be used only with Fortran codes:

```
-C
```

Checks each reference to an array element, array section, or character substring for correctness. This way some array-bound violations can be detected.

```
-qinit=f90ptr
```

Changes the initial association status of pointers from undefined to dissassociated. This option applies to Fortran 90 and above. The default association status of pointers is undefined.

```
-qsigtrap[=<trap_handler>]
```

Sets up the specified trap handler to catch SIGTRAP exceptions when compiling a file that contains a main program. This option enables you to install a handler for SIGTRAP signals without calling the SIGNAL subprogram in the program.

### 6.1.1.3. C/C++ options

The following options apply only to C/C++ codes

```
-qformat=[<options_list>]
```

Warns of possible problems with string input and output format specifications. Functions diagnosed are printf, scanf, strftime, strftime family functions and functions marked with format attributes. `<options_list>` is a comma-separated list of one or more of the following suboptions:

- `all`  
  Turns on all format diagnostic messages.
- `exarg`  
  Warns if excess arguments appear in printf and scanf style function calls.
- `nlt`  
  Warns if a format string is not a string literal, unless the format function takes its format arguments as a va_list.
- `sec`  
  Warns of possible security problems in use of format functions.
- `y2k`
Warns of strftime formats that produce a 2-digit year.

zln

Warns of zero-length formats.

-qinfo=[<suboption>][,<groups_list>]

Produces or suppresses additional informational messages. <groups_list> is a colon-separated list. If a <groups_list> is specified along with a <suboption>, a colon must separate them. The suboptions are:

all

Enables all diagnostic messages for all groups.

private

Lists shared variables that are made private to a parallel loop.

reduction

Lists variables that are recognised as reduction variables inside a parallel loop.

The list of groups that can be specified is extensive. Only a few are given here. For a complete list, please refer to the manual page of the bgxlc compiler.

c99

C code that might behave differently between C89 and C99 language levels.

cls

C++ classes.

cmp

Possible redundancies in unsigned comparisons.

cnd

Possible redundancies or problems in conditional expressions.

gen

General diagnostic messages.

ord

Unspecified order of evaluation.

ppt

Trace of preprocessor actions.

uni

Uninitialised variables.

6.1.2. Compiler flags for using debuggers

In order to run your code under the control of a debugger, you might need to recompile your application including the following compiler flags (XL compilers):
Additionally, the flag `-qkeepparm` may be useful. When specified, it ensures that function parameters are stored on the stack even if the application is optimised. As a result, parameters remain in the expected memory location, providing access to the values of these incoming parameters to debuggers.

### 6.2. Debuggers

#### 6.2.1. STAT on JUQUEEN

**Note**

STAT is only available on JUQUEEN. At the date of the redaction of this document, it had not been installed on FERMI.

**Important**

In order to be able to use the graphical user interface, please make sure you are logged in with

```shell
ssh -X
```

If you are not directly connected to JUQUEEN, make sure you are using the `-X` option for all SSH connections and that your local system (laptop, PC) has a running X server.

STAT (Stack Trace Analysis Tool) is a tool developed by the Lawrence Livermore National Laboratory to quickly show groups of processes in a parallel application which exhibit similar behaviour. This tool scales to millions of processes.

This is very useful, for example, to quickly identify which processes are hanging and or to exhibit deadlock behaviour. After assessing the offending processes, you can easily debug only these particular processes with a full-featured debugger such as TotalView (Section 6.2.2).

STAT works by gathering, merging and showing the stack traces of all processes in a color-coded tree. The nodes of this tree are function names for the given number of processes. The connections between the nodes are the groups of processes that followed that call-path. See an example screenshot in Figure 8.

**Figure 8. STAT example.**

![STAT Example Diagram](http://www.paradyn.org/STAT/STAT.html)

The full documentation of STAT can be found at [http://www.paradyn.org/STAT/STAT.html](http://www.paradyn.org/STAT/STAT.html).
A quick "how to" for running STAT on JUQUEEN

Your program must be compiled with debugging symbols (-g).

Load the STAT module with the command `module load UNITE stat`.

Run the `stat-gui` command. It will open a window asking to attach to a process. As your program is not running yet, nothing will appear.

Submit your application as usual with `llsubmit`. It’s a good idea to have it set to send you an email when execution begins (by setting the `# @ notification = always` line in your submission script). Hint: You can quickly check the status of your job by using the command `llq -u [USERNAME]`.

When your application is running, you go to the GUI and press the Refresh Process list button. It should show the pid of runjob with your submission. It’s important that the STAT GUI is run from the same login node as the application was submitted from (that is, either juqueen1 or juqueen2). This can be verified with the aforementioned `llq -u [USERNAME]` command.

Attach STAT to the runjob process. It will connect to all processes briefly, and show the current stack trace for that execution at that moment (see Figure 9).

Figure 9. STAT example.

In this example, one of the processes is holding up the execution of the whole program caused by a simple `sleep()` call at rank 0 while the other ranks are waiting for it in a barrier.

When STAT captures a snapshot of the execution of a program it also pauses it. By clicking the resume button, execution goes ahead. One can sample the application again in order to get another snapshot of the execution.

6.2.2. TotalView

TotalView is a very powerful debugger supporting C, C++, Fortran 77, Fortran 90/95, PGI HPF and assembler programs and some of the features it offers are as follows:
- Multi-process and multi-threaded
- C++ support (templates, inheritance, inline functions)
- Fortran 90/95 support (user types, pointers, modules)
- 1D + 2D array data visualisation
- Support for parallel debugging (MPI: automatic attach, message queues, OpenMP, pthreads)
- Scripting and batch debugging
- Memory debugging
- Reverse debugging with ReplayEngine (not available on Blue Gene/Q systems)

6.2.2.1. General usage

At start, TotalView will launch three windows: the root window (Figure 10), the startup-parameter window (Figure 11) and the process window (Figure 12).

Figure 10. TotalView root window.

![Figure 10. TotalView root window.](image)

Figure 11. TotalView startup-parameters window.

![Figure 11. TotalView startup-parameters window.](image)
In the startup-parameter window (Figure 11), you have four tabs: Debugging Options, Arguments, Standard I/O and Parallel. If you wish to activate the memory debugging, check the corresponding box in the tab Debugging Options. If you would like to change or add the arguments passed to your application or to runjob, you can do so under Arguments. Please do not change anything in Parallel. Once you have made all the needed changes, click on OK.

Now click on GO in the process window of TotalView (Figure 12). TotalView will proceed executing the runjob command and launch your application. This may take several minutes depending on the size of the block you have requested (which is the number of cores you asked for).

Finally, a dialog box (Figure 13) appears. Click on YES and after a few seconds the source code of the main program of your application appears in the process window and you can start debugging your code.

Since a detailed description of the usage of TotalView is far beyond the scope of this guide, please refer to the TotalView documentation (Rogue Wave Software) [http://www.roguewave.com/support/product-documentation/totalview-family.aspx#totalview] for a user's guide and further information about TotalView.
6.2.2.2. Using TotalView interactively on JUQUEEN

Important

In order to be able to use the graphical user interface, please make sure you are logged in with

```
ssh -X
```

If you are not directly connected to JUQUEEN, make sure you are using the \(-X\) option for all SSH connections and that your local system (laptop, PC) has a running X server.

In order to debug your program with TotalView, load the UNITE and totalview modules first:

```
module load UNITE totalview
```

The most common way to use TotalView (like any other debugger) is interactively with a graphical user interface. In order to do so, start your the TotalView launch script \(lltv\) (after compilation of your application with the appropriate flags, Section 6.1.2). Example:

```
lltv -n <nodes> : -default_parallel_attach_subset=<rank-range> runjob -a -- exe <program> -p <num>
```

This will start the program \(<program>\) with \(<nodes>\) and \(<num>\) processes per node, attaching TotalView to ranks \(<rank-range>\). The subset specification \(<rank-range>\) can be in one of these forms:

- \(rank\): that rank only,
- \(rank1-rank2\): all ranks between rank1 and rank2 inclusive,
- \(rank1-rank2:stride\): every strideth rank between rank1 and rank2.

A rank specification can be either a number or "max" (the last rank in the MPI job).

6.2.2.3. Using TotalView in batch mode on FERMI

In order to launch a TotalView session on FERMI, you need to establish a tunneled connection to its VNC server. It will open a graphical display on your local machine. In order to do that, a specific procedure needs to be followed (explained below).

Preparation

In order to use TotalView properly, one software (two for Windows users) has to be installed on the user’s local workstation:

- The usage of TotalView from a remote host requires a VNC (Virtual Network Computing) connection. VNC is a graphical desktop sharing system to remotely control another computer. While the VNC Server is already installed on FERMI, the user has to download and install VNC Viewer (available on the website http://www.realvnc.com/download/viewer/).

- There are many ways to establish a remote connection from Windows, where ssh isn’t available. Cygwin, a Linux-like environment for Windows (available at http://cygwin.com/setup.exe), is recommended. During installation, be sure to select OpenSSH from the list of available packages (use the internal search engine if you don’t find it in the packages list).

Establishing a VNC connection

1. On FERMI, load the tightvnc module:

   ```
   module load tightvnc
   ```

2. Execute the script vncserver_wrapper

   ```
   vncserver_wrapper
   ```
VNC Server will be opened and a display number will be assigned. Instructions will also appear on the screen containing all the information needed to proceed with the connection establishment.

Instructions example:

New display: 4

On your personal workstation, open a ssh tunneling connection:
MyPC# ssh -L 5904:localhost:5904 -L 5804:localhost:5804 -N amaran10@login01.fermi.cineca.it

(don't wait for the prompt: the command has to continue until you finish your work with VNC).

Windows users need to install Cygwin (Linux-like environment for Windows):

download setup.exe from www.cygwin.com , add Openssh package from the mirror you have selected

On your personal workstation, either,
in another shell, run the vncviewer client:

MyPC# vncviewer localhost:4
(on your personal workstation the vnc client can have a different name)

OR

in your browser (with java plugin installed), look at http://localhost:5804

WARNING: You need to export the DISPLAY variable in the job script to redirect X stream to your login/submit node:

export DISPLAY=fen01:4

When your work is completed, you have to

1) close the vnc client on your pc
2) kill the vnc server on login01.fermi.cineca.it
login01.fermi.cineca.it# vncserver -kill :4
3) close the ssh tunneling connection by issuing a ^C.

Note

If it is the first time you are using this module, you will be asked to choose a VNC password. Remember this password since you will use it every time you open VNC Viewer on your local machine.

3. Open a local terminal on your machine (if Linux) or a Cygwin shell (if Windows). Copy/paste in there the following line:

ssh -L 59<xx>:localhost:59<xx> -L 58<xx>:localhost:58<xx> -N <username>@login<no>.fermi.cineca.it

<xx> is your VNC display number, <username> is your username on FERMI, and <no> is the number of the front-end node you’re logged into (01, 02, 07 or 08). You will find the correct line to copy/paste on your vncserver_wrapper instructions which has all the variables already set.
Note

If asked for a password, input the password you normally use for connecting on FERMI.

4. Open VNC Viewer.

On Linux, open another local shell and type:

```
vncviewer localhost:<xx> (＜xx＞ is your VNC display number, 04 in the example)
```

On Windows, double click on the VNC Viewer icon (created on your desktop by installation) and write `localhost:<xx>` when asked for the server (`＜xx＞` is your VNC display number, 04 in the example).

In both cases, you will be asked for your VNC password.

Set and launch a TotalView batch job

1. Inside your job script, you have to load the appropriate module and export the DISPLAY environment variable:

   ```
   module load profile/advanced totalview
   export DISPLAY=fen<no>:<xx>
   ```

   where `＜xx＞` and `＜no＞` are as above (you’ll find the correct DISPLAY name to export in `vncserver_wrapper` instructions).

2. TotalView execution line will be as follows:

   ```
   totalview runjob -a <runjob arguments: --np, --exe, --args...>
   ```

   after the mandatory `-a` flag, you have to insert every runjob option and argument you need in the same way you're used to. For example:

   ```
   totalview runjob -a --np 4096 --ranks-per-node 32 : ./myexe.x
   ```

3. Submit your job. When it starts running, you will find a TotalView window opened on your VNC Viewer display, where you can start your debugging session. Closing TotalView will also kill the job, and vice versa.

   You can use TotalView with all the functionalities described in its user guide. Just remember, when specifying the parallel settings, to select “Blue Gene” as your parallel system, and to choose the number of tasks and nodes according to what you asked for on your job script.

