bullx cluster suite

Application Developer's Guide

extreme computing

REFERENCE
86 A2 22FA 04

extreme computing

bullx cluster suite

Application Developer's Guide

Software

August 2010

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

REFERENCE
86 A2 22FA 04

Trademarks and Acknowledgements

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.

Table of Contents

List of Figures

List of Tables

Preface

Scope and Objectives

The purpose of this guide is to describe the tools and libraries included in the **bullx cluster suite** delivery that allow the development, testing and optimal use of application programs on **Bull** Extreme Computing clusters. In addition, various Open Source and proprietary tools are described.

Intended Readers

This guide is for Application Developers and Users of Bull extreme computing clusters.

Prerequisites

The installation of all hardware and software components of the cluster must have been completed. The cluster Administrator must have carried out basic administration tasks (creation of users, definition of the file systems, network configuration, etc).

Bibliography

Refer to the manuals included on the documentation CD delivered with your system OR download the latest manuals for your **bullx cluster suite** release, and for your cluster hardware, from: http://support.bull.com/

The *bullx cluster suite Documentation* CD-ROM (86 A2 12FB) includes the following manuals:

- *bullx cluster suite Installation and Configuration Guide* (86 A2 19FA)
- *bullx cluster suite Administrator's Guide* (86 A2 20FA)
- *bullx cluster suite Application Developer's Guide* (86 A2 22FA)
- *bullx cluster suite Maintenance Guide* (86 A2 24FA)
- *bullx cluster suite High Availability Guide* (86 A2 25FA)
- *InfiniBand Guide* (86 A2 42FD)
- *Authentication Guide* (86 A2 41FD)
- *SLURM Guide* (86 A2 45FD)
- *Lustre Guide* (86 A2 46FD)

The following document is delivered separately:

- The *Software Release Bulletin* (SRB) (86 A2 80EJ)

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

For **Bull System Manager**, refer to the *Bull System Manager* documentation suite.

For clusters that use the **PBS Professional** Batch Manager, the following manuals are available on the *PBS Professional CD-ROM*:

- *Bull PBS Professional Guide* (86 A2 16FE)
- *PBS Professional Administrator's Guide*
- *PBS Professional User's Guide* (on the *PBS Professional CD-ROM*)

For clusters that use **LSF**, the following manuals are available on the LSF CD-ROM:

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

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)

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.

Chapter 1. Introduction to the Extreme Computing Environment

The term **extreme computing** describes the development and execution of large scientific applications and programs that require a powerful computation facility, which can process enormous amounts of data to give highly precise results.

bullx cluster suite is a software suite that is used to operate and manage a Bull extreme computing cluster of Xeon-based nodes. These clusters are based on Bull platforms using **InfiniBand** stacks or with **Gigabit Ethernet** networks. **bullx cluster suite** includes both Bull proprietary and Open Source software, which provides the infrastructure for optimal interconnect performance.

A Bull extreme computing cluster includes an administrative network based on a 10/100 Mbit or a Gigabit Ethernet network, and a separate console management network.

The **bullx cluster suite** delivery also provides a full environment for development, including optimized scientific libraries, MPI libraries, as well as debugging and performance optimization tools.

This manual describes these software components, and explains how to work within the **bullx cluster suite** environment.

1.1 Software Configuration

1.1.1 Operating System and Installation

bullx cluster suite is based on a standard Linux distribution, combined with a number of Open Source applications that exploit the best from the Open Systems community. This combined with technology from Bull and its partners, results in a powerful, complete solution for the development, execution, and management of parallel and serial applications simultaneously.

Its key features are:

- Strong manageability, through Bull's systems management suite that is linked to state-of-the-art workload management software.
- High-bandwidth, low-latency interconnect networks.
- Scalable high performance file systems, both distributed and parallel.

All cluster nodes use the same Linux distribution. Parallel commands are provided to supply users and system administrators with single-system attributes, which make it easier to manage and to use cluster resources.

Software installation is carried out by first creating an image on a node, loading this image onto the Management Node, and then distributing it to the other nodes using the **Image Building and Deployment** (KSIS) utility. This distribution is performed via the administration network.

1.2 Program Execution Environment

When a user logs onto the system, the login session is directed to one of several nodes where the user may then develop and execute their applications. Applications can be executed on other cluster nodes apart from the user login system. For development, the environment consists of:

- Standard Linux tools such as **GCC** (a collection of free compilers that can compile C/C++ and FORTRAN), **GDB Gnu Debugger**, and other third-party tools including the **Intel** FORTRAN Compiler, the Intel C Compiler, Intel MKL libraries and Intel Debugger **IDB**.
- Optimized parallel libraries that are part of the bullx cluster suite. These libraries include the **Bull MPI2** and **bullx mpi** message-passing library. **Bull MPI2** complies with the MPI1 and 2 standards and is a high performance, high quality native implementation. **Bull MPI2** exploits shared memory for intra-node communication. It includes a trace and profiling tool, enabling data to be tracked.
- **Modules** software provides a means for predefining and changing environments. Each one includes a compiler, a debugger and library releases which are compatible with each other. So it is easy to invoke one given environment in order to perform tests and then compare the results with other environments.

1.2.1 Resource Management

The resource manager is responsible for the allocation of resources to jobs. The resources are provided by nodes that are designated as compute resources. Processes of the job are assigned to and executed on these allocated resources.

Both **Gigabit Ethernet** and **InfiniBand** clusters use the **SLURM** (Simple Linux Utility for Resource Management) open-source, highly scalable cluster management and job scheduling system. **SLURM** has the following functions.

- It allocates compute resources, in terms of processing power and Computer Nodes to jobs for specified periods of time. If required the resources may be allocated exclusively with priorities set for jobs.
- It is also used to launch and monitor jobs on sets of allocated nodes, and will also resolve any resource conflicts between pending jobs.
- It helps to exploit the parallel processing capability of a cluster.

See The *SLURM Guide* for more information.

1.2.2 Batch Management

Different possibilities exist for handling batch jobs for extreme computing clusters.

- **PBS-Professional**, a sophisticated, scalable, robust Batch Manager from **Altair Engineering** is supported as a standard. **PBS Pro** can also be integrated with the **MPI** libraries.

See The *Bull PBS Professional Guide*, *PBS-Professional Administrator's Guide* and *User's Guide* available on the **PBS-Pro CD-ROM** delivered for the clusters, which use PBS-Pro, and the PBS-Pro web site http://www.pbsgridworks.com.

Important PBS Pro does not work with SLURM and should only be installed on clusters which do not use SLURM.

- **LSF**, a batch manager from **Platform™** Company for managing and accelerating batch workload processing for compute-and data-intensive applications is optional on Bull extreme computing.

See The *LSF Installation and Configuration Guide* available on the LSF CD-ROM for more information.

1.2.3 Parallel processing and MPI libraries

A common approach to parallel programming is to use a message passing library, where a process uses library calls to exchange messages (information) with another process. This message passing allows processes running on multiple processors to cooperate.

Simply stated, a **MPI** (Message Passing Interface) provides a standard for writing message-passing programs. A **MPI** application is a set of autonomous processes, each one running its own code, and communicating with each other through calls to subroutines of the MPI library.

Bull MPI2 and **bullx MPI**, Bull's second-generation MPI library, are included in the **bullx cluster suite** delivery. The **Bull MPI2** library enables dynamic communication with different device libraries, including InfiniBand **(IB)** interconnects, Ethernet/IB/EIB socket devices or single machine devices. **Bull MPI2** is fully integrated with the **SLURM** resource manager.

bullx MPI is based on the Open Source **Open MPI** project. **Open MPI** is a **MPI-2** implementation that is developed and maintained by a consortium of academic, research, and industry partners. **Open MPI** offers advantages for system and software vendors, application developers and computer science researchers.

This library enables dynamic communication with different device libraries, including **InfiniBand (IB)** interconnects, socket Ethernet/IB devices or single machine devices.

bullx MPI conforms to the **MPI-2** standard.

See *Chapter 2* for more information on MPI Libraries.

1.2.4 Data and Files

Application file I/O operations may be performed using locally mounted storage devices, or alternatively, on remote storage devices using either **Lustre** or the **NFS** file systems. By using separate interconnects for administration and I/O operations, the Bull cluster system administrator is able to isolate user application traffic from administrative operations and monitoring. With this separation, application I/O performance and process communication can be made more predictable while still enabling administrative operations to proceed.

See The *Lustre Guide* for more information on Lustre.

Chapter 2. Parallel Libraries

A common approach to parallel programming is to use a message passing library, where a process uses library calls to exchange messages (information) with another process. This message passing allows processes running on multiple processors to cooperate.

Simply stated, a **MPI** (Message Passing Interface) provides a standard for writing message-passing programs. A **MPI** application is a set of autonomous processes, each one running its own code, and communicating with each other through calls to subroutines of the **MPI** library.

Programming with MPI

It is not in the scope of the present guide to describe how to program with MPI. Please, refer to the Web, where you will find complete information.

2.1 MPIBull2

MPIBull2 is a second generation MPI library. This library enables dynamic communication with different device libraries, including InfiniBand **(IB)** interconnects, socket Ethernet/IB/EIB devices or single machine devices.

MPIBull2 conforms to the MPI-2 standard.

2.1.1 MPIBull2_1.3.x features

MPIBull2_1.3.x includes the following features:

- It only has to be compiled once, supports the NovaScale architecture, and is compatible with the more powerful interconnects.
- It is designed so that both development and testing times are reduced and it delivers high performance on **NovaScale** architectures.
- Fully compatible with **MPICH2 MPI** libraries. Just set the library path to get all the **MPIBull2** features.
- Supports both MPI 1.2 and MPI 2 standard functionalities including
  - Dynamic processes (**osock** only)
  - One-sided communications
  - Extended collectives
  - Thread safety (see the *Thread-Safety* Section below)
  - **ROMIO** including the latest patches developed by Bull
- Multi-device functionality: delivers high performance with an accelerated multi-device support layer for fast interconnects. The library supports:
  - Sockets-based messaging (for **Ethernet**, **SDP**, **SCI** and **EIP**)
  - Hybrid shared memory-based messaging for shared memory
  - InfiniBand architecture multirails driver Gen2

- Easy Runtime Selection: makes it easy and cost-effective to support multiple platforms. With the **MPIBull2** Library, both users and developers can select drivers at runtime easily, without modifying the application code. The application is built once and works for all interconnects supported by Bull.
- Ensures that applications achieve high performance, and maintain a high degree of interoperability with standard tools and architectures.
- Common feature for all devices:
  - **FUTEX** (Fast User mode muTEX) mechanism in user mode.

2.1.2 MPIBull2 Compilers and Wrappers

The **MPIBull2** library has been compiled with the latest **Intel** compilers, which, according to Bull's test farms, are the fastest ones available for the **Xeon** architecture. Bull uses Intel **Icc** and **Ifort** compilers to compile the **MPI** libraries. It is possible for the user to use their own compilers to compile their applications for example **gcc**.

In order to check the configuration and the compilers used to compile the MPI libraries look at the **${mpibull2_install_path}/share/doc/compilers_version** text file.

MPI applications should be compiled using the **MPIBull2 MPI** compiler wrappers:

C programs:	mpicc your-code.c	
C++ programs:	mpiCC your-code.cc	
	or	
	mpic++ your-code.cc	(for case-insensitive file systems)
F77 programs:	mpif77 your-code.f	
F90 programs:	mpif90 your-code.f90	

Compiler wrappers allow the user to concentrate on developing the application without having to think about the internal mechanics of MPI. They simply add various command line flags and invoke a back-end compiler; they are not compilers in themselves.

See The wrapper man pages for more information.

The command below is used to override the compiler type used by the wrapper. Use either the **–cc**, **-fc** -, and **cxx** option according to the wrapper type (C, Fortran and C++).

```
mpi_user >>> mpicc -cc=gcc prog.c -o prog
```

2.1.2.1 Linking wrappers

When using compiling tools, the wrappers need to know which communication device and a linking strategy they should use. The compiling tools parse as long as some of the following conditions have been met:

- The device and linking strategy has been specified in the command line using the **-sd** options.
- The environment variables **DEF_MPIDEV**, **DEF_MPIDEV_LINK** (required to ensure compatibility), **MPIBULL2_COMM_DRIVER**, and **MPIBULL2_LINK_STRATEGY** have been set.

- The preferences have already been set up; the tools will use the device they find in the environment using the **MPIBULL2-devices** tool.
- The tools take the system default, using the dynamic socket device.

Note It is possible to obtain better performance using the **–fast/-static** options to link statically with one of the dependent libraries, as shown in the commands below.

```
mpicc -static prog.c
mpicc -fast prog.c
```

2.1.3 Configuring MPIBull2

Important **MPIBULL2 is usually installed in the /opt/mpi/mpibull2-<version> directory. The environmental variables MPI_HOME, PATH, LD_LIBRARY_PATH, MAN_PATH, PYTHON_PATH will need to be set or updated. These variables should not be set by the user. Use the setenv_mpibull2.{sh,csh} environment setting file, which may be sourced from the ${mpibull2_install_path}/share directory by a user or added to the profile for all users by the administrator.**

MPIBull2 may be used for different architectures including standalone **SMP**s, **Ethernet**, **Infiniband** or **Quadrics** Clusters.

You have to select the device that will use **MPIBull2** before launching an application with **MPIBull2**.
The list of possible devices available is as follows:

- **osock** is the default device. This uses sockets to communicate and is the device of choice for **Ethernet** clusters.
- **oshm** should be used on a standalone machines, communication is through shared memory.
- **ibmr_gen2**, otherwise known as **InfiniBand multi-rail gen2**. This works over **InfiniBand**'s verbs interface.

The device is selected by using the **mpibull2-devices** command with the **–d** switch, for example, enter the command below to use the shared memory device:

```
mpi_user >>> mpibull2-devices -d=oshm
```

For more information on the **mpibull2-devices** command, see the following sections.

2.1.4 Running MPIBull2

The MPI application requires a launching system in order to spawn the processes onto the cluster. **Bull** provides the **SLURM** Resource Manager as well as the **MPD** subsystem.

For **MPIBull2** to communicate with **SLURM** and **MPD**, the **PMI** interface has to be defined. By default, **MPIBull2** is linked with **MPD**'s **PMI** interface.

If you are using **SLURM**, you must ensure that **MPIBULL2_PRELIBS** includes **-lpmi** so that your **MPI** application can be linked with **SLURM**'s **PMI** library.

See The *SLURM Guide* and *Sections 2.1.5.3* in this chapter for more information on MPD.

2.1.5 MPIBull2 Advanced features

2.1.5.1 MPIBull2 Linking Strategies

Designed to reduce development and testing time, MPIBull2 includes two linking strategies for users.

Firstly, the user can choose to build his application and link dynamically, leaving the choice of the MPI driver until later, according to which resources are available. For instance, if a small Ethernet cluster is the only resource available, the user compiles and links dynamically, using an **osock** driver, whilst waiting for access to a bigger cluster via a different InfiniBand interconnect and which uses the ibmr_gen2 driver at runtime.

Secondly, the User might want to use an out-of-the-box application, designed for a specific MPI device. Bull provides the combination of a MPI Core and all its supported devices, which enables static libraries to be linked to by the User's application.

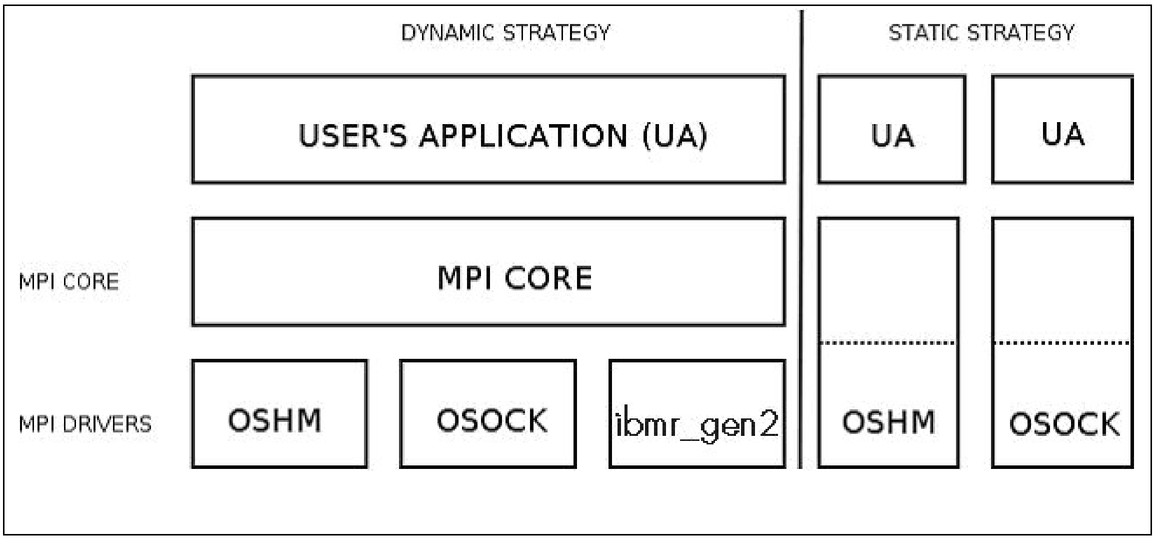


Figure 2-1. MPIBull2 Linking Strategies

2.1.5.2 Thread-safety

If the application needs an MPI Library which provides MPI_THREAD_MULTIPLE thread-safety level, then choose a device which supports thread safety and select a *_ts device. Use the mpibull2-device commands.

