HPC

BAS5 for Xeon

Application Tuning Guide

REFERENCE
86 A2 23FA 00

HPC

BAS5 for Xeon

Application Tuning Guide

Software

March 2009

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

REFERENCE
86 A2 23FA 00

The following copyright notice protects this book under Copyright laws which prohibit such actions as, but not limited to, copying, distributing, modifying, and making derivative works.

Copyright © Bull SAS 2009

Printed in France

Trademarks and Acknowledgements

We acknowledge the rights of the proprietors of the trademarks mentioned in this manual.

All brand names and software and hardware product names are subject to trademark and/or patent protection.

Quoting of brand and product names is for information purposes only and does not represent trademark misuse.

The information in this document is subject to change without notice. Bull will not be liable for errors contained herein, or for incidental or consequential damages in connection with the use of this material.

Preface

Scope and Objectives

The purpose of this guide is to describe the use of the tools which enable application program optimization for Bull Advanced Server (**BAS**) for Xeon High Performance Clusters.

Intended Readers

This guide is for application programmers who wish to tune and optimize their code so that it fully exploits all the processing power available.

Prerequisites

The installation of all the hardware and software components of the HPC system must have been completed.

Structure

This guide is organized as follows:

Chapter 1. Looks at the *Performance Monitoring and Application Profiling Tools* used to identify areas where application program performance could be improved.

Chapter 2. *Coding and Compiling Optimization.* Looks at some coding tips and compiling options to help improve the performance of your application on the Bull HPC platform. Guidelines are given in order to ensure that the application program runs as efficiently as is possible.

Chapter 3. *Program Execution Optimization.* Describes how to optimize and launch your program.

Chapter 4. *Message Passing Interface Optimization.* Looks at some optimization tips for the Message Passing Interface(MPI).

Chapter 5. *Lustre File System Optimization.* Describes how the Lustre (CFS) parallel file system should be optimized.

Appendix A Describes *Amdahl's Law.*

Bibliography

Refer to the manuals included on the documentation CD delivered with you system OR download the latest manuals for your Bull Advanced Server **(BAS)** release, and for your cluster hardware, from: http://support.bull.com/

The Bull *BAS5 for Xeon Documentation* CD-ROM (86 A2 12FB) includes the following manuals:

- Bull *HPC BAS5 for Xeon Installation and Configuration Guide* (86 A2 19FA)
- Bull *HPC BAS5 for Xeon Administrator's Guide* (86 A2 20FA)
- Bull *HPC BAS5 for Xeon User's Guide* (86 A2 22FA)

- Bull *HPC BAS5 for Xeon Maintenance Guide* (86 A2 24FA)
- Bull *HPC BAS5 for Xeon Application Tuning Guide* (86 A2 23FA)
- Bull *HPC BAS5 for Xeon High Availability Guide* (86 A2 25FA)

The following document is delivered separately:

- The *Software Release Bulletin* (SRB) (86 A2 68EJ)

mportant

The Software Release Bulletin contains the latest information for your BAS delivery. This should be read first. Contact your support representative for more information.

In addition, refer to the following:

- Bull *Voltaire Switches Documentation CD* (86 A2 79ET)
- *Bull System Manager* documentation

For clusters which use the **PBS Professional** Batch Manager:

- PBS Professional 10.0 *Administrator's Guide (*on the *PBS Professional CD-ROM)*
- PBS Professional 10.0 User's Guide (on the PBS Professional CD-ROM)

For clusters which use LSF:

- *LSF Installation and Configuration Guide* (86 A2 39FB) *(*on the LSF CD-ROM*)*
- *Installing Platform LSF on UNIX and Linux (*on the LSF CD-ROM*)*

For clusters which include the Bull Cool Cabinet:

- Site Preparation Guide (86 A1 40FA)
- R@ck'nRoll & R@ck-to-Build Installation and Service Guide (86 A1 17FA)
- Cool Cabinet Installation Guide (86 A1 20EV)
- Cool Cabinet Console User's Guide (86 A1 41FA)
- Cool Cabinet Service Guide (86 A7 42FA)

Web links

http://www.linuxhpc.org/

Highlighting

- Commands entered by the user are in a frame in '`Courier`' font, as shown below:

```
mkdir /var/lib/newdir
```

- System messages displayed on the screen are in '`Courier New`' font between 2 dotted lines, as shown below.

```
Enter the number for the path :
```

- Values to be entered in by the user are in ‘Courier New’, for example:

  COM1

- Commands, files, directories and other items whose names are predefined by the system are in ‘**Bold**’, as shown below:

  The **/etc/sysconfig/**dump file.

- The use of *Italics* identifies publications, chapters, sections, figures, and tables that are referenced.

- < > identifies parameters to be supplied by the user, for example:

  <node_name>

WARNING
A Warning notice indicates an action that could cause damage to a program, device, system, or data.

Table of Contents

List of Figures

Chapter 1. Performance Monitoring and Application Tools

This chapter looks at the Performance Monitoring and Application Profiling Tools to be used to identify areas where application program performance could be improved.

The following topics are described:

- 1.1 *Tools for Optimizing HPC Performance*
- 1.2 *System Monitoring Tools*
- 1.3 *Ganglia Cluster Performance Monitoring*
- 1.4 *IOstat*
- 1.5 *dstat*
- 1.5 *mpianalyser and profilecomm*
- 1.7 *PAPI*
- 1.8 *Profiling Programs – HPC Toolkit*
- 1.9 *Intel® VTune™ Performance Analyzer for Linux*

1.1 Tools for Optimizing HPC Performance

What will interest the user who wants to improve performance, is to optimize the use of all the resources of the system and to track down any possible bottlenecks resulting from the development, compilation and execution of individual parts, or from the whole application program. Various tools are available to help the user in these tasks.

The first measurement step is to determine the run-time for a specific aspect of the program. The **time** command is used to measure this.

Secondly, it is important to examine the factors which determined this time. Which resources were used and for how long? Can saturated resources be identified or, alternatively, those which are underutilized.

To do this different methods exist, according to the type of program that is being analyzed and also according to the objectives of the user. For example, is the goal to optimize the behavior of the machine for a given program (benchmark), or is it to improve the operation of the program itself on a particular machine or network?

If there is no Input/Output problem, then the quality of algorithms should be analyzed using a profiling approach which focuses on the parts of the program which consume most system resources.

If a program uses a **MPI** (Message Passing Interface) code, each process can be analyzed separately.

If the objective is to optimize a program, the level of detail provided by these tools is generally enough. On the other hand, if more information about the machine is needed, more hardware-oriented tools which provide good metrics will have to be used.

Note	Intel® Trace Tools (Trace Analyzer and Trace Collector) and Intel® Vtune™ Peformance Analyzer are proprietary software available from Intel.

The tools referred to in this chapter should be used in the sequence indicated above to determine where the performance of the application could be improved.

If the need is to focus on the performance of the parallel applications that use the MPI (Message Passing Interface) then either profilecomm or proprietary software such as Intel Trace Analyzer / Collector tools can be used. These tools display trace information graphically.

Intel Vtune is used to perform post mortem analysis of the output after the application has completed its execution and they cannot be used during run-time. HPC Toolkit, an open source tool based on PAPI, is included in the BAS5 for Xeon delivery. This profiles the application in a similar way as Intel Vtune.

1.2 System Monitoring Tools

1.2.1 Time

The first determinant to find is the run-time for a specific operation; this will be used as a yardstick in the optimization process. Different benchmark operations, similar to those defined in the call to tender, can be used.

The time command is used to measure the duration of execution for a particular operation. The execution time is reported in terms of user CPU time, system CPU time, and real time.

The etime function is used to give the time of execution for a particular part of the application program.

1.3 Ganglia Cluster Performance Monitoring

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as those used in Bull HPC systems. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies including **XML** for data representation, **XDR** for compact, portable data transport, and **RRDtool (Round Robin Database tool)** for data storage and visualization. It uses carefully engineered data structures and algorithms to achieve a very low per-node overhead and high concurrency.

Bull System Manager – HPC Edition uses a GUI to display the Ganglia data for the hardware system. This can be used to monitor the performance of the systems and detect any variations within it.

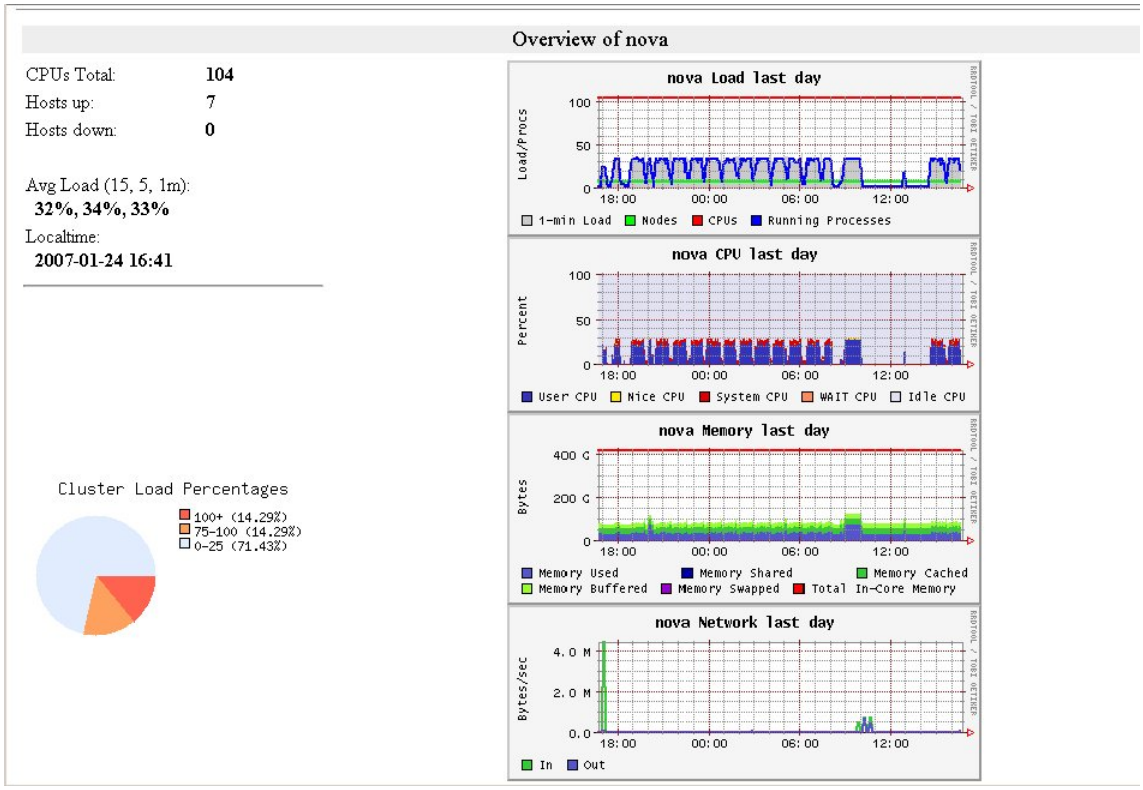

Figure 1-1. Ganglia overview of a Cluster

The parameters which enable the calculation of the performance of the cluster are collected on all nodes by **Ganglia**. The results may be viewed within **Bull System Manager - HPC Edition** by clicking on the Group Performance or Global Performance button.

Different categories of data are collected, including the following:

- Processors
- Memory
- Disks
- Network (admin)
- Interconnect
- Lustre (for systems which use the Lustre file system)

1.3.1 Group Performance Global View

This view displays diagrams of difference performance metrics for a selected set of nodes. Each diagram shows the evolution of the metric concerned over a user-defined period of time.

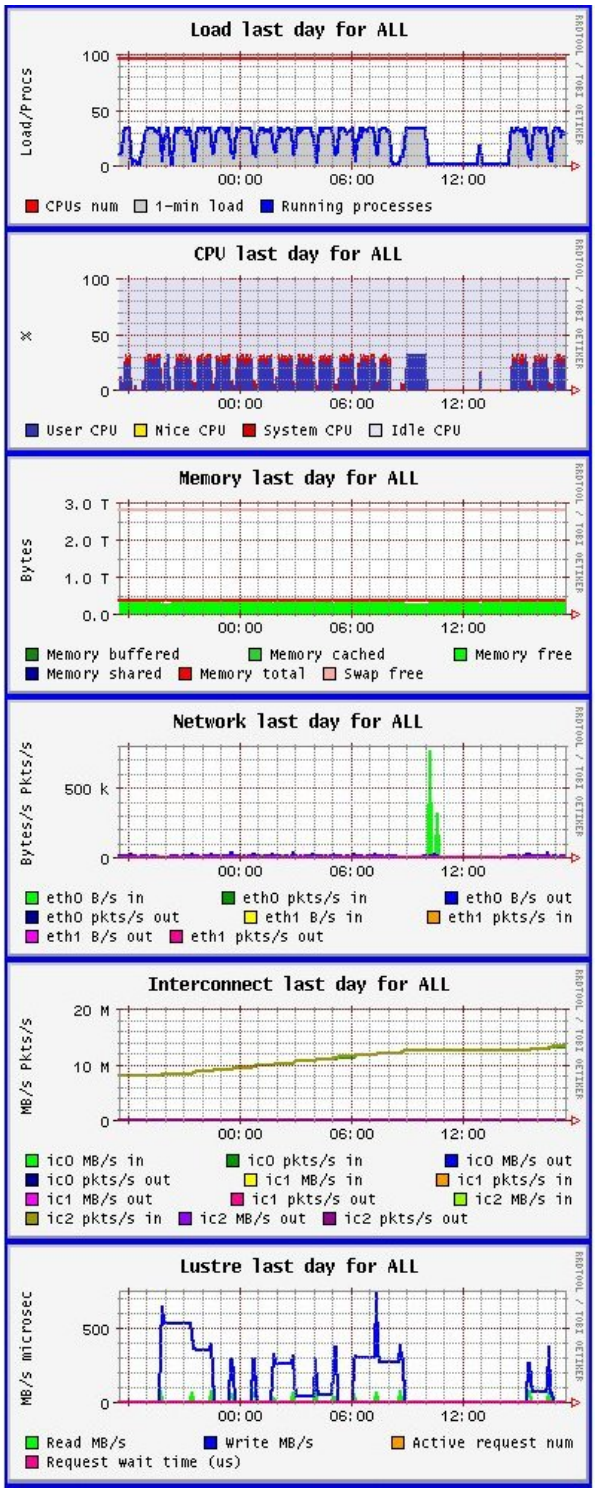

Figure 1-2. Ganglia Group Performance Global view

Clicking on a diagram will display graphs with more detailed information.

1.3.2 Detailed Cluster Performance View

This view displays the global performance diagram for each type of metrics and the diagrams for the ten first nodes which are sorted in ascending or descending order of the metric value.

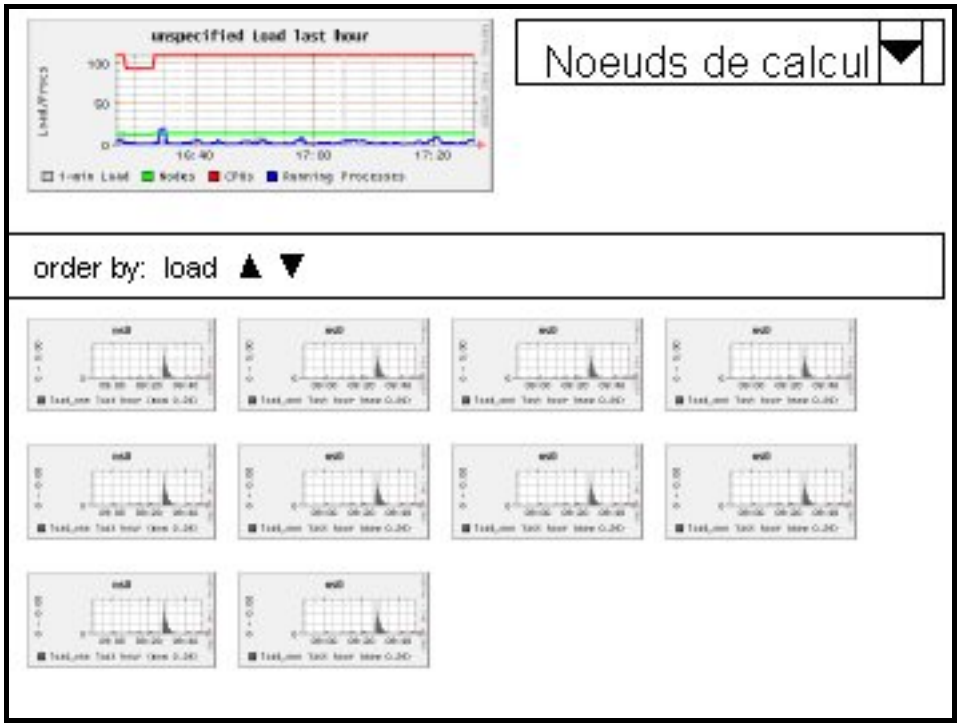

Figure 1-3. Ganglia detailed performance view

Using the monitoring data it is possible to identify areas where performance is being lost and as a result to make changes to the system or the application, and then verify the changes to see if the performance has improved.

1.4 IOstat

The **iostat** Linux command is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates. The **iostat** command generates reports that can be used to change a system's configuration to better balance the input/output load between physical disks.

Performance problems may be the result of too many files being repeatedly opened, read and written to, and then closed. This type of problem is indicated by increasing seek times and may be identified using **iostat.**

The first report generated by the **iostat** command provides statistics for the time elapsed since the system was first booted. Each subsequent report covers the period of time since the previous report. The interval parameter stipulates the time period in seconds for each report.

The count parameter may be used with the interval parameter. These determine the number of reports generated and the time period for each report. If the interval parameter is used without the count parameter, the **iostat** command generates reports continuously.

All I/O statistics are collected each time the **iostat** command runs. The report consists of a CPU header row followed by a row of CPU statistics. On multiprocessor systems, CPU statistics are calculated system-wide as averages among all processors. A device header row is displayed followed by a line of statistics for each device that is configured.

The **iostat** command generates two types of reports, the CPU Utilization report and the Device Utilization report.

- On multiprocessor systems the CPU Utilization Report provides the CPU values which are global averages for all processors.
- The Device Utilization Report provides statistics either by physical device or by partition.

Examples

The following command displays four reports of extended statistics at two second intervals.

```
iostat -x 2 4
```

The following command displays six reports of extended statistics at two second intervals for devices **hda** and **hdb**.

```
iostat -x hda hdb 2 6
```

For more information on the formats of the reports and the commands which are available refer to the man page for **iostat**, alternatively look at http://linuxcommand.org/

1.5 dstat

dstat overcomes some of the limitations of iostat. The **dstat** command can be used to monitor systems during performance tuning tests, benchmarks, or troubleshooting. This command allows you to view all of your system resources instantly. You can compare disk usage in combination with interrupts from IDE controllers, or compare the network bandwidth numbers directly with the disk throughput (in the same interval).

dstat allows you to aggregate block device throughput for a certain disk set or network set, so that you can see the throughput for all the block devices that make up a single file system or storage system.

By default dstat's output is viewed in real-time, the data being displayed in coloured columns. However, it can also be saved in a file in a **CSV** format that can be imported into **Gnumeric** or **Excel** so that the data can be viewed graphically. The counters can be configured so that they appear in the order that makes the most sense for your cluster.

1.5.1 dstat Plugins

Dstat includes external plugins for dedicated counters. It is open source and written in **python** allowing new specific counters to be developed for your cluster. The plugins include the following:

dstat_app	The most expensive process on the system
dstat_freespace	See the disk usage per partition
dstat_nfs3	The NFS v3 client operations
dstat_nfsd3	The NFS v3 server operations
dstat_postfix	Counters of the different queues (needs postfix)
dstat_thermal	CPU temperature

1.5.2 dstat performance impact

Before running any tests check what impact **dstat** in terms of resource usage. Use the **-t** option together with the **–debug** option to examine performance time variations, according to whether or not a plugin is loaded. If the impact is higher than expected, then reduce the number of stats or remove the expensive stats.