Caution

Due to license issues, you are NOT allowed to run TotalView sessions with more than 1024 processes simultaneously. An error message will appear if you try to do so. You can always check the status of the licenses with the command:

```
lmstat -c $LM_LICENSE_FILE -a
```

Close the VNC session

When you’re done with your debugging session, please follow these instructions in order to close your VNC session properly:

1. Close VNC Viewer on your local machine;

2. Kill VNC Server on FERMI:

   ```
   vncserver -kill :<x>
   ```
<x> is the usual VNC display number, without the initial 0 (if it was present). Once again, the correct command to input is written on your `vncserver_wrapper` instructions;

3. On your first local shell (Cygwin shell if Windows), close the SSH tunneling connection with CTRL+C.

### 6.2.3. GDB

GDB, the Gnu Project Debugger, is available as a default command line application for debugging both front-end or back-end applications, compiled either with the GNU or IBM XL compilers. With GDB, it is possible to examine the behaviour of your software at runtime, break it on certain conditions and manipulate its parameters to look at the effects. It is also possible to make a post-mortem analysis by reading binary core files. They can help to find where in the code the problem arose.

However, all these functionalities aren't fully exploitable on Blue Gene/Qs. In particular, since there is no support for parallel debugging, you can only debug a single process or thread of your parallel code at a time.

For general instructions about how to use GDB, you can check the following webpage http://sourceware.org/gdb/current/onlinedocs/gdb/ or this quick reference http://www.stanford.edu/class/cs107/other/gdbrefcard.pdf.

**Debugging front-end applications**

For applications compiled for the front-ends (and not for the compute nodes), it's easy to launch GDB:

```
gdb ./myexe
```

Inside GDB, you can look at your program with all the commands and functionalities the debugger offers, from running the application in real-time to reading its source code at the moment of its crash.

**Post-mortem analysis of compute node executions**

Applications compiled to run on the Blue Gene/Q compute nodes (CN) need their specific version of GDB:

```
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bqg-linux-gdb ./myexe
```

Since you aren't running on a compute node, you can't use GDB with all its functionalities. It is not possible to run a simulation inside the debugger (see the next section on how to connect to a running execution).

You can, however, analyse the core files created by your failing code to determine the source line where your application has stopped. These core files are generated by your execution and can be used to debug your application after it ends (post-mortem analysis). In order to generate the core files (that need to be in binary format), you have to add some environment variables to your runjob command line:

```
runjob --envs BG_COREDUMPONEXIT=1 --envs BG_COREDUMPBINARY=* ... : myexe
```

You can specify the process whose core file you want to generate by replacing the asterisk (which means all the processes) with its rank number. If you want the core from more than one process, you can separate their ranks with commas:

```
runjob --envs BG_COREDUMPONEXIT=1 --envs BG_COREDUMPBINARY=1,3 ... : myexe
```

Once you have the binary core file, you can launch GDB to analyse it by specifying the core file as a parameter:

```
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bqg-linux-gdb ./myexe <corefile>
```

The debugger will point at the last instruction the application tried to execute. You can use commands like `print`, `list` or `where` in order to understand what happened to your software.

**Using GDB on running applications**

It is possible to attach GDB to a process running on a compute node by associating to it a tool called `gdbserver`. It can be associated only to one process (rank) at a time. For debugging multiple processes, you have to launch multiple tools (up to 4) at a time.
The complete procedure for attaching GDB to a running process is explained below. On FERMI, there is an easier method provided by a module (see the following section).

1. First of all, submit your job as usual:

   ```
   llsUBmit <jobscript>
   ```

2. Then, get your jobID (fen04.29286.0 in the example):

   ```
   llq -u $USER
   Id Owner Submitted ST PRI Class Running On
   ---------------------- --------- ---------- -- --- ---------- ---------
   fen04.29286.0 amaran10 12/18 12:19 R 50 parallel fen06
   ```

3. Use it for getting the BG Job ID (78473 in the example):

   ```
   llq -l <jobID> | grep "Job Id"
   ```

   A number called “BG Job ID” (different from the usual job ID) will be displayed.

   ```
   llq -l fen04.29286.0 | grep "Job Id"
   BG Job Id: 78473
   ```

4. Start the GDB-server tool:

   ```
   start_tool --tool /bgsys/drivers/ppcfloor/ramdisk/distros/cios/sbin/gdbtool --args "--rank=<rank #> --listen_port=10000" --id <BG Job ID>
   ```

   `<rank #>` is the rank of the process you want to attach to. The listen port value is by default (no need to change it).

5. Get the IP address for your process (10.15.0.81 in the example):

   ```
   dump_proctable --id <BG Job ID> --rank <rank #> --host <sn_host>
   ```

   `<sn_host>` should be replaced by sn01-io on FERMI and by juqueens1c1 on JUQUEEN. An IP address will be displayed, referring to the IO node of the card where your process is running.

   ```
   dump_proctable --id 78473 --rank 0 --host sn01-io
   Rank   I/O node IP address   pid
   0      10.15.0.81            0x00000000
   ```

6. Launch GDB (back-end version):

   ```
   /bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bq-linux-gdb ./myexe
   ```

7. Inside GDB, connect remotely to your job process:

   ```
   (gdb) target remote <IP address>:10000
   ```

   `<IP address>` is the value you got from dump_proctable; 10000 is the default listen port.

   ```
   (gdb) target remote 10.15.0.81:10000
   ```

You are now connected to your running application on the process you asked for. Even this time, GDB functionalities are limited and commands like `run` or `continue` won’t work. You can, however, get information about the current state of your run (useful, for example, if you’re stuck on a deadlock) with commands like `print`, `list` or `where`.
**gdb-setup on FERMI**

You can complete the setup procedure for using GDB on running processes in an easier and more user-friendly way. A module containing a script that automatically executes some of the steps of the above described procedure (gdb-setup) must be loaded. In order to use it, you first need to load the module `gdb`:

```
module load gdb
```

After the module is loaded, you can use the setup script with:

```
gdb-setup -j <job_id> -r <rank #>
```

where `<job_id>` is the jobID of the running job you want to debug and `<rank #>` is the rank (MPI task) of the specific process you want to examine.

```
module load gdb
gdb-setup -j fen03.46368 -r 3
```

The script automatically finds the BG Job ID, launches `gdbtool` and finds the IP address of the I/O node with `dump_proctable`. Then it prints the instructions for the connection from GDB:

```
Setup has finished successfully! You can now start your debugging session following these instructions:
```

Open `gdb` (back-end version), indicating the path and the name of the executable running inside your job (what you want to debug):

```
> be-gdb ./myexe
```

Inside `gdb`, input the following:

```
target remote 10.15.0.52:10019
```

WARNING: quitting `gdb` while remotely debugging will also kill the job!

You should open a different terminal if you want to analyse a different rank (i.e. a different `gdbtool`) for the same job.

You can now launch `be-gdb`, the back-end version of GDB, and remotely connect to the job with the “target remote” line printed by `gdb-setup`. All the functionality limitations on the compute nodes still stand.

### 6.3. Analysing core dump files

**Note**

If you would like to analyse core dump files, please compile your application with the `-g` option.

**6.3.1. Core dump file analysis using addr2line**

`addr2line` is a simple UNIX command that translates program addresses into file names and line numbers. It obtains information from core files in order to find the source file and the line on which your program stopped its execution.

**Warning**

Line numbers can be retrieved only if the corresponding source file as been compiled with the `-g` option.

If an application stops abruptly when there are no particular environment variables specified, a core file is generated in lightweight text file format. It is quite difficult to interpret it but there is a section which can be helpful: the list of hexadecimal addresses written between the tags `+++STACK` and `---STACK` and, in particular, the values under the column `Saved Link Reg`. They represent the function call chain up to the point where your executable file stopped. It is worth noticing that in the case of multithreaded (i.e. OpenMP) applications, multiple function chains can be found in the core file (one for each thread).
If your application did not generate core files, it is possible to force their creation at exit by setting the environment variable \texttt{BG\_COREDUMPONEXIT} to 1. This could be very useful for applications which, even if not crashing, are (for example) deadlocked in an MPI call. Example:

\texttt{runjob --envs BG\_COREDUMPONEXIT=1 --np 1024 ...}

This is the procedure for reading the hexadecimal addresses with \texttt{addr2line}:

1. Copy the lines of the \texttt{Saved Link Reg} column in a new text file.
2. Replace in such lines the first eight 0s with \texttt{0x} and save.
   Example: 0000000001b2678 -> 0x018b2678
3. Use \texttt{addr2line} and give the executable file and this text as input: file:
   \texttt{addr2line -e ./myexe < addresses.txt}

A list of functions will be displayed. It represents the history of the last nested calls your job executed until its crash. Each line provides the full path of the source code file where the function resides and the corresponding line number where it called the subsequent function (or crashed, if it is the last one on the list).

**Tip**

It is possible to write a script to automatise this procedure. An example is given in the "Blue Gene/Q Application Development" redbook.

You can also ask \texttt{addr2line} to tell you the path for a single address. For example:

\texttt{addr2line -e ./myexe 0x018b2678}

### 6.3.2. TotalView

If an application aborts due to an error, the current status of the memory usage of the application can be written to disk (core dump files) before the execution stops. Due to the fact that writing core files from thousands of processes takes (too) much time and uses (too) much disk space, the generation of complete core files is deactivated. Instead, lightweight text file cores are generated by failing processes.

TotalView doesn't know how to work with lightweight core files. It needs full process memory dumps. The generation of full core dumps can be enabled by exporting the environment variable \texttt{BG\_COREDUMPDISABLED} to * in your job command file:

\texttt{runjob --envs BG\_COREDUMPDISABLED=* <other runjob options> : application.x}

where \texttt{application.x} is your application.

**Important**

Use this option with care, because a core dump file is generated \textit{for each process}. Running with 16,000 MPI tasks means that 16,000 core files will be generated! Before using this option, try to reproduce the error with the least number of tasks possible.

You can specify the process whose core file you want to generate by replacing the asterisk (which means all the processes) with its rank number. If you want the core file from more than one process, you can separate their ranks with commas:

\texttt{runjob --envs BG\_COREDUMPONEXIT=1 --envs BG\_COREDUMPBINARY=1,3 ... : application.x}

Start TotalView as described in Section 6.2.2.1. After the source code of your application appears in the process window, go to the menu \texttt{File} and select \texttt{New Program}. Select \texttt{Open a core file} in the dialog box which appears and choose a core dump file. The Process window displays the core dump file, with the Stack Trace, Stack Frame, and Source Panes showing the state of the process when it dumped the core file. The title bar of the Process
Window names the signal that caused the core dump. The right arrow in the line number area of the Source Pane indicates the value of the program counter (PC) at the moment when the process encountered the error.

7. Performance analysis

7.1. Performance analysis tools

7.1.1. gprof

**gprof** - the GNU Profiler - is the basic tool to analyse program execution profiles. It is a good starting point for profiling your application. **gprof** produces an execution profile determined from the call graph profile file (**gmon.out** by default). The profile file is created by programs that are compiled and/or linked with the `-p` or `-pg` option. It works for C and Fortran programs and both GNU and XL compilers.

To produce an execution profile of your application you need to compile and link it with the `-p` or `-pg` option (-pg is recommended to get the higher level of profiling). After the program execution is finished, the profiling data is collected in **gmon.out.x** files, where *x* corresponds to the MPI rank of the process that produced the data. This is an extended functionality of the GNU toolchain available on Blue Gene/Q systems.

In order to get the most basic timer-tick profiling information at the machine instruction level, use the `-p` option and only on the link command. Without any additional options in the compile command, there will be very little overhead added to the program execution time. Furthermore, no call graph information will be collected.

In order to proceed with a procedure level profiling, you need to include the `-p` option in the compile command for all application source code files. This will also collect call graph information. An additional performance overhead will occur (probably not excessive). Note also that compilation with aggressive optimisation options may alter the structure of the reported call graph due to possible compiler-based code reorganisation.

To enable the full level of profiling, you need to include the `-pg` option in all compile and link commands. When you do this, the call-graph, statement-level, basic block, and machine-instruction profiling data are collected. This level of profiling will introduce the most overhead in an overall program performance.

To analyse the data collected in the **gmon.out.x** files during the program execution, you need to start gprof on a front-end node to obtain a flat profile:

```
gprof program-binary profile-file-list
```

For multi-threaded (i.e. OpenMP) applications, **gprof** does not profile threads by default (only the master thread). If you want to track all the threads of a process, you have to set the `BG_GMON_START_THREAD_TIMERS` environment variable to `all`. In this case, every thread will be analysed including the internal MPI communication threads. If you don't want to track them, set `BG_GMON_START_THREAD_TIMERS` to `nocomm`.

By default, only the 32 first MPI processes (ranks 0 to 31) are profiled. This should be enough for most users. It is possible to select the processes that will be tracked by setting the `BG_GMON_RANK_SUBSET` environment variable. The values it accepts are in the `M` or `M:N` or `M:N:S` formats with `M` the first rank to profile, `N` the last one and `S` a stride.

If you use the `--sum` option in the gprof command line, it will merge all **gmon.out.x** files into one **gmon.sum** file.

In cases where a large number of **gmon.out.x** files are collected, for example, if a large number of processes are profiled, users may find it impossible to process all of these files due to the limit on input arguments in the Linux system. To overcome this limitation, use the `--sum` option on the gprof command line. This will instruct gprof to search the current directory for all **gmon.out.x** files in a continuous sequence.