Note Thread-safety within the MPI Library requires data locking. Linking with such a library may impact performance. A loss of around 10 to 30% has been observed on micro-benchmarks.

Not all MPI Drivers are delivered with a thread-safe version. Devices known to support MPI_THREAD_MULTIPLE include osock and oshm.

2.1.5.3 Using MPD

MPD is a simple launching system from MPICH-2.

To use it, you need to launch the MPD daemons on Compute hosts.

If you have a single machine, just launch **mpd &** and your MPD setup is complete.

If you need to spawn MPI processes across several machines, you must use **mpdboot** to create a launching ring on the cluster. This is done as follows:

1. Create the hosts list:

```
mpi_user >>> export cluster_machines="host1 host2 host3 host4"
```

2. Create the file used to store host information:

```
mpi_user >>> for i in $cluster_machines; do echo "$i" >> machinefiles; done
```

3. Boot the MPD system on all the hosts:

```
mpi_user >>> mpdboot -n $(cat $clustermachines | wc -l) -f machinefiles
```

4. Check if everything is OK:

```
mpi_user >>> mpdtrace
```

5. Run the application or try hostname:

```
mpi_user >>> mpiexec -n 4 ./your_application
```

MPI Process Daemons (MPD) run on all nodes in a ring like structure, and may be used in order to manage the launching of the different processes. MPIBull2 library is PMI compliant, which means it can interact with any other PMI PM. This software has been developed by ANL. In order to set up the system the MPD ring must firstly be knitted using the procedure below:

6. At the $HOME prompt edit the **.mpd.conf** file by adding something like **MPD_SECRETWORD=your_password** and `chmod 600` to the file.

7. Create a boot sequence file. Any type of file may be used. The MPD system will by default use the **mpd.hosts** file in your **$HOME** directory if a particular file is not specified in the boot sequence. This contains a list of hosts, separated by carriage returns. Semi-colons can be added to the host to specify the number of CPUs for the host, for example.

```
host1:4
host2:8
```

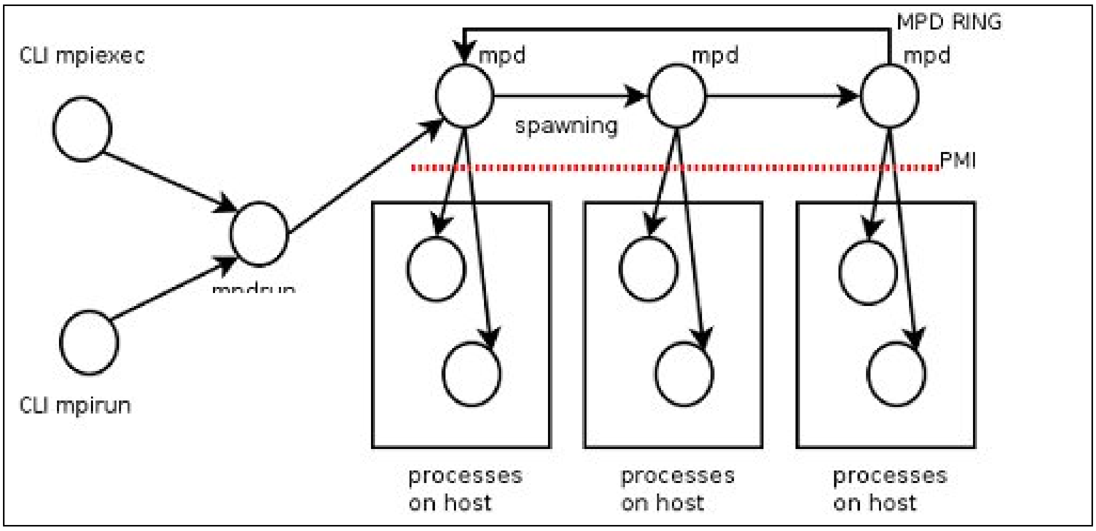


Figure 2-2. MPD ring

8. Boot the ring by using the **mpdboot** command, and specify the number of hosts to be included in the ring.

```
mpdboot -n 2 -f myhosts_file
```

Check that the ring is functioning correctly by using the **mpdtrace** or **mpdringtest** commands. If everything is okay, then jobs may be run on the cluster.

2.1.5.4 Dynamic Process Services

The main goal of these services is to provide a means to develop software using multi-agent or master/server paradigms. They provide a mechanism to establish communication between newly created processes and an existing MPI application (**MPI_COMM_SPAWN**). They also provide a mechanism to establish communication between two existing **MPI** applications, even when one did not 'start' the other (**MPI_PUBLISH_NAME**).

MPI_PUBLISH_NAME structure

MPI_PUBLISH_NAME (service_name, info, port_name)

IN	service_name	a service name to associate with the port (string)
IN	info	implementation-specific information (handle)
IN	port_name	a port name (string)

Although these paradigms are useful for extreme computing clusters there may be a performance impact. **MPIBull2** includes these Dynamic Process Services, but with some restrictions:

- Only the **osock** socket **MPI** driver can be used with these dynamic processes.
- A **PMI** server implementing spawn-answering routines must be used as follows.
  - For all Bull clusters the **MPD** sub-system is used - see the sections above for more details.
  - For clusters that use **SLURM**, a **MPD** ring must be deployed once SLURM's allocation has been guaranteed.
  - **PBS Professional** clusters can use **MPD** without any restrictions.

- The quantity of processes which can be spawned depend on the reservation previously allocated with the Batch Manager /Scheduler (if used).

See The chapter on *Process Creation and Management* in the **MPI-2.1** Standard documentation available from http://www.mpi-forum.org/docs/ for more information.

MPI Ports Publishing Example

	Sever	Client
Command	`mpiexec -n 1 ./server`	`mpiexec -n 4 ./toy`
Process	**(MPI_Open_port)** + **(MPI_Publish_name)** MPIBull2 1.3.9-s (Astlik) MPI_THREAD_FUNNELED (device osock) Server is waiting for connections **(MPI_Comm_accept)** Master available, Received from 0 Now time to merge the communication **(MPI_Comm_merge)** Establish communication with 1st slave Accept communication to port Slave 1 available Slave 2 available Disconnected from slave, Send message to Master	MPIBull2 1.3.9-s (Astlik) MPI_THREAD_FUNNELED (device osock) **(MPI_Get_attribute)** Got the universe size from server **(MPI_Lookup_name)** Lookup found service attag#0$port#35453$description#10.11.0.11 $ifname#10.11.0.11$ port [x4] **(MPI_Comm_connect)** + **(MPI_Send / MPI_Recv)** Sent stuff to the commInter Recv stuff to the commInter Master Process at work, merge comm Master: number of tasks to distribute: 10 Sent a message to the following MPI process Sent stuff to the commInter Recv stuff to the commInter Slave Process at work, merge comm Sent stuff to the commInter Recv stuff to the commInter Slave Process at work, merge comm Sent stuff to the commInter Recv stuff to the commInter Slave Process at work, merge comm Process 1 with 1 Threads runs at work 1: Got task from 900001 to 1000000 Merged and disconnected **(MPI_Comm_disconnect)** Assigned tasks: —0 0–1 [x10]

	Slave 3 available Disconnected from slave, Send message to Master (MPI_Comm_Unpublish_name) (MPI_Close_Port)	[compute] I give up 3: Wallclock Time: 45.2732 1: Wallclock Time: 45.2732 Unpublishing my service toyMaster 2: Wallclock Time: 45.2732 Closing my port of connection (master) master disconnected from 1 master disconnected from 2 master disconnected from 3 Master with 1 Threads joins computation (univ: 1) disconnected from server 0: Wallclock Time: 45.2757

2.1.6 MPIBull2 Tools

2.1.6.1 MPIBull2-devices

This tool may be used to change the user's preferences. It can also be used to disable a library. For example, if the program has already been compiled and the intention is to use dynamic MPI Devices that have already been linked with the MPI Core, then it is now possible to specify a particular runtime device with this tool. The following options are available with **MPIBULL2-devices**

-dl Provides list of drivers. This is also supported by MPI wrappers.

-dlv Provides list of drivers with versions of the drivers.

```
mpi_user >>> mpibull2-devices -dl
```

```
MPIBULL2 Communication Devices :
+ Original Devices :
*oshm   : Shared Memory device, to be used on a single machine [static][dynamic]
*osock  : Socket protocol (can be used over IPoIB, SDP, SCI...) [static][dynamic]
******
```

-c Obtains details of the user's configuration.

```
mpi_user >>> mpibull2-devices -c
```

```
MPIBULL2 home : /install_path
User prefs    :
  \__ Directory                  : /home_nfs/mpi_user/.MPIBull2/
  \__ Custom devices             : /home_nfs/mpi_user/.MPIBull2//site_libs
  \__ MPI Core flavor            : Standard / Error detection on
  \__ MPI Communication Driver   : oshm (Shared Memory device, to be used on
a single machine) [static][dynamic]
```

-d=xxx Sets the communication device driver specified.

```
mpi_user >>> mpibull2-devices -d=ibmr_gen2
```

2.1.6.2 mpibull2-launch

This meta-launcher connects to the process manager specified by the user. It is used to ensure compatibility between different process manager launchers, and also to allow users to specify their custom key bindings.

The purpose of **mpibull2-launch** is to help users to retain their launching commands. **mpibull2-launch** also interprets user's special key bindings, in order to allow the user to retain their preferences, regardless of the cluster and the **MPI** library. This means that the user's scripts will not need changing, except for the particular environment variables that are required.

The **mpibull2-launch** tool provides default key bindings. The user can check them using the **--metahelp** option. If the user wishes to check some of the **CPM** (Cluster Process Manager) special commands, they should use **--options** with the **CPM** launch name command (e.g. **--options srun**).

Some tool commands and 'device' functionalities rely on the implementation of the **MPI** components. This simple tool maps key bindings to the underlying **CPM**. Therefore, a unique command can be used to launch a job on a different CPM, using the same syntax. **mpibull2-launch** system takes in account the fact that a user might want to choose their own key bindings. A template file, named **keylayout.tmp1**, may be found in the tools RPM, and can be used to construct individual key binding preferences.

Launching a job on a cluster using mpibull2-launch

For a **SLURM CPM** use a command similar to the one below and set **MPIBULL2_LAUNCHER=srun** to make this command compatible with the **SLURM CPM**.

```
mpibull2-launch -n 16 -N 2 -ptest ./job
```

Example for a user who wants to use the Y key for the partition

```
PM Partition to use+Y:+partition:
```

The user should edit a file using the format found in the example template, and then add custom bindings using the **–custom_keybindings** option. The + sign is used to separate the fields. The first field is the name of the command, the second the short option, with a colon if an argument is needed, and the third field is the long option.

2.1.6.3 mpiexec

This launcher connects to the MPD ring.

2.1.6.4 mpirun

This launcher connects to the MPD ring.

2.1.7 MPIBull2 – Example of use

2.1.7.1 Setting up the devices

When compiling an application the user may wish to keep the makefiles and build files, which have already been generated. Bull has taken this into account. The code and build files can be kept as they are. All the user needs to do is to set up a few variables or use the **MPIBULL2-devices** tool.

During the installation process, the **/etc/profile.d/mpibull2.sh** file will have been modified by the System Administrator according to the user's needs. This file determines the default settings (by default the rpm sets the **osock** socket/TCP/IP driver). It is possible to override these settings by using environment variables – this is practical as it avoids modifying makefiles - or by using the tools options. For example, the user can statically link their application against a static driver as shown below. The default linking is dynamic, and this enables drive modification during runtime. Linking statically, as shown below, overrides the user's preferences but does not change them.

```
mpi_user >>> mpicc -sd=ibmr_gen2 prog.c -o prog
mpicc : Linking statically MPI library with device (ibmr_gen2)
```

The following environment variables may also be used

MPIBULL2_COMM_DRIVER	Specifies the default device to be linked against
MPIBULL2_LINK_STRATEGY	Specifies the link strategy (the default is dynamic) (this is required to ensure compatibility)
MPIBULL2_MPITOOLS_VERBOSE	Provides information when building (the default is verbose off)

```
mpi_user >>> export DEF_MPIDEV=ibmr_gen2
mpi_user >>> export MPIBULL2_MPITOOLS_VERBOSE=1
mpi_user >>> mpicc prog.c -o prog
mpicc : Using environment MPI variable specifications
mpicc : Linking dynamically MPI library with device (ibmr_gen2)
```

2.1.7.2 Submitting a job

If a user wants to submit a job, then according to the process management system, they can use **MPIEXEC, MPIRUN, SRUN or MPIBULL2-LAUNCH** to launch the processes on the cluster (the online man pages gives details of all the options for these launchers)

2.1.8 MPIBull2 and NFS Clusters

To use **MPI** and **NFS** together, the shared NFS directory must be mounted with the no attribute caching (**noac**) option added; otherwise the performance of the Input/Output operations will be impacted. To do this, edit the **/etc/fstab** file for the **NFS** directories on each client machine in a multi-host **MPI** environment.

Note All the commands below must be carried out as root.

Run the command below on the NFS client machines:

```
grep nfs_noac /etc/fstab
```

The **fstab** entry for **/nfs_noac** should appear as below:

```
/nfs_noac /nfs_noac nfs bg,intr,noac 0 0
```

If the **noac** option is not present, add it and then remount the **NFS** directory on each machine using the commands below.

```
umount /nfs_noac
mount /nfs_noac
```

To improve performance, export the **NFS** directory from the **NFS** server with the **async** option.

This is done by editing the **/etc/exports** file on the **NFS** server to include the **async** option, as below.

Example

The following is an example of an export entry that includes the **async** option for **/nfs_noac**:

```
grep nfs_noac /etc/exports
```

```
 /nfs_noac       *(rw,async)
```

If the **async** option is not present, add it and export the new value:

```
exportfs -a
```

2.1.9 MPIBull2 Debuggers

2.1.9.1 Parallel gdb

With the **mpiexec** launching tool it is possible to add the Gnu DeBugger in the global options by using **-gdb**. All the **gdb** outputs are then aggregated, indicating when there are differences between processes. The **-gdb** option is very useful as it helps to pinpoint faulty code very quickly without the need of intervention by external software.

Refer to the **gdb** man page for more details about the options, which are available.

2.1.9.2 Totalview

Totalview is a proprietary software application and is not included in the **bullx cluster suite** distribution. See *Chapter 8* for more details.

It is possible to submit jobs using the **SLURM** resource manage with a command similar to the format below or via MPD.

```
totalview srun –a <args> ./prog <progs_args>
```

Alternatively, it is possible to use MPI process daemons (**MPD**) and to synchronize **Totalview** with the processes running on the MPD ring.

```
mpiexec -tv <args> ./prog <progs_args>
```

2.1.9.3 MARMOT MPI Debugger

MARMOT is an **MPI** debugging library. **MARMOT** surveys and automatically checks the correct usage of the MPI calls and their arguments made during runtime. It does not replace classical debuggers, but is used in addition to them.

The usage of the MARMOT library will be specified when linking and building an application. This library will be linked to the application and to the **MPIBULL2** library.
It is possible to specify the usage of this library manually by using the **MPIBULL2_USE_MPI_MARMOT** environment variable, as shown in the example below;

```
export MPIBULL2_USE_MPI_MARMOT=1
mpicc bench.c -o bench
```

or by using the **-marmot** option with the MPI compiler wrapper, as shown below:

```
mpicc -marmot bench.c -o bench
```

See The documentation in the share section of the marmot package, or go to http://www.hlrs.de/organization/amt/projects/marmot/ for more details on Marmot.

2.1.10 MPIBull2 parameters

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, Quadrics** etc.). 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 later to restore the set of parameters, combined in exactly the same way.

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.

2.1.11 Usage

SYNOPSIS

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

Actions

The following actions are possible for the **mpibull2-param**s 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 display all possible values, including the default. Use the **-i** option to list all information.

Examples

This command 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 -l all shm
```

This command 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 -li all
```

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

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

-f [l[iv]] [FNAME]
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

This command will list all family names with the string **band** in their names.

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

For each family name with the string **band** inside, this command will list all the parameters and current values.

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

-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

This command will set the ROMIO_LUSTRE parameter in the current environment.

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

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

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

-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

This command will, for example, try to save all the MPI parameters into the file named Thu_Feb_14_15_50_28_2010.

```
mpibull2-params -sv
```

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

This command will restore all modified parameters to default.

```
mpibull2-params -r
```

-h
Displays the help page

2.1.12 Family names

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

Some of the parameter family names that are possible for **bullx cluster suite** are listed below.

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
Infiniband_RDMA_IBMR_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 included in the parameter families used within your cluster environment.

2.1.13 Managing your MPI environment

Bull provides different **MPI** libraries for different user requirements. In order to help users manage different environment configurations, Bull also ships Modules that can be used to switch from one MPI library environment to another. This relies on the module software – see *Chapter 6*.

The directory used to store the module files is **/opt/mpi/modulefiles/**, into which the different module files that include the **mpich**, **vltmpi** libraries for **InfiniBand**, and **MPIBull2** environments are placed.