See http://dag.wieers.com/home-made/dstat/ for more information

1.6 mpianalyser and profilecomm

mpianalyser is an integrated framework which uses the **PMPI** interface to analyze the behaviour of MPI programs.

Profilecomm is a part of **mpianalyser** and is dedicated to MPI application profiling. It has been designed to be:

- Light: it uses few resources and so does not slow down the application.
- Easy to run: it is used to characterize the MPI communications in a program. Communication matrices are constructed with it. **Profilecomm** is a "post-mortem" tool, which does not allow on-line monitoring.

Data is collected as long as the program is running. At the end of the program, data is written into a file for future analysis.

readpfc is a tool with a command line interface which handles the data that has been collected. Its main uses are the following:

- To display the data collected.
- To export communication matrices in a format that can be used by other applications.

Data collected

The **profilecomm** module provides the following information:

- Communication matrices
- Execution time
- Table of calls of MPI functions
- Message size histograms
- Topology of the execution environment.

1.6.1 Communication Matrices

The **profilecomm** library collects separately the point to point communications and the collective communications. It also collects the number of messages and the volume that the sender and receiver have exchanged. Finally, the library builds 4 types of communication matrices:

- Communication matrix of the number of point to point messages
- Communication matrix of the volume (in bytes) of point to point messages
- Communication matrix of the number of collective messages
- Communication matrix of the volume (in bytes) of collective messages

The volume only indicates the payload of the messages.

In order to compute the standard deviation of messages size, two other matrices are collected. They contain the sum of squared messages sizes for **point-to-point** and for collective communications.

In order to get more precise information about messages sizes, each numeric matrix can be split into several matrices according to the size of the messages. The number of partitions and the size limits can be defined through the **PFC_PARTITIONS** environment variable. In a point to point communication, the sender and receiver of each message is clearly identified, this results in a well defined position in the communication matrix.

In a collective communication, the initial sender(s) and final receiver(s) are identified, but the path of the message is unknown. The **profilecomm** library disregards the real path of the messages. A collective communication is shown as a set of messages sent directly by the initial sender(s) to the final receiver(s).

Execution Time

The measured execution time is the maximum time interval between the calls to **MPI_Init** and **MPI_Finalize** for all the processes. By default the processes are synchronized during the measurements. However, if necessary, the synchronization may be by-passed using an option of the **profilecomm** library.

Call Table

The call table contains the number of calls for each profiled function of each process. For collective communications, since a call generates an unknown number of messages, the values indicated in the call table do not correspond to the number of messages.

Histograms

Profilecomm collects two messages size histograms, one for point-to-point and one for collective communications. Each histogram contains the number of messages for sizes 0, 1 to 9, 10 to 99, 100 to 999, ..., 10^8 to 10^9-1 and bigger than 10^9 bytes.

Topology of the Execution Environment

The **profilecomm** module registers the topology of the execution environment, so that the machine and the CPU on which each process is running can be identified, and above all the intra- and inter-machine communications made visible.

1.6.2 Profilecomm Data Collection

When using **profilecomm** there are 2 separate operations – data collection, and then its analysis.

Using Profilecomm

To be profiled by **profilecomm**, an application must be linked with the **MPI Analyser** library.

profilecomm is disabled by default, to enable it, set the following environment variable:

```
export MPIANALYSER_PROFILECOMM=1
```

When the application finishes, the results of the data collection are written to a file (**mpiprofile.pfc** by default). By default this file is saved in a format specific to **profilecomm**, but it is possible to save it in a text format. The **readpfc** command enables **.pfc** files to be read and analysed.

1.6.3 Profilecomm Options

Different options may be specified for **profilecomm** using the **MPIPROFILE_OPTIONS** environment variable.

For example:

```
export MPIPROFILE_OPTIONS="-f foo.pfc"
```

Some of the options that modify the behavior of **profilecomm** when saving the results in a file are below:

-f *file*, -filename *file*
Saves the result in the **file** file instead of the default file (**mpiprofile.txt** for text format files and **mpiprofile.pfc** for **profilecomm** binary format files).

-t, -text
Saves the result in a text format file, readable with any text editor or reader. This format is useful for debugging purpose but it is not easy to use beyond 10 processes.

-b, -bin
Saves the results in a **profilecomm** binary format file. This is the default format. The **readpfc** command is required to work with these files.

-s, -sync
Synchronizes the processes during the time measurements. This option is set by default.

-ns, -nosync
Doesn't synchronize the processes during the time measurements.

-v32, -volumic32
Use 32 bit volumic matrices. This can save memory when profiling application with a large number of processes. A process must not send more than 4GBs of data to another process.

-v64, -volumic64
Use 64 bits volumic matrices. This is the default behavior. It allows the profiling of processes which exchanges more than 4GBs of data.

Examples

To profile the **foo** program and save the results of the data collection in the default file **mpiprofile.pfc**:

```
$ MPIPROFILE=profilecomm prun –p my_partion –N 1 -n 4./foo
```

To save the results of the data collection in the `foo.pfc` file:

```
$ MPIPROFILE=profilecomm MPIPROFILE_OPTIONS="-f foo.pfc" prun –p
my_partion –N 1 -n 4./foo
```

To save the result of the collect in text format in the `foo.txt` file:

```
$ MPIPROFILE=profilecomm MPIPROFILE_OPTIONS="-t -f foo.txt" prun –p
my_partion –N 1 -n 4./foo
```

1.6.4 Messages Size Partitions

Profilecomm allows the numeric matrices to be split according to the size of the messages. This feature is activated by setting the PFC_PARTITIONS environment variable. By default, there is only one partition, i.e. the numeric matrices are not split.

The **PFC_PARTITIONS** environment variable must be of the form **[partitions:] [limits]** in which **partitions** represents the number of partitions and **limits** is a comma separated list of sorted numbers representing the size limits in bytes.

If **limits** is not set, **profilecomm** uses the built-in default limits for the requested partition number.

Example 1 - 3 partitions using the default limits (1000, 1000000):

```
$ export PFC_PARTITIONS="3:"
```

Example 2 - 3 partitions using user defined limits (in this case, the partition number can be safely omitted):

```
$ export PFC_PARTITIONS="3:500,1000"
```

Or

```
$ export PFC_PARTITIONS="500,1000"
```

1.6.5 Profilecomm Data Analysis

To analyze data collected with **profilecomm** the **readpfc** command and other tools, including spreadsheets, can be used. The main features of **readpfc** are the following:

- Displaying the data contained in **profilecomm** files.
- Exporting communication matrices in a standard file format.

readpfc syntax

```
readpfc [options] [file]
```

If `file` is not specified, **readpfc** reads the default file **mpiprofile.pfc** in the current directory.

Readpfc output

The main feature of **readpfc** is to display the information contained in the seven different sections of a **profilecomm** file. These are:

- Header
- Point to point
- Collective
- Call table
- Histograms
- Statistics
- Topology.

Header Section:

Displays information contained into the header of a **profilecomm** file. The more interesting fields are:

- **Elapsed Time** – indicates the length of the data collection.
- **World size** - indicates the number of processes.
- **Number of partitions** – indicates the number of partitions.
- **Partitions limits** – indicates the list of size limits for the messages partitions (only used if there are several partitions).

The other fields are less interesting for the final users but are used internally by **readpfc**.

Example:

```
Header:
 Version: 2
 Flags: little-endian
 Header size: 40 bytes
 Elapsed time: 9303 us
 World size: 4
 Number of partitions: 3
 Partitions limits:  1000  1000000
 num_intsz: 4 bytes (32 bits)
 num_volsz: 8 bytes (64 bits)
```

1.6.6 Point to Point Communications

- For point to point communication matrices, use the following. The number of communication messages is displayed first, then the volume. If either the
- **--numeric-only** or **--volumic-only** options are used then only one matrix is displayed accordingly.

Example:

```
Point to point:
 numeric (number of messages)
      0   1.1k      0      0 |   1.1k
   1.1k      0      0      0 |   1.1k
      0      0      0   1.1k |   1.1k
      0      0   1.1k      0 |   1.1k

 volumic (Bytes)
      0 818.8k      0      0 | 818.8k
 818.8k      0      0      0 | 818.8k
      0      0      0 818.8k | 818.8k
      0      0 818.8k      0 | 818.8k
```

If the file contains several partitions and the `-J/--split` option is set then this command displays as many numeric matrices as there are partitions. Example:

```
Point to point:
 numeric (number of messages)
 0 <= msg size < 1000
      0    800      0      0 |    800
    800      0      0      0 |    800
```

```
      0      0      0    800 |    800
      0      0    800      0 |    800

 1000 <= msg size < 1000000
      0    300      0      0 |    300
    300      0      0      0 |    300
      0      0      0    300 |    300
      0      0    300      0 |    300

 1000000 <= msg size
      0      0      0      0 |      0
      0      0      0      0 |      0
      0      0      0      0 |      0
      0      0      0      0 |      0

 volumic (Bytes)
      0 818.8k      0      0 | 818.8k
 818.8k      0      0      0 | 818.8k
      0      0      0 818.8k | 818.8k
      0      0 818.8k      0 | 818.8k
```

If the **-r/--**rate option is set then the messages rate and data rate matrices are shown instead of communications matrices. These rates are the average rates for all execution times not the instantaneous rates. Example:

```
Point to point:
 message rate (msg/s)
      0 118.2k      0      0 | 118.2k
 118.2k      0      0      0 | 118.2k
      0      0      0 118.2k | 118.2k
      0      0 118.2k      0 | 118.2k

 data rate (Bytes/s)
      0 88.01M      0      0 | 88.01M
 88.01M      0      0      0 | 88.01M
      0      0      0 88.01M | 88.01M
      0      0 88.01M      0 | 88.01M
```

1.6.7 Collective Section

The collective section is equivalent to the point to point section for collective communication matrices. Example:

```
Collective:
 numeric (number of messages)
      0    102    202    102 |    406
    102      0      0    100 |    202
    202      0      0      0 |    202
    102    100      0      0 |    202

 volumic (Bytes)
      0 409.6k 421.6k 409.6k | 1.241M
 12.04k      0      0    12k | 24.04k
 421.6k      0      0      0 | 421.6k
 12.04k 409.6k      0      0 | 421.6k
```

1.6.8 Call table section

This section contains the call table. If the `--ct-total-only` option is activated, only the total column is displayed. Example:

```
Call table:
                     0      1      2      3      4      5      6      7  Total
Allgather            0      0      0      0      0      0      0      0      0
Allgatherv           0      0      0      0      0      0      0      0      0
Allreduce            2      2      2      2      2      2      2      2     16
Alltoall             0      0      0      0      0      0      0      0      0
Alltoallv            0      0      0      0      0      0      0      0      0
Bcast              200    200    200    200    200    200    200    200   1.6k
Bsend                0      0      0      0      0      0      0      0      0
Gather               0      0      0      0      0      0      0      0      0
Gatherv              0      0      0      0      0      0      0      0      0
Ibsend               0      0      0      0      0      0      0      0      0
Irsend               0      0      0      0      0      0      0      0      0
Isend                0      0      0      0      0      0      0      0      0
Issend               0      0      0      0      0      0      0      0      0
Reduce             200    200    200    200    200    200    200    200   1.6k
Reduce_scatter       0      0      0      0      0      0      0      0      0
Rsend                0      0      0      0      0      0      0      0      0
Scan                 0      0      0      0      0      0      0      0      0
Scatter              0      0      0      0      0      0      0      0      0
Scatterv             0      0      0      0      0      0      0      0      0
Send              1.1k   1.1k   1.1k   1.1k   1.1k   1.1k   1.1k   1.1k   8.8k
Sendrecv             0      0      0      0      0      0      0      0      0
Sendrecv_replace     0      0      0      0      0      0      0      0      0
Ssend                0      0      0      0      0      0      0      0      0
Start                0      0      0      0      0      0      0      0      0
```

1.6.9 Histograms Section

This section contains the message sizes histograms. It shows the number of messages whose size is zero, between 1 and 9, between 10 and 99, ..., between 10^8 and 10^9-1 and greater than 10^9.

Example:

```
Histograms of msg sizes
size   pt2pt   coll  total
   0       0      0      0
   1     800      6    806
  10    1.2k      6 1.206k
 100    1.2k    500   1.7k
1000    1.2k    500   1.7k
104        0      0      0
105        0      0      0
106        0      0      0
107        0      0      0
108        0      0      0
109        0      0      0
```

1.6.10 Statistics Section

This section displays statistics computed by **readpfc**. These statistics are based on the information contained in the data collection file. This section is divided into two or three sub-sections:

- The *General statistics* section contains statistics for the whole application.
- The *Per process average* section contains average per process.
- The *Messages sizes partitions* section displays the distribution of messages among the partitions. This section is only present if there are several partitions.
- For each statistic we distinguish point to point communications from collective communications.

Example:

```
General statistics:
Total time: 0.009303s  (0:00:00.009303)
                        pt2pt |       coll |      total
Messages count  |        4400 |       1012 |       5412
Volume          |    3.2752MB |  2.10822MB |  5.38342MB
Avg message size|       744B |  2.08322kB |       995B
Std deviation   |     1216.4 |     1989.1 |     1488.4
Variation coef. |     1.6341 |    0.95481 |     1.4963
Frequency msg/s |    472.966k |   108.782k |   581.748k
Throughput B/s  | 352.06MB/s | 226.62MB/s | 578.68MB/s

Per process average:
                        pt2pt |       coll |      total
Messages count  |        1100 |        253 |       1353
Volume          |     818.8kB |  527.054kB |  1.34585MB
Frequency msg/s |    118.241k |   27.1955k |   145.437k
Throughput B/s  | 88.015MB/s | 56.654MB/s | 144.67MB/s

Messages sizes partitions:
                                |   pt2pt count |    coll count  |   total
count
         0 <= sz < 1000         |  3.2e+03  73% |  5.1e+02  51% |  3.7e+03  69%
      1000 <= sz < 1000000      |  1.2e+03  27% |    5e+02  49% |  1.7e+03  31%
   1000000 <= sz                |         0   0% |         0   0% |         0   0%
```

The message sizes partitions should be examined first.

Where:

Total time	Total execution time between MPI_Init and MPI_Finalize.
Messages count	Number of sent messages.
Volume	Volume of sent messages (bytes).
Avg message size	Average size of messages (bytes).
Std deviation	Standard deviation of messages size.
Variation coef.	Variation coefficient of messages size.
Frequency msg/s	Average frequency of messages (messages per second).
Throughput B/s	Average throughput for sent messages (bytes per second).

1.6.11 Topology Section

This section shows the distribution of processes on nodes and processors. This distribution is displayed in two different ways.

- First, for each process the node and the CPU in the node where it is running and secondly, the list of running processes for each node.

Example- 8 processes running on 2 nodes.

```
Topology:
8 process on 2 hosts
process hostid  cpuid
      0      0      0
      1      0      1
      2      0      2
      3      0      3
      4      1      0
      5      1      1
      6      1      2
      7      1      3

host  processes
   0  0 1 2 3
   1  4 5 6 7
```

1.6.12 Display Options

The following options allow different information to be displayed:

-a, --all
Displays all the information. Equivalent to **–ghimst**.

-c, --collective
Displays collective communication matrices.

-g, --topology
Displays the topology of execution environment.

-h, --header
Displays header of the **profilecomm** file.

-i, --histograms
Displays messages size histograms.

-j, --joined
Displays entire numerics matrices (i.e. not split). This is the default.

-J, --splitted
Display numerics matrices split according to messages size.

-m, --matrix, --matrices
Displays communication matrix (matrices). Equivalent to **–cp**.

-n, --numeric-only
Does not display volume matrices. This option cannot be used simultaneously with the **-v/--volumic-only** option.

-p, --p2p, --pt2pt
Displays point to point communication matrices.

-r, --rate, --throughput
Displays messages rate and data rate matrices instead of communications matrices.

-s, --statistics
Computes and displays some statistics regarding MPI communications.

-S, --scalable
Displays all scalable information; this means all information whose size is independent of number of processes. Useful when there is a great number of processes. Equivalent to **histT.**

--square-matrices
Displays the matrices containing the sum of the squared sizes of messages. These matrices are used for standard deviation computation and are useless for final users. This option is mainly provided for debugging purposes.

-t, --calltable
Displays the call table.

-T, --ct-total-only
Displays only the *Total* column of the call table. By default **readpfc** displays also one column for each process.

-v, --volumic-only
Does not display numeric matrices. This option cannot be used simultaneously with **-n/--numeric-only** option.

1.6.13 Exporting a Matrix or an Histogram

The communication matrices and the histograms can be exported in different formats which can be understood by other software programs, for example spreadsheets. Three formats are available: **CSV** (Comma Separated Values), **MatrixMarket** (not available for histogram exports) and **gnuplo**t.

It is also possible to have a graphical display of the matrix or the histogram, which is better for matrices with a large number of elements. Obviously, it is also possible to include the graphics in a report. Seven graphic formats are available: PostScript, Encapsulated PostScript, SVG, xfig, EPSLaTeX, PSLaTeX and PSTeX. All these formats are vectorial, which means the dimensions of the graphics can be modified if necessary.