7.1.1.1. gprof examples

The following example shows the flat profile for a simple parallel program which operates on matrices. The profile shows that the program spends most of its time in the functions called `mat_cmp` and `mat_mul`. There is also a significant part of the runtime consumed by `sin` and `cos` calculations.
Example 1. Flat profile

Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>cumulative self</th>
<th>self calls</th>
<th>s/call</th>
<th>total s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>s/call</td>
<td></td>
</tr>
<tr>
<td>30.60</td>
<td>4655.25</td>
<td>4655.25</td>
<td>48</td>
<td>96.98</td>
<td>mat_cmp</td>
</tr>
<tr>
<td>27.23</td>
<td>8798.60</td>
<td>4143.35</td>
<td>64</td>
<td>64.74</td>
<td>mat_mul</td>
</tr>
<tr>
<td>23.02</td>
<td>12301.23</td>
<td>3502.63</td>
<td></td>
<td></td>
<td>sin</td>
</tr>
<tr>
<td>19.10</td>
<td>15207.63</td>
<td>2906.40</td>
<td></td>
<td></td>
<td>cos</td>
</tr>
<tr>
<td>0.00</td>
<td>15213.24</td>
<td>0.36</td>
<td>16</td>
<td>0.00</td>
<td>mmap</td>
</tr>
<tr>
<td>0.00</td>
<td>15213.76</td>
<td>0.00</td>
<td></td>
<td></td>
<td>main</td>
</tr>
</tbody>
</table>

Each row in a profile corresponds to a function or subroutine in the analysed program. Functions are sorted by decreasing runtime spent. The meaning of the profile fields is as follows:

- **time**: The percentage of the time the program spent in this function or subroutine (exclusive of the functions or subroutines called inside it).
- **cumulative seconds**: Total number of seconds that the program spent executing this function, plus the time spent in all the functions above it.
- **self seconds**: The number of seconds for this function only (without taking into account the functions or subroutines called inside it).
- **calls**: The total number of times the function has been called.
- **s/call**: The average time in seconds spent in this function per call.
- **total s/call**: The average time in seconds spent in this function and its descendants per call.
- **name**: The name of the function.

If the **calls** and **s/call** fields in a table are blank, the corresponding function was never called or the information cannot be determined. In this example, the **sin** and **cos** functions are not profiled because versions from the math library were used and these versions are not compiled with profiling enabled.

Example 2. Line level flat profile

Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>cumulative self</th>
<th>self total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Time 1</td>
<td>Time 2</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>mat_mul</td>
<td>24.51</td>
<td>4018.92</td>
</tr>
<tr>
<td>__sin</td>
<td>11.44</td>
<td>5894.25</td>
</tr>
<tr>
<td>mat_cmp</td>
<td>8.67</td>
<td>7315.83</td>
</tr>
<tr>
<td>mat_cmp</td>
<td>7.93</td>
<td>8616.51</td>
</tr>
<tr>
<td>__cos</td>
<td>7.13</td>
<td>9786.35</td>
</tr>
<tr>
<td>mat_cmp</td>
<td>7.13</td>
<td>10955.63</td>
</tr>
<tr>
<td>mat_mul</td>
<td>6.47</td>
<td>12017.09</td>
</tr>
<tr>
<td>__sin</td>
<td>4.89</td>
<td>12819.35</td>
</tr>
<tr>
<td>__cos</td>
<td>4.13</td>
<td>13496.15</td>
</tr>
<tr>
<td>__cos</td>
<td>2.92</td>
<td>13975.28</td>
</tr>
<tr>
<td>__sin</td>
<td>1.55</td>
<td>14228.67</td>
</tr>
<tr>
<td>__cos</td>
<td>1.49</td>
<td>14472.77</td>
</tr>
<tr>
<td>__cos</td>
<td>1.44</td>
<td>14709.26</td>
</tr>
<tr>
<td>__sin</td>
<td>1.43</td>
<td>14945.63</td>
</tr>
<tr>
<td>__sin</td>
<td>1.43</td>
<td>15179.81</td>
</tr>
<tr>
<td>__sin</td>
<td>1.08</td>
<td>15591.18</td>
</tr>
<tr>
<td>__cos</td>
<td>0.90</td>
<td>15738.35</td>
</tr>
<tr>
<td>__sin</td>
<td>0.82</td>
<td>15872.69</td>
</tr>
<tr>
<td>__sin</td>
<td>0.80</td>
<td>16004.63</td>
</tr>
<tr>
<td>__cos</td>
<td>0.76</td>
<td>16129.12</td>
</tr>
<tr>
<td>__cos</td>
<td>0.76</td>
<td>16253.48</td>
</tr>
<tr>
<td>__cos</td>
<td>0.71</td>
<td>16370.64</td>
</tr>
<tr>
<td>__cos</td>
<td>0.06</td>
<td>16380.59</td>
</tr>
<tr>
<td>__sin</td>
<td>0.03</td>
<td>16385.94</td>
</tr>
<tr>
<td>mat_mul</td>
<td>0.01</td>
<td>16387.07</td>
</tr>
<tr>
<td>mat_mul</td>
<td>0.01</td>
<td>16387.98</td>
</tr>
<tr>
<td>mat_mul</td>
<td>0.01</td>
<td>16388.81</td>
</tr>
<tr>
<td>mmap</td>
<td>0.00</td>
<td>16391.47</td>
</tr>
<tr>
<td>mat_mul</td>
<td>0.00</td>
<td>16396.00</td>
</tr>
<tr>
<td>mat_mul</td>
<td>0.00</td>
<td>16396.00</td>
</tr>
<tr>
<td>main</td>
<td>0.00</td>
<td>16396.00</td>
</tr>
</tbody>
</table>

In order to analyse the run-time profile at source code line-level accuracy, use the `-l` option with the gprof command. For the complete information on line-by-line profiling, add the `-g` and `-qfullpath` options to the compile command of the target source files. The example above shows line-by-line profile output from gprof. The profile consists of the same information as the previous flat profile but with execution time samples shown for actual lines in the source code (`cannon_cmp.c`). Note that the generation of a line-by-line profile for complex programs can take a significant amount of time.

The call graph shows how much time was spent in each function and its child functions. The next example came from the same simple program.

**Example 3. Call graph**
Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.00% of 15213.76 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>57.8</td>
<td>0.00</td>
<td>8798.60</td>
<td>16/16</td>
<td>generic_start_main [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4655.25</td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4143.35</td>
<td>mat_cmp [3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4143.35</td>
<td>mat_mul [4]</td>
</tr>
<tr>
<td>[2]</td>
<td>57.8</td>
<td>0.00</td>
<td>8798.60</td>
<td>16/16</td>
<td>generic_start_main [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4655.25</td>
<td>main [1]</td>
</tr>
<tr>
<td>[3]</td>
<td>30.6</td>
<td>4655.25</td>
<td>0.00</td>
<td>48/48</td>
<td>mat_cmp [3]</td>
</tr>
<tr>
<td>[4]</td>
<td>27.2</td>
<td>4143.35</td>
<td>0.00</td>
<td>64/64</td>
<td>mat_mul [4]</td>
</tr>
<tr>
<td>[5]</td>
<td>23.0</td>
<td>3502.63</td>
<td>0.00</td>
<td></td>
<td>sin [5]</td>
</tr>
<tr>
<td>[6]</td>
<td>19.1</td>
<td>2906.40</td>
<td>0.00</td>
<td></td>
<td>cos [6]</td>
</tr>
<tr>
<td>[7]</td>
<td>0.0</td>
<td>0.36</td>
<td>0.00</td>
<td></td>
<td>mmap [7]</td>
</tr>
</tbody>
</table>

A detailed description of the call graph fields can be found in the gprof output. The gprof call graph shows entries separated into horizontal rows. Each entry row in the call graph consists of several lines. The line corresponding to the index number on the left gives the current function. The lines above this line list the functions that called the current function; the lines below it list the functions that the current function called.

7.1.2. Performance Application Programming Interface (PAPI)

PAPI is a library and associated utilities for portable access to hardware/system performance counters. A core component for CPU/processor counters provides the total numbers of instructions, floating-point operations, branches predicted/taken, cache accesses/misses, TLB misses, cycles, stall cycles, etc. Additional counters provide the total numbers of events and bytes transferred with the 5D-torus network, the quantity of I/O performed and some events related to compute node kernels (CNK). Predefined events are derived from available native counters.

On JUQUEEN, to be able to use PAPI, you need to load the following modules: module load UNITE papi

On FERMI, you need to load the PAPI module: module load papi

The PAPI API can be used for manually configuring and assessing counters. However, it is more commonly used by tools such as Scalasca (Section 7.1.3) or TAU (Section 7.1.4). Since hardware counters are shared resources, they can’t be used simultaneously by performance measurement tools and the subject application itself.

7.1.2.1. Further information

Since a complete description of PAPI is far beyond the scope of this guide, please see the PAPI homepage [http://icl.cs.utk.edu/] for further information.
7.1.3. SCALASCA

Scalasca is a software tool that supports performance optimisation of parallel programs by measuring and analysing the runtime behaviour. The analysis identifies potential performance bottlenecks, in particular those concerning communication and synchronisation, and offers guidance in exploring their causes.

Analysing your code with Scalasca consists of the following steps:

1. Instrumenting the application code
2. Analysing the instrumented code by running it with measurement configured
3. Examining the analysis results
4. Optional: Refining the measurement configuration and rerunning the program

7.1.3.1. Instrumenting your code

First, you need to load the corresponding Scalasca modules.

On FERMI: module load bgq-xl scalasca

On JUQUEEN: module load UNITE scalasca

You can obtain a short introductory help by running the scalasca command without arguments.

After you have loaded the Scalasca module, you need to recompile your application in order to instrument it for using Scalasca. Simply prepend your compiler command with:

scalasca -instrument

For instance, if you are using the mpixlf90_r compiler wrapper to compile your application, use scalasca -instrument mpixlf90_r. Example:

Example 4. Makefile modified for Scalasca instrumentation

... 
PRE = scalasca -instrument 
F90 = $(PRE) mpixlf90_r 
... 
%.o: %.f90
   $(F90) $(F90FLAGS) -c -o $@ $<
$(EXE): $(OBJ)
   $(F90) $(F90FLAGS) $(LDFLAGS) -o $@ $(OBJ)
...

7.1.3.2. Analysing (running) the instrumented code

In order to analyse your code with Scalasca, run the instrumented executable file of your application the same way as you do without Scalasca; just prepend scalasca -analyze to your usual runjob command line. You may also need to quote certain runjob options by enclosing them in quotation marks as shown in the following job command file example:

Example 5. Job command file for a Scalasca experiment on JUQUEEN

#@job_name         = scalasca_profile
#@output           = scalasca_prof.out
Best Practice Guide - Blue Gene/Q v1.1.1

Example 6. Job command file for a Scalasca experiment on FERMI

```bash
@error            = scalasca_prof.err
@job_type         = bluegene
@bg_size          = 64
@wall_clock_limit = 01:00:00
@queue

module load UNITE scalasca
scalasca -analyze runjob --ranks-per-node 8 --envs "OMP_NUM_THREADS=8" \
         --np 512 : application.x args
```

This job file will launch your application on 64 compute nodes with 512 MPI processes, each with 8 OpenMP threads, and perform a Scalasca profile analysis.

You will then obtain a directory named `.epik_application_8p512x8_sum` where the Scalasca results are stored. In addition, information is printed to STDOUT and STDERR. The following are examples:

Example 7. STDOUT example of a Scalasca run

```
[00000] EPIK: Created new measurement archive ./epik_application_8p512x8_sum
[00000] EPIK: Activated ./epik_hydro_8p512x8_sum [NO TRACE] (0.345s)
[00000] EPIK: MPI-2.2 initialized 512 ranks providing thread support level 1 as requested
...     [STDOUT of application.x]
...
[00000] EPIK: Closing experiment ./epik_application_8p512x8_sum
[00000] EPIK: Largest definitions buffer 33021 bytes
[00000] EPIK: 107 unique paths (108 max paths, 8 max frames, 0 unknowns)
[00000] EPIK: Unifying... done (0.387s)
[00000] EPIK: Collating... done (4.422s)
[00000] EPIK: Closed experiment ./epik_application_8p512x8_sum (4.815s) maxHeap(*)=189.022/96459.500MB
```

Example 8. STDERR example of a Scalasca run

```
S=C=A=N: Scalasca 1.4.3 runtime summarization
S=C=A=N: ./epik_application_8p512x8_sum experiment archive
S=C=A=N: Thu May 16 16:29:04 2013: Collect start
/bgsys/drivers/ppcfloor/bin/runjob --ranks-per-node 8 \
        --envs "OMP_NUM_THREADS=8" --np 512 \
```
7.1.3.3. Examine Scalasca analysis results

Example 6 (or Example 5) will produce profile data of the application run of application.x in an `epik_application_8p512x8_sum` directory.