Important It is recommended that when a file is created, for example in the **99-mpimodules.sh** and **99-mpimodules.sh .csh**, it is added to the **/etc/profile.d/** directory. The line below should be pasted into this file. This will make the configuration environment available to all users.

```
module use -a /opt/mpi/modulefiles
```

1. Run the following command to check which modules are available:

```
module av
```

This will give output similar to that below:

```
------------------- /opt/mpi/modulefiles ------------------
mpibull2/1.2.8-1.t      mpich/1.2.7-p1          vltmpi/24-1
```

2. Run the command to see which modules are loaded:

```
module li
```

This will give output similar to that below:

```
Currently Loaded Modulefiles:
  1) oscar-modules/1.0.3
```

3. Run the following commands to change the MPI environments, according to your needs:

```
module load mpich
module li
```

```
Currently Loaded Modulefiles:
  1) oscar-modules/1.0.3   2) mpich/1.2.7-p1
```

4. Run the command to check which **MPI** environment is loaded:

```
which mpicc
```

This will give output similar to that below:

```
/opt/mpi/mpich-1.2.7-p1/bin/mpicc
```

5. Run the command below to remove a module (e.g. **mpich**):

```
module rm mpich
```

6. Then load the new **MPI** environment by running the load command, as shown in the example below:

```
module load mpibull2
```

2.2 bullx MPI

bullx MPI is based on the Open Source **Open MPI** project. **Open MPI** is an **MPI-2** implementation that is developed and maintained by a consortium of academic, research, and industry partners. **Open MPI** offers advantages for system and software vendors, application developers and computer science researchers.

This library enables dynamic communication with different device libraries, including **InfiniBand (IB)** interconnects, socket Ethernet/IB devices or single machine devices.

bullx MPI conforms to the **MPI-2** standard.

Note As **bullx MPI** is based on **Open MPI**, most of the documentation available for Open MPI also applies to **bullx MPI**. You can therefore refer to http://open-mpi.org/faq/ for more detailed information

2.2.1 Quick Start for bullx MPI

Important **bullx MPI** is usually installed in the **/opt/mpi/bullxmpi-<version>** directory. To use it, you can either:
* use the **mpivars.{sh,csh}** environment setting file, which may be sourced from the **${bullxmpi_install_path}/bin** directory by a user or added to the profile for all users by the administrator.
* use module files bundled with bullx MPI (see *Chapter 6* for more information on modules)

2.2.2 Compiling with bullx MPI

MPI applications should be compiled using **bullx MPI** wrappers:

C programs:	mpicc your-code.c
C++ programs:	mpiCC your-code.cc
	or
	mpic++ your-code.cc (for case-insensitive file systems)
F77 programs:	mpif77 your-code.f
F90 programs:	mpif90 your-code.f90

Wrappers to compilers simply add various command line flags and invoke a back-end compiler; they are not compilers in themselves.

2.2.3 Running with bullx MPI

bullx MPI comes with a launch command : **mpirun**.

mpirun is a unified processes launcher. It is highly integrated with various batch scheduling systems, auto-detecting its environment and acting accordingly.

Running with no batch scheduler

mpirun can be used with no batch scheduler. You only need to specify the Compute Nodes list:

```
$ cat hostlist
node1
node2
$ mpirun -hostfile hostlist -np 4 ./a.out
```

Running with SLURM

mpirun is to be run inside a SLURM allocation. It will auto-detect the number of cores and the node list. Hence, **mpirun** needs no arguments.

```
salloc -n 2 mpirun ./a.out
```

Running with PBS Professional

To launch a job in a **PBS** environment, just use **mpirun** with your submission:

```
#!/bin/bash
#PBS -l select=2:ncpus=1
mpirun ./a.out
```

Running with LSF

In a LSF environment, **mpirun** will also automatically detect all the arguments and can therefore be used simply, as below:

```
#!/bin/bash
#BSUB -n 8
mpirun ./a.out
```

2.2.4 Configuring and tuning bullx MPI

Parameters in **bullx MPI** are set using the **MCA** (Modular Component Architecture) subsystem.

2.2.5 Obtaining Details of the MPI Configuration

The **ompi_info** command is used to obtain the details of your **bullx MPI** installation - components detected, compilers used, and even the features enabled. The **ompi_info -a** command can also be used, this adds the list of the **MCA** subsystem parameters at the end of the output.

Output Example

```
MCA btl: parameter "btl" (current value: <none>, data source: default
value)
Default selection set of components for the btl framework (<none>
means use all components that can be found)
```

The parameter descriptions are defined using the following template:

```
MCA <section> : parameter “<param>” (current value: <val>, data
source: <source>)
                <Description>
```

2.2.6 Setting the MCA parameters

MCA parameters can be set in 3 different ways, Command Line, Environment Variables and Files.

Note The parameters are searched in the following order - Command Line, Environment Variables and Files.

Command line

The Command line is the highest-precedence method for setting **MCA** parameters. For example:

```
shell$ mpirun --mca btl self,sm,openib -np 4 a.out
```

This sets the **MCA** parameter **btl** to the value of **self,sm,openib** before running **a.out** using four processes. In general, the format used for the command line is **"–mca <param_name> <value>"**.

Note When setting multi-word values, you need to use quotes to ensure that the shell and **bullx MPI** understand that they are a single value. For example:

shell$ mpirun –mca param "value with multiple words" ...

Environment Variables

After the command line, environment variables are searched. Any environment variable named **OMPI_MCA_<param_name>** will be used. For example, the following has the same effect as the previous example (for **sh**-flavored shells):

```
shell$ OMPI_MCA_btl=self,sm,openib
shell$ export OMPI_MCA_btl
shell$ mpirun -np 4 a.out
```

Or, for **csh**-flavored shells:

```
shell% setenv OMPI_MCA_btl “self,sm,openib”
shell% mpirun -np 4 a.out
```

Note When setting environment variables to values with multiple words quotes should be used, as below:

```
# sh-flavored shells
shell$ OMPI_MCA_param="value with multiple words"
# csh-flavored shells
shell% setenv OMPI_MCA_param "value with multiple words"
```

Files

Finally, simple text files can be used to set MCA parameter values. Parameters are set one per line (comments are permitted). For example:

```
# This is a comment
# Set the same MCA parameter as in previous examples
mpi_show_handle_leaks = 1
```

Note Quotes are not necessary for setting multi-word values in **MCA** parameter files. Indeed, if you use quotes in the **MCA** parameter file, they will be treated as part of the value itself.

Example

```
# The following two values are different:
param1 = value with multiple words
param2 = "value with multiple words"
```

By default, two files are searched (in order):

1. **$HOME/.openmpi/mca-params.conf**: The user-supplied set of values takes the highest precedence.

2. **/opt/mpi/bullxmpi-x.x.x/etc/openmpi-mca-params.conf**: The system-supplied set of values has a lower precedence.

More specifically, the **MCA** parameter **mca_param_files** specifies a colon-delimited path of files to search for **MCA** parameters. Files to the left have lower precedence; files to the right are higher precedence.

Keep in mind that, just like components, these parameter files are only relevant where they are "visible". Specifically, **bullx MPI** does not read all the values from these files during start-up and then send them to all nodes for the job. The files are read on each node during the start-up for each process in turn. This is intentional: it allows each node to be customised separately, which is especially relevant in heterogeneous environments.

Chapter 3. MPI Profiling with mpianalyser and profilecomm

mpianalyser is a profiling tool, developed by Bull for its own **MPI** implementation. This is a non-intrusive tool, which allows the display of data from counters that has been logged when the application runs.

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.

3.1 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 obtain 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).

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.2 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.

3.3 profilecomm Data Collection

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

3.3.1 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.

3.3.2 profilecomm Options

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

For example:

```
export PFC_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**:

```
$ MPIANALYSER_PROFILECOMM=1 srun –p my_partion –N 1 -n 4./foo
```

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

```
$ MPIANALYSER_PROFILECOMM=1 PFC_OPTIONS="-f foo.pfc" srun –p
my_partion –N 1 -n 4./foo
```

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

```
$ MPIANALYSER_PROFILECOMM=1 PFC_OPTIONS="-t -f foo.txt" srun –p
my_partion –N 1 -n 4./foo
```

3.3.3 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"
```

Note **profilecomm** supports a maximum of 10 partitions only.

3.4 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 standard file formats.

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

Note The header, histograms, statistics and topology sections are not included in the output when the **-t**, **-text** text format options are used.

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)
```

3.4.1 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
```

3.4.2 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
```

3.4.3 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
```

3.4.4 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
```

3.4.5 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).

3.4.6 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
```

3.5 Profilcomm Data Display Options

The following options can be used to display the data:

-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.

3.5.1 Exporting a Matrix or an Histogram

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

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.

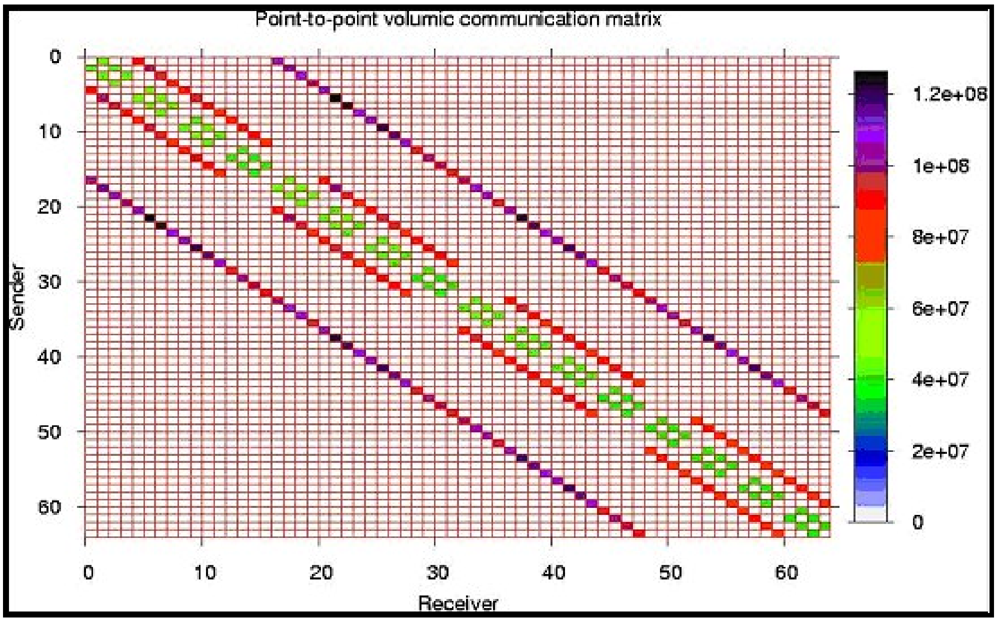


Figure 3-1. An example of a communication matrix

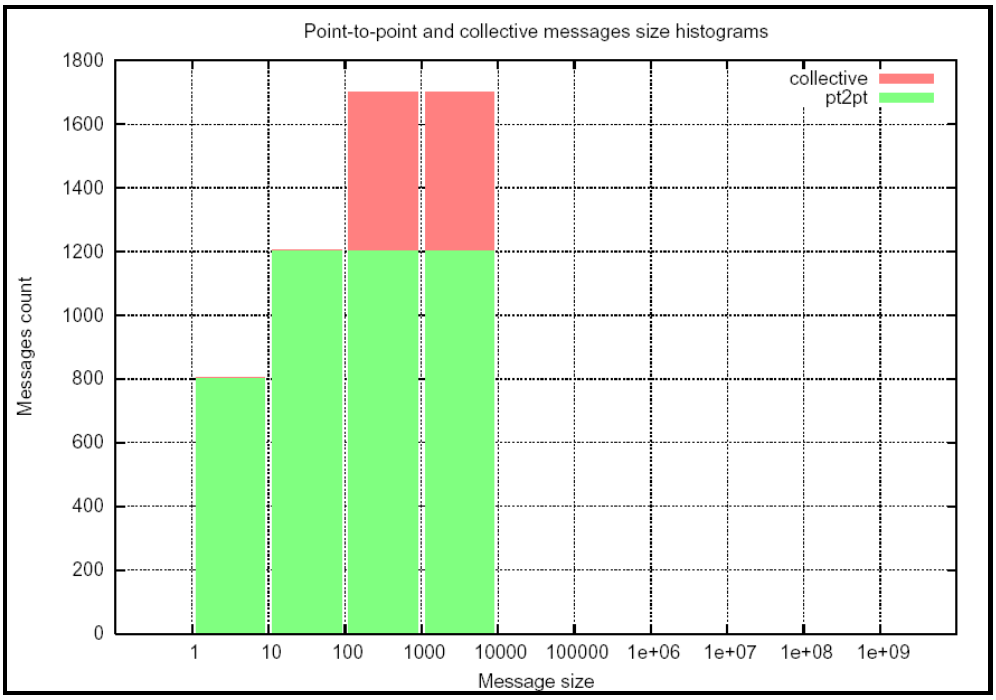


Figure 3-2. An example of a histogram

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:

Important 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.0 -f ps -o foo.ps foo.pfc
$ ls foo.ps
```

```
foo.ps
```

3.5.2 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
```

Chapter 4. Scientific Libraries

This chapter describes the following topics:

- 4.1 *Overview*
- 4.2 *Bull Scientific Studio*
- 4.3 *Intel Scientific Libraries*
- *4.4 NVIDIA CUDA Scientific Libraries*

Important See the *Software Release Bulletin* for details of the Scientific Libraries included with your delivery.

4.1 Overview

Scientific Libraries include tested, optimized and validated functions that spare users the need to develop such subprograms themselves.

The advantages of scientific libraries are:

- Portability
- Support for different types of data (real, complex, double precision, etc.)
- Support for different kinds of storage (banded matrix, symmetrical, etc.)

The following sets of scientific libraries are available for Bull Extreme Computing clusters.

Bull Scientific Studio is included in the bullx cluster suite delivery and includes a range of Open Source libraries that can be used to facilitate the development and execution of a wide range of applications.

Proprietary scientific libraries that have to be purchased separately are available from Intel®, and from NVIDIA® for those clusters which include NVIDIA graphic card accelerators.

4.2 Bull Scientific Studio

Bull Scientific Studio is based on the Open Source Management Framework (OSMF), and provides an integrated set of up-to-date and tested mathematical scientific libraries that can be used in multiple environments. They simplify modeling by fixing priorities, ensuring the cluster is in full production for the maximum amount of time, and are ideally suited for large multi-core systems.

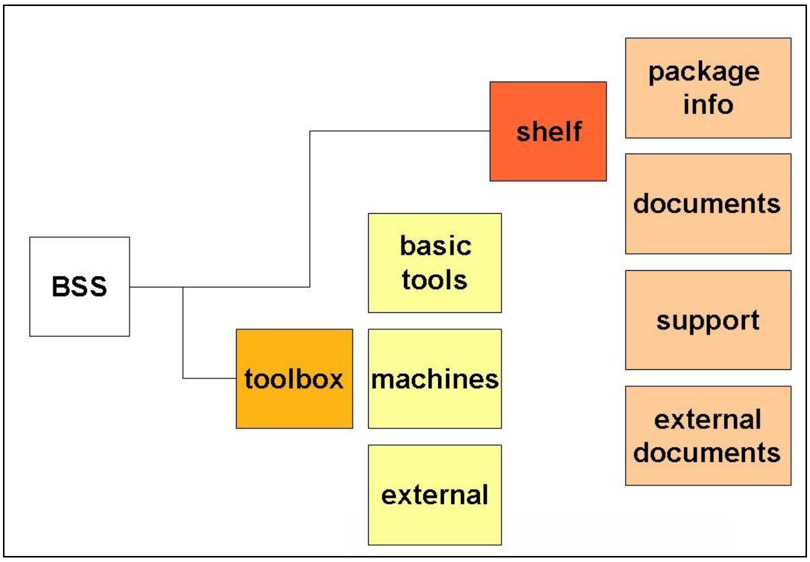


Figure 4-1. Bull Scientific Studio structure

4.2.1 Scientific Libraries and Documentation

The scientific libraries are delivered with the tools included in Bull **Scientific Studio** for developing and running your application.

All the libraries included in Bull **Scientific Studio** are documented in a two RPM files called **SciStudio_shelf** and **OpenS_shelf** as shown in *Figure 4-1*. These files are included in the bullx cluster suite delivery and can be installed on any system. The install paths are:

/opt/scilibs/ SCISTUDIO_SHELF/SciStudio_shelf -<version>

/opt/opens/OPENS_SHELF/OpenS_shelf-<version>/

The **SciStudio_shelf** and the **OpenS_shelf** rpm are generated for each release and contain the documentation for each library included in the release. The documentation for each library is included in the directory for each library based on the type of library. All of the Scientific Studio libraries are found in **/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>** and the OpenS library documentation is found under **/opt/opens/OPENS_SHELF/OpenS_shelf-<versions>**.
For example, the SciStudio libraries are found under **/SCISTUDIO_SHELF/SciStudio_shelf-<version>/<library name>**, for example, the **SCIPORT** documentation is included in the folder

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SCIPORT/sciport-<version>

If there are multiple versions of a library then there is a separate directory for each version number.
A typical documentation directory structure for a shelf rpm files is shown below:

Packaging information

- Configuration information
- README, notice
- Changelogs
- Installation

Documentation

- HowTos, tips
- Manuals
- Examples/tutorials

Support

- Troubleshooting
- Bug reports
- FAQs

External documents

- Documents related to the subject
- Weblinks

The following scientific libraries are included in Bull Scientific Studio.