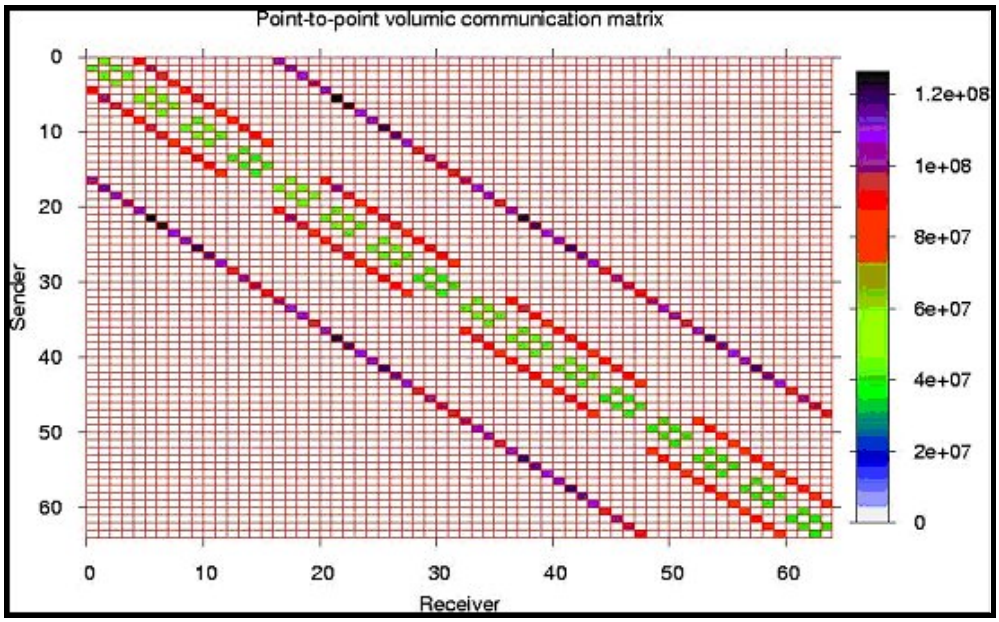

Figure 1-4. An example of a communication matrix

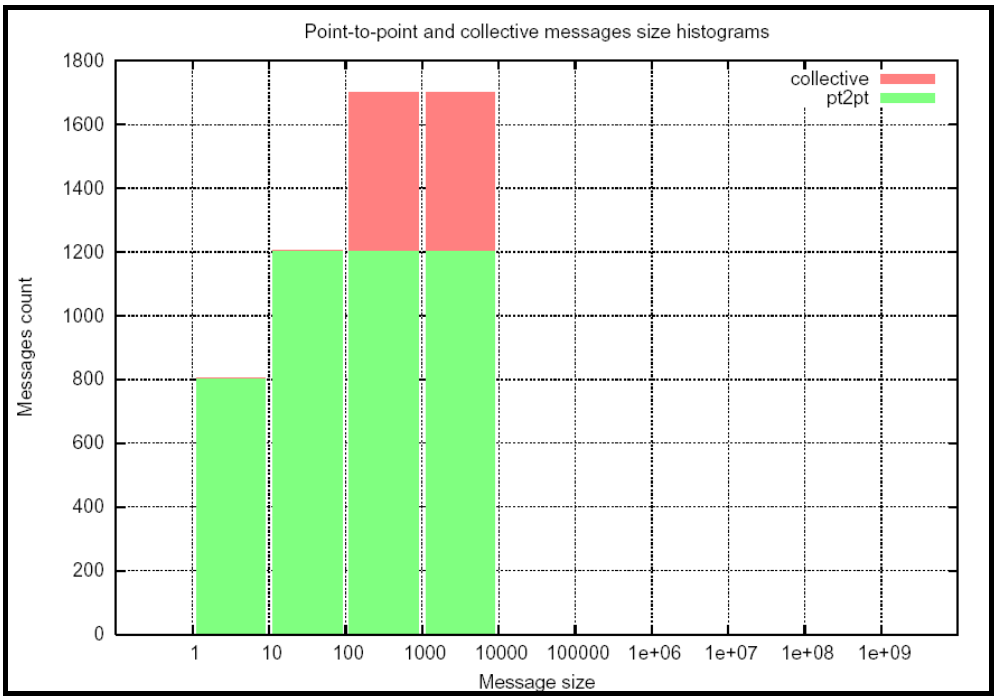

Figure 1-5. An example of a histogram

The following options may be used when exporting matrices:

--csv-separator sep	Modifies CSV delimiter. Default delimiter is comma ",". Some software programs prefer a semicolon ";".
-f format, --format *format*	Chooses export format. Default format is CSV (Comma Separated Values).
help	lists available export formats
csv	export in CSV format
mm, market, MatrixMarket	export in MatrixMarket format

gp, gnuplot	export in a format used by pfcplot so that a graphical display of the matrix can be produced
ps, postscript	export in PostScript format
eps	export in Encapsulated PostScript format
svg	export in Scalable Vector Graphics format
fig, xfig	export in xfig format
epslatex	export in LaTex and Encapsulated PostScript format
pslatex	export in LaTex format and PostScript inline
pstex	export in Tex format and PostScript inline

The available values are the following:

When using epslatex two files are written: xxx.tex and xx.eps. The filename indicated in the –o option is the name of the Latex file.

--logscale[=base]
Uses a logarithmic color scale. Default value for logarithm basis is 10; this basis can be modified using the **base** argument. This option is only relevant when exporting in a graphical format.

--nogrid
Does not display the grid on a graphical representation of the matrix.

-o file, --output *file*
Specifies the file name for an export file. The default filenames are **out.csv**, **out.mm, out.dat, out.ps, out.svg, out.fig** or **out.tex**, according to export format. This option is only available with the –x option.

--palette *pal*
Uses a personalized colored palette. This option is only relevant when exporting in a graphical format. This palette must be compatible with the **`defined`** function of gnuplot, for instance: `--palette '0 "white", 1 "red", 2 "black"'` or `--palette '0 "#0000ff", 1 "#ffff00", 2 "ff0000"'`

--title *title*
Uses a personalized title for a graphical display. The default title is *Point-to-point/collective numeric/volumic communication* matrix, according to the exported matrix.

-x object, --*export object*
Exports a communication matrix or histogram specified by the ***`object`*** argument. Values for **object** are the following:

help	List of available matrices and histograms
pn[.part], np[.part]	Point-to-point numeric communication matrix. The optional item part is the partition number for split matrices. If part is not set, the entire matrix (i.e.the sum of the split matrices) is exported
pv, vp	Point to point volumic communication matrix
cn[.part], nc[.part]	Collective numeric communication matrix
cv, vc	Collective volumic communication matrix
ph, hp	Point-to-point messages size histogram
ch, hc	Collective messages size histogram
th, ht	Total messages size histogram (collective and point-to-point)
ah, ha	Both point-to-point and collective messages size histograms (all histograms)

Other options

-H, --help, --usage
Displays help messages

-q, --quiet
Does not display help warning messages (error messages continue to be displayed).

-V, --version
Displays program version.

Examples

- To display all information available in foo.pfc file, enter:

```
$ readpfc -a foo.pfc
```

This will give information similar to that below

```
Header:
 Version: 2
 Flags: little-endian
 Header size: 40 bytes
 Elapsed time: 9303 us
 World size: 4
 Number of partitions: 3
 Partitions limits:  1000  1000000
 num_intsz: 4 bytes (32 bits)
 num_volsz: 8 bytes (64 bits)
[...]
Topology:
4 process on 1 hosts
process hostid  cpuid
      0      0      0
      1      0      1
      2      0      2
      3      0      3

host  processes
   0  0 1 2 3
```

- To display a point to point numerical communication matrix:

```
$ readpfc -pn foo.pfc
```

```
Point to point:
 numeric (number of messages)
      0   1.1k      0      0 |   1.1k
   1.1k      0      0      0 |   1.1k
      0      0      0   1.1k |   1.1k
      0      0   1.1k      0 |   1.1k
```

- To export the collective volumic communication matrix in CSV format in the default file:

```
$ readpfc –x cv foo.pfc
```

```
Warning: No output file specified, write to default (out.csv).
```

```
$ ls out.csv
```

```
out.csv
```

- To export the first part (small messages) of point to point numerical communication matrices in PostScript format in the foo.ps file:

```
$ readpfc -x np.o -f ps -o foo.ps foo.pfc
$ ls foo.ps
```

```
foo.ps
```

1.6.14 pfcplot, histplot and gnuplot

The **pfcplot** script converts matrices into graphic using **gnuplot** . It is generally used by **readpfc** , but can be used directly by the user who wants more flexibility. The matrix must be exported with the **–f gnuplot** option to be read by **pfcplot**.

For more details enter:

```
man pfcplot
```

Users who have particular requirements can invoke **gnuplot** directly. To do this the matrix must be exported with **gnuplot** format or with CSV format, choosing space as the separator.

Important

Due to the limitations of gnuplot, one null line and one null column are added to the exported matrix in gnuplot format.

Histplot is the equivalent of **pfcplot** for histograms. Like **pfcplot**, it can be used directly by users but it is not user-friendly. More details are available from the man page:

```
man histplot
```

1.7 PAPI

PAPI (Performance API) is used for the following reasons:

- To provide a solid foundation for cross-platform performance analysis tools,
- To present a set of standard definitions for performance metrics on all platforms,
- To provide a standard API among users, vendors and academics.

PAPI supplies two interfaces:

- A high-level interface, for simple measurements,
- A low-level interface, programmable, adaptable to specific machines and linking the measurements.

PAPI should only be used by specialists interested in optimizing scientific programs. These specialists can focus on code sequences using PAPI functions.
Top and **PAPI** are all open source tools.

1.7.1 High-level PAPI Interface

The high-level API provides the ability to start, stop and read the counters for a specified list of events. It is particularly well designed for programmers who need simple event measurements, using PAPI preset events.

Compared with the low-level API the high-level is easier to use and requires less setup (additional calls). However this ease of use leads to a somewhat higher overhead and the loss of flexibility.

Note Earlier versions of the high-level API are not thread safe. This restriction has been removed with PAPI 3.

Below is a simple code example using the high-level API:

```
#include <papi.h>

#define NUM_FLOPS 10000
#define NUM_EVENTS 1

main()
{
int Events[NUM_EVENTS] = {PAPI_TOT_INS};
long_long values[NUM_EVENTS];

/* Start counting events */
if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
   handle_error(1);

/* Defined in tests/do_loops.c in the PAPI source distribution */
do_flops(NUM_FLOPS);

/* Read the counters */
if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
   handle_error(1);

printf("After reading the counters: %lld\n",values[0]);

do_flops(NUM_FLOPS);
```

```
/* Add the counters */
if (PAPI_accum_counters(values, NUM_EVENTS) != PAPI_OK)
   handle_error(1);
printf("After adding the counters: %lld\n", values[0]);

/* double a,b,c; c+= a* b; 10000 times  */
do_flops(NUM_FLOPS);

/* Stop counting events */
if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
   handle_error(1);

printf("After stopping the counters: %lld\n", values[0]);
}
```

Possible Output:

```
After reading the counters: 441027
After adding the counters: 891959
After stopping the counters: 443994
```

Note that the second value (after adding the counters) is approximately twice as large as the first value (after reading the counters). This is because **PAPI_read_counters** resets and leaves the counters running, then **PAPI_accum_counters** adds the current counter value into the **values** array.

1.7.2 Low-level PAPI Interface

The low-level API manages hardware events in user-defined groups called **Event Sets**. It is particularly well designed for experienced application programmers and tool developers who need fine-grained measurements and control of the PAPI interface. Unlike the high-level interface, it allows both PAPI preset and native event measurements.

The low-level API features the possibility of getting information about the executable and the hardware, and to set options for multiplexing and overflow handling. Compared with high-level API, the low-level API increases efficiency and functionality.

An Event Set is a user-defined group of hardware events (preset or native) which, all together, provide meaningful information. The users specify the events to be added to the Event Set and attributes such as the counting domain (user or kernel), whether or not the events are to be multiplexed, and whether the Event Set is to be used for overflow or profiling. PAPI manages other Event Set settings such as the low-level hardware registers to use, the most recently read counter values and the Event Set state (running / not running).

Following is a simple code example using the low-level API. It applies the same technique as the high-level example.

```
#include <papi.h>
#include <stdio.h>

#define NUM_FLOPS 10000

main()
{
int retval, EventSet=PAPI_NULL;
```

```
long_long values[1];

/* Initialize the PAPI library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
if (retval != PAPI_VER_CURRENT) {
  fprintf(stderr, "PAPI library init error!\n");
  exit(1);
}

/* Create the Event Set */
if (PAPI_create_eventset(&EventSet) != PAPI_OK)
    handle_error(1);

/* Add Total Instructions Executed to our Event Set */
if (PAPI_add_event(EventSet, PAPI_TOT_INS) != PAPI_OK)
    handle_error(1);

/* Start counting events in the Event Set */
if (PAPI_start(EventSet) != PAPI_OK)
    handle_error(1);

/* Defined in tests/do_loops.c in the PAPI source distribution */
do_flops(NUM_FLOPS);

/* Read the counting events in the Event Set */
if (PAPI_read(EventSet, values) != PAPI_OK)
    handle_error(1);

printf("After reading the counters: %lld\n",values[0]);

/* Reset the counting events in the Event Set */
if (PAPI_reset(EventSet) != PAPI_OK)
  handle_error(1);

do_flops(NUM_FLOPS);

/* Add the counters in the Event Set */
if (PAPI_accum(EventSet, values) != PAPI_OK)
   handle_error(1);
printf("After adding the counters: %lld\n",values[0]);

do_flops(NUM_FLOPS);

/* Stop the counting of events in the Event Set */
if (PAPI_stop(EventSet, values) != PAPI_OK)
    handle_error(1);

printf("After stopping the counters: %lld\n",values[0]);
 }
```

Possible output:

```
After reading the counters: 440973
After adding the counters: 882256
After stopping the counters: 443913
```

Note that PAPI_reset is called to reset the counters, because PAPI_read does not reset the counters. This lets the second value (after adding the counters) to be approximately twice as large as the first value (after reading the counters).

For more details, please refer to PAPI man and documentation, which are installed with the product in /usr/share directory.

1.8 Profiling Programs – HPC Toolkit

HPC Toolkit provides a set of profiling tools that help you to improve the performance of the system. These tools perform profiling operations on the executables and display information in a user-friendly way.

The main advantage of **HPC Toolkit** over other profiling tools is that you do not need to include profiling options and to re-compile the executable.

Note In this section, the term "executable" refers to a Linux program file, in ELF (Executable and Linking Format) format.

Prerequisites:

- **hpcviewer** requires the **Java Runtime Environment** from the **RHEL5.3-Supplementary-for-EM64T** CDROM to be installed on the Management Node. See chapter 3 in the **BAS5 for Xeon** *Installation and Configuration Guide* for more details.
- The executable must contain debugging information (if not, there will be no correspondence between counters and code at source line level)
- The executable must be dynamically linked because HPC Toolkit overloads the default initialization functions to call PAPI.
- The executable must not use ANSI libstdc++. (The constructor being static with the current libstdc++ at the present time, using HPC Toolkit with such an executable produces a SIGSEGV).

1.8.1 HPC Toolkit Tools

HPC Toolkit provides four main capabilities:

- *analysis* of an executable to recover the program structure
- *measurement* of performance metrics as the executable runs
- *correlation* of the performance metrics with the program structure
- *presentation* of the performance metrics with the associated source code

---

Note HPC Toolkit provides the most complete performance information when working with fully-optimized executables that include line map information within the object code. Since compilers often provide line map information for fully-optimized code, this requirement need not require a special build process.

HPC Toolkit includes the following tools:

hpcstruct *analyzes* an executable to determine its static program structure. The goal is to search for execution loops and to identify the corresponding source code procedures, loop nests, functions, and inlined code.

hpcrun-flat *measures* the execution of an executable by statistical sampling of the hardware performance counters to create flat profiles. A flat profile is an IP histogram, where IP is the instruction pointer.

hpcprof-flat *correlates* the raw profiling data from **hpcrun-flat** with the program structure file produced by **hpcstruct**. **hpcprof-flat** generates high level metrics in the form of a performance database called the Experiment database. The Experiment database is in Experiment XML format for use with **hpcviewer**.

hpcprofit *correlates* flat profile metrics with either source code structure or object code and generates textual output suitable for a terminal. **hpcprofit** also generates textual dumps of profile files.

hpcviewer *presents* the Experiment database produced by **hpcprof-flat** by allowing the user to quickly and easily view the performance database generated by **hpcprof-flat.**

1.8.2 Display Counters

The **hpcrun-flat** tool uses the hardware counters as parameters. To know which counters are available for your configuration, use the **papi_avail** command or the **hpcrun-flat** tool itself:

(1) papi_avail:

```
papi_avail
```

```
Available events and hardware information.
--------------------------------------------------------------------------
Vendor string and code   : GenuineIntel (1)
Model string and code    : 32 (1)
CPU Revision : 0.000000
CPU Megahertz: 1600.000122
CPU's in this Node : 6
Nodes in this System: 1
Total CPU's  : 6
Number Hardware Counters : 12
Max Multiplex Counters   : 32
--------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.
Name          Code        Avail    Deriv Description (Note)
PAPI_TOT_CYC   0x8000003b  Yes      No    Total cycles
PAPI_L1_DCM0   x80000000   Yes      No    Level1 data cache  misses
PAPI_L1_ICM0   x80000001   Yes      No    Level 1  instruction cache misses
PAPI_L2_DCM0   x80000002   Yes      Yes   Level 2 data cache misses
...
PAPI_FSQ_INS   0x80000064  No       No    Floating point square root
instructions
PAPI_FNV_INS   0x80000065  No       No    Floating point inverse instructions
PAPI_FP_OPS    0x80000066  Yes      No    Floating point operations
--------------------------------------------------------------------------
Of 103 possible events, 60 are available, of which 17 are derived.
```

The following counters are particularly interesting: PAPI_TOT_CYC (number of CPU cycles) and PAPI_FP_OPS (number of floating point operations).

To display more details use the **papi_avail -d** command.

(2) hpcrun-flat:

```
hpcrun-flat [informational-options]
```

Informational Options:

–l, --events-short List available events (some may not be profilable)

–L, --events-long Similar to events-short but with more information

--paths Print paths for external PAPI and MONITOR

-V, --version Print version information

-h, --help Print help

Example:

```
hpcrun-flat -l
```

```
*** Hardware information ***
------------------------------------------------------------------------------
Vendor string and code  : GenuineIntel (1)
Model string and code   : 32 (0)
CPU Revision            : 5
CPU Megahertz           : 1599
CPU's in this Node      : 16
Nodes in this System    : 1
Total CPU's             : 16
Number Hardware Counters: 12
Max Multiplex Counters  : 32
==============================================================================
*** Wall clock time ***
WALLCLK     wall clock time (1 millisecond period)
==========================================================================
*** Available PAPI preset events ***
--------------------------------------------------------------------------
Name            Description
--------------------------------------------------------------------------
PAPI_L1_DCM     Level 1 data cache misses
PAPI_L1_ICM     Level 1 instruction cache misses
PAPI_L2_DCM     Level 2 data cache misses
PAPI_L2_ICM     Level 2 instruction cache misses
...
PAPI_L3_TCW     Level 3 total cache writes
PAPI_FP_OPS     Floating point operations
Total PAPI events reported: 60
==========================================================================
*** Available native events ***
--------------------------------------------------------------------------
Name                        Description
--------------------------------------------------------------------------
ALAT_CAPACITY_MISS_ALL      ALAT Entry Replaced -- both integer and
                            floating point instructions
ALAT_CAPACITY_MISS_FP       ALAT Entry Replaced -- only floating point
                            instructions
ALAT_CAPACITY_MISS_INT      ALAT Entry Replaced -- only integer
                            instructions...
BRANCH_EVENT                Execution Trace Buffer Event Captured.
                            Alias to ETB_EVENT
Total native events reported: 637
==========================================================================
```

1.8.3 Using HPC Toolkit

Important

It is necessary to run one of these sequences in order to produce complete results that allow you to view metrics and analyze performance:

- **hpcstruct, hpcrun-flat, hpcprof-flat, hpcviewer**
- **hpcstruct, hpcrun-flat, hpcprofttt**

1.8.3.1 Step 1: Analyzing the executable code (hpcstruct)

hpcstruct analyzes an executable to determine its static program structure. **hpcstruct** recovers the program structure from the executable's object code and writes a Program XML file (type=PGM) that describes that structure. This XML file is used by **hpcprof-flat** or **hpcprofttt**.

hpcstruct works best with highly optimized binaries produced by C, C++, and FORTRAN programs.

Note Default values for options and switches are shown in curly brackets.

Syntax:

```
hpcstruct [options] executable > program_structure_XML_file
```

General Options:

-v, --verbose [<n>]	Verbose mode; generate progress messages to *stderr* (standard error output) at verbosity level <n>
-V, --version	Print version information
-h, --help	Print help information
–debug [<n>]	Use debug level <n> {1}
–debug-proc <glob>	Debug the structure recovery for procedures matching the procedure glob <*glob*>

Recovery and Output Options:

-i, --irreducible-interval-as-loop-off	Do not treat irreducible intervals as loops
-f, --forward-substitution-off	Assume that forward substitution does not occur (helpful for handling erroneous PGI debugging info)
-p <list>, --canonical-paths <list>	Ensure that the program structure tree only contains the files found in the colon-separated <list>. May be passed multiple times.
-n, --normalize-off	Turn off scope tree normalization

-u, --unsafe-normalize-off	Turn off potentially unsafe normalization
-c, --compact	Generate compact output
-s, --symbolic-only	Include only those program structure tree nodes that have DWARF line number information
-d, --skip-disconnected-nodes	Skip those program structure tree nodes that are disconnected from the enclosing procedure

Example:

```
hpcstruct -v 2 -s smath.exe >smath.psxml
```

```
> hpcstruct -v 2 -s smath.exe >smath.psxml
msg: Building scope tree for [MAIN__] ...
msg: Building scope tree for [ft250_] ...
msg: Building scope tree for [xyz1_] ...
```

hpcstruct writes the Program XML structure tree for the **smath.exe** program to the file **smath.psxml**. All nodes without line number information are ignored because the **-s** option was used.