To interactively explore the analysis results with Scalasca's CUBE GUI, use the following command:

```
scalasca -examine -s epik_application_8p512x8_sum
```

**Important**

To use the `cube3` GUI, you must be logged in with

```
ssh -X
```

If you are not directly connected to the Blue Gene/Q, make sure you are using the `-X` option for all SSH connections and that your local system (laptop, PC) has a running X server.

On Juqueen, you can obtain further information on how to use the GUI with:

```
module help cube
```

and refer to the Scalasca documentation [http://www.scalasca.org/download/documentation/documentation.html].

The file `epik_application_8p512x8_sum/epik.conf` contains the values of all Scalasca environment variables that were in effect during the run of the experiment. When necessary, adjusting one or more of these environment variables can be done to configure subsequent measurement runs; for example, adding one or more PAPI hardware counters (Section 7.1.2).

In addition, the file `epik_application_8p512x8_sum/epik.path` contains the call tree of your application execution.

To further analyse the results, execute the following command:

```
scalasca -examine -s epik_application_8p512x8_sum
```

You then get an additional file, `epik_application_8p512x8_sum/epik.score`, which contains a table where all functions of your application are listed along with the type of function (such as an MPI function, a user function, etc.) and the execution times of each function (absolute and relative).

Finally, Scalasca provides you with an estimate of the amount of disk space needed to perform a trace experiment, a time-stamped protocol of all events executed by your application:

```
... Estimated aggregate size of event trace (total_tbc): 8111014272 bytes
Estimated size of largest process trace (max_tbc): 35620782 bytes
...
```

Using this example, when you run a trace experiment, you will need about 8 GiB of disk space (`total_tbc`). Furthermore, the largest trace size for a single process will be around 34 MiB (`max_tbc`) which exceeds the default trace buffer size of 10 MiB. To avoid disruptive flushing of trace buffers to disk during measurement, larger buffers can be configured by setting the environment variable `ELG_BUFFER_SIZE` to at least `max_tbc` and/or refining the measurement by specifying a filter file to reduce measurement overhead.
A measurement filter can be used to exclude some functions or subroutines from the experiment; for example, if they introduce too much overhead and are not particularly relevant for your analysis or they belong to external libraries. To use a measurement filter, simply list the names of all the functions (one per line) in an ASCII file filter.txt. The names of the routines to be used in the filter can be identified from the epik.score file. Add this filter to Scalasca analysis runs by using the option -f filter.txt.

### 7.1.3.4. Refine the measurement configuration

To analyse the wasted waiting time associated with communication and synchronisation inefficiencies, such as late senders and load imbalances, a Scalasca trace experiment can be configured and the instrumented application rerun. To minimise measurement collection and storage overhead, remember to configure a measurement filter and/or adequate trace buffer sizes, best determined from the score report of a prior summary experiment as discussed in Section 7.1.3.3.

Here is an example job command file to perform this trace experiment:

**Example 9. Job command file for a Scalasca trace experiment on JUQUEEN**

```bash
#@job_name         = scalasca_trace
#@output           = scalasca_trace.out
#@error            = scalasca_trace.err
#@job_type         = bluegene
#@bg_size          = 64
#@wall_clock_limit = 01:00:00
#@queue

module load UNITE scalasca
export ELG_BUFFER_SIZE=36000000
scalasca -analyze -t -f filter.txt runjob --ranks-per-node 8 \  
    --envs "OMP_NUM_THREADS=8" --np 512 : application.x args
```

**Example 10. Job command file for a Scalasca trace experiment on FERMI**

```bash
#@job_name         = scalasca_trace
#@output           = scalasca_trace.out
#@error            = scalasca_trace.err
#@job_type         = bluegene
#@bg_size          = 64
#@wall_clock_limit = 01:00:00
#@queue

module load bgq-xl scalasca
export ELG_BUFFER_SIZE=36000000
scalasca -analyze -t -f filter.txt runjob --ranks-per-node 8 \  
    --envs "OMP_NUM_THREADS=8" --np 512 : application.x args
```

The -t option specifies that a trace will be collected and analysed, and then stored in an experiment directory epik_application_128_trace. The results of this experiment can be examined as described in Section 7.1.3.3.

### 7.1.3.5. Scalasca example

Scalasca has been used to analyse the execution performance of the three-dimensional reservoir simulator **PFLOTRAN** on JUGENE (a Blue Gene/P system, ancestor of JUQUEEN) using a test case comprised of 850x1000x160 grid cells with 15 chemical species for 2E9 degrees of freedom; although it scaled well to 65,536 processes, various inefficiencies became increasingly significant. Developed by LANL/ORNL/PNNL, PFLOTRAN combines...
solvers for non-isothermal multi-phase groundwater flow and reactive multi-component contaminant transport. It consists of approximately 80,000 lines of Fortran90 and employs PETSc, LAPACK, BLAS and HDF5 I/O libraries; the application uses MPI directly but also indirectly via these libraries.

The PFLOTRAN application and PETSc library were both instrumented using Scalasca, resulting in over 1,100 routines in an initial summary analysis report. Those routines not found on execution callpaths to MPI operations were filtered during subsequent measurements with the result that measurement dilation, with respect to the uninstrumented application execution, was reduced to an acceptable 5-15%. For executions of 10 simulation timesteps, each process required a trace buffer size (ELG_BUFFER_SIZE) of 32 MiB; the total trace size grew to 2.0 TiB with 65,536 processes, taking an additional 25 minutes for Scalasca trace collection and parallel analysis.

Figure 14 shows the Scalasca analysis report explorer GUI (CUBE) with a trace analysis of a 8192-process execution on JUGENE. Load imbalance on 6-7% of processes along the top/front edge is clearly visible and due to the computational characteristics of that part of the problem geometry. A more significant imbalance was found which arose from the decomposition of grid cells onto processes. Both of these imbalances, in (local) computation time, combine to manifest as waiting time in associated MPI communication and synchronisation which are quantified by Scalasca trace analysis (8% of total execution time as highlighted in Figure 14). Even after addressing these imbalances, the large numbers of MPI_Allreduce operations performed in each simulation timestep are the primary scalability inhibitor which dominates execution time at large scales. MPI File I/O done with HDF5 also constitutes 8% of total execution time at this large scale. Although the initialisation cost can be amortised over longer simulation times, it was also identified as an important area to be reworked by the developers to improve large-scale performance.

Figure 14. Scalasca analysis report explorer GUI (CUBE) showing trace analysis of PFLOTRAN application execution on JUGENE with 8192 MPI processes.

7.1.3.6. Further information

Scalasca analysis reports (cubefiles) can also be examined with the TAU/ParaProf GUI (Section 7.1.4). Since a complete description of the Scalasca tool is far beyond the scope of this guide, please see the Scalasca homepage [http://www.scalasca.org/] for further information.

7.1.4. TAU

The TAU performance system is a toolkit integrating a variety of instrumentation, measurement, analysis and visualisation components, and with extensive bridges to and from other performance tools such as Scalasca and
Vampir. ParaProf profile displays of a Scalasca analysis report are demonstrated in Figure 15. TAU is highly customizable with multiple profiling and tracing capabilities, targets all parallel programming/execution paradigms and is ported to a wide range of computer systems.

Figure 15 shows a color-coded routine execution time profile (upper right), a routine-wise performance breakdown for all processes (upper left), a distribution histogram of MPI_Allreduce time by processes (lower right) and a three-dimensional profile view. Note that, by default, ParaProf shows both routine and callpath profiles simultaneously and one or the other typically needs to be disabled (as done in this example).

**Figure 15.** TAU/ParaProf GUI displays a Scalasca trace analysis report of a PFLOTRAN application execution with 8192 MPI processes.

On FERMI, you need to load the corresponding module before using TAU:

```
module load tau
```

On JUQUEEN, you need to load the following modules:
**Important**

To be able to use the TAU paraprof GUI, you must be logged in with

```
ssh -X
```

If you are not directly connected to the Blue Gene/Q, make sure you are using the `-X` option for all SSH connections and that your local system (laptop, PC) has a running X server.

### 7.1.4.1. Further information

Since a complete description of the TAU tool is far beyond the scope of this guide, please see the [TAU homepage](http://tau.uoregon.edu/) for further information.

### 7.2. Hints for interpreting results

Interpreting performance analysis outputs is not easy. There are a few tips that can be followed in a first approach:

- Examine the routines where most of the time is spent to see if they can be optimised in any way. The gprof (Section 7.1.1), SCALASCA (Section 7.1.3) and TAU (Section 7.1.4) tools can help you for that.

- Look for load imbalance in the code. A large difference in computing time between different parallel tasks is the usual symptom. SCALASCA (Section 7.1.3) and TAU (Section 7.1.4) are the most appropriate tools.

- High values of time spent in the MPI functions and subroutines usually indicate something wrong in the code. The reason could be due to load imbalance, bad communication patterns, or difficulties of scaling to the specified number of tasks. Once again, SCALASCA (Section 7.1.3) and TAU (Section 7.1.4) are perfectly appropriate tools for this.

- High values of cache misses could mean a bad memory access in an array or loop. Examine the arrays and loops in the problematic function to see if they can be reorganised to access the data in a more cache-friendly way. PAPI (Section 7.1.2) will give you these values.

The Tuning chapter in this guide has a lot of information on how to optimise both serial and parallel code. Combining profiling information with the optimisation techniques is the best way to improve the performance of your code.

### 7.2.1. Spotting load-imbalance

By far the largest obstacle to scaling applications out to large numbers of parallel tasks is the presence of *load-imbalance* in the parallel code. Spotting the symptoms of load-imbalance in performance analysis are key to optimising the scaling performance of your code. Typical symptoms include:

For MPI codes  
large amounts of time spent in MPI_BARRIER or MPI collectives (which often include an implicit barrier).

For hybrid MPI+OpenMP codes  
large amounts of time in "_omp_barrier" or "_omp_barrierp".

The location of the accumulation of load-imbalance time often gives an indication of where in your program the load-imbalance is occuring.

### 8. Tuning and optimisation

#### 8.1. Single core optimisation

The single core performance of the application needs to be analysed and optimised in order to achieve the best possible performance on the Blue Gene/Q system. In this section, we present and discuss specific advanced compiler flags. We also show how to ensure that specific optimisations have been applied by the compiler. At the end, we
present a few optimisation strategies through examples. IBM XL compilers will be used as a reference since they support many more architecturally specific features of the Blue Gene/Q compute chip than the GNU compilers.

8.1.1. Advanced and aggressive compiler flags

The IBM XL compiler provides several flags which enable aggressive optimisation. Note that usage of some of these flags may result in relaxed conformance to the IEEE floating-point standard, alter results of computations or even produce errors. Therefore aggressive optimisation flags should be used with caution. Each and every optimisation applied to the code should be investigated through testing (e.g. by checking the results of the execution).

The most important advanced compiler flags and their meanings are listed below:

1. IBM XL compiler optimisation levels: \(-O\)n \((n=0, 2, 3, 4, 5)\)

   The IBM XL compiler supports five optimisation levels. The default optimisation level is \(-O0\). This optimisation level performs only quick local optimisations such as constant folding and elimination of local common subexpressions. This default setting should be used only for debugging or testing purposes.

   The \(-O2\) optimisation level is the best combination for compilation speed and runtime performance. Most developers should choose this option as a starting point for preparing their optimised binaries. This represents a set of low-level transformations that apply to subprogram or compilation unit scope and can include some inlining. The most important optimisations that are applied (if possible) at \(-O2\) level are:

   • Eliminating redundant code
   • Performing basic loop optimisations
   • Structuring the code to take advantage of \(-qarch\) and \(-qtune\) settings

   The set of optimisations in the \(-O2\) portfolio might vary according to the compiler version. In some cases it might be useful to increase the memory available to optimisations at \(-O2\) level by setting an appropriate value with the \(-qmaxmem\) option. For instance, specifying \(-qmaxmem=-1\) allows the optimiser to use memory as needed. The default setting at \(-O2\) optimisation level is \(-qmaxmem=8192\) (8192 kilobytes).

   At the \(-O3\) optimisation level, the compiler performs additional optimisations which could be both memory and compile-time intensive. This optimisation level switches on some of the more advanced and aggressive optimisations which can potentially alter the semantics of the program. Most importantly, at the \(-O3\) level, conformance to IEEE rules is decreased. In many cases, it is recommended to use the \(-qstrict\) along with \(-O3\). It prevents the compiler from applying aggressive optimisations that might alter the semantics of the program. By default, \(-qnostrict\) is set by the \(-O3\) optimisation level. The most important optimisations enabled at this level are:

   • Loop scheduling
   • Inlining
   • Indepth aliasing analysis
   • Analysis of larger program regions

   Additionally, the \(-O3\) optimisation level activates the following compiler options: \(-qhopt=level=0:novector:fastmath,-qfloat=rsqrt:fltint:nonrngchk,-qmaxmem=-1,-qnostrict\).

   The \(-O4\) optimisation level is the same as \(-O3\) except that it adds the following:

   • Sets the \(-qhopt\) option to \(level=1\).
   • Sets the \(-qipa\) option to the default \(level=1\) setting.