4.2.2 Scientific Library Versions

4.2.2.1 MPI versions

The libraries that use MPI have been built in two versions. One with the MPIBULL2 library, and one with the bullx MPI library. For this release, the <mpi-version> referred to in this chapter below are:

For MPIBULL2:	mpibull2_1.3.9
For bullx MPI:	bullxmpi_1.0.2

4.2.2.2 Build Version

The <buildversion> referred in this section is either 9010.Bull or Bull.9010.

The libraries listed below are the only libraries that use the 9010.Bull build version:

Blacs, BlockSolve95, fftw2, ParMETIS, sciport

4.2.3 BLACS

BLACS stands for Basic Linear Algebra Communication Subprograms.

BLACS is a specialized communications library that uses message passing. After defining a process chart, it exchanges vectors, matrices and blocks and so on. It can be compiled on top of MPI systems.

BLACS uses MPIBull2 or bullx MPI libraries. More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

```
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/BLACS/blacs-<ver>
```

4.2.3.1 Using BLACS

BLACS is located in the following directory:

```
/opt/scilibs/BLACS/blacs-<version>/<mpi-version>_<buildversion>
```

The libraries include the following:

libblacsCinit_MPI-LINUX-0.a
libblacsF77init_MPI-LINUX-0.a
libblacs_MPI-LINUX-0.a

4.2.3.2 Testing the Installation of the Library

The installation of the library can be tested using the tests found in the following directory:

/opt/scilibs/BLACS/blacs-<version>/<mpi-version>_<buildversion>/tests

Setting Up the Environment

First, the MPI_HOME and LD_LIBRARY_PATH variables must be set up to point to the MPI libraries that are to be tested. For example when using mpibull2 libraries:

```
export MPI_HOME=/opt/mpi/mpibull2-<version>/
export PATH=$MPI_HOME/bin:$PATH
export LD_LIRARY_PATH=$MPI_HOME/lib:$LD_LIRARY_PATH
```

Running the Tests

Then, run the tests as follows:

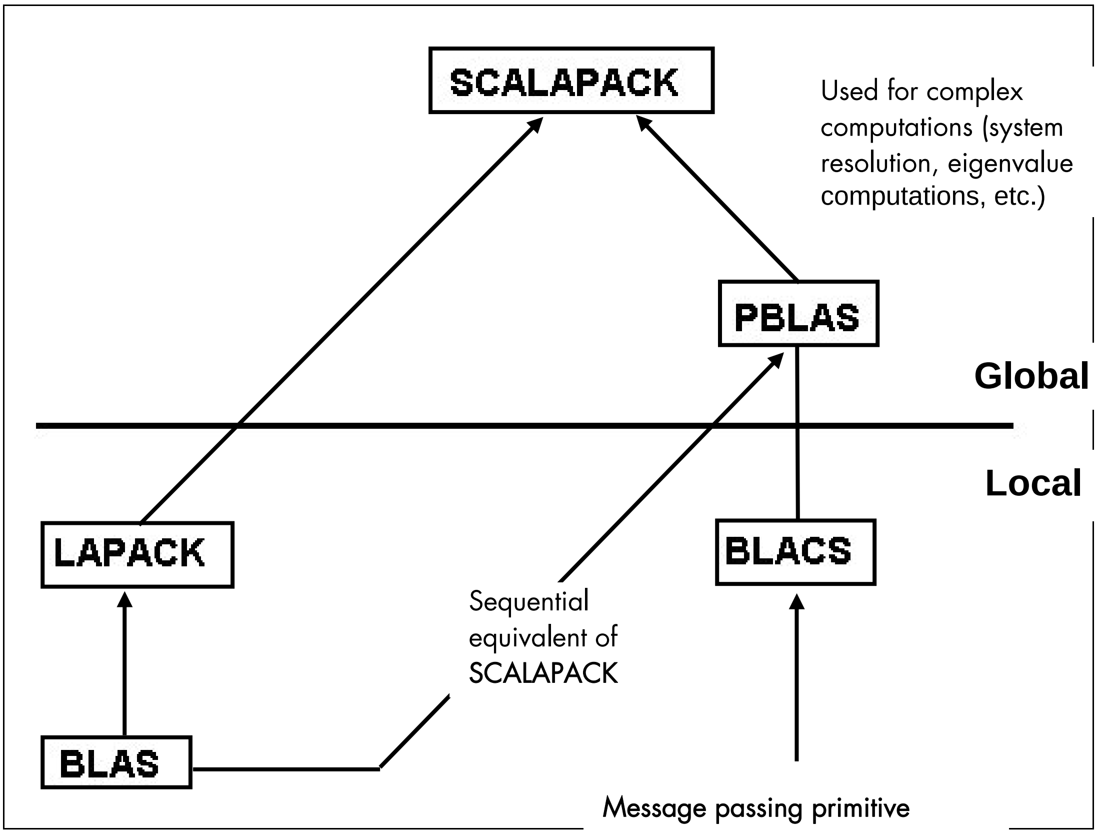
```
mpirun -np 4 xCbtest_MPI-LINUX-0
mpirun -np 4 xFbtest_MPI-LINUX-0
```

4.2.4 SCALAPACK

SCALAPACK stands for: SCALable Linear Algebra PACKage.

This library is the scalable version of LAPACK. Both libraries use block partitioning to reduce data exchanges between the different memory levels to a minimum. SCALAPACK is used above all for eigenvalue problems and factorizations (LU, Cholesky and QR). Matrices are distributed using BLACS.

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under: /opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SCALAPACK/ScaLAPACK-<ver>



SCALAPACK
Used for complex computations (system resolution, eigenvalue computations, etc.)
PBLAS
Global
Local
LAPACK
BLACS
Sequential equivalent of SCALAPACK
BLAS
Message passing primitive

Figure 4-2. Interdependence of the different mathematical libraries (Scientific Studio and Intel)

4.2.4.1 Using SCALAPACK

Local component routines are called by a single process with arguments residing in local memory.
Global component routines are synchronous and parallel. They are called with arguments that are matrices or vectors distributed over all the processes.

SCALAPACK uses MPIBull2/bullx MPI.

The default installation of this library is as follows:

/opt/scilibs/SCALAPACK/ScaLAPACK-<version>/<mpi-version>_<buildversion>

The following library is provided:

Libscalapack.a

Several tests are provided in the following directory:

/opt/scilibs/SCALAPACK/ScaLAPACK-<version>/<mpi-version>_<buildversion>/tests

4.2.5 Blocksolve95

BlockSolve95 is a scalable parallel software library primarily intended for the solution of sparse linear systems that arise from physical models, especially problems involving multiple degrees of freedom at each node.

BlockSolve95 uses the MPIBull2/bullx MPI library.

The default installation of this library is as follows:

/opt/scilibs/BLOCKSOLVE95/BlockSolve95-<version>/<mpi-version>_<buildversion>/lib/lib0/linux

The following library is provided:

libBS95.a

Some examples are also provided in the following directory.

/opt/scilibs/BLOCKSOLVE95/BlockSolve95-<version>/<mpi-version>_<buildversion>/examples

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/BLOCKSOLVE95/BlockSolve95-<ver>

4.2.6 lapack

lapack_sci is a set of Fortran 77 routines used to resolve linear algebra problems such as the resolution of linear systems, eigenvalue computations, matrix computations, etc. However, it is not written for a parallel architecture.

The default installation of this library is as follows:

/opt/scilibs/LAPACK_SCI/lapack_sci-<version>

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf–<version>/ LAPACK_SCI -<version>

4.2.7 SuperLU

This library is used for the direct solution of large, sparse, nonsymmetrical systems of linear equations on high performance machines. The routines will perform an LU decomposition with partial pivoting and triangular systems solves through forward and back substitution. The factorization routines can handle non-square matrices, but the triangular solves are performed only for square matrices. The matrix commands may be pre-ordered, either through library or user supplied routines. This pre-ordering for sparse equations is completely separate from the factorization.

Working precision iterative refinement subroutines are provided for improved backward stability. Routines are also provided to equilibrate the system, estimate the condition number, calculate the relative backward error and estimate error bounds for the refined solutions. SuperLU_Dist is used for distributed memory.

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SUPERLU_DIST/SuperLU_DIST-<version>
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SUPERLU_MT/SuperLU_MT-<version>
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SUPERLU_SEQ/SuperLU_SEQ-<version>

SuperLU Libraires

The following SuperLU Libraries are provided:

/opt/scilibs/SUPERLU_DIST/SuperLU_DIST-<version>/<mpi-version>_<buildversion>/lib/superlu_lnx_x86_64.a

/opt/scilibs/SUPERLU_MT/SuperLU-MT-<version>/lib/ superlu_mt_PTHREAD.a
/opt/scilibs/SUPERLU_SEQ/SuperLU-SEQ-3.0 /lib/superlu_x86_64.a

Tests are provided for each library under the following directory:
/opt/scilibs/SUPERLU_<type>/SuperLU_<type>-<version>/test directory

4.2.8 FFTW

FFTW stands for the Fastest Fourier Transform in the West. FFTW is a C subroutine library for computing a discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and using both real and complex data.

There are three versions of FFTW in this distribution. They are located in the following directories:

/opt/scilibs/FFTW/FFTW3-<version>/lib
/opt/scilibs/FFTW/fftw-2<version>/<mpi-version>_<buildversion>/lib

Tests are also available in the following directory:

/opt/scilibs/FFTW/fftw-<version>/test

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/FFTW/fftw-<version>

4.2.9 PETSc

PETSc stands for Portable, Extensible Toolkit for Scientific Computation. PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communications (see http://www.mcs.anl.gov/mpi for more details).

The PETSc library is available in the following directory:

/opt/scilibs/PETSC/PETSc-<version>/<mpi-version>_<buildversion>/lib/linux-intel-opt/

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/PETSC/PETSc -<version>

4.2.10 NETCDF/sNETCDF

NetCDF (Network Common Data Form) allows the management of input/output data. NetCDF is an interface for array-oriented data access, and is a library that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data.

The library is located in the following directories:

/opt/scilibs/NETCDF/netCDF-<version>/<mpi-version>_<buildversion>/bin
/opt/scilibs/NETCDF /netCDF-<version>/<mpi-version>_<buildversion>/include
/opt/scilibs/NETCDF /netCDF-<version>/<mpi-version>_<buildversion>/lib
/opt/scilibs/NETCDF /netCDF-<version>/<mpi-version>_<buildversion>/man

/opt/scilibs/SNETCDF/snetCDF-<version>/bin
/opt/scilibs/SNETCDF/snetCDF-<version>/include
/opt/scilibs/SNETCDF/snetCDF-<version>/lib
/opt/scilibs/SNETCDF/snetCDF-<version>/man

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/NETCDF/netCDF-<version>

4.2.11 pNETCDF

Parallel-NetCDF library provides high-performance I/O while still maintaining file-format compatibility with Unidata's NetCDF. NetCDF (Network Common Data Form) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data.

The library is located in the following directories:

/opt/scilibs/PNETCDF/pNetCDF-<version>/<mpi-version>_<buildversion>/bin
/opt/scilibs/PNETCDF /pNetCDF-<version>/<mpi-version>_<buildversion>/include
/opt/scilibs/PNETCDF /pNetCDF-<version>/<mpi-version>_<buildversion>/lib
/opt/scilibs/PNETCDF /pNetCDF-<version>/<mpi-version>_<buildversion>/man

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/PNETCDF/pNetCDF-<version>

4.2.12 METIS and PARMETIS

METIS is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices. The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel *k*-way, and multi-constraint partitioning schemes developed in our lab.

ParMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. ParMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel Adaptive Mesh Refinement computations and large scale numerical simulations.

The libraries for ParmMETIS are located in the following directory:

/opt/scilibs/PARMETIS/ParMETIS<version>/<mpi-version>_<buildversion>/lib

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/PARMETIS/ParMETIS-<version>

4.2.13 SciPort

SCIPORT is a portable implementation of CRAY SCILIB that provides both single and double precision object libraries. SCIPORTS provides single precision and SCIPORTD provides double precision.

The libraries for SCIPORT can be found in the following directory:

/opt/scilibs/SCIPORT/sciport-<versions>/lib

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SCIPORT/sciport-<version>

4.2.14 gmp_sci

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface.

The main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc.

GMP is carefully designed to be as fast as possible, both for small operands and for huge operands. The speed is achieved by using full words as the basic arithmetic type, by using fast algorithms, with highly optimized assembly code for the most common inner loops for a lot of CPUs, and by a general emphasis on speed.

GMP is faster than any other big num library. The advantage for GMP increases with the operand sizes for many operations, since GMP uses asymptotically faster algorithms.

The libraries for GMP_SCI can be found in the following directory:

/opt/scilibs/GMP_SCI/gmp_sci-<version>/lib
/opt/scilibs/GMP_SCI/gmp_sci-<version>/include
/opt/scilibs/GMP_SCI/gmp_sci-<version>/info

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/GMP/gmp -<version>

4.2.15 MPFR

The MPFR library is a C library for multiple-precision, floating-point computations with correct rounding. MPFR has continuously been supported by the INRIA (Institut National de Recherche en Informatique et en Automatique) and the current main authors come from the CACAO and Arénaire project-teams at Loria (Nancy, France) and LIP (Lyon, France) respectively. MPFR is based on the GMP multiple-precision library.
The main goal of MPFR is to provide a library for multiple-precision floating-point computation which is both efficient and has a well-defined semantics.

The libraries for MPFR can be found in the following directory:

/opt/scilibs/MPFR/MPFR-<version>/lib
/opt/scilibs/MPFR/MPFR-<version>/include
/opt/scilibs/MPFR/MPFR-<version>/share

More information is available from the documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

```
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/MPFR/MPFR-<version>
```

4.2.16 sHDF5/pHDF5

The HDF5 technology suite includes :

- A versatile data model that can represent very complex data objects and a wide variety of metadata.
- A completely portable file format with no limit on the number or size of data objects in the collection.
- A software library that runs on a range of computational platforms, from laptops to massively parallel systems, and implements a high-level API with C, C++, Fortran 90, and Java interfaces.
- A rich set of integrated performance features that allow for access time and storage space optimizations.
- Tools and applications for managing, manipulating, viewing, and analyzing the data in the collection

The libraries for sHDF5/pHDF5 can be found in the following directory:

```
/opt/scilibs/PHDF5/pHDF5-<version>/<mpi-version>_<buildversion>/lib
/opt/scilibs/PHDF5/pHDF5-<version>/<mpi-version>_<buildversion>/bin
/opt/scilibs/PHDF5/pHDF5-<version>/<mpi-version>_<buildversion>/include
/opt/scilibs/PHDF5/pHDF5-<version>/<mpi-version>_<buildversion>/doc

/opt/scilibs/SHDF5/sHDF5-<version>/lib
/opt/scilibs/SHDF5/sHDF5-<version>/bin
/opt/scilibs/SHDF5/sHDF5-<version>/include
/opt/scilibs/SHDF5/sHDF5-<version>/doc
```

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

```
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/PHDF5/pHDF5-<version>
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SHDF5/sHDF5-<version>
```

4.2.17 ga/Global Array

The Global Arrays (GA) toolkit provides an efficient and portable *'shared-memory'* programming interface for distributed-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays, without the need for explicit cooperation with other processes. Unlike other shared-memory environments, the GA model exposes the non-uniform memory access (NUMA) characteristics of the high performance computers to the programmer, and takes into account the fact that access to a remote portion of the shared data is slower than to the local portion. The location information for the shared data is available, and direct access to the local portions of shared data is provided.

The libraries for ga are located in the following directory:

```
/opt/opens/GA/ga-<version>/<mpi-version>_<buildversion>/lib
```

More information is available from documentation included in the OpenS_shelf rpm. When this is installed, the documentation files will be located under:

```
/opt/opens/OPENS_SHELF/OpenS_shelf-<version>/GlobalArray /ga-<version>
```

4.2.18 gsl

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software provided under the GNU General Public License. The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite. The complete range of subject areas covered by the library includes:

- Complex Numbers
- Special Functions
- Permutations
- BLAS Support
- Eigensystems
- Quadrature
- Quasi-Random Sequences
- Statistics
- N-Tuples
- Simulated Annealing
- Interpolation
- Chebyshev Approximation
- Discrete Hankel Transforms
- Minimization
- Physical Constants
- Discrete Wavelet Transforms
- Roots of Polynomials
- Vectors and Matrices
- Sorting
- Linear Algebra
- Fast Fourier Transforms
- Random Numbers
- Random Distributions
- Histograms
- Monte Carlo Integration
- Differential Equations
- Numerical Differentiation
- Series Acceleration
- Root-Finding
- Least-Squares Fitting
- IEEE Floating-Point
- Basis splines

The gsl libraries can be found in the following directory:

```
/opt/scilibs/GSL/GSL-<version>/lib
/opt/scilibs/GSL/GSL-<version>/bin
/opt/scilibs/GSL/GSL-<version>/include
/opt/scilibs/GSL/GSL-<version>/doc
```

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

```
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/GSL/gsl-<version>
```

4.2.19 pgapack

PGAPack is a general-purpose, data-structure-neutral, parallel genetic algorithm package developed by Argonne National Laboratory

The libraries for pga can be found in the following directory:

```
/opt/scilibs/ PGAPACK/pgapack-<version>/<mpi-version>_<buildversion>/lib
/opt/scilibs /PGAPACK/pgapack-<version>/<mpi-version>_<buildversion>/doc
/opt/scilibs /PGAPACK/pgapack-<version>/<mpi-version>_<buildversion>/include
/opt/scilibs /PGAPACK/pgapack-<version>/<mpi-version>_<buildversion>/man
```

More information is available from the documentation included in the **SciStudio_shelf rpm**. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/PGAPACK/pgapack-<version>

4.2.20 valgrind

Valgrind is an award-winning instrumentation framework for building dynamic analysis tools. There are **Valgrind** tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use **Valgrind** to build new tools. The **Valgrind** distribution currently includes five production-quality tools: a memory error detector, a thread error detector, a cache and branch-prediction profiler, a call-graph generating cache profiler, and a heap profiler. It also includes two experimental tools: a data race detector, and an instant memory leak detector.