1.8.3.2 Step 2: Measuring the execution (hpcrun-flat)

hpcrun-flat is a flat statistical sampling-based profiler. It supports multiple sample sources during one execution and creates an Instruction Pointer (IP) histogram, or flat profile, for each sampled source. It can profile complex applications and can be used in conjunction with parallel process launchers.

The executable executes under control of **hpcrun-flat**. For an event e and a period p, after every p instances of e, a counter associated with the current IP is incremented.

When the executable terminates, **hpcrun-flat** writes the histogram into a file with the name *executable*.hpcrun-flat.*hostname*.*pid*.*tid*. This file is known as a profile file and contains a histogram of counts for each load module.

The user can abort the process by sending the Interrupt signal (INT or Ctrl-C). **hpcrun-flat** will write the partial profile. This technique is useful for programs that run a long time or are not well-behaved.

Syntax 1:

```
hpcrun-flat [profiling-options] [--] executable [executable-arguments]
```

General Options:

`--`	The special option '`--`' stops the **hpcrun-flat** option processing; this is useful when the program specified by `executable` takes arguments of its own.
--debug [<n>]	Run with debug level <n>. {1}

Profiling Options:

–e <event>[:<period>] - -event <event>[:<period>]
An event to profile and its corresponding sample period. <event> can be a PAPI or native processor event. This option can be passed multiple times. It is recommended that a period always be specified. {PAPI_TOT_CYC:999999}

-r [<yes|no>], - -recursive [<yes|no>],
Profile process spawned by executable_name. {no}

-t <each|all>, - -threads <each|all>
Select thread profiling mode. With each, separate profiles are generated for each thread. With all, profiles of all threads are combined. Only POSIX threads are supported. {each}

-o <outpath>, - -output<outpath>
Directory for output data {.}

- -**papi-flag <flag>** Profile style flag {PAPI_POSIX_PROFIL}

Notes

- Because **hpcrun-flat** uses LD_PRELOAD to initiate profiling, it cannot be used to profile *setuid* commands. For the same reason, it cannot profile statically linked applications.
- Some events are not compatible. To resolve this problem, specify a period of time for each event using the **:period** parameter. When this option is specified **hpcrun-flat** retrieves each event in sequence, thus avoiding conflicts.
- The WALLCLK event can be used to profile the "wall" clock. It may be used only once, cannot be used with another event, and cannot have a period specified. The WALLCLK event cannot be used in a multithreaded process.

Examples:

```
hpcrun-flat -e PAPI_TOT_INS -e PAPI_TOT_CYC -o hpcrun.data -- smath.exe
```

```
>hpcrun-flat -e PAPI_TOT_INS -e PAPI_TOT_CYC -o hpcrun.data -- smath.exe
hpcrun-flat [pid 24024, tid 0x0]:
Using output file hpcrun.data/smath.exe.hpcrun-flat.sysj.24024.0x0
 The computed answer is:     500500
```

To retrieve the counters for 3000 events, enter:

```
hpcrun -e PAPI_TOT_INS:3000 -e PAPI_TOT_CYC:3000 ...
```

1.8.3.3 Step 3: Correlating flat metrics with program structure (hpcprof-flat)

hpcprof-flat generates high level metrics from raw profiling data produced by **hpcrun-flat** and correlates it with logical source code abstractions produced by hpcstruct.

Syntax 1:

```
hpcprof-flat [options] [output-options] [correlation-options]
<profile-file>
```

The inputs to this usage of **hpcprof-flat** are (1) the Program XML file created by the **hpcstruct** tool and (2) the profile files created by the **hpcrun-flat** tool. If the Program XML file is not provided, **hpcprof-flat** will default to correlation using the line map information.

By default, **hpcprof-flat** generates an Experiment database file (Experiment XML format) to be used with **hpcviewer** as well as a configuration file that can be used as input to a subsequent invocation of **hpcprof-flat**.

General Options:

-v, --verbose [<n>] Verbose mode; generate progress messages to stderr (standard error output) at verbosity level <n>

-V, --version Print version information

-h, --help Print help information

–debug [<n>] Use debug level <n> {1}

Source Structure Correlation Options:

-I <path>, --include <path> Use <path> when searching for source files. A '*' after the last slash indicates recursion. This option may be used multiple times.

-S <file>, --structure <file> Use the program structure file <file> generated by the hpcstruct tool. This option may be used multiple times (e.g., for shared libraries).

Output Options:

-o <db-path>, --db <db-path>, –output <db-path>
Specify experiment database name <db-path> {./experiment-db}

–src [yes|no], --source[yes|no]
Indicates if source code files should be copied into experiment database. {yes}

Output Format Options:

Select different output formats and optionally specify the output filename *file* (located within the Experiment database). The output is sparse in the sense that it ignores program areas without profiling information. (Set *file* to '-' to write to *stdout*.)

-x [*file*], **--experiment** [*file*]
Default. Experiment XML format. {`experiment.xml`}. NOTE: To disable, set *file* to no.

–csv [*file*] Comma-separated-value format. It includes flat scope tree and loops and is useful for downstream external tools. {`experiment.csv`}

Example:

```
hpcprof-flat -S smath.psxml hpcrun.data/*
```

```
> hpcprof-flat -S smath.psxml hpcrun.data/*
msg: Copying source files reached by PATH/REPLACE options to experiment-db
msg: Writing final scope tree (in XML) to experiment.xml
```

Syntax 2:

```
hpcprof-flat [options] [output-options] --config <config-file>
```

The general options and the output options are as listed above for **hpcprof-flat**, Syntax 1. However, the correlation options are contained in the configuration file and cannot be specified on the command line.

<config-file> is a configuration file generated by a previous **hpcprof-flat** activity and optionally edited by the user. The configuration file syntax is briefly described in Section *Configuration File Syntax*, on page1-38.

Example:

For example, the config.xml file produced by the above **hpcprof-flat** command can be modified to insert a computed metric that computes the cycles per instruction:

```
<METRIC name="CPI" displayName="CPI" percent="false">
  <COMPUTE>
    <math>
      <apply> <divide/>
        <ci>PAPI_TOT_CYC</ci>
        <ci>PAPI_TOT_INS</ci>
      </apply>
    </math>
  </COMPUTE>
</METRIC>
```

```
hpcprof-flat -S smath.psxml --config experiment-db/config.new
```

```
> hpcprof-flat -S smath.psxml --config experiment-db/config.new
msg: Computed METRIC CPI: CPI = (PAPI_TOT_CYC / PAPI_TOT_INS)
msg: Copying source files reached by PATH/REPLACE options to experiment-db
msg: Writing final scope tree (in XML) to experiment.xml
```

When the experiment.xml file is viewed with **hpcviewer**, it will show three columns of metrics, the native metrics for the PAPI_TOT_CYC and PAPI_TOT_INS events as well as a computed metric for CPI.

1.8.3.4 Step 3a: Correlating flat metrics with program structure (hpcproftt)

hpcproftt provides an alternative to **hpcprof-flat** and **hpcviewer**. **hpcproftt** correlates profile metrics with either *source code structure* (the first and default mode) or *object code* (second mode) and generates textual output suitable for a terminal. **hpcproftt** also supports a third mode in which it generates textual dumps of profile files. In all modes, **hpcproftt** expects a list of profile files as input.

hpcproftt defaults to *source structure* correlation mode. When `--source` is not specified, the default switches are `{pgm, lm}`; with `--source`, the default switch is `{sum}`.

Syntax 1: Source Structure Correlation

```
hpcproftt [--source] [options] <profile-file>...
```

In source mode, **hpcproftt** first creates raw metrics for every native event in the profile files and creates any derived metrics specified by the `--metric` option. It then correlates the metrics to the program structure based on the **hpcstruct** output file specified by the `--structure` option. If this file is not specified, a simple structure is computed from the load module's line map. **hpcproftt** finally generates the metric summaries and annotated source files to *stdout*. Each summary compares a source structure element, such as a procedure, with all other elements of that type throughout the program. Structure elements include Program, Load Module, File, Procedure, Loop, and Statement. The desired elements are chosen by switches specified with the `--source` option.

General Options:

-v, --verbose [<n>] Verbose mode; generate progress messages to *stderr* (standard error output) at verbosity level <n>

-V, --version Print version information

-h, --help Print help information

--debug [<n>] Use debug level <n> {1}

Source Structure Correlation Switches:

--source[=all,sum,pgm,lm,f,p,l,s,src] or
--src[=all,sum,pgm,lm,f,p,l,s,src]

Correlate metrics to source code structure. Without –source, the default is {pgm,lm}; with, it is {sum}

all	all summaries plus annotated source files
sum	all summaries
pgm	program summary
lm	load module summary
f	file summary
p	procedure summary
l	loop summary
s	statement summary
src	annotate source files; equiv to –srcannot '*'

Source Structure Correlation Options:

--srcannot <glob> Annotate source files with path names that match file glob <glob>. Protect globs from the shell with 'single quotes'. May pass multiple times.

-M <metric>, --metric <metric>
Show a supplemental or different metric set. <metric> is one of the following:

sum	Additionally show Mean, RStdDev, Min, Max
sum-only	Show only Mean, RStdDev, Min, Max

-I <path>, --include <path>
Use <path> when searching for source files. A '*' after the last slash indicates recursion. This option may be used multiple times.

-S <file>, --structure <file>
Use the program structure file <file> generated by the **hpcstruct** tool. This option may be used multiple times (e.g., for shared libraries).

Example of Source Structure Correlation:

```
hpcproftt --source hpcrun.data/*
```

```
>hpcproftt --source hpcrun.data/*
================================================================================
Metric definitions. column: name (nice-name) [units] {details}:
   1: PAPI_TOT_INS [events] {Instructions completed:999999 ev/smpl}
   2: PAPI_TOT_CYC [events] {Total cycles:999999 ev/smpl}
================================================================================
Program summary (row 1: sample count for raw metrics):
--------------------------------------------------------------------------------
     421      253
4.21e+08 2.53e+08
================================================================================
Load module summary:
--------------------------------------------------------------------------------
  97.62%   98.42%  smath.exe
   2.38%    1.58%  /lib/tls/libm-2.3.4.so
================================================================================
File summary:
--------------------------------------------------------------------------------
  97.62%   98.42%  [smath.exe]smathz.f
   1.19%    0.79%  [/lib/tls/libm-2.3.4.so]~~~<unknown-file>~~~
   1.19%    0.79%  [/lib/tls/libm-2.3.4.so]<built-in>
================================================================================
Procedure summary:
--------------------------------------------------------------------------------
  94.06%   94.07%  [smath.exe]<smathz.f>ft250_
   2.38%    2.37%  [smath.exe]<smathz.f>MAIN__
   1.19%    1.98%  [smath.exe]<smathz.f>xyz1_
   0.48%    0.00%  [/lib/tls/libm-2.3.4.so]<~~~<unknown-file>~~~>atan
   0.48%    0.79%  [/lib/tls/libm-2.3.4.so]<<built-in>>POW_COMMON
   0.24%    0.00%  [/lib/tls/libm-2.3.4.so]<<built-in>>COMMON_PATH
   0.24%    0.00%  [/lib/tls/libm-2.3.4.so]<~~~<unknown-file>~~~>sinh
   0.24%    0.00%  [/lib/tls/libm-2.3.4.so]<~~~<unknown-file>~~~>log10
   0.24%    0.40%  [/lib/tls/libm-2.3.4.so]<~~~<unknown-file>~~~>sqrt
   0.24%    0.00%  [/lib/tls/libm-2.3.4.so]<<built-in>>_SINCOS_COMMON2

...
```

Syntax 2:

```
hpcproftt --object[=s] [options] <profile-file>
```

In object mode, **hpcproftt** performs fine-grained correlation and generates annotated object code. It will create raw metrics for every native event in only **one** profile file.

Object Correlation Switches:

--object[=s]
--obj[=s] Correlate metrics with object code by annotating object code procedures and instructions. {}

s intermingle source line info with object code

Object Correlation Options:

–obj-values Show raw metrics as values instead of percents

–obj-threshold <n> Prune procedures with an event count < n {1}

Note On some architectures, delays between event triggers, interrupt generation, and sampling of the IP mean that an event may be associated with a different instruction from the one that caused the event. This gap may be as many as 50 to 70 instructions in length.

Example of Object Code Correlation:

```
hpcproftt --source hpcrun.data/smath.exe.hpcrun-flat.sysj.24024.0x0
```

```
>hpcproftt --object=s hpcrun.data/smath.exe.hpcrun-flat.sysj.24024.0x0
================================================================================
Load module: smath.exe
--------------------------------------------------------------------------------
Metric definitions. column: name (nice-name) [units] {details}:
   1: PAPI_TOT_INS [samples] {Instructions completed:999999 ev/smpl}
   2: PAPI_TOT_CYC [samples] {Total cycles:999999 ev/smpl}

Metric summary for load module (totals):
       411      249

Procedure: MAIN__ (MAIN__)
----------------------------------------------------------
Metric definitions. column: name (nice-name) [units] {details}:
   1: PAPI_TOT_INS [samples] {Instructions completed:999999 ev/smpl}
   2: PAPI_TOT_CYC [samples] {Total cycles:999999 ev/smpl}
Metric summary for procedure (percents relative to load module):
        10        6
     2.43%    2.41%
Metric details for procedure (percents relative to procedure):
smathz.f:1
0x4000000000001260:                    [MII]
0x4000000000001266:                    [MII]
0x400000000000126c:                    [MII]
0x4000000000001270:                    [MII]
0x4000000000001276:                    [MII]
0x400000000000127c:                    [MII]
0x4000000000001280:                    [MFB]       nop.m 0x0
smathz.f:259
0x4000000000001286:                    [MFB]       nop.m 0x0
```

```
0x4000000000000128c:                    [MFB]       nop.m 0x0
0x4000000000001290:                    [MMI]
0x4000000000001296:                    [MMI]
...
```

Syntax 3:

```
hpcproftt --dump <profile-file>
```

This form of the **hpcproftt** command will generate textual representation of raw profile data.

Example:

```
hpcproftt --dump hpcrun.data/*
```

```
> hpcproftt --dump hpcrun.data/*
================================================================================
hpcrun.data/smath.exe.hpcrun-flat.sysm.29041.0x0
================================================================================
--- ProfileData Dump ---
{ ProfileData: hpcrun.data/smath.exe.hpcrun-flat.sysm.29041.0x0 }
  { LM: /lib/ld-2.3.4.so, loadAddr: 0x2000000000000000 computed=0 }
    { EventData: PAPI_TOT_INS, period: 999999, outofrange: 0, overflow: 0 }
    { EventData: PAPI_TOT_CYC, period: 999999, outofrange: 0, overflow: 0 }
  { LM: /lib/libdl-2.3.4.so, loadAddr: 0x2000000000470000 computed=0 }
    { EventData: PAPI_TOT_INS, period: 999999, outofrange: 0, overflow: 0 }
    { EventData: PAPI_TOT_CYC, period: 999999, outofrange: 0, overflow: 0 }
  { LM: /lib/libgcc_s-3.4.6-2.so.1, loadAddr: 0x20000000001c0000 computed=0 }
    { EventData: PAPI_TOT_INS, period: 999999, outofrange: 0, overflow: 0 }
    { EventData: PAPI_TOT_CYC, period: 999999, outofrange: 0, overflow: 0 }
  { LM: /lib/tls/libc-2.3.4.so, loadAddr: 0x2000000000200000 computed=0 }
    { EventData: PAPI_TOT_INS, period: 999999, outofrange: 0, overflow: 0 }
    { EventData: PAPI_TOT_CYC, period: 999999, outofrange: 0, overflow: 0 }
  { LM: /lib/tls/libm-2.3.4.so, loadAddr: 0x2000000000100000 computed=0 }
    { EventData: PAPI_TOT_INS, period: 999999, outofrange: 0, overflow: 0 }
      { 0x2000000000115b60: 1 }
      { 0x2000000000116200: 1 }
      { 0x2000000000116450: 1 }
      { 0x2000000000117200: 1 }
      { 0x2000000000117890: 1 }
...
```

1.8.3.5 Step 4: Presenting the results (hpcviewer)

The **hpcviewer** tool displays the counters values for each code line (Figure 1-6 below).

hpcviewer uses the Experiment XML file generated by **hpcprof-flat**.

Syntax:

```
hpcviewer [experiment-database-file]
```

[experiment-database-file] is the name of the Experiment database file produced by **hpcprof-flat** or **hpcproftt**. When [experiment-database] is not specified, **hpcviewer** will prompt the user to select the Experiment database file from a directory window.

The **hpcviewer** window is divided into three panes.

- The *source pane*, at the top of the screen, contains the source code associated with the entity currently selected in the navigation pane.
- The *navigation pane*, at the lower left, presents a hierarchical tree-based structure that identifies the display of the performance data. This pane can include load modules, source files, procedures, loops, and source lines.

  The buttons in the navigation pane control flatten and zoom. From left to right, the four buttons are:

flatten	replaces each top-level scope with its children. Useful to view and rank peers together.
unflatten	inverse of flatten. Makes previously hidden nodes visible again.
up arrow	zooms to show only information for the selected line and its descendants
down arrow	zooms out by reversing a prior zoom operation

- The *metric pane* displays the performance metrics associated with the entities to the left in the navigation pane. Entities in the tree view of the navigation pane are sorted at each level by the metric selected in the metrics pane. Sort order can be reversed by clicking on the arrow at the head of the selected column.

The following figure shows an example of the **hpcviewer** screen. There is a column for each event specified in **hpcrun-flat** as well as a third column for the computed metric that was added by **hpcprof-flat**.

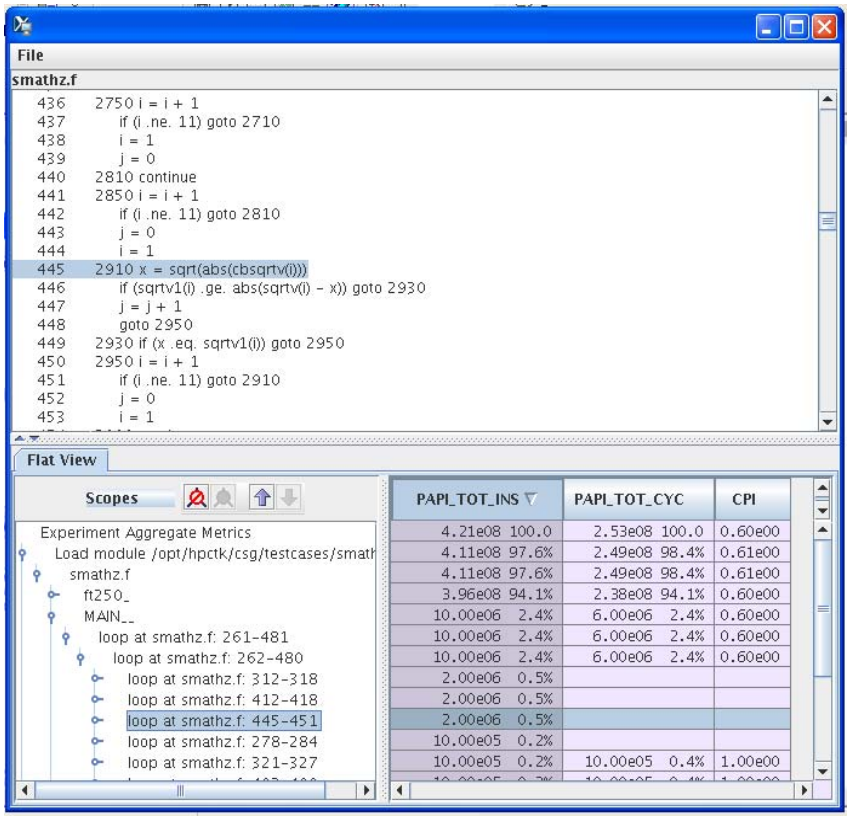

Figure 1-6. View of the counter values, using hpcviewer

1.8.4 Configuration File Syntax

A configuration file is an XML document of type `HPCVIEW`. The following top-level elements are used in the configuration file:

*	`<HPCVIEW>`	Begin document.
	`<TITLE name="my-title"/>`	my-title names the Experiment database.
	`<PATH name="path"/>`	A set of PATH directives specifying path names to search for source files. path is a relative or absolute path containing source code to which performance data is correlated. In order to recursively search a directory, append an escaped '*' after the last slash, e.g., /mypath/* (escaping is for the shell).
	`<REPLACE in="old-path-prefix" out="new-path-prefix" />`	A set of REPLACE directives can be used to define one path prefix to operationally match another prefix occuring in profile data files or in a program structure file. This is useful when trying to compare performance metrics between machines with different file structures, e.g., because the executables or the source files are installed in different places.
	`<STRUCTURE name="program.psxml"/>`	One or more STRUCTURE directives providing program structure files created by hpcstruct
*	`<METRIC name="name" displayName="name-in-display" display="true|false" percent="true|false"> ... </METRIC>`	One or more metrics.
*	`</HPCVIEW>`	End document.