   The \(-O5\) optimisation level is the same as \(-O4\) except that it also adds the \(-qipa=level=2\) option.
2. Higher-Order Loop Analysis and Transformations (HOT)

Higher-order loop analysis and transformations are controlled by \(-\text{qhot}\) flag. By default, this option is disabled (i.e. the \(-\text{qnohot}\) compiler option is enabled). There are additional settings which can be specified for \(-\text{qhot}\). The most important ones are listed below.

- \(-\text{qhot} = \text{arraypad}[\text{number}]\): The compiler pads every array in the code by the \text{number} amount. If \text{number} is not specified, not all arrays are necessarily padded and different arrays might be padded by different amounts.

- \(-\text{qhot} = \text{level}=0\): The compiler performs a subset of the high-order transformations such as distribution, loop interchange and loop tiling. This level of \(-\text{qhot}\) optimisation is set by the \(-\text{O3}\) flag and it implies the fastmath and nvector suboptions.

- \(-\text{qhot} = \text{level}=1\): The compiler performs the full set of high-order transformations and it implies the fastmath and vector suboptions.

- \(-\text{qhot} = \text{level}=2\): The compiler performs some other more aggressive loop transformations such as cache reuse or parallelisation (the latter requiring the \(-\text{qsmp}\) option).

All high-order transformation levels described above can be accompanied by fastmath or nofastmath and vector or nvector suboptions. If fastmath is switched on, it enables the replacement of intrinsic math calls with MASS library scalar equivalents. If vector is switched on, it converts loop-array operations into MASS library vector calls wherever possible. When vector or fastmath is used, the \(-\text{qnostrict}\) option will also be set.

If only \(-\text{qhot}\) is specified, the default level and suboptions are \(-\text{qhot} = \text{level}=1:\text{fastmath}:\text{vector}\). \(-\text{O2}\).

3. Interprocedural analysis

The \(-\text{qipa}\) compiler option activates a class of optimisations known as interprocedural analysis (IPA). IPA refers to many different techniques of analysis and optimisation of the whole program. Most of these techniques rely on the call graph or data flow analysis of the program. One of the best known IPA techniques is inlining, a method which enables the inlining of the whole body of a particular function in the place where the function is called. It is extremely useful in the case of small functions that are frequently invoked in the program (e.g. in a loop).

There are three basic levels of IPA on the Blue Gene/Q system. The first level \(-\text{qipa} = \text{level}=0\) performs only minimal interprocedural analysis and optimisations. The \(-\text{qipa} = \text{level}=1\) adds inlining, limited alias analysis and limited call-site tailoring. The highest level, \(-\text{qipa} = \text{level}=2\), performs full data flow and alias analysis. Additionally, the programmer can use a set of suboptions that can instruct the compiler and give valuable information about the selected functions such as names of functions which represent program exits or a set of functions that do not directly refer to any global variable. To see the full listing of those options, please refer to the compiler's manual (i.e. \text{bgxlc -qreport}).

The \(-\text{qipa}\) option should be used both at the compile and link steps.

4. SIMD vectorisation

The \(-\text{qsimd}\) option enables the compiler to automatically take advantage of vector instructions. The usage and impact on the performance of Blue Gene/Q applications will be discussed in Section 8.1.2, “Enabling vectorisation on the Blue Gene/Q architecture”.

Higher level optimisation methods may generate code which does not fully conform to IEEE rules. Therefore, all high level optimisation should be used with caution. Numerical results should be checked after usage of such optimisations. There are a few compiler switches which could prevent the compiler from producing code incompatible with IEEE rules. One of them is the \(-\text{qstrict}\) option which is often used together with the \(-\text{O3}\) optimisation level. The other one is the \(-\text{qfloat}\) option which can improve the accuracy of floating-point calculations. Please refer to the compiler's manual to see the possible suboptions.
It is useful to know which optimisations and what kind of code transformations were actually applied to the code. Here, we list a few options that can be useful when working on single node performance with XL compilers:

- **-qlist**: Produces a compiler listing that includes an object listing,
- **-qlistopt**: Produces a compiler listing that displays all the options that were in effect when the compiler was invoked,
- **-qreport**: Generates for each source file `name.ext` a file `name.lst` with pseudo code and a description of the kind of code optimisations which were actually performed during compilation,
- **-qsourcetext/html>**: Produces a compiler listing that includes source code.

An informative, full and human-readable listing will be generated if you invoke the compiler with the `-qreport -qlist -qsource` options. Note that compiler listings are generated during compilation and therefore do not affect the performance of the application.

Each physical core of a PowerPC A2 chip provides hardware support for 4 parallel threads. One possibility of using this hardware feature is to parallelise each MPI process with OpenMP or Pthreads programming model (these are the recommended mixed programming models for Blue Gene/Q). However, sometimes it is a very difficult or even impossible task. In those situations, it is often worthwhile to let the compiler try to auto-parallelise some of the program loops. This compiler feature is enabled by the `-qsmp=auto` option. Compiling the code with `-qreport` will produce information about successful auto-parallelisations. To use `-qsmp=auto`, compile your code with the compilers which have the `_r` suffix in their names (e.g. `mpixlc_r`, `mpixlf95_r`).

### 8.1.2. Enabling vectorisation on the Blue Gene/Q architecture

The text of this section has partly been taken from “Redbook: IBM System Blue Gene Solution: Blue Gene/Q Application Development”.

The Blue Gene/Q hardware contains the quad-processing extension (QPX) to the IBM Power Instruction Set Architecture. The computational model of the QPX architecture is a vector single instruction, multiple data (SIMD) model with four execution slots and a register file that contains 32 registers with 256 bits. Each of the 32 registers contains four elements of 64 bits. Each of the execution slots operates on one vector element. These elements are referred to as vector registers or quad registers.

Programmers and users of Blue Gene/Q systems should always look for vectorisation opportunities in their codes. There are two methods of using the QPX floating-point instruction set when using the IBM XL compiler for Blue Gene/Q:

- Automatic compiler SIMDisation,
- Vector intrinsics functions or assembly code.

In order to perform vector instructions, the addresses of the variables should be 32-byte aligned. With automatic compiler SIMDisation, this might be automatically adjusted by the compiler. If not, programmers should ensure proper alignment.

If your program is compiled with the XL compiler for Blue Gene/Q, SIMDisation is enabled by default. The compiler attempts to automatically transform code to efficiently use the QPX floating-point instruction set. Simdisation is enabled at all optimisation levels but more aggressive and effective SIMDisation occurs at higher optimisation levels. Simdisation is controlled by the `-qsimd` option and can be disabled by using the `-qsimd=noauto` option.

Compiler listings generated with the `-qreport` option can give additional information about SIMDisation. Examples of such useful messages are shown below:

An attempt to SIMD vectorize failed because the loop contain variables with a non-vectorizable alignment.
An attempt to SIMD vectorize failed because of a data dependence.
An attempt to SIMD vectorize failed because the iteration count is too small.
Loop (loop index 3 with nest-level 2 and iteration count 2048) at mxm.c <line 40> was SIMD vectorized.

Specific QPX vector intrinsics are not covered by this section. For basic information and examples of usage, please see "Redbook: IBM System Blue Gene Solution: Blue Gene/Q Application Development". For a full list of the vector compiler built-in functions, please see "IBM XL C/C++ for Blue Gene/Q, V12.1 Language Reference, GC14-7364-00" and "IBM XL Fortran for Blue Gene/Q, V14.1 Language Reference, GC14-7369-00".

8.1.3. Optimisation strategies

The impact of single core optimisation on code performance will be presented here with a simple example of the product of 2 matrices written in the C language. We start with two alternative versions of the code: The first one (Example 11) is a naive solution where loops are programmed in the I-J-K order. It is very easy to observe that this approach will produce highly nonoptimal code in terms of cache usage. Matrices in C language are stored in memory row-by-row and accessing the \( B[k][j] \) element in each iteration of the innermost loop will surely generate high numbers of cache misses. The second (Example 12) is an alternative version of the code in which loops are programmed in the I-K-J order. We will go through a few compiler optimisation steps and try to understand what the compiler is actually doing with the code and how this affects the performance of both versions.

**Example 11. Simple C example program performing matrix multiplication (with loops in I-J-K order)**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define N 2048

double A[N][N];
double B[N][N];
double C[N][N];

int main(int argc,char **argv) {
    int i,j,k;
    struct timeval start;
    struct timeval stop;
    double time;
    const double micro = 1.0e-06;

    mat_init(A,B,C);

    gettimeofday(&start,NULL);
    for (i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            for (k=0; k<N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&stop,NULL);
    time = (stop.tv_sec - start.tv_sec)
           + micro * (stop.tv_usec - start.tv_usec);
    printf("Matmul time=%f\n",time);
}
```
Example 12. Cache-friendly version of the simple C example program performing matrix multiplication (with loops in I-K-J order)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
define N 2048

double A[N][N];
double B[N][N];
double C[N][N];

int main(int argc,char **argv) {

    int i,j,k;
    struct timeval start;
    struct timeval stop;
    double time;
    const double micro = 1.0e-06;

    mat_init(A,B,C);

    gettimeofday(&start,NULL);
    for (i=0; i<N; i++) {
        for(k=0; k<N; k++) {
            for (j=0; j<N; j++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&stop,NULL);

    time = (stop.tv_sec - start.tv_sec) + micro * (stop.tv_usec - start.tv_usec);
    printf("Matmul time=%f\n",time);
    return 0;
}
```

• Step 1 - basic optimisation level

Start by compiling both codes with the first recommended optimisation level:

-O2

The time spent in the main program loops is 448.31 seconds for the I-J-K version and 56.18 seconds for the I-K-J version. As we can see, the I-K-J version of the code shows much better performance. However, it gives only 305.8 MFlop/s which is roughly 2.4% of the theoretical peak performance of a single compute core of a PowerPC A2 chip.

• Step 2 - more compiler optimisations

Next, compile the code with the -O3 option. However, to ensure that the compiler does not perform aggressive code transformations, add the -qstrict option:
-O3 -qstrict

Computation time is 41.55 seconds for the I-J-K version. Surprisingly, one can discover that the time for I-K-J version is roughly the same, approximately 48.81 s. To understand what actually happened, recompile the code with the -qreport -qsource -qlist options. We can see that, in both cases, the compiler applied loop unrolling to the outermost loop:

Outer loop has been unrolled 4 time(s).

As one can also see from the optimisation listing, the compiler was unable to vectorise the code due to the -qstrict option:

Loop was not SIMD vectorized because the floating point operation is not vectorizable under -qstrict.

The performance is still very weak with 413.5 MFlop/s (3.2% of the theoretical peak performance of a single compute core of a PowerPC A2 chip).

- Step 3 - full -O3 compiler optimisations and vectorisation

Next, compile the code with the full set of third level optimisations:

-03

Computational time is now 15.69 s for both the I-J-K version and I-K-J version. The following set of optimisations were applied by the compiler to both codes:

0  37  1  Outer loop has been unrolled 4 time(s).
0  37  1  Loop was not SIMD vectorized because the loop is not the innermost loop.
0  38  2  Outer loop has been unrolled 4 time(s).
0  39  3  Loop with nest-level 2 and iteration count 2048 was SIMD vectorized.

Additionally, in the case of the I-J-K version, there was a loop interchange applied:

0  37  1  Loop interchanging applied to loop nest.

We can see, then, that the compiler produces very similar machine code for both versions by transforming version I-J-K into I-K-J by loop interchanging. Additionally, the innermost loops have been SIMD vectorised. Performance is 1094.9 MFlop/s (8.5% of the theoretical peak performance of a single compute core of a PowerPC A2 chip).

- Step 4 - aggressive compiler optimisations

Next, compile the code with the -O3 option and enable aggressive high order transformations with the -qhot option.

-03 -qhot

Computation time is now 448.04 s for both versions of the code. This performance decrease was unexpected. Again, to understand what actually happened, we need to analyse the actual optimisations which the compiler applied to the code. The -qreport -qsource -qlist options produce some interesting information
which should not be ignored. In the **OBJECT SECTION, OPTIMIZATION** of the listing, we can see that the main program loops were replaced by nonoptimal DGEMM call:

```c
__xl_dgemm(&$$__xl_matmul__transab_N0,&$$__xl_matmul__transab_N0,
&$$__xl_matmul__J0,&$$__xl_matmul__I0,&$$__xl_matmul__K0,
&$$__xl_matmul__ALPHA_10,((char *)&b  + 0u),&$$__xl_matmul__LDB0,
((char *)&a  + 0u),&$$__xl_matmul__LDA0,&$$__xl_matmul__BETA_00,
((char *)&c  + 0u),&$$__xl_matmul__LDC0)
```

This example shows that it is extremely important, when using XL compilers, to always start your single core optimisation from the lowest recommended optimisation level. Usage of higher order transformations enabled by the `-qhot` compiler option resulted in a performance decrease which would not have been identified if we had started the optimisation at the `-qhot` level.