The libraries for **Valgrind** are located in the following directories:

/opt/opens/VALGRIND_OPENS/valgrind_OpenS–<version>/share/doc/valgrind/
/opt/opens/VALGRIND_OPENS/valgrind_OpenS-<version>/bin
/opt/opens/VALGRIND_OPENS/valgrind_OpenS-<version>/valgrind/include
/opt/opens/VALGRIND_OPENS/valgrind_OpenS-<version>/valgrind/lib
/opt/opens/VALGRIND_OPENS/valgrind_OpenS-<version>/include/valgrind/vki/
/opt/opens/VALGRIND_OPENS/valgrind_OpenS-<version>/man
/opt/opens/VALGRIND_OPENS/valgrind_OpenS-<version>/lib/valgrind/amd64-linux
/opt/opens/VALGRIND_OPENS/valgrind_OpenS-<version>/lib/valgrind/ x86-linux

More information is available from documentation included in the **SciStudio_shelf** rpm. When this is installed, the documentation files will be located under:

/opt/opens/OPENS_SHELF/OpenS_shelf-<version>/VALGRIND/valgrind-<version>

4.2.21 Hypre

Hypre is a library for solving large, sparse linear systems of equations on massively parallel computers. The main features of this library are:

- **Scalable preconditioners. Hypre** contains several families of preconditioned algorithms focused on the scalable solution of very large sparse linear systems. These algorithms include structured multigrid and element-based algebraic multigrid
- **Implementation of a suit of common iterative methods.** Hypre provides commonly used Krylov-based iterative methods to be used with its scalable preconditioners. These include Conjugate Gradient and GMRES for symmetrical and unsymmetrical matrices, respectively.
- **Intuitive grid-centric interfaces.** Hypre provides data structures to represent and manipulate sparse matrices through interfaces. Each interface provides access to several solvers without the need to write new interface codes. These interfaces include stencil-based structured/semi-structured interfaces, finite-element based unstructured interface, and a linear algebra based interface.
- **Configuration Options.** Hypre can be installed in several computer platforms by simply setting a set of installation parameters. These parameters or options include compilers, optimization modes, and versions of MPI and BLAS routines particular to the users' computational environment. In most cases, users only need to type a **configure** command followed by a **make** command.

- **Dynamic configuration of parameters.** Hypre works for users with different levels of expertise. More experience users can take further control of the solution process through various tuning parameters.
- **User defined interfaces for multiple languages.** Hypre currently supports Fortran and C languages.

The libraries for **Hypre** are located in the following directory:

/opt/scilibs/HYPRE/Hypre-<version>/lib
/opt/scilibs/HYPRE/Hypre-<version>/doc

More information is available from documentation included in the **SciStudio_shelf rpm**. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/Hypre/Hypre-<version>

4.2.22 ML

ML, Sandia's main multigrid preconditioning package. **ML** is designed to solve large sparse linear systems of equations arising primarily from elliptic **PDE** discretizations. ML is used to define and build multigrid solvers and preconditioners, and it contains black-box classes to construct highly-scalable smoothed aggregation preconditioners. ML preconditioners have been used on thousands of processors for a variety of problems, including the incompressible **Navier-Stokes** equations with heat and mass transfer, linear and nonlinear elasticity equations, the Maxwell equations, semiconductor equations, and more.

The libraries for **ML** are located in the following directory:

/opt/scilibs/ML/ml-<version>/doc
/opt/scilibs/ML/ml-<version>/lib
/opt/scilibs/ML/ml-<version>/include

More information is available from documentation included in the **SciStudio_shelf rpm**. When this is installed, the documentation files will be located under:

/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/ML/ml-<version>

4.2.23 spooles

SPOOLES is a library for solving sparse real and complex linear systems of equations, written in the C language using object oriented design. The following functionalities are included:

1. Compute multiple minimum degree, generalized nested dissection and multi-section orderings of matrices with symmetric structure.
2. Factor and solve square linear systems of equations with symmetric structure, with or without pivoting for stability. The factorization can be symmetric **LDL^T**, **Hermitian LDL^H**, or non-symmetric LDU. A direct factorization or a drop tolerance factorization can be computed. The factors and solve can be done in serial mode, multithreaded with Solaris or POSIX threads, or with MPI.
3. Factor and solve overdetermined full rank systems of equations using a multifrontal QR factorization, in serial or using POSIX threads.

4. Solve square linear systems using a variety of Krylov iterative methods. The preconditioner is a drop tolerance factorization, constructed with, or without pivoting, for stability.

The libraries for SPOOLES can be found in the following directory:

/opt/scilibs/SPOOLES/pgapack-<version>/<mpi-version>_<buildversion>/lib
/opt/scilibs/ SPOOLES/pgapack-<version>/<mpi-version>_<buildversion>/doc
/opt/scilibs/SPOOLES/pgapack-<version>/<mpi-version>_<buildversion>/include

More information is available from the documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SPOOLES/spooles-<version>

4.2.24 Open Trace Format (OTF)

Detailed program analysis of massively parallel programs requires the recording of event based performance data during run-time, and its visualisation with appropriate software tools like the Vampir framework.

The specification of a powerful trace file format has to fulfill a large number of requirements. It must allow an efficient collection of the event based performance data on a parallel program environment. On the other hand it has to provide a fast and comprehensive access to the large amount of performance data and the corresponding event definitions by the analysis software tools.

To improve scalability for very large and massively parallel traces the Open Trace Format (OTF) is developed at ZIH as a successor format to the Vampir Trace Format (VTF3).

The Open Trace Format makes use of a portable ASCII encoding. It distributes single traces to multiple so called streams with one or more files each. Merging of records from multiple files is done transparently by the OTF library. The number of possible streams is not limited by the number of available file handles.

The read/write library should be used as a portable interface for third party software. The library supports efficient parallel and distributed access to trace data and offers selective reading access regarding arbitrary time intervals, process selection and record types. Optional auxiliary information can assist this selective access.

The software package contains additional tools for trace data conversion or preparation. More information can be found in the documentation provided.

The libraries for OTF are located in the following directory:

/opt/scilibs/OTF/OTF-<version>/share
/opt/scilibs/OTF/OTF-<version>/lib
/opt/scilibs/OTF/OTF-<version>/include
/opt/scilibs/OTF/OTF-<version>/bin

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/OTF/OTF-<version>

4.2.25 scalasca

SCALASCA is a performance analysis tool developed with the goal of making the optimization of parallel applications on large-scale systems both more effective and more efficient. It can automatically detect inefficient program behavior and highlight opportunities for performance improvement. SCALASCA builds on the idea of searching event traces of parallel applications for execution patterns indicating inefficient behavior. During the search process, SCALASCA classifies the detected pattern instances by category and quantifies their significance for every program phase and system resource involved. The results are made available to the user in a flexible graphical user interface, which can be investigated using varying levels of granularity. A distinctive feature of SCALASCA in comparison to its predecessor KOJAK, is that it achieves a higher degree of scalability by analyzing the trace data in parallel.

The libraries for SCALASCA are located in the following directory:

```
/opt/scilibs/SCALASCA/scalasca-<version>/doc
/opt/scilibs/SCALASCA/scalasca-<version>/lib
/opt/scilibs/SCALASCA/scalasca-<version>/include
/opt/scilibs/SCALASCA/scalasca-<version>/bin
/opt/scilibs/SCALASCA/scalasca-<version>/example
```

More information is available from documentation included in the SciStudio_shelf rpm. When this is installed, the documentation files will be located under:

```
/opt/scilibs/SCISTUDIO_SHELF/SciStudio_shelf-<version>/SCALASCA/scalasca-<version>
```

4.3 Intel Scientific Libraries

Note The scientific libraries in this section are all **Intel**® proprietary libraries and must be bought separately.

4.3.1 Intel Math Kernel Library

This library, which has been optimized by **Intel** for its processors, contains, among other things, the following libraries: **BLAS**, **LAPACK** and **FFT**.

The Intel Cluster **MKL** is a fully thread-safe library.

The library is located in the **/opt/intel/mkl<release_nb>/** directory.

To use it, the environment has to be set by updating the **LD_LIBRARY_PATH** variable:

```
export LD_LIBRARY_PATH=/opt/intel/Compiler/<maj_ver_nb>/<min_ver_nb>/mkl/em64t:$LD_LIBRARY_PATH
```

Example

```
export LD_LIBRARY_PATH=/opt/intel/Compiler/<11.1>/<069>/mkl/em64t:$LD_LIBRARY_PATH
```

4.3.2 BLAS

BLAS stands for Basic Linear Algebra Subprograms.

This library contains linear algebraic operations that include matrixes and vectors. Its functions are separated into three parts:

- Level 1 routine to represent vectors and vector/vector operations.
- Level 2 routines to represent matrixes and matrix/vector operations.
- Level 3 routines mainly for matrix/matrix operations.

This library is included in the Intel MKL package.

For more information see www.netlib.org/blas.

4.3.3 PBLAS

PBLAS stands for Parallel Basic Linear Algebra Subprograms.

PBLAS is the parallelized version of **BLAS** for distributed memory machines. It requires the cyclic distribution by matrix block that the **BLACS** library offers.

This library is included in the **Intel MKL** package.

4.3.4 LAPACK

LAPACK stands for Linear Algebra PACKage.

This is a set of Fortran 77 routines used to resolve linear algebra problems such as the resolution of linear systems, eigenvalue computations, matrix computations, etc. However, it is not written for a parallel architecture.

This library is included in the **Intel MKL** package.

4.4 NVIDIA CUDA Scientific Libraries

For clusters that include **NVIDIA Tesla** graphic accelerators, the **NVIDIA** Compute Unified Device Architecture **(CUDA™) Toolkit,** including versions of the **CUFFT** and the **CUBLAS** scientific libraries, is installed automatically on the **LOGIN**, **COMPUTE** and **COMPUTEX** nodes.

mportant The CUFFT and CUBLAS libraries are not ABI compatible by symbol, by call, or by libname with the libraries included in Bull Scientific Studio. The use of the NVIDIA CUBLAS and CUFFT libraries needs to be made explicit and is exclusive to systems that include the NVIDIA Tesla graphic accelerators.

4.4.1 CUFFT

CUFFT, the **NVIDIA® CUDA™** Fast Fourier Transform (**FFT**) library is used for computing discrete Fourier transforms of complex or real-valued data sets. The **CUFFT** library provides a simple interface for computing parallel FFTs on a Compute Node connected to a **Tesla** graphic accelerator, allowing the floating-point power and parallelism of the node to be fully exploited.

FFT libraries vary in terms of supported transform sizes and data types. For example, some libraries only implement **Radix-2** FFTs, restricting the transform size to a power of two, while other implementations support arbitrary transform sizes. The **CUFFT** library delivered with bullx cluster suite supports the following features:

- 1D, 2D, and 3D transforms of complex and real-valued data.
- Batch execution of multiple 1D transforms in parallel.
- 2D and 3D transforms in the [2, 16384] range in any dimension.
- 1D transforms up to 8 million elements.
- In-place and out-of-place transforms for real and complex data.

The interface to the **CUFFT** library is the header file **cufft.h**. Applications using **CUFFT** need to link against the **cufft.so** Linux **DSO** when building for the device, and against the **cufftemu.so** Linux DSO when building for device emulation.

See The CUDA **CUFFT Library** document available from www.nvidia.com for more information regarding types, API functions, code examples and the use of this library.

4.4.2 CUBLAS

CUBLAS is an implementation of **BLAS** (Basic Linear Algebra Subprograms) on top of the **NVIDIA® CUDA™** driver. The library is self-contained at the **API** level, that is, no direct interaction with the CUDA driver is necessary.

The basic model by which applications use the **CUBLAS** library is to create matrix and vector objects in the memory space of the **Tesla** graphics accelerator, fill them with data, call a sequence of **CUBLAS** functions, and, finally, load the results back to the host. To accomplish this, **CUBLAS** provides helper functions for creating and destroying objects in the graphics accelerator memory space, and for writing data to and retrieving data from these objects.

Because the CUBLAS core functions (as opposed to the helper functions) do not return an error status directly (for reasons of compatibility with existing BLAS libraries), CUBLAS provides a separate function, that retrieves the last recorded error to help debugging.

The interface to the CUBLAS library is the header file **cublas.h**. Applications using CUBLAS need to link against the **cublas.so** Linux DSO when building for the device, and against the **cublasemu.so** Linux DSO when building for device emulation.

See The **CUDA CUBLAS Library** document available from www.nvidia.com for more information regarding functions for this library.

Chapter 5. Compilers

This chapter describes the following topics:

5.1 Overview

Compilers play an essential role in exploiting the full potential of Xeon® processors. Bull therefore recommends the use of **Intel**® C/C++ and **Intel**® **Fortran** compilers.

GNU compilers are also available. However, these compilers are unable to compile/link any program which uses **MPI_Bull**. For **MPI_Bull** programs it is essential that Intel compilers are used.

Alternatively, clusters that use **NVIDIA Tesla** graphic accelerators connected to the Compute Nodes will use the compilers supplied with the **NVIDIA CUDA™ Toolkit** and **Software Development Kit.**

5.2 Intel Tools

5.2.1 Intel® Fortran Compiler Professional Edition for Linux

The current version of the Intel® Fortran compiler is version 11. This supports the Fortran 95, Fortran 90, Fortran 77, Fortran IV standards whilst including many features from the Fortran 2003 language standard.

The main features of this compiler are:

- Advanced optimization features including auto-vectorization, High-Performance Parallel Optimizer (HPO), Interprocedural Optimization (IPO), Profile Guided Optimization (PGO) and Optimized Code Debugging.
- Multi-threaded Application Support including OpenMP and Auto Parallelization to convert serial applications into parallel applications to fully exploit the processing power that is available
- Data preloading
- Loop unrolling

The Professional Edition includes the **Intel® Math Kernel Library (Intel® MKL)** with its optimized functions for maths processing. It is also compatible with GNU products.
It also supports big endian encoded files. Finally, this compiler allows the execution of applications, which combine programs written in C and Fortran.

See www.intel.com for more details.

Different versions of the compiler may be installed to ensure compatibility with the compiler versions used to compile the libraries and applications on your system.

Note It may be necessary to contact the System Administrator to ascertain the location of the compilers on your system. The paths shown in the examples below may vary.

To specify a particular environment use the command below.

```
source /opt/intel/Compiler/<maj_ver_nb>/<min_ver_nb>/bin/ifortvars.sh intel64
```

For example:

- To use version `11.1.069` of the Fortran compiler:

```
source /opt/intel/Compiler/11.1/069/bin/ifortvars.sh intel64
```

- To display the version of the active compiler, enter:

```
ifort --version
```

- To obtain the compiler documentation go to:

```
/opt/intel/Compiler/11.1/069/Documentation
```

Remember that if you are using **MPI_Bull** then a compiler version has to be used which is compatible with the compiler originally used to compile the MPI library.

5.2.2 Intel® C^{++} Compiler Professional Edition for Linux

The current version of the Intel C^{++} compiler is version 11.

The main features of this compiler are:

- Advanced optimization features including auto-vectorization, High-Performance Parallel Optimizer (HPO), Interprocedural Optimization (IPO), Profile Guided Optimization (PGO) and Optimized Code Debugging.
- Multi-threaded Application Support including OpenMP and Auto Parallelization to convert serial applications into parallel applications to fully exploit the processing power that is available
- Data preloading
- Loop unrolling

The Professional Edition includes **Intel® Threading Building Blocks (Intel® TBB), Intel Integrated Performance Primitives (Intel® IPP)** and the **Intel® Math Kernel Library (Intel® MKL)** with its optimized functions for maths processing. It is also compatible with GNU products.

See www.intel.com for more details.

Different versions of the compiler may be installed to ensure compatibility with the compiler version used to compile the libraries and applications on your system.

Note It may be necessary to contact the System Administrator to ascertain the location of the compilers on your system. The paths shown in the examples below may vary.

To specify a particular environment use the command below:

```
source /opt/intel/Compiler/<maj_ver_nb>/<min_ver_nb>/bin/iccvars.sh intel64
```

For example:

- To use version `11.1.069` of the C/C++ compiler:

```
source /opt/intel/Compiler/11.1/069/bin/iccvars.sh intel64
```

- To display the version of the active compiler, enter:

```
icc --version
```

- To obtain the compiler documentation go to:

```
/opt/intel/Compiler/11.1/069/Documentation
```

Remember that if you are using **MPI_Bull** then a compiler version has to be used which is compatible with the compiler originally used to compile the MPI library.

5.2.3 Intel Compiler Licenses

Three types of Intel ® Compiler licenses are available:

- **Single User:** allows one user to operate the product on multiple computers as long as only one copy is in use at any given time.
- **Node-Locked:** locked to a node, allows any user who has access to this node to operate the product concurrently with other users, limited to the number of licenses purchased.
- **Floating:** locked to a network, allows any user who has access to the network server to operate the product concurrently with other users, limited to the number of licenses purchased.