* element is required

A metric may be of two types, **native** or **derived**. Metrics are introduced using the METRIC element and contain several attributes:

name.	A unique name used when creating derived metrics that are expressions of other metrics.
displayName.	Name to be displayed. Not necessarily unique.
display.	Controls metric visibility. A metric used only as input to a computed metric need not be displayed.

percent. Indicates whether the viewer should display a column of percentages computed as the ratio of the metric for this scope to the metric for the whole program. Percents are useful when metrics are computed by summing contributions from descendants in the scope tree, but are meaningless for computed metrics such as ratio of flops/memory access in a scope.

The elements that appear inside the METRIC element determine its type. A metric may be of two types: **native** (type=FILE) or **derived** (type=COMPUTE).

1.8.4.1 Native or FILE Metrics

This type of metric appears in profile information generated by **hpcrun-flat** or by **hpcproftt**:

```
<METRIC name="m1" ...>
  <FILE name="file1" select="short-name-in-file1" type="HPCRUN|PROFILE"/>
</METRIC>
```

Because a file may contain multiple metrics, the `FILE` element has an optional 'select' attribute to identify a particular metric within the file. Metrics are identified by their 'shortName' values, typically zero-based indices. The default 'select' value is 0 and corresponds to the first metric.

1.8.4.2 Derived or COMPUTE Metrics

Derived metrics are specified by a `COMPUTE` element containing a MathML equation in terms of metrics defined earlier in the HPCVIEW document.

hpcprof-flat supports the following operands:

- **constants**: `<cn>2</cn`
- **variables**: `<ci>m1</ci>` (used to refer to other metrics)

and the following MathML operators (used within `<apply>`):

- **negation**: `<minus/>` (1-ary)
- **subtraction**: `<minus/>` (2-ary)
- **addition**: `<plus/>` (n-ary)
- **multiplication**: `<times/>` (n-ary)
- **division**: `<divide/>` (2-ary)
- **exponentiation**: `<power/>` (2-ary)
- **minimum**: `<min/>` (n-ary)
- **maximum**: `<max/>` (n-ary)
- **mean (arithmetic)**: `<mean/>` (n-ary)
- **standard deviation**: `<sdev/>` (n-ary)

Consider the examples from the previous sections with two native metrics for `PAPI_TOT_CYC` (cycles) and `PAPI_TOT_INS` (instructions).

The file `config.xml` from example, produced by **hpcprof-flat**, contains the following elements, including only native metrics:

```
<HPCVIEW>
<TITLE name=""/>
<STRUCTURE name="smath.psxml"/>
<METRIC name="PAPI_TOT_INS" displayName="PAPI_TOT_INS" sortBy="true">
  <FILE name="hpcrun.data/smath.exe.hpcrun-flat.sysj.29041.0x0"
     select="0" type="HPCRUN"/>
</METRIC>
<METRIC name="PAPI_TOT_CYC" displayName="PAPI_TOT_CYC">
  <FILE name="hpcrun.data/smath.exe.hpcrun-flat.sysj.29041.0x0"
    select="1" type="HPCRUN"/>
</METRIC>
</HPCVIEW>
```

The file `config.new` from example, produced by **hpcprof-flat** and subsequently edited by the user, contains the following elements, including both native and derived metrics:

```
<HPCVIEW>
<TITLE name=""/>
<STRUCTURE name="smath.psxml"/>
<METRIC name="PAPI_TOT_INS" displayName="PAPI_TOT_INS" sortBy="true">
  <FILE name="hpcrun.data/smath.exe.hpcrun-flat.sysj.29041.0x0"
    select="0" type="HPCRUN"/>
</METRIC>
<METRIC name="PAPI_TOT_CYC" displayName="PAPI_TOT_CYC">
  <FILE name="hpcrun.data/smath.exe.hpcrun-flat.sysj.29041.0x0"
    select="1" type="HPCRUN"/>
</METRIC>
<METRIC name="CPI" displayName="..." percent="false">
  <COMPUTE>
    <math>
      <apply> <divide/>
        <ci>PAPI_TOT_CYC</ci>
        <ci>PAPI_TOT_INS</ci>
      </apply>
    </math>
  </COMPUTE>
</METRIC>
</HPCVIEW>
```

1.8.5 More Information

For more detailed information about HPC Toolkit go to:

http://hpctoolkit.org/man/hpctoolkit.html

1.9 Intel® VTune™ Performance Analyzer for Linux

Intel® VTune™ Performance Analyzer provides both **Sampling** and **Call Graph** analysis to identify where time and resources are being used by applications, libraries and drivers. Sampling should be used first because of its low overhead and in order to identify application modules which require more analysis using Call Graphs. Sampling is usually best for code that predominantly uses loops, whilst Call Graphs are usually better for code that branches.

Sampling

Intel® VTune™ Performance Analyzer uses system-wide, event-based sampling to find bottlenecks with a low overhead (typically less than 5 percent). Events and processes are sampled over a time period and then may be analyzed at different levels - operating system process, thread, module executable, function/method, individual line of source code, or individual machine/assembly language instructions - to identify specific bottlenecks. Problems such as cache misses and branch mis-predictions are easily identified.

Call Graphs

Call Graphs determine calling sequences within algorithms and graphically display critical paths. They also highlight the critical path, the preceding functions and calls which resulted in the time or resource bottleneck.

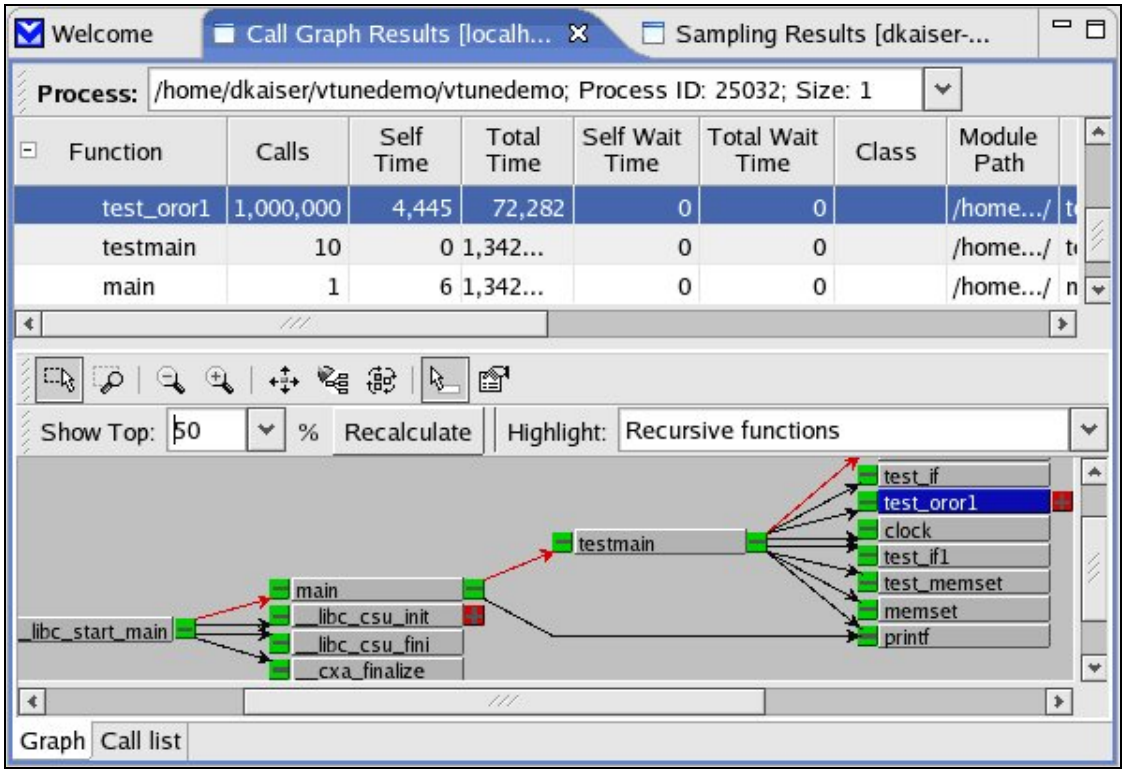

Figure 1-7. A Call Graph showing the critical path in red

Figure 1-7 shows both a table and graph view. When a table entry is selected the function is highlighted in the graph, and vice versa. The critical path for the function is clearly visible.

Identify Performance Improvements

Intel® VTune™ Performance Analyzer looks at an application at machine instruction level. These are annotated and any latencies or stalls are identified. Possible changes to the application are highlighted, and the performance of the new code is compared with the original code to verify improvements in the performance.

Adapted to HPC clusters

Intel® VTune™ Performance Analyzer is adapted for HPC clusters:

- Users can share a large system for simultaneous Call Graph performance analyses.
- Sampling is supported on systems with 128 or more processors using local buffering per CPU for minimum inter-node contention.
- Dedicated events are used to measure parallelism, core sharing of the bus and cache, and modified data sharing by threads for tuning multi-core **Intel®** processors. These identify opportunities to improve threading, tune multi-core sharing of the bus and cache, and optimize cache-line usage.
- Remote profiling minimizes the performance impact on the target system by running the user interface on a separate Windows® PC which is connected to the system.

Intel® Performance Analyzer is proprietary software and has to be bought directly from Intel.

See http://www.intel.com/ for more details.

Chapter 2. Coding and Compiling Optimization

This chapter looks at some coding tips and compiling options to help improve the performance of your application on the Bull HPC platform. Guidelines are given in order to ensure that the application program runs as efficiently as is possible.

The following topics are described:

- 2.1 *Application Code Optimization*
- 2.2 *Compiler Optimization Options*

2.1 Application Code Optimization

Application code optimization is hotly debated and an enormous amount of material has been written on the subject. Some of the guidelines produced are common sense regarding the use of good programming technique. The parallel processing capability means that more than ever your code must be tidily organized and streamlined. Also, of course, the structure and requirements of each application is different, bringing with it its own constraints and limitations.

Sometimes the simplest change to your application can produce the biggest gains in resource use. At all times a scientific approach must be taken with all optimizations measured and verified against existing values.

This chapter contains some general programming guidelines and pointers to ensure that the compilation is as efficient as is possible.

Throughout are tips and pieces of advice resulting from the experience of Bull's High Performance Computing Benchmarking and Software team.

2.1.1 Alias Usage

Aliasing is when a pointer points to the same memory zone across several iterations. Thus it is possible to increase the optimization level for the compiler as long as the developer can ensure that there are not two pointers using the same memory zone. In this case the FORTRAN and C compiler option **-fno-alias** is used to restrict alias usage.

2.1.2 Improving Loops

Loops are very powerful programming devices which in a few lines can result in a high amount of date processing and optimization. Some, if not all, of the basic loop structures – switching, partitioning, factoring, hoisting, fusion, distribution and unrolling – will be part of most programmers' repertoire. Obviously, these optimizations have to be used carefully, with a good knowledge of the application, to ensure that all data dependencies are respected.

Loops automatically allow for parallelism in terms of program scheduling and structure. They also enable the programmer to identify code parallelizing possibilities which may not have been obvious initially.

Array Loop Optimizations

Some optimizations for arranging arrays in memory are as follows:

- C Arrange as a series of lines
- Fortran Arrange as a series of columns

It is essential that data which is placed within one memory location is streamed smoothly, and the data flow for a particular object which is placed in the same memory location is not broken. Again the following options can be used:

- C Internal loop for columns
- Fortran Internal loop for lines

1. Switching, if possible, within loops is useful to align the access to arrays with their position in memory.

```
      do i = 1, N
         do j = 1, N
            A(i,j) = 1/B(i,j)
         end do
   end do
   do j = 1, N
      do i = 1, N
         A(i,j) = 1/B(i,j)
      end do
   end do
```

2. The partitioning of loops allows their granularity to be adapted to the memory hierarchy. The computation is done by blocs which are not necessarily aligned. This works well when all the loops may be switched.

```
   do i = 1, N
   do j = 1, N
         A(i,j) = 1/B(i,j)
         end do
   end do
```

```
   do jj = 1, N, sj
   do ii = 1, N, si
      do j = jj, jj+sj-1
         do i = ii,ii+si-1
               A(i,j) = 1/B(i,j)
               end do
            end do
         end do
   end do
```

3. Fusion combines loops within in the same cycle, thus eliminating the need for temporary arrays. Distribution makes it possible to build parallel loops.

```
      do i = 1, N
         A(i) = ...
   end do
   do i = 1, N
      B(i) = ... A(i) ...
   end do
```

```
do i = 1, N
             A(i) = ...
             B(i) = ... A(i) ...
end do
```

4. Scalars can be increased to remove any dependences resulting from the memory re-use.

```
   do i = 1, N
   T=f(i)
   A(i) = A(i)+T*T
   end do
```

```
   do  i = 1, N
   T[i]=f(i)
        A(i) = A(i)+T[i]*T[i]
   end do
```

Loop Peeling

Loop peeling is a traditional optimization which is used for loops with a low number of iterations. It acts to explicitly extract the first iterations from the loop in order to avoid having to have them returned to the loop, which may result in a high overhead for a low number of iterations. This approach is particularly appropriate for Intel® Itanium® 2 platforms which use the software pipeline intensively - up to 10 levels of operation.

2.1.3 C++ Programming Hints

The following hints originate from Intel's programming tutorial:

- Use the `const` modifier as much as is possible.
- Use local variables rather than global or static variables e.g.

```
int limit;                    int limit;
int function()                   int function()
{                             {
for (i=0; i<limit...)            int my_limit = limit;
}                                for (i=0; i<my_limit...)
                              }
```

- Use static variables rather than global ones e.g.

```
int flag;                          static int flag;
/* flag used only in this file */  /* flag used only in this file */
```

- Use procedures like warning(), error(), exception(), assert() and err().
- Use inline functionality for functions which are used a lot or are small in size.
- Use for or while loops instead of do while loops.
- Use **int** data types for arrays instead of unsigned **int** data types.
- Let the compiler handle prefetching except in the case when there is a problem. In this case the PREFETCH directive is used.

2.1.4 Memory Tips

- Minimize the use of the pointers.
- Use addresses based on the arrays rather than pointers.

```
int *src = src_array;
int *dst = dest_array;
for (i=0; i<10; i++)          for (i=0; i<10; i++)
{                             {
*dst++ = *src++;              dest_array[i] = src_array[i];
}                             }
```

- Use the **restrict** keyword for better control.

2.1.5 Application code performance impedances

The following points may be counter-productive in terms of application performance:

- Reusing the same code for unrelated computations.
- Unnecessary branching and procedure calls.
- Optimizing by hand, for example, loop unrolling and prefetching.
- Writing functions in assembly code.
- Dead code and empty function calls.
- Using the # pragma pack directive and the unaligned keyword. These can lead to misalignment.

2.1.6 Interprocedural Optimization (IPO)

Application performance for programs which contain a lot of small and frequently used functions can be improved considerably using IPO. IPO reduces the number of branches in code, reduces overhead calls through inlining functions and performs interprocedural memory analysis in order to keep critical date in registers across function boundaries.

Keep the following points in mind:

- Uses static variables and static functions, and avoid assigning function addresses or variable addresses to global variables.

  Unless the compiler can detect the whole program, it has no knowledge about the overall use of global variables, external functions, or static variable and static functions whose addresses are taken and assigned to a global variable or function pointer.
- If IPO does not inline automatically, uses the `inline` keyword in C++, and `_inline` in C.
- Avoid passing pointers into a function as a parameter and then assigning them to a global variable. The code below hinders IPO. x is a global variable and p is a pointer.

```
int *x;
foo()
{
   int y;
   bar(&y);
}
bar (p)
{
   x = p
}
```

2.2 Compiler Optimization Options

One of the most important ways of generating efficient executables is to closely examine the compiler optimization options. A single set of optimization options doesn't exist. You have to find the best set of options according to the characteristics of the source code. In addition, each source file can be compiled using different options. Finally, compiler directives can be inserted into the source code in order help the compiler to optimize the program.

2.2.1 Starting Options

Before using advanced optimization options, it is advisable to generate a reference executable using the default compilation optimization options. Advanced optimization option time differences will be analyzed against this execution time.

The default optimization options for the Intel FORTRAN Compiler are the following:

-72	Sets the number of column in the source code to 72.
-O2	Level 2 optimizations (software pipelining, unrolling, inlining).
-align	Memory aligning.
-nomodule	Compilation with F90 modules located in the current directory.
-Zp8	8 bytes alignment.

The default optimization options for the Intel C/C++ Compiler are the following:

-O2	Optimizations of level 2 (software pipelining, unrolling, inlining).
-Ob1	Enables inlining of functions declared with the __inline keyword.
-alias-args	Assume arguments may be aliased.
-falias	Assume aliasing in the program.
-ffnalias	Assume aliasing within functions.

2.2.2 Intel C/C++ and Intel Fortran Optimization Options

Once the reference execution time is collected, more aggressive optimization options can be activated.

The following optimization commands may be activated on both C/C++ and Fortran compilers:

-O3	Level 3 optimizations (**-O2** optimizations plus more aggressive optimizations such as prefetching and loop transformations).
-ip	Enables more interprocedural optimizations for single file compilations.
-ipo	When this option is specified, the compiler performs inline function expansion for calls to functions defined in separate files.
-Qoption,f,-ip_ninl_min_stats=n	Modifies the number of inlining levels (by default this is 15)

-Qoption,f,-ip_ninl_max_total_stats=n	Modifies the number of lines added when inlining (by default n = 2000)
-static	Causes the executable to link all the libraries statically.
-fno-alias	Specifies that aliasing should not be assumed in the program.
-fno-fnalias	Specifies that aliasing should not be assumed within functions, but should be assumed across calls.
-ftz	Enables denormal results to be flushed to zero.

Loop unrolling is an optimization option whereby instructions called in multiple iterations of a loop are combined so that only a single iteration is necessary. This technique is particularly useful for parallel processing. Performance is improved as result of the reduction in the number of overhead instructions that have to be executed for a loop, which in turn reduces branching and improves cache hit rate. However, this option has to be handled carefully.