• **Step 5 - ESSL dgemm**

The code substitution presented in the previous step resulted in a performance decrease. The DGEMM function of the XL compiler (`xl_dgemm`) is surely not optimised for execution on the Blue Gene/Q architecture. It is possible to obtain high performance with proper code substitution by using DGEMM of ESSL as shown below:

**Example 13. Cache-friendly version of the simple C example program performing matrix multiplication (with loops in I-K-J order)**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <essl.h>

#define N 2048

double A[N][N];
double B[N][N];
double C[N][N];

int main(int argc,char **argv) {

    int i,j,k;
    struct timeval start;
    struct timeval stop;
    double time;
    const double micro = 1.0e-06;
    double one = 1.0, zero = 0.0;

    gettimeofday(&start,NULL);
    dgemm("T","T",N,N,N,one,&A,N,&B,N,zero,&C,N);
    gettimeofday(&stop,NULL);
    time = (stop.tv_sec - start.tv_sec)
        + micro * (stop.tv_usec - start.tv_usec);
    printf("Matmul time=%f\n",time);

    return 0;
}
```

The above example can be compiled with the following commands:

```
bgxlc -O3 -qtune=qp -qarch=qp -I/opt/ibmmath/include -c mxm_dgemm.c
```
bgxlc_r -O3 -qtune=qp -qarch=qp mxm_dgemm.o -L/opt/ibmmath/lib64 -lesslbg -L/opt/ibmcmp/xlf/bg/14.1/lib64 -lxlf90_r -lxlopt -lxl -lxlfmath -lxlsmp -lrt -o mxm_dgemm

The resulting computation time is 3.34 s which gives us 5143.6 MFlop/s (40% of the theoretical peak performance of a single compute core of a PowerPC A2 chip).

- **Step 6 - ESSL SMP dgemm**

Each physical core of the PowerPC A2 chip has built-in support for up to 4 hardware threads. In order to obtain high single core performance, we will try to use all 4 hardware threads within one physical core. First, we will link the above ESSL DGEMM example with the SMP version of the ESSL library:

bgxlc -O3 -qtune=qp -qarch=qp -I/opt/ibmmath/include -c mxm_dgemm.c

bgxlc_r -O3 -qtune=qp -qarch=qp mxm_dgemm.o -L/opt/ibmmath/lib64 -lesslsmpbg -L/opt/ibmcmp/xlf/bg/14.1/lib64 -lxlf90_r -lxlopt -lxl -lxlfmath -lxlsmp -lrt -o mxm_dgemm

Second, we need to ensure that all 4 execution threads will use a single physical core. One of the ways of doing this is to set the environment variables OMP_NUM_THREADS to 4 and BG_THREADLAYOUT to 2.

Computation time is now 1.62 s which corresponds to 10604.8 MFlop/s (82.9% of the theoretical peak performance of a single compute core of a PowerPC A2 chip). We consider that this is nearly the highest possible single core performance for matrix multiplication of this particular problem size on Blue Gene/Q architecture.

Single core optimisation of codes on the Blue Gene/Q architecture should, therefore, be seen as a process which should be started from the lowest recommended compiler optimisation level. Programmers must be aware of code transformations which are applied by the compiler since in some cases compiler optimisations may decrease code performance. We should always pay attention to optimisations such as vectorisation, loop interchange, loop unrolling and code substitution with optimised (or nonoptimised) library calls. Additionally, since PowerPC A2 chips provide support for up to 4 hardware threads within each physical core, we should always try to apply thread level parallelism to processes.

### 8.2. Advanced MPI usage

#### 8.2.1. Tuning and environment variables

The Blue Gene/Q MPI implementation provides several environment variables that affect its behaviour. Setting these environment variables can allow a program to run faster, or if set incorrectly, might cause the program to not run at all.

**Important**

These variables only take effect if they are set via the --envs of the runjob command (for example, runjob --envs PAMID_THREAD_MULTIPLE=1 <other runjob options> : application.x).

If you use non-blocking MPI subroutines (such as MPI_Isend, MPI_Irecv...), you can obtain near perfect overlap of communications by computations by setting the PAMID_THREAD_MULTIPLE to 1. For a pure MPI application or a hybrid application initialised with the MPI_Init_thread() function without MPI_THREAD_MULTIPLE, PAMID_THREAD_MULTIPLE is set to 0 and the transmission of the message only begins when the receive is posted via a call to MPI_Recv or to MPI_Wait (a call to MPI_Irecv does not allow the communication to start). But if set to 1, MPI uses additional helper threads which ensure that the communications progress as soon as the messages are ready to be sent. Therefore, a non-blocking communication can begin immediately without waiting for the receiver to be ready. This variable can be used in all cases except if you compiled your application with a "legacy" MPI library (see Section 5.3.1). Note also, that if you are using all 64 hardware threads of a compute node, there are no hardware threads available for MPI and, therefore, this functionality will not work (the environment variable is simply ignored).
Some other interesting environment variables are listed below:

- **PAMID_COLLECTIVES=** <integer>

  Controls whether optimised collectives are used. The possible values are 0 (optimised collectives are not used; only MPICH point-to-point, based collectives are used) or 1 (optimised collectives are used). The default is 1.

- **PAMID_EAGER=** <bytes>

  Sets the message size cutoff for the switch from the eager to the rendezvous protocol. The default size is 4097 bytes. The MPI rendezvous protocol is optimised for maximum bandwidth. However, there is an initial handshake between the communication partners, which increases the latency. In case your application uses many short messages, you might want to decrease the message size (even down to 0). On the other hand, if your application can be mapped well to the TORUS network and uses mainly large messages, increasing the size limit might lead to a better performance. The 'K' and 'M' multipliers can be used in this value, such as "16K" or "1M". This environment variable is identical to PAMID_RZV. See also PAMID_EAGER_LOCAL.

- **PAMID_EAGER_LOCAL=** <bytes>

  Sets the message size cutoff for the switch from the eager to the rendezvous protocol (see PAMID_EAGER for further information) when the destination rank is local (i.e. on the same node). The default size is 4097 bytes. The 'K' and 'M' multipliers can be used in the value. For example, "16K" or "1M" can be used. This environment variable is identical to PAMID_RZV_LOCAL.

- **PAMID_NUMREQUESTS=** <integer>

  Controls how many asynchronous collectives to issue before barriering. Default is 1. Setting it higher may slightly improve performance but will use more memory for queueing unexpected data.

- **PAMID_RZV=** <integer>

  See PAMID_EAGER.

- **PAMID_RZV_LOCAL=** <integer>

  See PAMID_EAGER_LOCAL.

### 8.2.2. Mapping tasks on node topology

Mapping is defined as the assignment of processes to Blue Gene/Q cores. Usually, these are MPI processes, each one identified by its MPI rank. However, this same mapping concept applies if a different communication protocol is used.

When a job is run on a Blue Gene/Q system, a compute block is booted which is made up of a number of compute nodes. The compute blocks can be either small blocks of one or more nodeboards within a single midplane, or large blocks of one or more midplanes.

The network topology (connections between the compute nodes) for Blue Gene/Qs is a five-dimensional (5D) torus or mesh, with direct links between the compute nodes which are nearest to each other in the ±A, ±B, ±C, ±D and ±E directions. A midplane, which consists of 512 compute nodes, is referred to as a 4x4x4x4x2 torus. Every block on Blue Gene/Q, large or small, has an E dimension of size 2. The torus direction between two midplanes inside the same rack is the D dimension. When cables go down a row of racks, the direction is the C dimension. When cables go down a column of racks, the direction is the B dimension. The remaining direction, which can go down a row or column (or both), is the A dimension. When two sets of cables go down a row or column, the longest cables define the A dimension.

When communication involves the nearest neighbours on the torus network, a large fraction of the theoretical peak bandwidth can be obtained. In order to significantly improve scaling for a number of applications, particularly at large processor counts, it is possible to control the placement of MPI ranks so that communication remains local.
This can be done by mapping MPI ranks onto the Blue Gene/Q torus network in a way that preserves locality for nearest-neighbour communication.

Mapping might not be important for jobs that use one midplane (512 nodes) or less of the Blue Gene/Q system due to the compact shape of a midplane, $<A,B,C,D,E> = <4,4,4,4,2>$, and the high degree of connectivity. Mapping can be particularly useful for applications which have a regular Cartesian topology and are dominated by nearest-neighbour boundary exchange, particularly for large configurations. For jobs using a batch job scheduler, mapping requires information about the shape of the block, not just the number of nodes, and special key words might be required to define the shape.

The default mapping places MPI ranks on the system in ABCDET order where the rightmost letter increments first, and where $<A,B,C,D,E>$ are torus coordinates and $T$ is the processor ID in each node ($T = 0$ to $N - 1$ with $N$ being the number of processes per node). If the job uses the default mapping and specifies one process per node, the following assignments are made:

- MPI rank 0 is assigned to coordinates $<0,0,0,0,0,0>$.
- MPI rank 1 is assigned to coordinates $<0,0,0,1,0,0>$.
- MPI rank 2 is assigned to coordinates $<0,0,1,0,0,0>$.
- MPI rank 3 is assigned to coordinates $<0,0,1,1,0,0>$.

The assignments continue like this, first incrementing the E coordinate, then the D coordinate, and so on, until all of the processes are mapped. Note that the E dimension is always 2 on the Blue Gene/Q system.

The default mapping is often a good choice. The mapping can also be controlled by passing arguments to the runjob command or, alternatively, by constructing a specially ordered communicator in the application.

Using a customised map file provides the most flexibility. The syntax for the map file is simple. It must contain one line for each MPI rank in the Blue Gene/Q block with six integers on each line which are separated by spaces. The six integers specify the $<A,B,C,D,E,T>$ coordinates for each MPI process rank. The first line in the map file gives the coordinates of the MPI rank 0, the second line gives MPI rank 1, and so on. It is important to ensure that the map file is consistent with a unique relationship between MPI rank and $<A,B,C,D,E,T>$ location. The “T” coordinate in the map file ranges from 0 to $N - 1$ with $N$ being the number of ranks per node.

The runjob command for the Blue Gene/Q system provides two methods to specify the mapping. For example, you can add `--mapping TEDCBA` to request TEDCBA order, where A increments first. All permutations of ABCDET are permitted. You can also create a customised map file and use `--mapping my.map` where `my.map` is the name of the map file.

### 8.2.2.1. Mapping file example

To understand the meaning of mapping, let us consider a small block consisting of 64 compute nodes. The shape of this block would be 2x2x4x2x2 and the connection would be a partial torus 00101 where only C and E dimensions are torus.

Suppose we run a job on this block with 2 ranks per node (and thus with a total of 128 MPI tasks), the default mapping ABCDET would be:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
</tr>
<tr>
<td>1</td>
<td>000010</td>
</tr>
<tr>
<td>2</td>
<td>000011</td>
</tr>
<tr>
<td>3</td>
<td>000100</td>
</tr>
<tr>
<td>4</td>
<td>000100</td>
</tr>
</tbody>
</table>

-------------------> MPI task 0
-------------------> MPI task 1
-------------------> MPI task 2
-------------------> MPI task 3
-------------------> MPI task 4

-------------------> MPI task 5
-------------------> MPI task 6
-------------------> MPI task 7
-------------------> MPI task 8
-------------------> MPI task 9
-------------------> MPI task 10
-------------------> MPI task 11
-------------------> MPI task 12
-------------------> MPI task 13

-------------------> MPI task 14
-------------------> MPI task 15
-------------------> MPI task 16
-------------------> MPI task 17
-------------------> MPI task 18
-------------------> MPI task 19
-------------------> MPI task 20
-------------------> MPI task 21
-------------------> MPI task 22
-------------------> MPI task 23

-------------------> MPI task 24
-------------------> MPI task 25
-------------------> MPI task 26
-------------------> MPI task 27
-------------------> MPI task 28
-------------------> MPI task 29
-------------------> MPI task 30
-------------------> MPI task 31
-------------------> MPI task 32
-------------------> MPI task 33

-------------------> MPI task 34
-------------------> MPI task 35
-------------------> MPI task 36
-------------------> MPI task 37
-------------------> MPI task 38
-------------------> MPI task 39
-------------------> MPI task 40
-------------------> MPI task 41
-------------------> MPI task 42
-------------------> MPI task 43

-------------------> MPI task 44
-------------------> MPI task 45
-------------------> MPI task 46
-------------------> MPI task 47
-------------------> MPI task 48
-------------------> MPI task 49
-------------------> MPI task 50
-------------------> MPI task 51
-------------------> MPI task 52
-------------------> MPI task 53

-------------------> MPI task 54
-------------------> MPI task 55
-------------------> MPI task 56
-------------------> MPI task 57
-------------------> MPI task 58
-------------------> MPI task 59
-------------------> MPI task 60
-------------------> MPI task 61
-------------------> MPI task 62
-------------------> MPI task 63
0 0 0 1 0 1 ----------------> MPI task 5
0 0 0 1 1 0 ----------------> MPI task 6
0 0 0 1 1 1 ----------------> MPI task 7
0 0 1 0 0 0 ----------------> MPI task 8
0 0 1 0 0 1 ----------------> MPI task 9
0 0 1 0 1 0 ----------------> MPI task 10
0 0 1 0 1 1 ----------------> MPI task 11
0 0 1 1 0 0 ----------------> MPI task 12
0 0 1 1 0 1 ----------------> MPI task 13
0 0 1 1 1 0 ----------------> MPI task 14
0 0 1 1 1 1 ----------------> MPI task 15
0 0 2 0 0 0 ----------------> MPI task 16

.........................................................
.........................................................
1 1 3 1 1 1 ----------------> MPI task 127

The mapping ABDETC, however, would be:

0 0 0 0 0 0 ----------------> MPI task 0
0 0 1 0 0 0 ----------------> MPI task 1
0 0 2 0 0 0 ----------------> MPI task 2
0 0 3 0 0 0 ----------------> MPI task 3
0 0 0 0 0 1 ----------------> MPI task 4
0 0 1 0 0 1 ----------------> MPI task 5
0 0 2 0 0 1 ----------------> MPI task 6
0 0 3 0 0 1 ----------------> MPI task 7
0 0 0 0 1 0 ----------------> MPI task 8
0 0 1 0 1 0 ----------------> MPI task 9
0 0 2 0 1 0 ----------------> MPI task 10
0 0 3 0 1 0 ----------------> MPI task 11
0 0 0 0 1 1 ----------------> MPI task 12
0 0 1 0 1 1 ----------------> MPI task 13
8.3. Advanced OpenMP usage

8.3.1. Thread affinity

On Blue Gene/Q, each OpenMP thread is assigned a hardware thread (and therefore a core) and the OpenMP thread keeps this assignment for its duration. The number of hardware threads that are available to a process is dependent on the number of processes in the node (see Section 4.2). The compute node kernel supports two algorithms for mapping threads. The `BG_THREADLAYOUT` environment variable allows you to choose the layout algorithm.

If the `BG_THREADLAYOUT=1` (default value), the software threads are assigned across the cores within the process before assigning software threads to additional hardware threads within a core. See example in Figure 16.

If the `BG_THREADLAYOUT=2`, the software threads are assigned to all hardware threads within a core before assigning software threads on other cores. See example in Figure 17.

**Figure 16. Thread affinity with BG_THREADLAYOUT=1 and 8 processes each with 8 threads.**
8.4. Memory optimisation

8.4.1. Memory consumption

On Blue Gene/Q systems, the available memory is relatively limited. It is very useful, therefore, to know how much memory an application uses and how much memory is free.

Integrating the following routine (in C) will allow you to track the memory usage of your application:

Example 14. Function to measure memory usage