The node-locked and floating licenses are managed by **FlexLM** from **Macrovision**. License installation, and **FlexLM** configuration, may differ according to your compiler, the license type, the number of licenses purchased, and the period of support for your product. Please check the Bull Product Designation document delivered with your compiler and follow the instructions contained therein.

5.2.4 Intel Math Kernel Library Licenses

Intel Math Kernel Library licenses are required for each Node on which you compile with **MKL**. However, the runtime libraries which are used on the compute nodes do not require a license fee.

5.3 GNU Compilers

GCC, a collection of free compilers that can compile both C/C++ and Fortran, is part of the installed Linux distribution.

5.4 NVIDIA nvcc C Compiler

For clusters which include NVIDIA Tesla graphic accelerators the NVIDIA Compute Unified Device Architecture (CUDA™) Toolkit is installed automatically on the LOGIN, COMPUTE and COMPUTEX nodes. The NVIDIA CUDA™ Toolkit provides a C development environment that includes the nvcc compiler. This compiler provides command line options to invoke the different tools required for each compilation stage.

nvcc's basic workflow consists in separating device code from host code and compiling the device code into a binary form or cubin object. The generated host code is outputted, either as C code that can be compiled using another tool, or directly as object code that invokes the host compiler during the last compilation stage.

Source files for CUDA applications consist of a mixture of conventional C++ 'host' code and graphic accelerator device functions. The CUDA compilation trajectory separates the device functions from the host code, compiles the device functions using proprietary NVIDIA compilers/assemblers, compiles the host code using the general purpose C/C++ compiler that is available on the host platform, and afterwards embeds the compiled graphic accelerator functions as load images in the host object file. In the linking stage, specific CUDA runtime libraries are added to support remote SIMD procedure calls and to provide explicit GPU manipulations, such as allocation of GPU memory buffers and host-GPU data transfer.

Important

Whenever double precision calculations are made on CUDA devices, it is imperative to use the **–arch=sm_13** switch on the **nvcc** command line, as shown below.

```
nvcc -arch=sm_13 code.cu -o executable
```

If the NVIDIA T20 accelerator is used the switch **-arch=sm_20** can be used.

See the NVIDIA CUDA docmentation available from www.nvidia.com for more information.

The compilation trajectory involves several splitting, compilation, preprocessing, and merging steps for each CUDA source file. These steps are subtly different for different modes of CUDA compilation (such as compilation for device emulation, or the generation of 'fat device code binaries'). It is the purpose of the CUDA nvcc compiler driver to keep the intricate details of CUDA compilation hidden from developers. Additionally, instead of being a specific CUDA compilation driver, nvcc mimics the behavior of general purpose compiler drivers, e.g. GCC, in that it accepts a range of conventional compiler options, for example to define macros and include/library paths, and to manage the compilation process. All non-CUDA compilation steps are forwarded to the general C compiler that is available on the platform.

5.4.1 Compiling with nvcc and MPI

Note Only C and C++ formats are accepted in the CUDA programming environment. Fortran programs should call the functions from C or C++ libraries. The user can program in any language (Python, etc.) as long as the C/C++ routines are called.

To use the CUDA programming environment with MPI:

1. Set the environment:

```
module load cuda
```

2. Compile the CUDA **.cu** files with the **nvcc** compiler to obtain **.o** object files.

```
nvcc -c kernel.cu
```

3. Link the object files with, for example, **mpif90** adding the **-lcudart** option to generate the executable.

```
mpif90 mpi_cuda.f90 kernel.o -lcudart -o executable
```

See The NVIDIA CUDA *Compute Unified Device Architecture Programming Guide* and *The CUDA Compiler Driver* document available from www.nvidia.com for more information.

5.5 Compiler Optimization Options

One of the most important ways of generating efficient executables is to examine closely the compiler optimization options. A single set of optimization options does not 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.

5.5.1 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).

-alias-args Assume arguments may be aliased.

-falias Assume aliasing in the program.

-ffnalias Assume aliasing within functions.

5.5.2 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.

Fortran Compilers

-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)

C/C++ Compilers

-Qoption,c,-ip_ninl_min_stats=n
Modifies the number of inlining levels (by default this is 15)

-Qoption,c,-ip_ninl_max_total_stats=n
Modifies the number of lines added when inlining (by default n = 2000)

Fortran and C/C++ Compilers

-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 the 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

5.5.3 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 and creates extra lapse 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.

5.5.4 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
```

5.5.5 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:

```
 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.

5.5.6 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:

- **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).

5.5.7 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 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 6. The User's Environment

This chapter describes how to access the extreme computing environment, how to use file systems, and how to use the modules package to switch and compare environments:

6.1 Cluster Access and Security

Typically, users connect to and use a cluster as described below:

- Users log on to the cluster platform either through Service Nodes or through the Login Node when the configuration includes these special Login Node(s). Once logged on to a node, users can then launch their jobs.
- Compilation is possible on all nodes which have compilers installed on them. The best approach is that compilers reside on Login Nodes, so that they do not interfere with performance on the Compute Nodes.

6.1.1 ssh (Secure Shell)

The **ssh** command is used to access a cluster node.

Syntax:

```
ssh [-l login_name] hostname | user@hostname [command]
```

```
ssh [-afgknqstvxACNTX1246] [-b bind_address] [-c cipher_spec]
    [-e escape_char] [-i identity_file] [-l login_name] [-m mac_spec]
    [-o option] [-p port] [-F configfile] [-L port:host:hostport]
    [-R port:host:hostport] [-D port] hostname | user@hostname [command]
```

ssh (ssh client) can also be used as a command to log onto a remote machine and to execute commands on it. It replaces **rlogin** and **rsh**, and provides secure encrypted communications between two untrusted hosts over an insecure network. X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel. **ssh** connects and logs onto the specified hostname. The user must verify his/her identity, using the appropriate protocol, before being granted access to the remote machine.

6.2 Global File Systems

The bullx cluster suite uses the **NFS** distributed file system.

6.3 Environment Modules

Environment modules provide a great way to customize your shell environment easily, particularly on the fly.

For instance an environment can consist of one set of compatible products including a defined release of a FORTRAN compiler, a C compiler, a debugger and mathematical libraries. In this way you can easily reproduce trial conditions, or use only proven environments.

The Modules environment is a program that can read and list module files returning commands; suitable for the shell to interpret, and most importantly for the **eval** command. Modulefiles is a kind of flat database which uses files.

In UNIX a child process can not modify its parent environment.
So how does Modules do this? Modules parses the given modules file and produces the appropriate shell commands to **set/unset/append/un-append** onto an environment variable. These commands are eval'd by the shell. Each shell provides some mechanism where commands can be executed and the resulting output can, in turn, be executed as shell commands. In the C-shell & Bourne shell and derivatives this is the **eval** command.

This is the only way that a child process can modify the parent's (login shell) environment. Hence the module command itself is a shell alias or function that performs these operations. To the user, it looks just like any other command.

The module command is only used in the development environment and not in other environments such as that for administration node.

See http://modules.sourceforge.net/ for more details.

6.3.1 Using Modules

The following command gives the list of available modules on a cluster.

```
module avail
```

```
------------------------ /opt/modules/version ------------------------
3.1.6

------------------- /opt/modules/3.1.6/modulefiles -------------------
dot         module-info null
module-cvs  modules     use.own

---------------------- /opt/modules/modulefiles ----------------------
oscar-modules/1.0.3 (default)
```

Modules available for the user are listed under the line **/opt/modules/modulefiles**.

The command to load a module is:

```
module load module_name
```

The command to verify the loaded modules list is:

```
module list
```

Using the **avail** command it is possible that some modules will be marked *(default):*

```
module avail
```

These modules are those which have been loaded without the user specifying a module version number. For example the following commands are the same:

```
module load configuration
module load configuration/2
```

The **module unload** command unloads a module.

The **module purge** command clears all the modules from the environment.

```
module purge
```

It is not possible to load modules which include different versions of **intel_cc** or **intel_fc** at the same time because they cause conflicts.

Module Configuration Examples

Note	The configurations shown below are examples only. The module configurations for bullx cluster suite will differ.

Configuration/1	intel_fc –version 10.0.046 intel_cc –version 10.0.066 intel_db –version 9.1.3 intel_mkl –version 9.0.017
Configuration/2	intel_fc –version 10.0.049 intel_cc –version 10.0.071 intel_db –version 9.1.3 intel_mkl –version 9.0.017
Configuration/3	intel_fc –version 10.0.061 intel_cc –version 10.0.071 intel_db –version 9.1.3 intel_mkl –version 9.0.017
Configuration/4	intel_fc –version10.0.019 intel_cc –version 10.0.022 intel_db –version 9.1.3 intel_mkl –version 90.017

Table 6-1.	Examples of different module configurations

6.3.2 Setting Up the Shell RC Files

A quick tutorial on Shell rc (run-command) files follows. When a user logs in and if they have **/bin/csh(/bin/sh)** as their shell, the first **rc** fire to be parsed by the shell is **/etc/csh.login** & **/etc/csh.cshrc (/etc/profile)** (the order is implementation dependent), and then the user's $HOME/.cshrc ($HOME/.kshenv) and finally $HOME/.login ($HOME/.profile).

All the other login shells are based on **/bin/csh** and **/bin/sh** with additional features and **rc** files. Certain environment variables and aliases (functions) need to be set for Modules to work correctly. This is handled by the Module init files in **/opt/modules /default/init**, which contains separate init files for each of the various supported shells, where the default is a symbolic link to a module command version.

Skeleton Shell RC (Dot) Files

The skeleton files provide a 'default' environment for new users when they are added to your system, this can be used if you do not have the time to set them up individually. The files are usually placed in **/etc/skel** (or wherever you specified with the **--with-skel-path=<path> option** to the configuration script), and contains a minimal set of **dot**' files and directories that every new user should start with.

The skeleton files are copied to the new user's $HOME directory with the **-m** option added to the **useradd** command. A set of sample dot'' files are located in **./etc/skel**. Copy everything but the .*.in and CVS files and directories to the skeleton directory. Edit and tailor for your system.

If you have a pre-existing set of skeleton files, then make sure the following minimum set exists: **.cshrc**, **.login**, **.kshenv**, **.profile**. These can be automatically updated with the command:

```
env HOME=/etc/skel/opt/modules/default/bin/add.modules
```

Inspect the new 'dot' files and if they are OK, then remove all the .*.old (original) files. An alternative way of setting-up the users' dot files can be found in **./ext**.
This model can be used with the **--with-dot-ext** configure option.

User Shell RC (Dot) Files

The final step for a functioning modules environment is to modify the user **dot** files to source the right files. One way to do this is to put a message in the **/etc/motd** telling each user to run the command:

```
/opt/modules/default/bin/add.modules
```

This is a script, which parses their existing dot files, prepending the appropriate commands to initialize the Modules environment.

The user can re-run this script and it will find and remember what modules they initially loaded and then strip out the previous module initialization and restore it with an upgraded one.

If the user lacks a necessary 'dot' file, the script will copy one over from the skeleton directory. The user will have to logout and login for it to come into effect.
Another way is for the system administrator to **su - username** to each user and run it interactively. The process can be semi-automated with a single line command that obviates the need for direct interaction:

```
su - username -c "yes | /opt/modules/modules/default/bin/add.modules"
```

Power users can create a script to directly parse the **/etc/passwd** file to perform this command. Otherwise, just copy the **passwd** file and edit it to execute this command for each valid user.

6.4 Module Files

Once the above steps have been performed, then it is important to have module files in each of the **modulefiles** directories. For example, the following module files will be installed:

```
--------- /opt/modules/3.0.9-rko/modulefiles ----------
dot          module-info modules     null        use.own
```

If you do not have your own module files in **/opt/modules/modulefiles** then copy 'null' to that directory. On some systems an empty modulefiles directory will cause a core dump, whilst on other systems there will be no problem. Use **/opt/modules/default/modulefiles/modules** as a template for creating your own module files.

For more information run:

```
module load modules
```

You will then have ready access to the module(1) modulefile(4) man pages, as well as the versions directory. Study the man pages carefully.

The version directory may look something like this:

```
--------------- /opt/modules/versions ---------------
3.0.5-rko 3.0.6-rko 3.0.7-rko 3.0.8-rko 3.0.9-rko
```

The model you should use for modulefiles is **name/version**. For example, **/opt/modules/modulefiles** directory may have a directory named **firefox** which contains the following module files: 301, 405c, 451c, etc.
When it is displayed with **module avail** it looks something like this:

```
firefox/301
firefox/405c
firefox/451c(default)
firefox/45c
firefox/46
```

The default is established with **.version** file in the **FireFox** directory and it looks something like this:

```
#%Module1.0###########################################################
##
## version file for Firefox
##
set ModulesVersion      "451c"
```

If the user does module load **firefox**, then the default firefox/451c will be used. The default can be changed by editing the **.version** file to point to a different module file in that directory. If no **.version** file exists then Modules will just use the last module in the alphabetical ordered directory listed as the default.

6.4.1 Upgrading via the Modules Command

The theory is that Modules should use a similar package/version locality as the package environments it helps to define. Switching between versions of the **module** command should be as easy as switching between different packages via the **module** command. Suppose there is a change from 3.0.5-rko to version 3.0.6-rko. The goal is to semi-automate the changes to the user 'dot' files so that the user is oblivious to the change.

The first step is to install the new module command & files to **/opt/modules/3.0.6-rko/**. Test it out by loading with '**module load modules 3.0.6-rko**'. You may get an error like: 3.0.6-rko (25):ERROR:152: Module 'modules' is currently not loaded. This is OK and should not appear with future versions.

Make sure you have the new version with **module –version**. If it seems stable enough, then advertise it to your more adventurous users. Once you are satisfied that it appears to work adequately well, then go into **/opt/modules** remove the old **default** symbolic link to the new versions.

For example

```
cd /opt/modules
rm default; ln -s 3.0.6-rko default
```

This new version is now the default and will be referenced by all the users that log in and by those that have not loaded a specific module command version.

6.5 The Module Command

Synopsis

```
module [ switches ] [ sub-command ] [ sub-command-args ]
```

The **Module** command provides a user interface to the Modules package. The Modules package provides for the dynamic modification of the user's environment via *modulefiles*.

Each *modulefile* contains the information needed to configure the shell for an application. Once the Modules package is initialized, the environment can be modified on a per-module basis using the module command which interprets *modulefiles*. Typically, *modulefiles* instruct the **module** command to alter or to set shell environment variables such as PATH, MANPATH, etc. *modulefiles* may be shared by many users on a system and users may have their own collection to supplement or replace the shared *modulefiles*.

The *modulefiles* are added to and removed from the current environment by the user. The environment changes contained in a *modulefile* can be summarized through the module command as well. If no arguments are given, a summary of the module usage and sub-commands are shown.

The action for the module command to take is described by the sub-command and its associated arguments.

6.5.1 modulefiles

modulefiles are the files containing **TCL** code for the Modules package.

modulefiles are written in the Tool Command Language, **TCL**(3) and are interpreted by the **modulecmd** program via the module(1) user interface. **modulefiles** can be loaded, unloaded, or switched on-the-fly while the user is working.

A modulefile begins with the magic cookie, '#%Module'. A version number may be placed after this string. The version number is useful as the format of **modulefiles** may change. If a version number does not exist, then **modulecmd** will assume the modulefile is compatible with the latest version. The current version for **modulefiles** will be 1.0. Files without the magic cookie will not be interpreted by **modulecmd**.

Each modulefile contains the changes to a user's environment needed to access an application. **TCL** is a simple programming language which permits **modulefiles** to be arbitrarily complex, depending on the needs of the application and the modulefile writer. If support for extended tcl (tclX) has been configured for your installation of modules, you may also use all the extended commands provided by tclX. **modulefiles** can be used to implement site policies regarding the access and use of applications.

A typical **modulefiles** file is a simple bit of code that sets or adds entries to the PATH, MANPATH, or other environment variables. **TCL** has conditional statements that are evaluated when the modulefile is loaded. This is very effective for managing path or environment changes due to different OS releases or architectures. The user environment information is encapsulated into a single modulefile kept in a central location. The same modulefile is used by all users independent of the machine. So, from the user's perspective, starting an application is exactly the same regardless of the machine or platform they are on.

modulefiles also hide the notion of different types of shells. From the user's perspective, changing the environment for one shell looks exactly the same as changing the environment for another shell. This is useful for new or novice users and eliminates the need for statements such as *if you're using the C Shell do this ..., otherwise if you're using the Bourne shell do this ...* Announcing and accessing new software is uniform and independent of the user's shell. From the modulefile writer's perspective, this means one set of information will take care of all types of shells.

Example of a Module file

```
#%Module1.0############################################################
####
##
## C/C++
##

set INTEL intel_cc

module-whatis   "loads the icc 10.1.011 (Intel C/C++) environment for
EM64T"

set iccroot /opt/intel/cce/10.1.011

prepend-path    PATH            $iccroot/bin
prepend-path    LD_LIBRARY_PATH $iccroot/lib
setenv          MANPATH         :$iccroot/man
prepend-path    INTEL_LICENSE_FILE
$iccroot/licenses:/opt/intel/licenses
```

6.5.2 Modules Package Initialization

The Modules package and the module command are initialized when a shell-specific initialization script is sourced into the shell. The script creates the module command as either an alias or function, creates Modules environment variables, and saves a snapshot of the environment in **${HOME }/.modulesbeginenv**. The module alias or function executes the modulecmd program located in ${MODULESHOME }/bin and has the shell evaluate the command's output. The first argument to modulecmd specifies the type of shell.