Unrolling options are:

-unroll0	Ending of unrolling
-unroll	Activation of unrolling
-unrollM	M is the maximum number of loops to be unrolled

2.2.3 Compiler Options which may Impact Performance

The following compiler options **must be avoided if possible** as they will lead to a loss in performance:

-assume dummy_aliases
This forces the compiler to assume that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use. These program semantics slow performance, so you should specify **-assume dummy_aliases** only for the subprograms called that depend on such aliases. The use of dummy aliases violates the FORTRAN-77 and Fortran 95/90 standards but occurs in some older programs.

-c
If you use **-c** when compiling multiple source files, also specify **-ooutputfile** to compile many source files together into one object file. Separate compilations prevent certain inter-procedural optimizations, used with multiple compiler invocations or with **-c** without the **-ooutputfile** option.

-check bounds
Generates extra code for array bounds checking at run time.

-check overflow
Generates extra code to check integer calculations for arithmetic overflow at run time. Once the program is debugged, omit this option to reduce executable program size and slightly improve run-time performance.

-fpe 0
Using this option slows program execution. It enables certain types of floating-point exception handling, which can be expensive.

-g
Generate extra symbol table information in the object file. Specifying this option also reduces the default level of optimization to **-O0** (no optimization). The **-g** option only slows your program down when no optimization level is specified, in which case **-g** turns on **-O0**, which slows the compilation down. If **-g, -O2** are specified, then the code runs at much the same speed as if **-g** were not specified.

-save
Forces the local variables to retain their values from the last invocation terminated. This may change the output of your program for floating-point values as it forces operations to be carried out in memory rather than in registers, which in turn causes more frequent rounding of your results.

-O0
Turns off optimizations. Can be used during the early stages of program development or when you use the debugger.

-vms
Controls certain VMS-related run-time defaults, including alignment. If you specify the **-vms** option, you may need to also specify the -align records option to obtain optimal run- time performance.

2.2.4 Flags and Environment Variables

-assume buffered_io with **FORT_BUFFERRED=TRUE**

-dryrun Gives non specific information regarding what has happened at the ld level.

KMP_STACKSIZE Allows the stack size to be increased. This works with **ulimit**

For example with **ulimit -s 1 024 000** or with **ulimit -S -s unlimited** the following command is used:

```
export KMP_STACKSIZE=250 000
```

2.2.5 Compiler Directives for Loops

The following directives are to be specified before the loops concerned:

#pragma For C and C++ programs

[Cc*!]DIR$ For Fortran programs.

The following pragmas can be used:

LOOP COUNT(N) Specifies the number of loop iterations for the pragma.

DISTRIBUTE POINT May be placed inside or outside of a loop.

[NO]UNROLL, UNROLL(N) Controls loop unrolling.

IVDEP Ignores vectorial dependences.

Example for **IVDEP**: The results generated using the **opt_report** option – see section 2.2.7:

```
 do i = 1, m
   if (a(i) .eq. 0) then      Resource II = 1
   b(i) = a(i) + 1       Recurrence II = 1
else                 Minimum II = 1
   b(i) = a(i)/c(i)   Last attempted II = 1
   endif         Estimated GCS II = 1
enddo
```

Modulo scheduling was successful but there was no overlap across iterations therefore the loop was not pipelined.

2.2.6 Options for Compiler Optimization Reports

The following options instruct the compiler to generate an optimization report:

-opt_report Instructs the compiler to generate an optimization report to **stderr.**

-opt-report-file<file> Instructs the compiler to generate an optimization report named ***<file>***.

-opt-report-level{min|med|max} Specifies the level of detail for the optimization report.

-opt-report-phase<phase> Specifies the optimizer phase **<phase>** to generate reports for. <*phase*> can be one of the following:

- **ecg_swp** : Code generator / software pipelining
- **hlo** : high level optimizer
- **ipo** : interprocedural optimizer

-opt-report-help Displays the logical names of optimizer phases available for report generation (using -opt-report-phase).

2.2.7 Compiling Tips

Consider both the -02 and -03 options

The best compiler options are very much dependent on the nature and structure of the program. The length of the vectors involved can be crucially important. In some circumstances the aggressive **–O3** optimizations may be counter-productive and generate inefficient code which does not match the expected performance in terms of time and resource use.

The less sophisticated option **-O2** generates more conservative code but may have a lower overhead.

Be careful when using loop unrolling options

The loop unrolling options – see section 2.2.2 can be counter productive in terms of performance. Register usage will be increased due to the need to store more temporary variables and code size will increase following unrolling, which is particularly undesirable for embedded applications.

These costs will have to be weighed up against the benefits achieved in terms in the reduction of the number of loop iterations for the program.

Try the option -O3 -unroll0

If a binary file compiled using the **-O2** option performs better than a binary compiled with the **-O3** option, it is often worth considering the combination **' -O3 -unroll0'**.

The implementation of unrolling when switching from **-O2** to **-O3** may prove to be counter-productive – see above. However, some, if not all, of other **-O3** optimization routines could be beneficial. This means that, generally speaking (depending on the program), the combination **-O3 -unroll0** may be the most effective.

Look at floating-point assist faults.

Floating-point assist faults (**FPAF**) are a mechanism which makes it possible to treat calculations implementing denormalized numbers (floating numbers with a zero mantissa). If these cannot be handled directly by the processor, then the OS will intervene with specific functions, leading to a potentially high time penalty. To see if the application generates **FPAFs**, use the command **dmesg** which shows system messages. The messages are of the type:

```
a.out(27243): floating-point assist fault At IP 4000000000032461, isr
0000020000000008
```

It should be noted that each line of this type may correspond to a variable number of occurrences. FPAF problems may be avoided as follows:

- By using the **-ftz** option, which changes denormalized numbers to zero. This is included as a default option with the **-O3** option, but not with lower optimization settings.

Chapter 3. Program Execution Optimization

This chapter contains a description of various ways that the execution of the program on the Bull HPC platforms can be made as smooth as possible exploiting all the computing power that is available. For information on the different platforms, application types, launching tools and specific execution optimization options refer to the Bull **BAS5 for Xeon** *User's Guide*.

The following topics are described:

- 3.1 *CPUSET*
- 3.2 *Tuning Performance for SLURM clusters*
- 3.3 *Avoiding Memory Access Stalls*

3.1 CPUSET

CPUSETs are lightweight objects in the **Linux** kernel that enable users to partition their multiprocessor machine by creating execution areas. A virtualization layer has been added so it becomes possible to split a machine in terms of CPUs.

The main motivation of this patch is to give the **Linux** kernel full administration capabilities concerning CPUs. CPUSETs are rigidly defined, and a process running inside this predefined area won't be able to run on other parts of the system.

This is useful for:

- Creating sets of CPUs on a system, and binding applications to them.
- Providing a way of creating sets of CPUs inside a set of CPUs so that a system administrator can partition a system among users, and users can further partition their partition among their applications.

3.1.1 Typical Usage of CPUSETS

- CPU-bound applications: Many applications (as it is often the case for HPC apps) used to have a "one process on one processor" policy using **sched_setaffinity()** to define this, but what if we have to run several such apps at the same time? One can do this by creating a CPUSET for each app.
- Critical applications: processors inside strict areas may not be used by other areas. Thus, a critical application may be run inside an area with the knowledge that other processes will not use its CPUs. This means that other applications will not be able to lower its reactivity. This can be done by creating a CPUSET for the critical application, and another for all the other tasks.

3.1.2 BULL CPUSETS

CPUSETS are integrated in the standard **Linux** kernel. However the **Bull** kernel includes the following additional CPUSET features:

Migration

Change on the fly the execution area for a whole set of processes (for example, to give more resources to a critical application). When you change the CPU list of a CPUSET all the processes that belong to the CPUSET will be migrated to stay inside the CPU list, if and as necessary.

Virtualization

Translate the masks of CPUs given to **sched_setaffinity()** so they stay inside the set of CPUs. With this mechanism processors are virtualized for the use of **sched_setaffinity()** and **/proc** information. Thus, any former application using this **system call** to bind processes to processors will work with virtual CPUs without any change. A new file is added to each CPUSET, in the CPUSET file system, to allow a CPUSET to be virtualized, or not.

3.1.3 pplace

pplace is a tool which offers finer control over the binding of threads and processes of an application to individual CPUs than **CPUSET**.

It may be used when using **OPENMP** for Benchmarking. **OpenMP** is an industry-standard parallel programming model which implements a fork-join model of parallel execution. With **OPENMP** the source thread or process is split into several parallel threads or processes. These include threads used for calculating and a monitor thread which controls the other threads. Care is required to bind the calculation threads to the CPUs using **pplace** only and not the monitoring threads.

SYNOPSIS

pplace -np <nb_cpus> -p <policy> [--name <process_name>] <command>

pplace will create a **CPUSET**, enable the process placement policy inside this **CPUSET**, and run the <command> inside this **CPUSET**.

OPTIONS

-np <nb_cpus>
Specify how many CPUs the application will use. A new **CPUSET**, with this number of CPUs will be created.

-p <policy>
Specify the placement policy-this policy is actually a comma-separated list of per-task policies. These policies can be:

ignore this task	(nothing)
bind task on next cpu	+
bind task on specific cpu	cpu number

The last policy becomes the default policy for all the tasks which follow. For instance:

-p 0,+ will place the first task on cpu0, the second task on cpu1, the third on cpu2, and so on.

-p 0,,,+ will place the first task on cpu0, ignore the second and third tasks, place the fourth task on cpu1, the fifth on cpu2, and so on.

-p 1, will place the first task on cpu1, and will ignore all other tasks.

--name <process_name> will only consider processes with name <process_name> for the placement. Note: only the 15 first characters of the name are taken into account.

-d debug. When the command terminates, **pplace** will print detailed information about how the process placement occurred. This can help you to choose your policy.

For the application developer individual calls to CPUs can be made in the source code using the command **Sched_setaffinity** which operates in the same way as **pplace**. The advantage which **pplace** offers is that this fixing of processes and threads can be made on the binaries without modifying the source code.

When the compiler uses **OPENMP** pragmas to generate a multithreaded application it uses runtime libraries from **Intel** and it is not possible to add individual calls in the manner of the **Sched_setaffinity** command. In this instance it may be advantageous to use **pplace** to control the CPU allocation.

3.2 Tuning Performance for SLURM clusters

3.2.1 Configuring and Sharing Consumable Resources in SLURM

SLURM, using the default node allocation plug-in, allocates nodes to jobs in exclusive mode, which means that even when all the resources within a node are not utilized by a given job, another job will not have access to these resources.

Nodes possess resources such as processors, memory, swap, local disk, etc. and jobs consume these resources. The **SLURM** exclusive use default policy may result in inefficient utilization of the cluster and of node resources.

SLURM provides a Consumable Resource plug-in which supports CPUs, Sockets, Cores, and Memory being configured as consumable resources. For 1.3+ SLURM versions consumable resources may be shared among jobs by the use of the per-partition **shared** setting.

See https://computing.llnl.gov/linux/slurm/documentation.html for details regarding the configuration and sharing of consumable resources for **SLURM**.

3.2.2 SLURM and Large Clusters

This section contains **SLURM** administrator information specifically for clusters containing 1,024 nodes or more. Virtually all **SLURM** components have been validated (through emulation) for clusters containing up to 65,536 compute nodes. Obtaining good performance at this scale requires some tuning and this section provides some basic information with which to get started.

3.2.2.1 Node Selection Plug-in (SelectType)

While allocating individual processors within a node is great for smaller clusters, the overhead of keeping track of the individual processors and memory within each node adds a significant overhead. For best scalability, it is recommended that the consumable resource plug-in ***select/cons_res*** is used and NOT ***select/linear***.

3.2.2.2 Job Accounting Gather Plug-in (JobAcctGatherType)

Job accounting relies on the **slurmstepd** daemon to periodically sample data on each Ccompute Node. The collection of this data will take compute cycles away from the application, inducing what is known as *system noise*. For large parallel applications, this system noise can impact application scalability.

For optimal application performance, it is best to disable job accounting, **jobacct_gather/none**. Consider the use of the job completion records parameter, **JobCompType,** for accounting purposes, as this entails far less overhead.

If job accounting is required, configure the sampling interval to a relatively large size (e.g. *JobAcctGatherFrequency=300*). Some experimentation may also be required to deal with collisions on data transmission.

3.2.2.3 Node Configuration

SLURM can track the amount of memory and disk space available for each Compute Node and use it for scheduling purposes, however this will entail an extra overhead. Optimize performance by specifying the expected configuration using the parameters that are available (**RealMemory, Procs**, and **TmpDisk)**. If the node is found to have fewer resources than the configured amounts, it will be marked as **DOWN** and not be used. Also, the **FastSchedule** parameter should be set.

While **SLURM** can easily handle a heterogeneous cluster, configuring the nodes using the minimal number of lines in the **slurm.conf** file will make administration easier and result in better performance.

3.2.2.4 Timers

The configuration parameter **SlurmdTimeout** determines the interval at which **slurmctld** routinely communicates with **slurmd**. Communications occur at half the **SlurmdTimeout** value. If a Compute Node fails, the time of failure is identified and jobs are no longer allocated to it. Longer intervals decrease system noise on Compute Nodes (these requests

are synchronized across the cluster, but there will be some impact on applications). For large clusters, **SlurmdTimeoutl** values of 120 seconds or more are reasonable.

3.2.2.5 MPICH2

If MPICH-2 is used, the **srun** command will manage the key-pairs used to bootstrap the application. Depending upon the processor speed and the architecture, the communication of key-pair information may require extra time. This can be done by setting the **PMI_TIME** environment variable before executing **srun** to launch the tasks. The default value of **PMI_TIME** is 500 and this is the number of microseconds allotted to transmit each key-pair.

The individual **slurmd** daemons on the Compute Nodes will initiate messages to the **slurmctld** daemon only when they start up or when **epilog** completes for a job. When a job that has been allocated a large number of nodes completes, a large number of messages may be sent by the **slurmd** daemons on these nodes to the **slurmctld** daemon, all at the same time. The *EpilogMsgTime* parameter may be used to spread this message traffic out over time and avoid message loss. Lost messages will be retransmitted, however this results in a delay in reallocating resources to new jobs.

3.2.2.6 TreeWidth parameter

SLURM uses hierarchical communications between the **slurmd** daemons in order to increase parallelism and improve performance. The *TreeWidth* configuration parameter controls the fanout of messages. The default value is 50, meaning each **slurmd** daemon can communicate with up to 50 other **slurmd** daemons and up to 2500 nodes can be contacted with two message hops. The default value will work well for most clusters. Optimal system performance can usually be achieved if *TreeWidth* is set to the square root of the number of nodes in the cluster for systems having no more than 2500 nodes, or the cube root for larger systems.

3.2.2.7 Hard Limits

The **srun** command automatically increases its open file limit to the hard limit in order to process all the standard input and output connections to the launched tasks. It is recommended that you set the open file hard limit to 8192 across the cluster.

3.2.3 SLURM Power Saving Mechanism

SLURM provides an integrated power saving mechanism from version 1.2.7 onwards. Nodes that remain idle for a configurable period of time can be placed in a power saving mode. The nodes will be restored to normal operation once work is assigned to them. Power saving is accomplished using the ***cpufreq*** governor that can change CPU frequency and voltage. Note that the *cpufreq* driver must be enabled in the Linux kernel configuration. While the **ondemand** governor can be configured to automatically alter the CPU performance based upon workload, **SLURM** provides somewhat greater flexibility for power management on a cluster. **SLURM** can alter the governors across the cluster, according to configurable rates, to prevent rapid changes in power demands. For example, starting a 1000 node job on an idle cluster could result in a high power surge.

This would be better supported using SLURM's options of increasing power demands in a gradual fashion.

3.2.3.1 Configuring Power Saving

A great deal of flexibility is offered in terms of when, and how, idle nodes are put into or removed from power save mode. The following configuration parameters are available:

- **SuspendTime**: Nodes becomes eligible for power saving mode after being idle for the number of seconds specified. A negative number disables power saving mode. The default value is -1 (disabled).
- **SuspendRate**: Maximum number of nodes to be placed into power saving mode per minute. A value of zero results in no limits being imposed. The default value is 60. Use this to prevent rapid drops in power requirements.
- **ResumeRate**: Maximum number of nodes to be placed into power saving mode per minute. A value of zero results in no limits being imposed. The default value is 60. Use this to prevent rapid increases in power requirements.
- **SuspendProgram**: Program to be executed to place nodes into power saving mode. The program executes as ***SlurmUser***, as configured in ***slurm.conf***. The argument to the program will be the names of nodes to be placed into power saving mode, using SLURM's hostlist expression format.
- **ResumeProgram**: Program to be executed to remove nodes from power saving mode. The program executes as ***SlurmUser***, as configured in ***slurm.conf***. The argument to the program will be the names of nodes to be removed from power saving mode, using SLURM's hostlist expression format.
- **SuspendExcNodes**: List of nodes that are excluded from power saving mode. Use SLURM's hostlist expression format. By default, no nodes are excluded.
- **SuspendExcParts**: List of partitions that are excluded from power saving mode. Multiple partitions may be specified using a comma separator. By default, no nodes are excluded.

While ***SuspendProgram*** and ***ResumeProgram*** execute as ***SlurmUser***. The program can take advantage of this to execute programs directly on the nodes as the *root* user as part of the **SLURM** infrastructure.

Example scripts are shown below:

```
#!/bin/bash
# Example SuspendProgram for cluster where every node has two CPUs
srun --uid=0 --no-allocate --nodelist=$1 echo powersave
>/sys/devices/system/cpu0/cpufreq
srun --uid=0 --no-allocate --nodelist=$1 echo powersave
>/sys/devices/system/cpu1/cpufreq

#!/bin/bash
# Example ResumeProgram for cluster where every node has two CPUs
srun --uid=0 --no-allocate --nodelist=$1 echo performance
>/sys/devices/system/cpu0/cpufreq
srun --uid=0 --no-allocate --nodelist=$1 echo performance
>/sys/devices/system/cpu1/cpufreq
```

The **srun –no-allocate** option allows ***SlurmUser*** and the **root** user only to spawn tasks directly on the Compute Nodes without actually creating a SLURM job. No other users have this permission (their requests will generate an invalid credential error message and the event will be logged). The **srun –uid** option allows ***SlurmUser*** and the **root** user only to execute a job as some other user. Then ***SlurmUser*** uses the **srun –uid** option, the **srun** command will try to set its user ID to that value in order to fully operate as the specified user. This will fail and **srun** will report an error to that effect. This does not prevent the spawned programs from running as **root** user. No other users have this permission (their requests will generate an invalid user id error message and the event will be logged).

The **slurmctld** daemon will periodically (every 10 minutes) log how many nodes are in power save mode using messages of this sort:

```
[May 02 15:31:25] Power save mode 0 nodes
...
[May 02 15:41:26] Power save mode 10 nodes
...
[May 02 15:51:28] Power save mode 22 nodes
```

Using these logs you can easily see the effect of SLURM's power saving support. You can also configure **SLURM** without **SuspendProgram** or **ResumeProgram** values to assess the potential impact of power saving mode before enabling it.

3.3 Avoiding Memory Access Stalls

For an application to be truly optimized it must run as fast as is possible on the system and at the same time avoid any possible memory access stalls whilst a data process is fetched from memory.

Compilers will automatically generate code optimized to exploit the maximum parallel processing capability possible with groups of instructions sequenced correctly. In many cases the easiest thing to do is leave the compiler to handle all the parallel optimizations.

Memory access stalls are caused when the data to be accessed is not loaded beforehand into the cache. This typically occurs when the memory access is random and not sequential. Two data structures which often use random-like data access are linked lists and hash table arrays.

Linked lists

Create the nodes in pre-allocated blocks of memory rather than taking them from the heap each time. This increases data locality which means that there is a good chance that the data node is already in memory as part of a data block which has already loaded.