```c
#include <stdio.h>
#include <stdlib.h>
#include <spi/include/kernel/memory.h>

void print_memusage()
{
    uint64_t shared, persist, heapavail, stackavail, stack, heap, guard, mmap;
    Kernel_GetMemorySize(KERNEL_MEMSIZE_GUARD, &guard);
    Kernel_GetMemorySize(KERNEL_MEMSIZE_SHARED, &shared);
    Kernel_GetMemorySize(KERNEL_MEMSIZE_PERSIST, &persist);
    Kernel_GetMemorySize(KERNEL_MEMSIZE_HEAPAVAIL, &heapavail);
    Kernel_GetMemorySize(KERNEL_MEMSIZE_STACKAVAIL, &stackavail);
    Kernel_GetMemorySize(KERNEL_MEMSIZE_STACK, &stack);
    Kernel_GetMemorySize(KERNEL_MEMSIZE_HEAP, &heap);
    Kernel_GetMemorySize(KERNEL_MEMSIZE_MMAP, &mmap);
    printf("MEMSIZE heap: %.2f/%.2f stack: %.2f/%.2f heapavail: %.2f MB
", (double)heap/(1024*1024), (double)heapavail/(1024*1024),
            (double)stack/(1024*1024), (double)stackavail/(1024*1024),
            double)mmap/(1024*1024) );
    printf("MEMSIZE shared: %.2f persist: %.2f guard: %.2f MB
", 
```
8.4.2. Memory fragmentation

If the memory allocations of your application fail, even though there is sufficient memory available, it can be due to memory fragmentation.

The operating system used on the Blue Gene/Q compute nodes (called CNK for Compute Node Kernel, see Section 2.5) is very simple. This is to have the least possible interference to the processes running on the Blue Gene/Q as well as to run the fastest possible. Accordingly, the CNK memory manager is very basic and is not capable of handling the phenomenon of memory fragmentation.

The memory can fragment when an application repeatedly makes dynamic allocations and deallocations. For example, if you allocate a certain number of small work arrays and thereby fill up the memory, after which you deallocate some of them, some holes will appear. If then you attempt to make a new allocation, the exploitation system will try to find a sufficiently large free place in which to put your new work array. If a place is not found (because your array is larger than the largest hole), it cannot allocate it even if the total available space is largely sufficient. A more advanced operating system is capable of allocating non-adjacent memory blocks or moving the blocks to prevent fragmentation. The approach used for memory management on Blue Gene/Q allows the allocations/deallocations to be done very rapidly but to the detriment of its flexibility.

Here are some suggestions for writing an application in order to prevent memory fragmentation:

- Allocate the work arrays all at the same time at the beginning of the application.
- Avoid multiple and repeated allocations/deallocations.
- Give preference to allocations/deallocations of fixed-size arrays during the execution of your application.

If you don't succeed in getting rid of this phenomenon, try to manage a part of the memory space yourself. This can be done by allocating a large space and distributing chunks of memory into it as needed during the execution. Be aware that this approach can prove to be very complex.

Example 15 is a small program in C which illustrates the problem of memory fragmentation:

Example 15. Memory fragmentation illustration

```c
#include <stdlib.h>
#define NBLOCKS 40
#define MB 1024*1024

int main(int argc,char **argv)
{
    int     i;
    size_t  size;
    char   *smallblocks[NBLOCKS], *bigblock;

    // Allocate 'small' blocks
    // With 1 rank-per-node, it will use all the memory (16 GiB)
    size = 400*MB;
    for (i=0;i<NBLOCKS;i++)
    {
        smallblocks[i] = (char *) calloc(size, sizeof(char));
        if(smallblocks[i]==NULL)
```
{  
    printf("'Allocation of smallblocks[%i] failed! ",i);  
    break;  
}

// Deallocate 1 smallblock every 2  
// After deallocation, we should have NBLOCKS/2  
// free blocks of 400 MiB  
for (i=0;i<NBLOCKS;i+=2) free(smallblocks[i]);

// Allocate big block  
// Total freespace should be at least 20*400 MiB=8000 MiB  
size = 600*MB;  
bigblock = (char *) malloc(size, sizeof(char));  
if(bigblock==NULL)  
{
    printf("'Allocation of bigblock failed!n'";  
    return EXIT_FAILURE;  
}

return EXIT_SUCCESS;

If we are running it with only one process on a full compute node, we obtain the following output during the execution:

runjob --ranks-per-node 1 --np 1 : ./a.out  
Allocation of bigblock failed!

2013-11-19 01:31:25.961 (WARN ) [0xfff68ea8b10]:6919:ibm.runjob_client.Job:  
  normal termination with status 1 from rank 0

8.4.3. Maximum memory available

Inside a compute node, the memory is distributed between the processes depending on the value you give to the  
--ranks-per-node option of the runjob command. The memory inside a node is statically mapped to the processes that are running on this node and is more or less evenly distributed between them.

Therefore, each process only has access to its own memory and cannot use the memory of another process even if this latter one does not need it. For example, if a job is run with 16 processes, each process will have around 1 GiB of available memory. If a process needs more memory, it can not allocate it even if all the other processes have free space. In this case, the program will fail as soon as one process tries to use more than its part.

In reality, the memory is not evenly distributed between the processes. The system tries to give more or less equal quantities but there are differences between the processes. This is due, among other things, to the fact that the kernel has to reserve some place for itself (16 MiB) and for the shared memory space (32 MiB by default; this can be changed with the BG_SHAREDMEMSIZE environment variable, but it should be kept at least at 32 MiB because the MPI library requires this).

In some cases, especially when there are a lot of processes per node, a certain process can have significantly less memory than the others. This is a problem with applications that are memory constrained. A better division of the memory is possible by setting the BG_MAPCOMMONHEAP environment variable to 1.

Here is an extract about this environment variable from the "Redbook: IBM System Blue Gene Solution: Blue Gene/Q Application Development": "This option obtains a uniform heap allocation between the processes; however, the trade off is that memory protection between processes is not as stringent. In particular, when using the
option, it is possible to write into another process’ heap. Normally this would cause a segmentation violation, but with this option set, the protection mechanism is disabled to provide a balanced heap allocation.”

8.5. Transactional memory

Transactional memory is a mechanism for controlling concurrent memory access in shared memory parallel programming. In classical shared memory programming models, it is a programmer's duty to ensure proper concurrency control so that parallel threads do not update the same resources at the same time. Traditionally, concurrency control is achieved with the use of thread locks. In some shared memory programming languages, locks can be implemented directly (e.g. in Pthreads library). Higher level shared memory programming models, such as OpenMP, indirectly introduce specific mechanisms to control the concurrent memory access by locks. One of these mechanisms is the critical pragma which restricts execution of a given structured block to a single thread at a time. The other one is the atomic pragma which ensures that a specific storage location is updated atomically (i.e. by only a single thread at a given moment).

Lock-based synchronisation between threads can lead to performance issues and should be avoided if possible. Lock mechanisms introduced in parallel loops cause some of the threads to wait to update protected data. This could dramatically slow down the execution of a loop. Transactional memory is an alternative to lock-based shared memory control. It works in a similar way to that of transactional databases. That is, all the read and write operations in a given memory region are grouped and executed as a single operation. Therefore, all threads can enter the critical region at the same time but the transactional memory mechanism must guarantee the atomicity, consistency and isolation properties of the operations.

8.5.1. Transactional memory support on the Blue Gene/Q

The Blue Gene/Q compute chip is one of the very first processors available for HPC which provides transactional memory hardware support. To use this hardware mechanism, developers must specify the code regions, so called transactional atomic regions, which should be visible as a single transactional operation for the system. The Blue Gene/Q hardware automatically detects memory read or write conflicts as well as threads that try to re-access the shared memory data or are stopped without updating it. To specify the transactional atomic region, the following compiler directives should be introduced in the source code:

* #pragma tm_atomic in C/C++ codes
* !TM$ TM_ATOMIC ... !TM$ END TM_ATOMIC in Fortran codes.

Moreover, special compilation is needed to generate code which uses the transactional memory on the Blue Gene/Q system. First of all, the thread-safe IBM XL compilers (compilers with _r in name) must be used. Secondly, the -qtm compiler option needs to be used in order to process the transactional memory directives in the program.

Following is a simple example of using the transactional memory compiler directives (as given in "IBM XL C/ C++ for Blue Gene/Q, V12.1, Compiler Reference"): 

**Example 16. Simple example of transactional memory directives usage**