The initialization scripts are kept in ${MODULESHOME }/init/shellname where shellname is the name of the sourcing shell. For example, a C Shell user sources the ${MODULESHOME }/init/csh script. The **sh**, **csh**, **tcsh**, **bash**, **ksh**, and **zsh** shells are all supported by **modulecmd**. In addition, **PYTHON** and **PERL** shells are supported which writes the environment changes to **stdout** as **PYTHON** or **PERL** code.

6.5.3 Examples of Initialization

In the following examples, replace ${MODULESHOME } with the actual directory name.

C Shell initialization (and derivatives)

```
source ${MODULESHOME }/init/csh module load modulefile modulefile
```

Bourne Shell (sh) (and derivatives)

```
${MODULESHOME }/init/sh module load modulefile modulefile
```

Perl

```
require "${MODULESHOME }/init/perl"; &module("load modulefile modulefile ");
```

6.5.4 Modulecmd Startup

Upon invocation, **modulecmd** sources **rc** files which contain global, user and *modulefile* specific setups. These files are interpreted as **modulefiles.**

Upon invocation of **modulecmd** module RC files are sourced in the following order:

1. Global RC file as specified by ${MODULERCFILE } or ${MODULESHOME }/etc/rc
2. User specific module RC file ${HOME }/.modulerc
3. All .module rc and .version files found during modulefile searches.

6.5.5 Module Command Line Switches

The module command accepts command line switches as its first parameter. These may be used to control output format of all information displayed and the module behaviour in the case of locating and interpreting module files.

All switches may be entered either in short or long notation. The following switches are accepted:

--force, -f
Force active dependency resolution. This will result in modules found using a **prereq** command inside a module file being loaded automatically. Unloading module files using this switch will result in all required modules which have been loaded automatically using the -f switch being unloaded. This switch is experimental at the moment.

--terse, -t
Display avail and list output in short format.

--long, -l
Display avail and list output in long format.

--human, -h
Display short output of the avail and list commands in human readable format.

--verbose, -v
Enable verbose messages during module command execution.

--silent, -s
Disable verbose messages. Redirect **stderr** to **/dev/null** if **stderr** is found not to be a **tty**. This is a useful option for module commands being written into **.cshrc** , **.login** or .profile files, because some remote shells (e.g. **rsh** (1)) and remote execution commands (e.g. **rdist**) get confused if there is output on **stderr**.

--create, -c
Create caches for module **avail** and module **apropos**. You must be granted write access to the ${MODULEHOME }/*modulefiles/* directory if you try to invoke module with the **-c** option.

--icase, -i
This is a case insensitive module parameter evaluation. Currently only implemented for the module apropos command.

--userlvl <lvl>, -u <lvl>

Set the user level to the specified value. The argument of this option may be one of:

novice	nov Novice
expert	exp Experienced module user
advanced	adv Advanced module user

6.5.6 Module Sub-Commands

- Print the use of each sub-command. If an argument is given, print the Module specific help information for the `modulefile`.

```
help [modulefile...]
```

- Load `modulefile` into the shell environment.

```
load modulefile [modulefile...]
add modulefile [modulefile...]
```

- Remove `modulefile` from the shell environment.

```
unload modulefile [modulefile...]
rm modulefile [modulefile...]
```

- Switch loaded `modulefile1` with `modulefile2`.

```
switch modulefile1 modulefile2
swap modulefile1 modulefile2
```

- Display information about a `modulefile`. The display sub-command will list the full path of the modulefile and all (or most) of the environment changes the modulefile will make when loaded. (It will not display any environment changes found within conditional statements).

```
display modulefile [modulefile...]
```

- List loaded modules.

```
show modulefile [modulefile...]
list
avail [path...]
```

- List all available modulefiles in the current MODULEPATH. All directories in the MODULEPATH are recursively searched for files containing the modulefile magic cookie. If an argument is given, then each directory in the MODULEPATH is searched for modulefiles whose pathname match the argument. Multiple versions of an application can be supported by creating a subdirectory for the application containing modulefiles for each version.

```
use directory [directory...]
```

- Prepend directory to the MODULEPATH environment variable. The --append flag will append the directory to MODULEPATH.

```
use [-a|--append] directory [directory...]
```

- Remove directory from the MODULEPATH environment variable.

```
unuse directory [directory...]
```

- Attempt to reload all loaded modulefiles. The environment will be reconfigured to match the saved ${HOME }/**.modulesbeginenv** and the modulefiles will be reloaded. The `update` command will only change the environment variables that the modulefiles set.

```
update
```

- Force the Modules Package to believe that no modules are currently loaded.

```
clear
```

Unload all loaded modulefiles.

```
purge
```

- Display the `modulefile` information set up by the module-**whatis** commands inside the specified modulefiles. If no modulefiles are specified, all the **whatis** information lines will be shown.

```
whatis [modulefile [modulefile...]]
```

- Searches through the **whatis** information of all modulefiles for the specified string. All module **whatis** information matching the search string will be displayed.

```
apropos string
keyword string
```

- Add modulefile to the shell's initialization file in the user's home directory. The startup files checked are .cshrc, .login, and .csh_variables for the C Shell; .profile for the Bourne and Korn Shells; **.bashrc, .bash_env**, and **.bash_profile** for the GNU Bourne Again Shell; **.zshrc, .zshenv**, and **.zlogin** for zsh. The .modules file is checked for all shells. If a **module load** line is found in any of these files, the modulefile(s) is(are) appended to any existing list of modulefiles. The 'module load' line must be located in at least one of the files listed above for any of the 'init' sub-commands to work properly. If the **module load** line is found in multiple shell initialization files, all of the lines are changed.

```
initadd modulefile [modulefile...]
```

- Does the same as **initadd** but prepends the given modules to the beginning of the list. `initrm modulefile [modulefile...]` Remove modulefile from the shell's initialization files.

```
initprepend modulefile [modulefile...]
```

- Switch `modulefile1` with `modulefile2` in the shell's initialization files.

```
initswitch modulefile1 modulefile2
```

- List all of the modulefiles loaded from the shell's initialization file.

```
initlist
```

- Clear all of the modulefiles from the shell's initialization files.

```
initclear
```

6.5.7 Modules Environment Variables

Environment variables are unset when unloading a modulefile. Thus, it is possible to load a modulefile and then unload it without having the environment variables return to their prior state.

MODULESHOME
This is the location of the master Modules package file directory containing module command initialization scripts, the executable program **modulecmd**, and a directory containing a collection of master modulefiles.

MODULEPATH
This is the path that the module command searches when looking for modulefiles. Typically, it is set to the master modulefiles directory, ${MODULESHOME }/modulefiles, by the initialization script. MODULEPATH can be set using **module use** or by the module initialization script to search group or personal modulefile directories before or after the master modulefile directory.

LOADEDMODULES
A colon separated list of all loaded modulefiles.

_LOADED_MODULEFILES_
A colon separated list of the full pathname for all loaded modulefiles.

MODULESBEGINENV
The filename of the file containing the initialization environment snapshot.

Files

/opt
The MODULESHOME directory.

${MODULESHOME}/etc/rc
The system-wide modules rc file. The location of this file can be changed using the MODULERCFILE environment variable as described above.

${HOME}/.modulerc
The user specific modules rc file.

${MODULESHOME}/*modulefiles*
The directory for system-wide modulefiles. The location of the directory can be changed using the MODULEPATH environment variable as described above.

${MODULESHOME}/bin/modulecmd
The modulefile interpreter that is executed upon each invocation of a module.

${MODULESHOME}/init/shellname
The Modules package initialization file sourced into the user's environment.

${MODULESHOME}/init/.modulespath
The initial search path setup for module files. This file is read by all shell init files.

${MODULEPATH}/.moduleavailcache
File containing the cached list of all modulefiles for each directory in the MODULEPATH (only when the avail cache is enabled).

${MODULEPATH}/.moduleavailcachedir
File containing the names and modification times for all sub-directories with an avail cache.

${HOME}/.modulesbeginenv
A snapshot of the user's environment taken when Modules are initialized. This information is used by the module update sub-command.

6.6 The NVIDIA CUDA Development Environment

For clusters which include **NVIDIA Tesla** graphic accelerators the **NVIDIA** Compute Unified Device Architecture **(CUDA™) Toolkit** is installed automatically on the **LOGIN**, **COMPUTE** and **COMPUTEX** nodes so that the **NVIDIA nvcc C** compiler is in place for the application.

CUDA is a parallel programming environment designed to scale parallelism so that all the processor cores available are exploited. As it builds on C extensions the **CUDA** development environment is easily mastered by application developers.

At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronizations.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel. Such decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables transparent scalability since each sub-problem can be scheduled to be solved on any of the available processor cores. A compiled CUDA program can therefore execute on any number of processor cores, and only the runtime system needs to know the physical processor count.

See The **NVIDIA CUDA** *Compute Unified Device Architecture Programming Guide* and the other documents in the **/opt/cuda/3.0/doc** directory for more information.

6.6.1 GPUSET library

For architectures that include several IOHs and multiple GPUs - see example below, the GPU access is non- uniform, and depends on which processor is running the application.

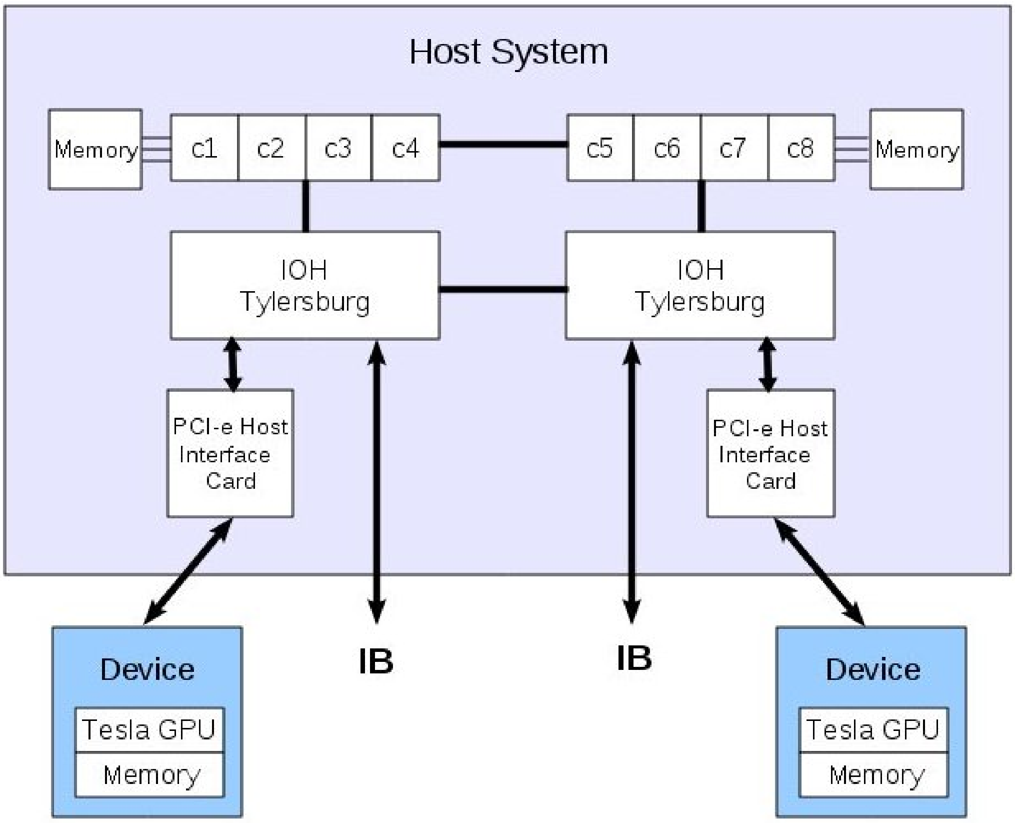


Figure 6-1. Typical architecture for NVIDIA Tesla GPUs and Bullx B5xx blades

In the **CUDA** environment, the **cudaSetDevice ()** function selects a **GPU** according to different parameters: memory size, the GPU version, etc., but NOT the position of the GPU in the machine. In an architecture with two IOHs, as shown in the example above, access to a GPU is penalized if the application runs on a CPU that is not connected to the same IOH as the GPU. Therefore, it is essential to ensure that the correct GPU is allocated to the application. The GPUSet library is used to do this as follows.

1. Install the **libgpuset** RPM:

```
yum install libgpuset
```

2. Preload the library by using the **LD_PRELOAD** variable, or by using a script, BEFORE launching the application that will use the GPU:

```
LD_PRELOAD=libgpuset;$LD_PRELOAD; <application name>
```

The library will override the call to **cudaSetDevice** and look for a **GPU** connected to the same **IOH** as the processor being used. The CUDA library will call **cudaSetDevice** using the GPU that is nearest, overriding the parameters set in the initial **cudaSetDevice** call, and so the application execution will not be penalized by non-uniform GPU access.

Note Even if a particular GPU has been specified by the original **cudaSetDevice** call, it will be ignored by the library and the nearest GPU will be used. Do not preload the library if, for the purposes of experimentation, etc., you would like to specify a particular GPU.

6.6.2 bullx cluster suite and CUDA

The CUDA development environment is based on the **NVIDIA (CUDA™) Toolkit**, which includes the **nvcc** compiler and runtime libraries. The **NVIDIA Software Developer Kit (SDK)**, including utilities and project examples, is also delivered.

The CUDA Toolkit is delivered as **RPMs** and installed in **/opt/cuda/3.0** and includes the **bin**, **lib** and **man** sub directories. These files are sourced to load the **CUDA** environment variables by, for example by using the command below:

```
source /opt/cuda/3.0/bin/cuda.sh
```

Alternatively, the module can be loaded from the command line, for example:

```
module load cuda
```

NVIDIA recommends that the SDK is copied into the file system for each user. To do this a **makefile** is used, this produces around 60 MBs of binaries and libraries for each user. The SDK is installed in the **/opt/cuda/3.0/sdk** directory. A patch has been applied to some of the files in order to suppress the relative paths that obliged the user to develop inside **SDK**. These patches are mainly related to the **CUDA** environment and the **MPI** options provided for the **nvcc** compiler and linker.

Programme examples are included in the **/opt/cuda/3.0/sdk/projects** directory. These programmes and the use of **SDK** are not documented; however the source code can be examined to obtain an idea of developing a program in the CUDA environment.

SDK will be delivered precompiled to save time for the user and includes macros to help error tracking.

6.6.3 NVIDA CUDA™ Toolkit and Software Developer Kit

The **NVIDIA CUDA™ Toolkit** provides a complete **C** development environment including:

- The **nvcc** C compiler
- **CUDA FFT** and **BLAS** libraries
- A visual profiler
- A **GDB** debugger
- **CUDA** runtime driver
- **CUDA** programming manual

The **NVIDIA CUDA Developer Software Developer Kit** provides CUDA examples, with the source code, to help get started with the **CUDA** environment. Examples include:

- Matrix multiplication
- Matrix transpose
- Performance profiling using timers
- Parallel prefix sum (scan) of large arrays
- Parallel Mersenne Twister (random number generation)

See The **CUDA Zone** at www.nvidia.com for more examples of applications developed within the CUDA environment, and for additional development tools and help.

6.7 OpenCL

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs and other processors. OpenCL provides parallel computing, using task-based and data-based parallelism.

The OpenCL programming model is very close to CUDA, sharing a range of computational interfaces. The main advantage of OpenCL is its portability. The same code can be launched on NVIDA cards, and also on other kinds of hardware (such as AMD GPUs).

OpenCL support is available for NVIDIA C1060/C1070 and C2050/C2070 accelerators that install the CUDA 3.0 RPM. This can be used instead of CUDA.

Note Only OpenCL v.1.0 can be used with CUDA 3.0 and not v.1.1

There is no need for a special compiler. All the OpenCL runtime libraries are located in the libOpenCL library. For example, run the command below to compile OpenCL code:

```
icc opencl_prog.c -o exec -lOpenCL
```

See http://www.khronos.org/registry/cl/ for more information on OpenCL

Chapter 7. Launching an Application

Platform	Application		Launching tool
Clusters with no Resource Manager	Serial		none
	Parallel	OpenMP	none
		bullx MPI	mpirun
		MPIBull2	mpiexec/mpirun (MPD)
Clusters with SLURM	Serial		srun
	Parallel	OpenMP on one node	salloc srun –c <no. of CPUs>
		MPIBull2	srun
		bullx MPI	salloc mpirun
		Hybrid (MPIBull2 + OpenMP)	srun –c <no. of CPUs per MPI task>
Clusters with PBS PRO	Serial		none
	Parallel	OpenMP on one node	none
		MPIBull2	mpiexec/mpirun (MPD)
		bullx MPI	mpirun
		Hybrid (MPIBull2 + OpenMP)	mpirun
Clusters with LSF	Serial		none
	Parallel	OpenMP on one node	none
		MPIBull2	mpirun.lsf
		bullx MPI	mpirun
		Hybrid (MPIBull2 + OpenMP)	mpirun

Table 7-1. Launching an application without a Batch Manager for different clusters

7.1 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 will not 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.

7.1.1 Typical Usage of CPUSETS

- CPU-bound applications: Many applications (as it is often the case for cluster 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.

7.1.2 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.