Hash tables

In order to prevent data access collisions when re-hashing new slots are allocated at random, with the result that the new slots are unlikely to be in the cache, which in turn leads to a memory access stall.

These stalls can be prevented as follows:

- Ensure data can be accessed sequentially. **Fortran** arrays should be accessed in column-major order whilst C arrays should be accessed in row-major order.
- If the data is such that it cannot be accessed sequentially then preload it using the --**builtin_prefetch** command. However, care has to be taken to ensure that when the data is preloaded it does not displace data which is in the process of being used. Otherwise, the displaced data will have to be reloaded constantly and thus impact performance.

Chapter 4. Message Passing Interface Optimization

This chapter looks at some optimization tips for the Message Passing Interface (MPI).

The following topics are described:

- 4.1 *Introduction*
- 4.2 *General Tips for MPI_Bull Usage*
- 4.3 *MPI-2 One-Sided Operations*
- 4.4 *mpibull2-params*

4.1 Introduction

Bull has developed a complete solution which helps to enable the full exploitation of NovaScale HPC platforms. To fully utilize the power of a cluster it is necessary to ensure that the application programs are executed in parallel and to take into account environmental factors including the part played by distributed and shared memory.

Bull's **BAS** (Bull Advanced Server) environment is dedicated to parallel programming and includes:

- **MPI** (Message Passing Interface) libraries **MPI_Bull2**– for more details on these libraries see the **BAS5 for Xeon** *User's Guide*.
- Performance monitoring tools including **HPC Toolkit.**
- The MPI profiling tool **profilecomm** is supplied as part of the **MPI_Bull** library and is used to identify hotspots or bottlenecks within the message passing for the application – see chapter 1 for more information on the data which **profilecomm** provides.
- Debugging/compilation tools.

4.1.1 MDM Optimization Tools

The Bull **MPI Data Mover Module (MDM)** which is incorporated in MPI_Bull2 library includes a trace tool, a profiling tool and a KDB module which may all be used for **MPI** profiling and optimization purposes.

The **trace tool** logs the most recent events for each processor. This tool which has the advantage of not influencing the behavior of the application helps to solve problems of concurrent access to data and of synchronization of processes.

The **KDB module** allows access to traces made when a crash occurs and thus enables more efficient error detection.

The **profiling tool** may be used to identify the critical parts of the application code. Its implementation is similar to that of the trace tool and involves a low loss of performance.

4.2 General Tips for MPI_Bull Usage

There follows some suggestions and points to be kept in mind when configuring the Message Passing Interface using the **MPI_Bull** libraries:

Wait until the sent buffer exchange is finished before modifying it

It is worth looking at the use of the synchronous and asynchronous exchanges when using **MPI_Bull**.

The developer may prefer to use asynchronous exchanges if he wishes to use the compute nodes for another application at the same time that the data is being exchanged for the first one.

Certain implementations of **MPI**, like **MPICH** use the device **ch_shmem**. This uses shared memory as an exchange zone and will tolerate the modification of the sending buffer before the call **MPI_WAIT** runs.

MPICH copies the buffer sent into a shared memory area when the **send** call is used and then the receiving process will copy this buffer into its own receiver buffer. Therefore two copies of the buffer are made and the buffer sent may be modified just after the **send** call without there being any consequence for the exchange.

MPI_Bull has a zero-copy mechanism provided by the **MDM** module which uses only one copy of the buffer to make the exchange. This module makes it possible for messages and data greater than 32Kb in size to be copied directly into the memory of the distant process.

However, it is necessary to pay close attention to buffer use when doing this. If an asynchronous exchange is initialized, it is absolutely prohibited to modify the buffer sent before the exchange is finished, i.e. the buffer sent should only be modified once the call **MPI_WAIT** is finished – this guarantees that the exchange is finished.

Look at the stack size for memory intensive applications

The definition of static buffers does not pose any problem for **MPI_Bull**. It is advisable, however, to look at the size of the stack for applications using a lot of memory, as it may not be large enough. A characteristic symptom of this type of problem is that the application is blocked, without an error message appearing. In this case, it is possible to modify the size of the stack by using the command:

```
$ ulimit – s unlimited
```

This command allows the stack size for the system to be modified. Care should be taken as the unlimited size extends up to the limit of the size of the hardware stack, which cannot be increased. This command can also be used to ascertain the stack size.

If possible avoid the MPI_Bsend function

The use of the **MPI_Bsend** function is possible with **MPI_Bull**, however it is not recommended. Although it can seem attractive as it excludes blockages, it uses an additional buffer (part of **MPI_Buffer_Attach**) into which the data to be sent is recopied. This use of an additional copy of the buffer can have a big impact on performance.

It is better to re-examine an algorithm which falls into deadlock and to use either synchronous or asynchronous **send** calls, rather than using a set-up with **MPI_Bsend**.

Use MPI_Sendrecv rather than MPI_Send and MPI_Recv

When implementing a parallel algorithm, it is important to keep in mind the possible simplifications that the **MPI_Bull** offers. The use of successive **sends** and then **recvs** between several processes may be an example of a complicated algorithm which could be simplified, and may even lead to errors of implementation and execution. In this case, you should verify that the send and receive calls are coordinated correctly. The use of **MPI_Sendrecv** leaves **MPI_Bull** to manage the exchange of the **send** and **recv** calls.

As far as is possible, use the **MPI_Sendrecv** call instead of successive calls to **MPI_Send** and then to **MPI_Recv**.

Use Collective operations whenever possible

It is worthwhile to bear in mind the collective operations which are possible with **MPI_Bull** and to simplify the code as much as is possible to take these into account. Often, a succession of point-to-point operations can be restructured and changed into a collective operation.

Do not use ANY_SOURCE

In accordance with the MPI-1 standard, it is possible to not specify the source for point-to-point operations, but to use the variable **ANY_SOURCE** which allows the "first source arising in the exchange" to be used.

As far as possible, it is advised to use an explicit source and not the **ANY_SOURCE** variable, which includes an additional overhead and consequently an impact on performance.

Whenever possible use intra-QBB transfers

The **MPI_Bull** library helps to minimize the overhead of data exchanges.

For messages greater than 32Kbs the **MPI_Bull** library uses the **MDM** module and the zero-copy mechanism to transfer the messages.

There is a light overhead for the zero-copy mechanism as this is done through the use of system calls but this is more efficient in terms of performance than the use of shared buffer zones.

For messages smaller than 32Kbs in size, the transfer of the data is carried out through the shared memory buffer and not through the **MDM** module –see the *Bull HPC BAS5 for Xeon User's Guide*.

For messages smaller than 32Kbs in size a shared memory zone is created on each QBB to optimize intra-QBB data transfer. It is advisable to prioritise the use of intra-QBB transfers as these provide better performance than inter-QBB transfers.

4.3 MPI-2 One-Sided Operations

Regarding the **MPI-2 One-Sided** functionality present in **MPI_Bull**, it is important to understand the implementation choices which were made and to have an idea of how the program works so that possible improvements in performance can easily be identified.

The **MPI-2** standard stipulates that **MPI_PUT**, **MPI_GET** and **MPI_ACCUMULATE** operations should be completed before the return call of corresponding synchronization function (e.g. **MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE,** etc).

Accordingly, Bull chose to program the MPI library so that the data exchange is carried out immediately the **MPI_PUT**, **MPI_GET** and **MPI_ACCUMULATE** functions are called.

4.4 mpibull2-params

mpibull2-params is a tool that is used to list/modify/save/restore the environment variables that are used by the **mpibull2** library and/or by the communication device libraries (**InfiniBand**, ….). The behaviour of the **mpibull2** MPI library may be modified using environment variable parameters to meet the specific needs of an application. The purpose of the **mpibull2-params** tool is to help **mpibull2** users to manage different sets of parameters. For example, different parameter combinations can be tested separately on a given application, in order to find the combination that is best suited to its needs. This is facilitated by the fact that **mpibull2-params** allow parameters to be set/unset dynamically.

Once a specific combination of parameters has been tested and found to be good for a particular context, they can be saved into a file by a mpibull2 user. Using the **mpibull2-params** tool, this file can then be used to restore the set of parameters, combined in exactly the same way, at a later date.

Notes

- The effectiveness of a set of parameters will vary according to the application. For instance, a particular set of parameters may ensure low latency for an application, but reduce the bandwidth. By carefully defining the parameters for an application the optimum, in terms of both latency and bandwidth, may be obtained.
- Some parameters are located in the **/proc** file system and only super users can modify them.

The entry point of the **mpibull2-params** tool is an internal function of the environment. This function calls an executable to manage the MPI parameter settings and to create two temporary files. According to which shell is being used, one of these two files will be used to set the environment and the two temporary files will then be removed. To update your environment automatically with this function, please source either the **$MPI_HOME/bin/setenv_mpibull2.sh** file or the **$MPI_HOME/bin/setenv_mpibull2.csh** file, according to which shell is used.

4.4.1 The mpibull2-params command

SYNOPSIS

```
mpibull2-params <operation_type> [options]
```

Actions

The following actions are possible for **mpibull2-params** command:

-l List the MPI parameters and their values

-f List families of parameters

-m Modify a MPI parameter

-d Display all modified parameters

-s Save the current configuration into a file

-r Restore a configuration from a file

-h Show help message and exit

Options

The following options and arguments are possible for the **mpibull2-params** command.

Note The options shown can be combined, for example, **-li** or can be listed separately, for example **–l –i.** The different option combinations for each argument are shown below.

-l [iv] [PNAME]

List current default values of all MPI parameters. Use the PNAME argument (this could be a list) to specify a precise MPI parameter name or just a part of a name. Use the **-v** (verbose) option to also display all possible values, including the default. Use the **-i** option to list all information.

Examples

```
mpibull2-params -l all shm
```

This will list all the parameters with the string 'all' or 'shm' in their name. **mpibull2-params -l | grep -e all -e shm** will return the same result.

```
mpibull2-params -li all
```

This will display all information - possible values, family, purpose, etc. for each parameter name which includes the string 'all'. This command will also indicate when the current value has been returned by **getenv()** i.e. the parameter has been modified in the current environment.

```
mpibull2-params -lv rom
```

This will display current and possible values for each parameter name which includes the string 'rom'. It is practical to run this command before a parameter is modified.

-f [l[iv]] [FNAME]

This will list all the default family names. Use the FNAME argument (this could be a list) to specify a precise family name or just a part of a name. Use the **-l** option to list all parameters for the family specified. **–l**, **-v** and **-i** options are as described above.

Examples

```
mpibull2-params -f band
```

List all family names with the string 'band' in their names.

```
mpibull2-params -fl band
```

For each family name with the string 'band' inside, list all the parameters and current values.

-m [v] [PARAMETER VALUE]

Modify a MPI PARAMETER with VALUE. The exact name of the parameter should be used to modify a parameter. The parameter is set in the environment, independently of the shell syntax (**ksh/csh**) being used. The keyword 'default' should be used to restore the parameter to its original value. If necessary, the parameter can then be unset in its environment. The **-m** operator lists all the modified MPI parameters by comparing all the MPI parameters with their default values. If none of the MPI parameters have been modified then nothing is displayed. The **–m** operator is like the **-d** option. Use the **-v** option for a verbose mode.

Examples

```
mpibull2-params -m mpibull2_romio_lustre true
```

This will set the ROMIO_LUSTRE parameter in the current environment.

```
mpibull2-params -m mpibull2_romio_lustre default
```

This will unset the ROMIO_LUSTRE parameter in the environment in which it is running and returns it to its default value.

-d [v]

This will display the difference between the current and the default configurations. Displays all modified MPI parameters by comparing all MPI parameters with their default values.

-s [v] [FILE]

This will save all modified MPI parameters into FILE. It is not possible to overwrite an existing file, an error will be returned if one exists. Without any specific arguments, this file will create a file named with the date and time of the day in the current directory. This command works silently by default. Use the **-v** option to list all modified MPI parameters in a standard output.

Example

```
mpibull2-params -sv
```

This command will, for example, try to save all the MPI parameters into the file named Thu_May_10_15_50_28_2007.

Output Example

```
save the current setting :
mpibull2_mpid_xxx=1
1 parameter(s) saved.
```

-r [v] [FILE]

Restore all the MPI parameters found in FILE and set the environment. Without any arguments, this will restore all modified MPI parameters to their default value. This command works silently, in the background, by default. Use the **-v** option to list all restored parameters in a standard output.

Example

```
mpibull2-params -r
```

Restore all modified parameters to default.

-h

Displays the help page

4.4.2 Family names

The command **mpibull2-params –f** will list the parameter family names that are possible for a particular cluster environment.

The parameter families which are possible for Bull HPC are listed below.

Quadrics_libElan_driver
LK_Ethernet_Core_driver
LK_IPv4_route
LK_IPv4_driver
OpenFabrics_IB_driver

Marmot_Debugging_Library
MPI_Collective_Algorithms
MPI_Errors
CH3_drivers
CH3_drivers_Shared_Memory
Execution_Environment
Elan
Elan_Hooks
Infiniband_RDMA_IMBR_mpibull2_driver
Infiniband_Gen2_mpibull2_driver
UDAPL_mpibull2_driver
IBA-VAPI_mpibull2_driver
MPIBull2_Postal_Service
MPIBull2_Romio

Run the command **mpibull2-params <fl> <family>** to see the list of individual parameters that are included in the parameter families used within your cluster environment.

Chapter 5. Lustre File System Optimization

This chapter describes how the **Lustre** parallel file system should be optimized.

The following topics are described:

- *5.1 Parallel File Systems - Introduction*
- *5.2 Monitoring Lustre Performance*
- *5.3 Lustre Optimization - Administrator*
- *5.4 Lustre Optimization – Application Developer*
- *5.5 Lustre File System Tunable Parameters*

5.1 Parallel File Systems - Introduction

To be fully optimized large cluster needs all its storage devices, and file systems of the input/output sub-system, to work in parallel with very high I/O rates and capable of accessing many processors at once. A distributed file system such as NFS is not sufficient for the requirements of the system.

A parallel file system provides network access to a file system distributed across different disks or storage devices on multiple independent servers or I/O nodes. Real files are split into several chunks of data or stripes, each stripe being written onto a different component in a cyclic distribution manner (striping).

For a parallel file system based on a client/server model, the servers are responsible for file system functionality and the clients provide access to the file system through a "mount" procedure. This mechanism provides a consistent namespace across the cluster and is accessible via the standard Linux I/O API.

I/O operations occur in parallel across multiple nodes in the cluster simultaneously. All files are spread across multiple nodes including the I/O buses and disks, therefore I/O bottlenecks are reduced and the overall I/O performance is increased.

For large cluster configurations HPC **Bull** integrates the **Lustre** parallel file system which is an open-source, object-based, Linux-based, POSIX-compliant system that offers very high performance.

There are separate sections in this chapter for the system administrator and the application developer. However, these are not exclusive, as any optimization of the Lustre file system will involve collaboration between these two. A lot of the optimizations need to be put in place when the platform is configured initially, and many aspects of application tuning cannot be done with user rights alone.

See The *Administrator's Guide* for a description of the different parts of the **Lustre** file system architecture and the command syntax.

5.2 Monitoring Lustre Performance

The I/O performance of an application depends on:

1. The application itself, in particular how Input/Output data is sent to the File System
2. The performance of the system including the Linux kernel and drivers, and the Lustre file system.
3. Hardware performance including networking cards, disk arrays and storage devices.

These three aspects are interrelated and the overall performance for an application on a cluster depends on having a balance between all three. Different means exist for monitoring the performance of the file system and for identifying potential improvements.

5.2.1 Ganglia

Ganglia is a scalable, distributed, open-source monitoring tool, and is included as part of **Bull System Manager – HPC Edition**. This provides information about the cluster distribution for the **Lustre** file system and can be used for monitoring its performance in real time. This is important as the file system may be used by several different users at the same time, and if there is a perceptible drop in performance then **Ganglia** will indicate where there is uneven access to the files. Ganglia also collects **Lustre** statistics from **/proc/fs/lustre** in order to measure different aspects of file system performance.

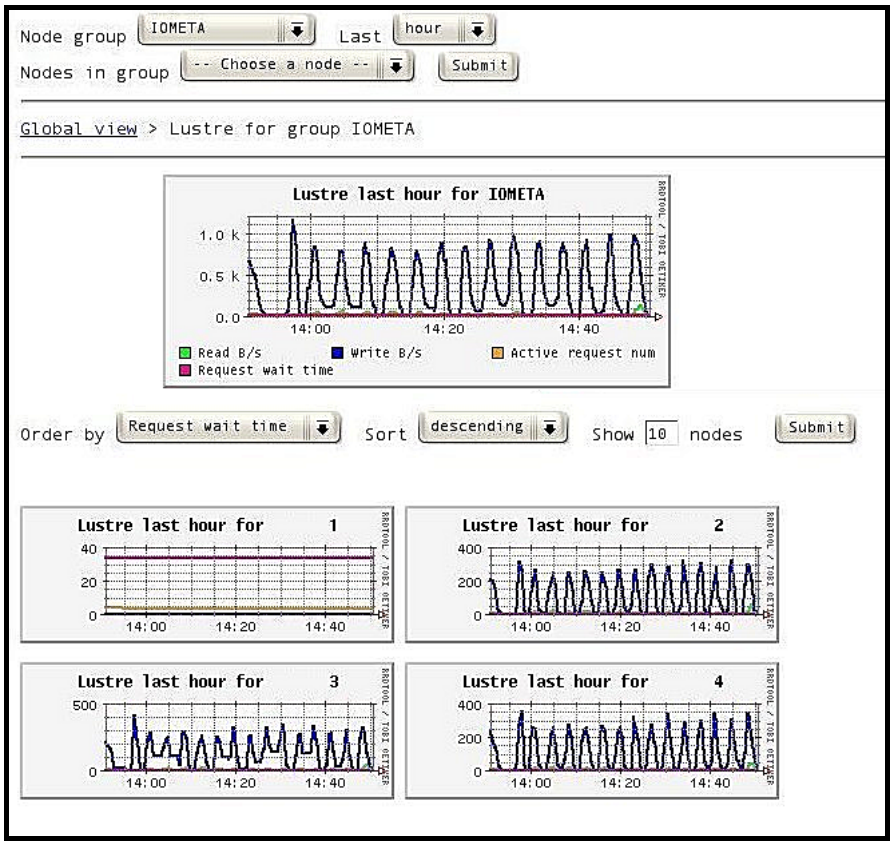

Figure 5-1 Ganglia Lustre monitoring statistics for a group of 4 machines with total accumulated values in top graph

5.2.2 Lustre Statistics System

Lustre itself collects a range of statistics. These are available in files in the proc filesystem. A list of these files can be retrieved by using the command:

```
find /proc/fs/lustre –name “*stats*”
```

5.2.3 Time

Time command – see Chapter 1 of this manual. Here are two examples with the **dd** command which is used exclusively for I/O operations.

Example 1

```
# time dd if=/dev/zero of=/tmp/testfile bs=1M count=100
```

```
100+0 records in
100+0 records out
104857600 bytes transferred in 0.738386 seconds (142009176 bytes/sec)

real    0m0.749s
user    0m0.001s
sys     0m0.476s
```

If the system time is high, as in example 1, then this is an indication that the resources being used for the application I/O are high.

Example 2

```
# time dd if=/dev/zero of=/home/testfile bs=1M count=1000
```

```
1000+0 records in
1000+0 records out
1048576000 bytes transferred in 44.987850 seconds (23307982 bytes/sec)

real    0m45.039s
user    0m0.012s
sys     0m5.262s
```

If the sum of the CPU user time and the system time is significantly lower than real time, as in example 2, then this may be an indication that, again, the resources being used for the application I/O operations are high.