```c
int v, w, z;
int a[10], b[10];
#pragma omp parallel for
for (v = 0; v < 10; v++)
    for (w = 0; w < 10; w++)
        for (z = 0; z < 10; z++) {
            //Use the tm_atomic directive
            //to indicate a transactional atomic region
            #pragma tm_atomic
            { 
                a[v] = a[w] + b[z];
            }
```

92
8.5.2. Environment variables and built-in functions

Programmers can control the Blue Gene/Q transactional memory mechanism with a number of environment variables. The most useful ones are listed below:

- **TM_MAX_NUM_ROLLBACK**: Maximum number of times a thread can roll back a particular transaction atomic region before failing the transaction and going into serialised execution (positive integer not greater than $2^{32}$, default is 10).

- **TM_REPORT_STAT_ENABLE**: Enables or disables statistics query built-in functions for transactional memory (possible values are YES or NO, default is NO).

- **TM_REPORT_NAME**: The name and location of the statistics log file (by default the log file is located in the current working directory and is named `tm_report.log.MPIrank`).

- **TM_REPORT_LOG**: Specifies how to create the statistics log file for transactional memory. (Possible values are: SUMMARY - the statistics log file is generated at the end of the program, FUNC - the statistics log file is updated at each call to `tm_print_stats`, ALL - the statistics log file is updated at each call to `tm_print_stats` and at the end of the program, VERBOSE - the same as ALL but generates more useful information about transactions.)

Programmers can also use the specific transactional memory built-in functions. The `speculation.h` header file must be included in the source file. Most of these built-in functions are related to transactional memory statistics:

- **void tm_get_stats (TmReport_t *stats)** - Can be used to obtain transactional memory statistics for a particular hardware thread in the program, `TmReport_t` type definition is given in the "IBM XL C/C++ for Blue Gene/Q, V12.1, Compiler Reference".

- **void tm_get_all_stats (TmReport_t *stats)** - Can be used to obtain transactional memory statistics for all the hardware threads in the program, `TmReport_t` type definition is given in the "IBM XL C/C++ for Blue Gene/Q, V12.1, Compiler Reference".

- **void tm_print_stats(void)** - Can be used to write transactional memory statistics for a particular hardware thread to a log file.

- **void tm_print_all_stats(void)** - Can be used to write transactional memory statistics for all the hardware threads to a log file.

8.5.3. Usage recommendations

It should be emphasised that hardware transactional memory is a rather new concept and developers should be very careful when introducing this mechanism in their codes. Although the use of transactional memory on Blue Gene/Q does not require large source code modifications (only single compiler directives), there are some switches and settings which need to be considered. The performance of the transactional memory regions depends largely on the granularity of computations. Therefore, all transactional memory parameters must be carefully selected for each individual problem size and its granularity.

Some examples and recommendations of Blue Gene/Q transactional memory hardware usage based on synthetic benchmarks have been presented in "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories" and "What scientific applications can benefit from hardware transactional memory?".

**Note**

Please note that using the hardware transactional memory does not guarantee an increase in performance and should be used with caution. Improper use of this mechanism may lead to a significant slowdown of selected regions of the application.
8.6. Thread level speculation

Thread Level Speculation (TLS), also known as Speculative MultiThreading (SpMT), is a dynamic parallelisation technique that depends on an out-of-order execution to achieve speedup on multiprocessor CPUs. It is a kind of speculative execution that occurs at the thread level as opposed to the instruction level.

Thread-level speculative execution uses hardware support that dynamically detects thread conflicts and rolls back conflicting threads for re-execution. It overcomes the analysis problems of compiler-directed code parallelisation. You can get significant performance gains in your applications by adding the compiler directives of thread-level speculative execution without rewriting the program code. Thread-level speculative execution is enabled with the `-qsmp=speculative` compiler option.

8.6.1. Rules for committing data

With thread-level speculative execution, tasks are committed according to the following rules:

- Before a task is committed, the data is in a speculative state.
- Tasks are committed in program order.
- Therefore, a task which is later in the program order can only be committed when all the earlier tasks have been committed. If a thread running a task encounters a conflict, all the threads running later tasks must roll back and retry. Eventually, all tasks are committed.

8.6.2. Thread-level speculative execution and OpenMP

It is also possible to write your code so that it can directly use the speculative execution of some OpenMP instructions.

The directives for thread-level speculative execution only take effect if you specify the `-qsmp=speculative` compiler option.

Speculative threads are not able to detect access conflicts in OpenMP `THREADPRIVATE` data. Accessing such data inside regions of thread-level speculative execution does not guarantee the same behaviour as regions being run by one thread.

8.6.2.1. Speculative DO

The `SPECULATIVE DO` directive instructs the compiler to speculatively parallelise a DO loop.

The `SPECULATIVE DO` directive precedes a `DO` loop. The optional `END SPECULATIVE DO` directive indicates the end of the `DO` loop.

This is an example of using the `SPECULATIVE DO` instruction:

```fortran
INTEGER :: k = 0
INTEGER :: i = 1
...
!SEP$ SPECULATIVE DO FIRSTPRIVATE(k)
DO i = 1, 10
   k = k + 1
END DO
...
END
```

1This option is valid only using XL compilers.
8.6.2.2. Speculative Sections

The `SPECULATIVE SECTIONS` directive instructs the compiler to speculatively parallelise sections of the code. In code blocks delimited by `SPECULATIVE SECTIONS` and `END SPECULATIVE SECTIONS`, you can use the `SPECULATIVE SECTION` directive to delimit program code segments.

The `SPECULATIVE SECTION` directive is optional for the first code segment inside the `SPECULATIVE SECTIONS` directive. The code segments after the first one must be preceded by a `SPECULATIVE SECTION` directive. You must use all the `SPECULATIVE SECTION` directives only in the lexical construct of the code segment that is delimited by the `SPECULATIVE SECTIONS` and `END SPECULATIVE SECTIONS` directives.

This is an example of using the `SPECULATIVE SECTIONS` instruction:

```fortran
INTEGER :: k = 0
INTEGER :: i = 1
...
!SEP$ SPECULATIVE SECTIONS FIRSTPRIVATE(k)
  k = k + 1
!SEP$ SPECULATIVE SECTION
  i = i - 1
!SEP$ END SPECULATIVE SECTIONS
...
END
```

8.7. I/O optimisation

This section briefly presents some general guidelines for efficient I/O on BlueGene/Q. Some helpful and readily available tools are presented in the subsections. A general understanding of the structure of FERMI and JUQUEEN's I/O subsystems and the organisation of their file systems is assumed. They are documented in this guide in Section 2.6 and Section 4.4.

8.7.1. General guidelines

8.7.1.1. Using the right file system

The scratch file system (WORK), accessed by means of the shell environment variable `$WORK`, is the file system of choice for jobs with demanding I/O. On JUQUEEN, direct reading or writing of files on the HOME file system is reasonable only for jobs which structurally use rather small-sized I/O requests - that is, less than 4 MiB per read or write action. On FERMI, the HOME file system is not available on the compute nodes.

8.7.1.2. Adhere to block size and pay attention to block alignment

As with most file systems, I/O operations on WORK as well as on HOME are most efficient when the requested sizes are exact multiples of the file system's block size and are kept well-aligned with file system block boundaries. The block size for the WORK file system is 4 MiB on JUQUEEN and 8 MiB on FERMI. The block size for the HOME file systems is 1 MiB on the two machines. Users who want to avoid hard-coding the block sizes can use the standard POSIX function `stat()`, or `fstat()` in the initialisation phase of their application. The field `st_blksize` of the returned `stat` structure contains the block size of the file system in bytes.

The primary focus of special libraries for parallel I/O, such as parallel HDF5 or parallel netCDF, is on organising output into independent portable platform formats. Doing efficient I/O is to some extent delegated to these libraries by using their routines but it is not their primary concern. These libraries are built on top of MPI I/O. Hints, rather than instructions, can be provided to MPI I/O to increase efficiency; `MPI_INFO` key value pairs can be used to specify desired buffer size, stripe size, collective buffering or data sieving optimisations, etc.
8.7.1.3. Optimise hierarchical I/O schemes by distributing the load over all available I/O nodes

I/O operations that stripe over a larger number of blocks are more efficient because they are able to use more of the underlying storage resources in parallel. However, they are costly in terms of buffer memory needed in user space and may prove difficult to organise on an architecture like the Blue Gene/Q which does not offer a large amount of memory per task. If such a hierarchical scheme, where presumably a subset of the tasks is doing large I/O operations, is implemented on Blue Gene/Q systems, it should be optimised to spread the work over all the I/O nodes available to the job. The IBM MPI implementation contains several Blue Gene/Q specific extensions that were added for this purpose: `MPIX_Pset_same_comm_create()` and `MPIX_Pset_diff_comm_create()` (see Section 5.3.2). Both are collective operations that create a set of communicators, of which each node only sees the one that corresponds to its own place in the topology. The first call creates communicators containing processes which run on nodes associated to the same I/O node. All I/O nodes are used if, for example, every node 0 in these communicators takes the role of a master node for the rest of the nodes. The second call creates a set of orthogonal communicators in which no two members of a given communicator use the same I/O node.

8.7.1.4. Consider not using a hierarchical scheme

The same efficiency that is associated with striping over a large number of file system blocks can, in principle, also be achieved by a large number of parallel tasks engaging in operations on the same file, each on its own fairly small number of blocks. From the perspective of file system organisation, the tasks must be orchestrated to operate on distinct file system blocks rather than operate on arbitrary ranges of a file which overlap in file system block usage. If all tasks are involved, doing I/O themselves, all I/O nodes are involved as well. However, task-local files must be avoided because handling of thousands of individual files can cause a severe performance bottleneck (see Section 8.7.1.5 for further information).

SIONlib (installed only on JUQUEEN, Section 8.7.5) which is a library that is currently being developed at Jülich, may be a helpful tool for automating the organisation of I/O along these lines.

8.7.1.5. Revise I/O of ported applications if their designs assume node local storage

Applications ported from commodity clusters, and other architectures that typically have node local scratch space, often have an I/O organisation in which all task-specific data (intermediate states kept for check pointing, error logs, final output data) are written to task-specific files stored on local storage for scratch. On these architectures, this is indeed a fairly straightforward and efficient way of making use of all the distributed I/O resources that the platform has allocated to the job. On FERMI and JUQUEEN, there is no distributed node local scratch space. Trivial adaptations of the application’s I/O organisation (which simply keep the multitude of task specific files and merely solve possible filename conflicts that can occur because the scratch file system on this platform has a global name space) lead to other severe performance issues. A multitude of directories and/or files must be created in the startup phase of a job, typically under a common root in the file system. This leads to severe “hot spots” in meta-data handling and thus to congestion and severe slowdown for the job, possibly even to slowdown of the I/O of unrelated jobs that experience delay from the busy meta-data servers. Revise such I/O schemes by an alternative that better fits Blue Gene/Q’s architecture; for example, a hierarchical scheme referred to in Section 8.7.1.3, or a non-hierarchical scheme, referred to in Section 8.7.1.4.

8.7.2. MPI I/O

MPI I/O is part of the MPI standard and is intended for dealing with parallel I/O. It offers routines that implement the opening, reading, writing and closing of files as collective actions. It is generic and flexible. Both hierarchical and non-hierarchical schemes can be implemented by means of MPI I/O. It also offers collective and non-collective routines to handle files that have been opened collectively.

MPI I/O introduces its own data types. MPI files are lists of MPI datatypes. Basic types are pre-defined. Derived types can be tailored to suit the application needs. MPI I/O is available on other platforms as well. Usage of MPI I/O however does not automatically tune file system access. The data types of MPI I/O mainly make sense to the application, but are not necessarily well aligned with respect to file system specific parameters.

Parallel versions of netCDF and HDF5 are built on top of MPI I/O to enable parallel file access.
8.7.3. HDF5

The HDF5 library is available on FERMI and JUQUEEN and can be loaded with

```
module load hdf5
```

Further information about how to use it can be obtained with

```
module help hdf5 and news hdf5 (last command only on JUQUEEN)
```

Further information about HDF5 is available on the HDF group website [http://www.hdfgroup.org/HDF5/].

8.7.4. netCDF

Several versions of netCDF are installed on FERMI and JUQUEEN. They can be loaded with

```
module load netcdf
```

Further information about how to use it can be obtained with

```
module help netcdf
```

Further information about netCDF are available here [http://www.unidata.ucar.edu/software/netcdf/].

8.7.5. SIONlib

SIONlib, currently available only on JUQUEEN, is a small library that focuses primarily on tuning massive parallel file access to the underlying storage system. It uses MPI to manage opening and closing of files as collective actions with standard POSIX I/O routines. It does not introduce its own datatypes. Rather, it takes the traditional UNIX view that for the I/O routines, a file is a generic stream of binary data. Therefore, its introduction into existing source code is not very intrusive. By replacing a limited number of I/O calls by SIONlib alternatives that internally take care of the orchestration of block size alignment, the tuning of massive parallel I/O operations to the underlying file system is significantly improved.

Several versions of SIONLib are available on JUGENE. You can choose which one you want to use by means of environment modules. To use the default version, simply enter:

```
module load sionlib
```

To see which versions are available, enter `module avail sionlib` and pick the version that suits you best.

This module command puts SIONlib tools, such as `sioncat`, `siondump`, and `sionsplit`, in your search path. These are for extracting data from a sion file, dumping its meta data, and splitting a sion files into separate ones. But most importantly, this command also puts the `sionconfig` tool into your search path. Use this tool to obtain the correct values for the include files and libraries that you need.

We assume usage of the IBM compiler by means of the `mpixlc_r` C compiler compiler wrapper script. To obtain the correct compiler switches for parallel I/O using SIONlib on JUQUEEN, use the `sionconfig` tool as follows: `sionconfig --be --mpi --cflags`. The option `--be` denotes the BlueGene "Back End" architecture as a target to generate code for, as opposed to the "Front End" (---fe) node architecture of the login nodes. The `sionconfig` tool output (with path details depending on the version) is similar to: `-I/usr/local/sionlib/v1.3p7/include -DBGQ -DSION_MPI -D_SION_BGQ` which can be used in your `Makefile` or build script.

To obtain the correct switches for the linking of objects with libraries use: `sionconfig --be --mpi --libs`. The output (with path details depending on the version) is similar to: `-L/usr/local/sionlib/v1.3p7/lib -lsion_64 -lsionser_64.

The following example would compile the file `mysource1.c` to `mysource1.o` and subsequently link that object file with the SIONlib libraries to produce the executable binary `mybinary1`. 
mpixlc_r -I/usr/local/sionlib/v1.3p7/include -DBGQ -DSION_MPI
   -D_SION_BGQ -c mysource1.c
mpixlc_r -o mybinary1 mysource1.o -L/usr/local/sionlib/v1.3p7/lib
   -lsion_64 -lsionser_64

Further information about SIONlib is available on the SIONlib website [http://www.fz-juelich.de/jsc/sionlib/].

## Further documentation