5.2.4 Iostat

When the host kernel has been configured to provide detailed I/O stats per partition the following information is available.

iostat can provide insight into the nature of I/O bottlenecks. It provides the nature and concurrency of requests being made of the attached storage devices.

iostat -x is invaluable for profiling the load on the storage devices which are part of a server node. The raw throughput numbers (wkB/s) combined with the requests per second (w/s) gives the average size of I/O requests to the device.

The service time indicates the amount of time it takes the device to respond to an I/O request. This sets the maximum number of requests that can be handled in turn when requests are not issued concurrently. Comparing this with the requests per second gives a measure of the amount of storage device concurrency.

Refer to the **iostat** man page for details regarding the meaning of the various columns in the output.

5.2.5 Llstat

llstat is a command which allows the examination of some of the Lustre statistics files. It 'decodes' the content by calculating statistics (min, max, mean, standard deviation) based on the contents of the file (sum and sum of squares).

See The Lustre Operations Manual available on http://manual.lustre.org Chapter 32.5.11 for more information.

5.2.6 Vmstat

The CPU use columns in the **vmstat** output file can be used to identify a node whose CPUs are completely occupied. On Metadata servers (**MDS**) and Object Storage Server (**OSS**) nodes, the I/O columns tell you how many blocks are flowing through the node's I/O subsystem. Coupled with the details of the attached storage for the node, it is then possible to determine if a subsystem in the node is the bottleneck.

Vmstat does not provide I/O block flow details for clients.

The columns that report swap activity can identify nodes that are having trouble keeping their working applications in memory.

5.2.7 Top

This tool is used to identify tasks which are using a lot of system resources. It can also be used to identify tasks which are not generating file system load, because they are using CPU or server threads which are struggling to obtain system resources on an overloaded node.

See *Chapter 1* for more details.

5.2.8 Strace

The **strace** command intercepts and records the system calls called and received by a running process and is used to measure I/O activity at the system level.

Two options which are useful are:

-e trace=file traces activity related to system calls for file activity

-tttT gives a microsecond resolution.

These two options combined allow the measurement and evaluation of the performance of file system calls. However whilst **strace** is being used the performance of the application may be impacted. It is possible to use **strace** command for each system call. For example, use **–e trace=write** option to analyze the write performance.

5.2.9 Application Code Monitoring

It is possible to add system calls in the code during the development of the application which can then be used to measure the I/O file performance of the application itself.

Another option is that any I/O operation debugging traffic is included within the **NFS** system and not the **Lustre** system in order to minimize any additional overhead in the use of system resources.

Note Increasing the debug level can lead to a major performance impact. Full debugging can slow the system by as much as 90%, compared to the default settings.

Benchmarking tools are easier to work with to study performance. Once the tuning changes have been made, a general purpose benchmarking tool can then be used to check for any adverse effects.

5.3 Lustre Optimization - Administrator

The **Lustre** system administrator will need to monitor the file system to check the overall performance, and to identify any areas where there may be possible degradation in the service. See the *Bull HPC Administrator's Guide* for more details. **Lustre** includes tools to monitor I/O performance. These can be used to evaluate any changes that are made to the application, and also to see if, and where, performance could be improved.

The application developer and large cluster administrator will need to be clear in their definition of the needs for the application at the outset, as some of the system configuration settings made when installing and configuring the system will impact the performance of the application. Some flexibility is possible, as there are parameters which can be modified after the cluster has been configured to meet the particular requirements of the application.

The developer needs to be aware that the Input/Output algorithms, specified for the application will have a big impact on performance and some performance compromises may have to be made for different parts of the application with respect to the overall performance for the complete program.

The main bottleneck within the whole system normally is the I/O speed of the data storage devices, as this is usually the slowest part of a cluster.

Note It may be possible to observe superlinear speedups for the I/O throughput using **Lustre** client cache.

Raw benchmarking data, including control data should be available for the systems. The objective is to get as close to these performance figures as is possible with the application in place.

Lustre is ideal for large sequential write I/O operations as used, for example, by checkpoint/restart. Using **Lustre** it should be possible to obtain 80 to 90% of the raw I/O performance figures (Write operations generally perform better than read operations).

Attention is particularly needed when small random I/O Metadata operations are being performed. This is because the data may be unevenly distributed throughout the system.

Several points have to be kept in mind when attempting to tune the file system:

- **Lustre** is a part of a shared file system which means that it will be difficult to obtain exclusive use of Interconnects and data storage devices. For clients who need to have exclusive use of the file system, it is possible to do this by mounting it directly on the clients. This is in contrast to CPUs and memory where an application can be given exclusive use easily. Overall there are a lot of variables which can impact an application's performance.
- System caches can have a positive effect on performance if all the I/O traffic takes place in the client cache – in fact it is possible that the application bandwidth may appear greater than the disk bandwidth. System caches can also have a negative effect as **Lustre**'s **readahead** option may impact performance.

- There are a lot of data pipelines within the Lustre architecture. Two in particular have a direct impact on performance. Firstly, the network pipe between clients and OSSs, and secondly the disk pipe between the OSS software and its backend storage. **Balancing these two pipes maximizes performances**.

The **Lustre** file system stores the file striping information in extended attributes (EAs) on the MDT. If the file system has large-inode support enabled (> 128bytes), then EA information will be stored inline (fast EAs) in the extra available space.

The table below shows how much stripe data can be stored inline for various inode sizes:

Inode size (Bytes)	# of stripes stored inline
128	0 (all EA would be stored in external block)
256	3
512	13
1024	35
2048	77
4096	163

Note It is recommended that MDT file systems be formatted with the inode large enough for the default number of stripes per file to improve performance and storage efficiency.

One needs to keep enough free space in the MDS file system for directories and external blocks. This represents ~512 Bytes per inode.

Lustre stripes the file data across the OSTs in a round-robin fashion.

Note It is recommended to stripe over as few objects as possible to limit network overhead and reduce the risk of data loss when there is an OSS failure.

The stripe size must be a multiple of the page size. The smallest recommended stripe size is 1 MB because **Lustre** tries to batch I/O into 1 MB blocks on the network.

5.3.1 Stripe Tuning

Check that enough stripes are being used

It is important to remember that the peak aggregate bandwidth for I/O to a single file is restricted by the number of stripes multiplied by peak bandwidth per server. No matter how many clients try to write to that file, if it only has one stripe, all of the I/O will go to only one server.

Check that files are striped evenly over the Object Storage Targets

Lustre will create stripes on consecutive OSTs by default, so files created at one time will be optimally distributed among OSTs, assuming there are enough stripes and/or files created

at that time. However, files created at different times may not have an optimal distribution among OSTs. To ascertain the file distribution, use the following command:

```
lfs getstripe
```

For more information refer to lfs man page.

If some servers appear to be receiving a disproportionate share of the I/O load check that the files are striped evenly over the OSTs.

If the I/O load is unbalanced for servers then use the lfs command to create a balanced set of files before the application starts, or if applicable, restructure the application so that Lustre striping is more efficient.

See *Chapter 25 - Striping and I/O Options* in the Lustre Operations Manual available on http://manual.lustre.org, for more information on File Striping.

5.4 Lustre Optimization – Application Developer

The main determinant on performance for the **Lustre** file system is the file size and how this is handled by the I/O devices. **POSIX** will handle the parallel distribution of the file, however, if the file is large performance may be impacted. The application developer has to decide if it is possible to chunk the program and thus gain performance.

The optimal level of performance is when the HPC platform I/O device read/write operations is as near as possible to that of the raw **Lustre** file system performance without the application running. Depending on the application program it should be possible to achieve 80% to 90% of the performance of a 'clean' HPC system.

One of the key questions to look at is to ascertain if the application performs I/O from enough client nodes to take full advantage of the aggregate bandwidth provided by the Object Storage Servers.

5.4.1 Striping Optimization for the Developer

Default striping settings are usually in the hands of the Lustre administrator who will normally use the default values. However, the application developer can also change these settings by using the **lfs** command on a per directory or on a per file basis. This controls the way parallel I/O operations are carried out for the files. Refer to the man page displayed under **lfs(1)** for more information.

Optimal striping settings depend primarily on the file size. It does not make sense to stripe a small file over several **OST**s, on the other hand it does make sense to stripe a big file over several **OST**s.

It is recommended to use the default striping settings configured by the Lustre administrator.

If the striping is to be changed, it is best to perform I/O tests with different striping configurations in order to find the best possible striping configuration.

5.4.2 POSIX File Writes

Being a high-performance distributed file system makes **Lustre** especially complex. By being POSIX compliant this complexity is simplified and no code modification is required whether a code is run on a local file system (**ext3, xfs**) or on **Lustre**. Only performance is enhanced.

There are several points to be kept in mind for file writes. Writes flow from the application that generates them to **OST**s where they are placed within the storage system. The network between the client and storage target needs to have capacity for the write traffic. It is also advisable to look for possible choke points along this path.

It should be said that these problems will only occur in extreme cases on the large cluster platform.

There must be enough write capacity for the application

If an application is to exploit a large network and disk pipes, it must generate a lot of write traffic, which can be cached on the client node and packaged into **RPC**s for the network.

There must be enough free memory on the node for use as a write cache. If the kernel cannot keep at least 4 MB in use for **Lustre** write caching, it cannot keep an optimal number of network transactions in progress at once.

There must be enough CPU capacity for the application to do the work which generates data for writing.

There must be enough storage space available

To prevent a situation in which Lustre puts application data into its cache, but then cannot write it to disk because the disk is full, Lustre clients must reserve disk space in advance. However, if it is unable to reserve this space as the OSTs are almost full, less than 2% space available, it must execute its writes synchronously with the server, instead of caching them for efficient bundling.

The degree to which this affects performance depends on how much your application would benefit from write caching. The **cur_dirty_bytes** file in the subdirectory of each OSC of **/proc/fs/lustre/osc/** on a client records the amount of cached writes which are destined for a particular storage target.

The maximum amount of cached data per OSC is determined by the **max_dirty_mb** value in the same directory. This is usually 4 MBs by default. Increasing this value will allow more dirty data to be cached on a client before it needs to flush to the OST, but also increases the time needed for other clients to read or overwrite the cached data once it has been written to the **OST**.

Server thread availability

Write **RPC**s arrive at the server and are processed synchronously by kernel threads (named ll_ost_*). **ps** will help to identify the number of threads that are in the D state indicating that they're busy servicing a request.

Vmstat –see section 5.2.6 - can give a rough approximation of the number of threads that are blocked processing I/O requests when a node is busy servicing only I/O **RPC**s. The number of threads sets an upper bound on the number of I/O **RPC**s that can be processed concurrently, which in turn sets an upper limit for the number of I/O requests that will be serviced concurrently by the attached storage.

5.4.3 Fortran

Particular attention may be necessary for the I/O operations of Fortran as opposed to C as the Fortran run time library may modify the way the I/O operations are programmed. Please refer to the *Chapter 2* for more information on Fortran compiler optimizations, or to the compiler documentation from Intel, or to the manual page for the **ifort** command with particular reference to the section on environmental variables. In particular, the environmental variables **FORT_BUFFERED**, **FORT_CONVERT*** and **F_UFMTENDIAN** should be looked at.

5.5 Lustre File System Tunable Parameters

WARNING

Changing tunable parameters of the lustre file system can render the file system non functional. It should be done with great care on production filesystems. The use of default values is recommended.

One scenario where tuning the file system is beneficial is a cluster with several file systems, some of which have clearly defined workloads. For these filesystems, the file system can then be tuned and optimized for this clearly defined workload. For example, if a file system is used only for checkpoint/restart purposes; the workload for this file system will probably consist of large sequential write and read I/O operations. It is then beneficial to tune the file system for this particular workload, particularly if the cluster has a large amount of memory.

Another example is when performing benchmarking: (temporary) changes may be applied in order to optimize benchmark through put.

This section describes the tuneable parameters of the Lustre file system. For the syntax refer to *Bull HPC Administrator's Guide* or the **lustre_util** man page.

See *Chapter 20.2 - Lustre I/O Tunables* in the *Lustre Operation Manual* on http://manual.lustre.org for more details.

5.5.1 Tuning Parameter Values and their Effects

max_read_ahead_mb

```
/proc/fs/lustre/llite/fs0/max_read_ahead_mb
```

This parameter defines the per-file read-ahead value for a client. Defaulting to 40MB **Read_ahead** is a two-edged sword: this can increase the read throughtput, but can be inefficient (if a file is read randomly rather than sequentially), and in turn detrimental as the memory which is wasted is not available elsewhere. The default value of 40MB is a general purpose value. It may be beneficial to increase this for sequential read workloads, whilst in other situations it may be better to disable it completely.

max_cache_mb

```
/proc/fs/lustre/llite/fs0/max_cache_mb
```

This parameter defines the maximum amount of inactive data cached by the client (the default value is ¾ of the RAM which is available).

max_dirty_mb

```
/proc/fs/lustre/osc/<...>/max_dirty_mb
```

This parameter has a value between 0 and 512MB.

This value controls the write back cache on the client per OSC. While it is beneficial to use larger values, the quantity of dirty data can become so high that, depending on the number of clients, it results in a significant amount time being needed to copy the data to disk.

max_page_per_rpc

```
/proc/fs/lustre/osc/<...>/max_page_per_rpc
```

This value should not be changed from the default value as the optimal value depends on the kernel page size.

max_rpc_in_flight

This value should not be changed from the default value as the optimal value depends on characteristics of the machine.

lru_size

```
/proc/fs/lustre/ldlm/ldlm/namespaces/<OSC name|MDC name>/lru_size
```

Increasing the default value is recommended for login nodes using **lustre** and for improving metadata performance.

debug

```
/proc/sys/lnet/debug
```

The debug level can impact the performance. This is the reason why it is disabled by default. When analyzing problems, different values may be used. The exact optimal value depends on the problem being analyzed: **full debugging** (**-1** value) can slow the filesystem noticeably and even mask the problem under diagnosis.

Note Increasing the debug level can lead to a major performance impact. Full debugging can slow the system by as much as 90%, compared to the default settings.

5.6 More Information

For more information on tuning Lustre file systems see the latest version of the *Lustre Operations Manual* available from http://manual.lustre.org

Appendix A. Amdahl's Law

Amdahl's Law states that the proportion of the program which can run in parallel – the variable *p* – can never reach 100%:

$$Speedup(n) = \frac{1}{(p/n) + (1-p)}$$

p = parallel fraction of the program

n = number of CPUs

In addition, the benefits resulting from augmenting the processing power available for an application will diminish proportionally as a result of hardware constraints and extra message passing latency. The examples below are simple illustrations of this point.

Example 1

p = 0.5 *n* = 10 Speedup = 1.82

p = 0.5 *n* = 15 Speedup = 1.88%

% Increase in Speedup for an extra 5 CPUs = 3.3%

Example 2

p = 0.95 *n* = 10 Speedup= 6.9

p = 0.95 *n* = 15 Speedup = 8.8

% Increase in Speedup for an extra 5 CPUs = 27.5%

Therefore, the higher the value of *p*, the greater the return for any addition to processing power. This applies equally to small increases in *p*, and where the numbers of CPUs involved may be considerably higher.

A key part of any program development is to identify and remove as many dependence constraints as is possible. Generally speaking, there is more to be gained from increasing *p*, than there is to be gained from simply adding additional processing power as Amadhl's law demonstrates.

The benefits to be gained from optimizing and improving the program itself will generally outweigh benefits gained from adding to the hardware's performance.

Glossary and Acronyms

A

ACL
Access Control List

API
Application Programmer Interface

B

BAS
Bull Advanced Server

BIOS
Basic Input Output System

BMC
Baseboard Management Controller

C

CGI
Common Gateway Interface.

ConMan
A management tool, based on telnet, enabling access to all the consoles of the cluster.

CMOS
Complementary Metal Oxide Semiconductor

Cron
A UNIX command for scheduling jobs to be executed sometime in the future. A cron is normally used to schedule a job that is executed periodically - for example, to send out a notice every morning. It is also a daemon process, meaning that it runs continuously, waiting for specific events to occur.

Cygwin
A Linux-like environment for Windows. The Bull cluster management tools use Cygwin to provide ssh support on a Windows system, enabling access in command mode from the Cluster management system.

D

DDN S2A
DataDirect Networks S2A

DNS
Domain Name Server. A server that retains the addresses and routing information for TCP/IP LAN users.

EIP
Encapsulated IP

EPIC
Explicit Parallel Instruction set Computing

EULA
End User License Agreement (Microsoft)

C

FDA
Fibre Disk Array

FSS
Fame Scalability Switch.

FWH
Firm Ware Hub

G

Ganglia
A distributed monitoring tool used to view information associated with a node, such as CPU load, memory consumption, network load.

GCC
GNU C Compiler

GNU
GNU's Not Unix

GPL
General Public Licence

GUI
Graphical User Interface

GUID
Globally Unique Identifier

H

HBA
Host Bus Adapter.

Hyper-Threading
Hyper-Threading technology is an innovative design from Intel that enables multi-threaded software applications to process threads in parallel within each processor resulting in increased utilization of processor execution resources. To make it short, it is to place two logical processors into a single CPU die.

HPC
High Performance Computing.

HSC
Hot Swap Controller

I

IDE
Integrated Device Electronics

IPMI
Intelligent Platform Management Interface

IPO
Interprocedural Optimization

K

KDC
Key Distribution Centre.

KDE
Kool Desktop Environment.

KSIS
Utility for image building and development.

KVM
Keyboard Video Mouse (allows the connection of the keyboard, video and mouse to the node)

L

LDAP
Lightweight Directory Access Protocol.

LKCD
Linux Kernel Crash Dump. A tool capturing and analyzing crash dumps.

LOV
Logical Object Volume.

LUN
Logical Unit Number

Lustre
Parallel file system managing the data shared by several nodes.

LVM
Logical Volume Manager.

M

MIB
Management Information Base.

MDS
MetaData Server.

MDT
MetaData Target.

MkCDrec
Make CD-ROM Recovery. A tool making bootable system images.

MPI
Message Passing interface.

N

Nagios
A powerful monitoring tool, used to monitor the services and resources of Bull HPC clusters.

NFS
Network File System.

NIC
Network Interface Card.

NPTL
Native POSIX Thread Library

NTFS
New Technology File System (Microsoft)

NTP
Network Time Protocol.

NVRAM
Non Volatile Random Access Memory

O

OpenSSH
Open Source implementation of the SSH protocol.

OSC
Object Storage Client.

OSS
Object Storage Server.

OST
Object Storage Targets.

P

PAPI
Performance Application Programming Interface.

PCI
Peripheral Component Interconnect (Intel)

PDSH
A parallel distributed shell.

PDU
Power Distribution Unit

PMU
Performance Monitoring Unit

PVFS
Parallel Virtual File System

PVM
Parallel Virtual Machine

R

ROM
Read Only Me

RPC
Remote Procedure Call

RPM
RedHat Package Manager

S

SAN
Storage Area Network.

SCSI
Small Computer System Interface

SIS

System Installation Suite.

SM

System Management

SMP

Symmetric Multi Processing. The processing of programs by multiple processors that share a common operating system and memory.

SSH

Secure Shell. A protocol for creating a secure connection between two systems.

Syslog-ng

Syslog New Generation, a powerful system log manager.

T

TORQUE

Tera-scale Open-source Resource and QUEue manager. A batch manager controlling and distributing the batch jobs on compute nodes.

U

Unit

Generally it is the set of nodes linked to the same Quadrics switch. One unit contains 4 cells. (See also *Cell*).

UID

User ID

V

VNC

Virtual Network Computing. It is used to enable access to Windows systems and Windows applications from the Bull NovaScale cluster management system.

W

WWPN

World Wide Port Name- a unique identifier in a Fibre Channel SAN.

X

XFS

eXtended File System

Index

L

M

N

O

P

R

S

T

V

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

REFERENCE
86 A2 23FA 00