7.2 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 that 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 that 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.

7.3 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.

7.3.1 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.

7.3.2 Improving Loops

Loops are very powerful programming devices, which in a few lines can result in a high amount of data 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. 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 blocks, 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 that is used for loops with a low number of iterations. It acts to 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.

7.3.3 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 that 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.

7.3.4 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.

7.3.5 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.

7.3.6 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 data 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
}
```

Chapter 8. Application Debugging Tools

8.1 Overview

There are two types of debuggers; symbolic ones and non-symbolic ones.

- A symbolic debugger gives access to a program's source code. This means that:
  - The lines of the source file can be accessed.
  - The program variables can be accessed by name.
- Whereas a non-symbolic debugger enables access to the lines of the machine code program only and to the top physical addresses.

The following debugging tools are described:

- 8.2 *GDB*
- 8.3 *IDB*
- 8.4 *TotalView*
- 8.5 DDT
- 8.6 *MALLOC_CHECK_ - Debugging Memory Problems in C programs*
- 8.7 *Electric Fence*

8.2 GDB

GDB stands for Gnu DeBugger. It is a powerful Open-source debugger, which can be used either through a command line interface, or a graphical interface such as XXGDB or DDD (Data Display Debugger). It is also possible to use an **emacs/xemacs** interface.

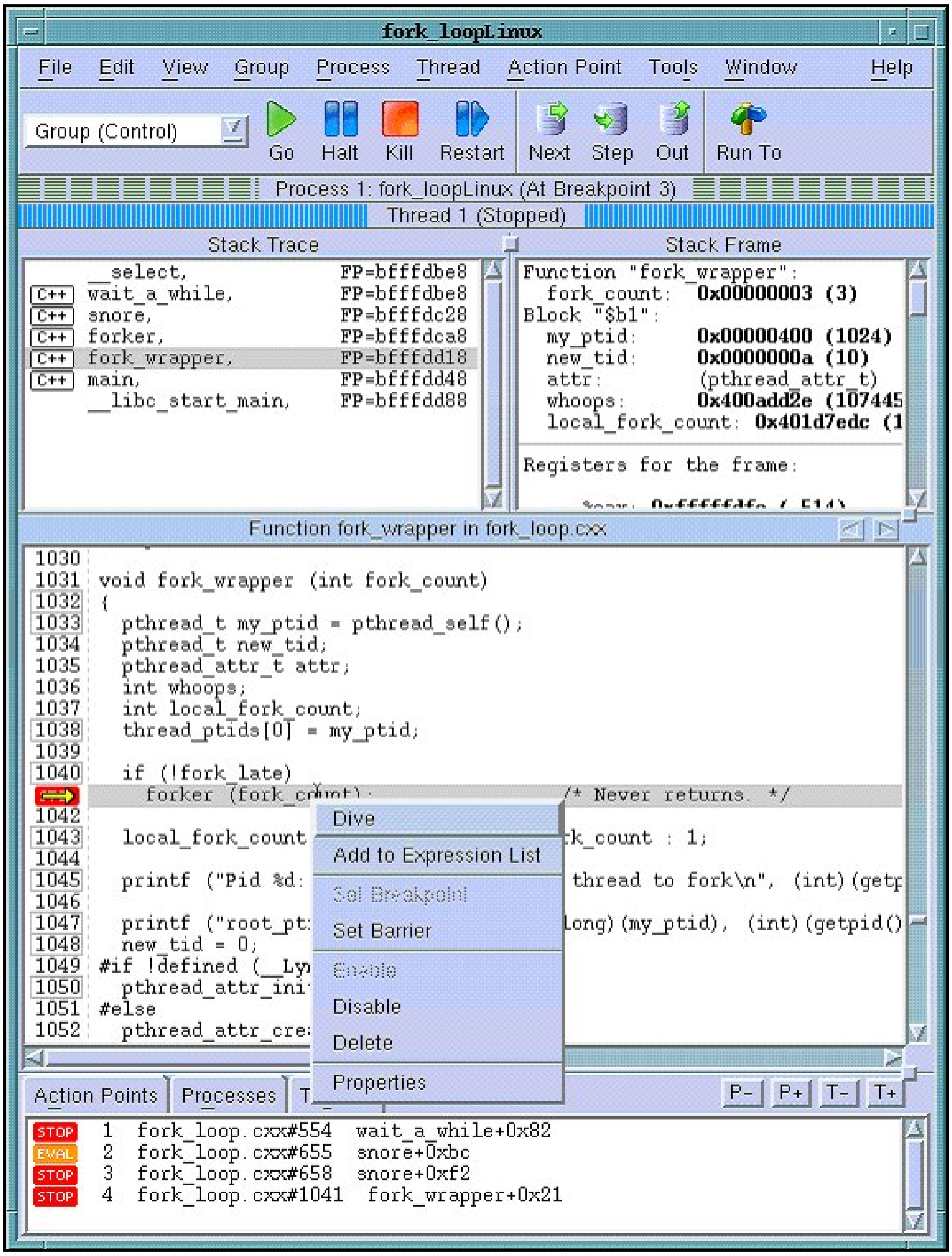
GDB supports parallel applications and threads.

GDB is published under the GNU license.

8.3 IDB

IDB is a debugger delivered with Intel compilers. It can be used with C/C++ and F90 programs.

8.4 TotalView



```
      __select,            FP=bfffdbe8
C++   wait_a_while,        FP=bfffdbe8
C++   snore,               FP=bfffdc28
C++   forker,              FP=bfffdca8
C++   fork_wrapper,        FP=bfffdd18
C++   main,                FP=bfffdd48
      __libc_start_main,   FP=bfffdd88
```

```
Function "fork_wrapper":
  fork_count:       0x00000003 (3)
Block "$b1":
  my_ptid:          0x00000400 (1024)
  new_tid:          0x0000000a (10)
  attr:             (pthread_attr_t)
  whoops:           0x400add2e (107445
  local_fork_count: 0x401d7edc (1

Registers for the frame:
```

```

void fork_wrapper (int fork_count)
{
  pthread_t my_ptid = pthread_self();
  pthread_t new_tid;
  pthread_attr_t attr;
  int whoops;
  int local_fork_count;
  thread_ptids[0] = my_ptid;

  if (!fork_late)
    forker (fork_count);                  /* Never returns. */

  local_fork_count                k_count : 1;

  printf ("Pid %d:                thread to fork\n", (int)(getp

  printf ("root_pt:               long)(my_ptid), (int)(getpid()
  new_tid = 0;
#if !defined (__Ly
  pthread_attr_ini
#else
  pthread_attr_cre
```

```
STOP  1  fork_loop.cxx#554   wait_a_while+0x82
EVAL  2  fork_loop.cxx#655   snore+0xbc
STOP  3  fork_loop.cxx#658   snore+0xf2
STOP  4  fork_loop.cxx#1041  fork_wrapper+0x21
```

Figure 8-1 Totalview graphical interface – image taken from http://www.totalviewtech.com/productsTV.htm

TotalView™ is a proprietary software application and is not included with the bullx cluster suite delivery. Totalview™ is used in the same way as standard symbolic debuggers for C, C++ and Fortran (77, 90 and HPF) programs. It can also debug **MPI** or **OpenMPI** applications. **TotalView**™ has the advantage of being a debugger which supports multi-processes and multi-threading. It can take control of the various processes or threads of the program and make it possible for the user to visualize the evolution of the execution in the same window or in different windows. The processes may be local or remote.
It works equally as well with mono-processor, SMP, clustered, distributed and MPP systems.

TotalView™ accepts new processes and threads exactly as generated by the application and regardless of the processor used for the execution. It is also possible to connect to a process started up outside TotalView™. Data tables can be filtered, displayed, and viewed in order to monitor the behavior of the program. Finally, you can descend *("call the components and details of...")* into the objects and structures of the program.

The program which needs debugging must be compiled with the '**- g**' option, and then breakpoints should be added to the program to control its execution.

TotalView™ is an XWindows application. Context-sensitive help provides you with basic information. You may download TotalView™ in the directory **/opt/totalview**.

Before running TotalView™, update your environment by using the following command:

```
source /opt/totalview/totalview-vars.sh
```

Then enter:

```
totalview&
```

See http://www.totalviewtech.com/productsTV.htm for additional information, and for copies of the documentation for **Totalview**™.

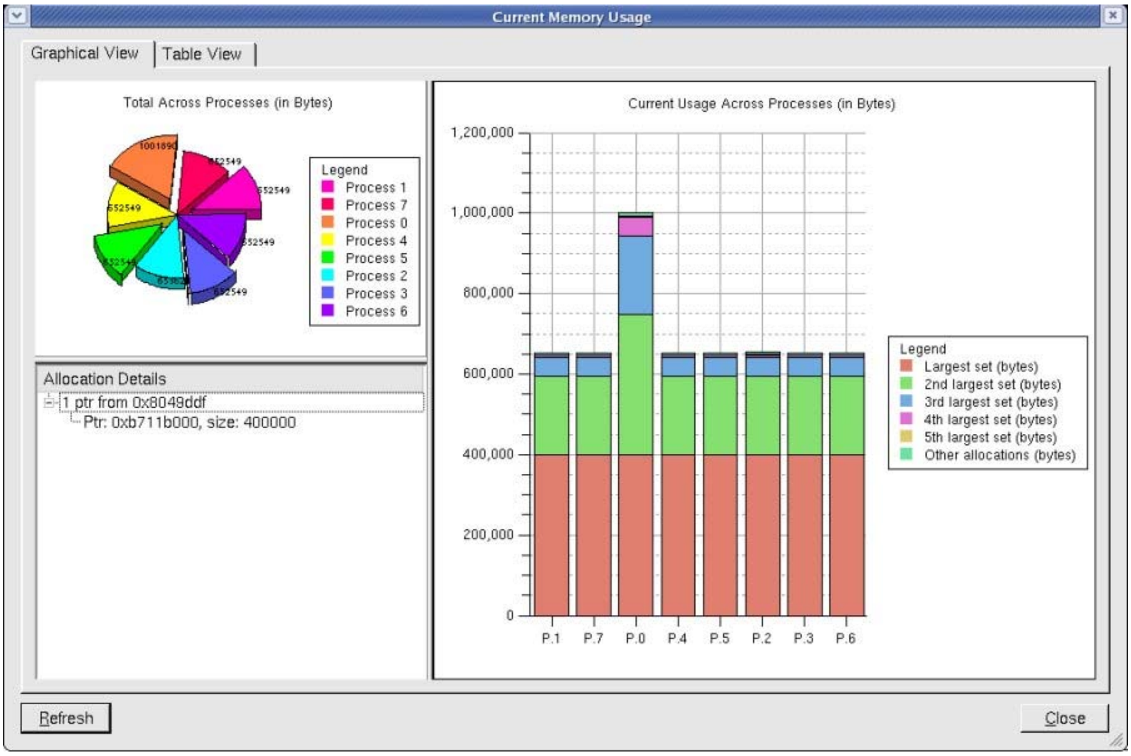
8.5 DDT

DDT™ is a proprietary debugging tool from **Allinea** and is not included with the **bullx cluster suite** delivery.

Its source code browser shows at a glance the state of the processes within a parallel job, and simplifies the task of debugging large numbers of simultaneous processes. DDT has a range of features designed to debug effectively - from deadlock and memory leak tools, to data comparison and group wise process control, and it interoperates with all known **MPIBull2** implementations

For multi-threaded or **OpenMP** development **DDT** allows threads to be controlled individually and collectively, with advanced capabilities to examine data across threads.

The Parallel Stack Viewer allows the program state of all processes and threads to be seen at a glance making parallel programs easier to manage.



Allocation Details

1 ptr from 0x8049ddf
Ptr: 0xb711b000, size: 400000

Figure 8-2. The Graphical User Interface for DDT

DDT can find memory leaks, and detect common memory usage errors before your program crashes.

A programmable STL Wizard enables C++ Standard Template Library variables and the abstract data they represent -including lists, maps, sets, multimaps, and strings – to be viewed easily.

Developers of scientific code have full access to modules, allocated data, strings and derived types for Fortran 77, 90, and 95.

MPI message queues can be examined in order to identify deadlocks in parallel code and data may be viewed in 3D with the multi-dimensional array viewer.

It is possible to run DDT with the PBS-Professional Batch Manager.

See http://allinea.com/ for more information refer.

8.6 MALLOC_CHECK_ - Debugging Memory Problems in C programs

When developing an application, the developer should ensure that all the buffers allocated during the run-time of the application are freed afterwards. However, even if he is vigilant, it is not unusual for memory leaks to be introduced into the code.

A simple way to detect these memory leaks is to use the environment variable MALLOC_CHECK __. This variable ensures that allocation routines check that each allocated buffer is freed correctly. The routines then become more 'tolerant' and allow byte overflows on both sides of blocks or for the block to be released again.
According to the value of MALLOC_CHECK __, when a release or allocation error appears the application behaves as follows:

- If MALLOC_CHECK __ is set to 1, an error message is written when exiting normally.
- If MALLOC_CHECK __ is set to 2, an error message is written when exiting normally and the process aborts. A core file is created. You should check that it is possible to create a core file by using the command *ulimit –c*. If not, enter the command *ulimit -c unlimited.*
- For any other value of MALLOC_CHECK __, the error is ignored and no message appears.

Example.c program

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 256

int main(void){

  char *buffer;

  buffer = (char *)calloc(256*sizeof(char));
  if(!buffer){
    perror(``malloc failed'');
    exit(-1);
  }

  strcpy(buffer, ``fills the buffer'');
  free(buffer);
  fprintf(stdout, ``Buffer freed for the first time'');
  free(buffer);
  fprintf(stdout,``Buffer freed for the second time'');
  return(0);

}
```

A program which is executed with the environmental variable MALLOC_CHECK __ set to 1 gives the following result:

$ export MALLOC_CHECK__=1

$./example

```
Buffer freed for the first time
Segmentation fault
```

$ ulimit –c 0

```
# The limit for the core file size must be changed to allow files bigger
than 0 bytes to be generated
```

```
$ ulimit –c unlimited    # Allows an unlimited core file to be generated
```

A program which is executed with the environmental variable MALLOC_CHECK __ set to 2 gives the following result:

```
$ export MALLOC_CHECK__=2
```

```
$ ./example
```

```
Buffer freed for the first time
Segmentation fault (core dumped)
```

Example Program Analysis using the GDB Debugger

The core file should be analyzed to identify where the problem is (the program should be compiled with the option - G):

```
$ gdb example -c core
```

```
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions. There is absolutely no
warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

Core was generated by `./example’.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x40097354 in mallopt () from /lib/libc.so.6
(gdb) bt
#0  0x40097354 in mallopt () from /lib/libc.so.6
#1  0x4009615f in free () from /lib/libc.so.6
#2  0x0804852f in main () at exemple.c:18
(gdb)
```

The **bt** command is used to display the current memory stack. In this example the last line indicates the problem came from line 18 in the main function of the **example.c** file. Looking at the **example.c** program on the previous page we can see that line 18 corresponds to the second call to the free function which created the memory overflow.

8.7 Electric Fence

Electric Fence is an open source **malloc** debugger for **Linux** and Unix. It stops your program on the exact instruction that overruns or under-runs a **malloc()** buffer.

Electric Fence is installed on the Management Node only.

Electric Fence helps you detect two common programming bugs:

- Software that overruns the boundaries of a **malloc()** memory allocation.
- Software that touches a memory allocation that has been released by **free()**.

You can use the following example, replacing **`icc --version`** by the command line of your program.

```
[test@host ]$LD_PRELOAD=/usr/local/tools/ElectricFence-
2.2.2/lib/libefence.so.0.0  icc --version
```

See http://perens.com/FreeSoftware/ for more information about Electric Fence.

Chapter 9. Application Profiling Tools

Different tools are available to monitor the performance of your application, and to help identify problems and to highlight where performance improvements can be made. These include:

- **PAPI**, an open source tool
- **HPC Toolkit**, an open source tool based on **PAPI**, is included in the **bullx cluster suite** delivery.
- **Intel Vtune** is used to perform post mortem analysis of the output after the application has completed its execution, and cannot be used during run-time.

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

9.1 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.
PAPI are all open source tools.

9.1.1 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.

9.1.2 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.

9.2 Profiling Programs with HPC Toolkit

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

An important advantage of HPC Toolkit over other profiling tools is that it does not require the use of compile-time profiling options or re-linking of the executable.

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

HPC Toolkit is designed to:

- *Work at binary level to ensure language independence*
  This enables HPC Toolkit to support the measurement and analysis of multi-lingual codes using external binary-only libraries.
- *Profile instead of adding code instrumentation*
  Sample-based profiling is less intrusive than code instrumentation, and uses a modest data volume.
- *Collect and correlate multiple performance metrics*
  Typically, performance problems cannot be diagnosed using only one type of event.
- *Compute derived metrics to help analysis*
  Derived metrics, such as the bandwidth used for the memory, often provide insights that will indicate where optimization benefits can be achieved.
- *Attribute costs very precisely*
  HPC Toolkit is unique in its ability to associate measurements in the context of dynamic calls, loops, and inlined code.

9.2.1 HPC Toolkit Workflow

The HPC Toolkit design principles led to the development of a general methodology, resulting in a workflow that is organized around four different capabilities:

- *Measurement* of performance metrics during the execution of an application
- *Analysis* of application binaries to reveal the program structure
- *Correlation* of dynamic performance metrics with the structure of the source code
- *Presentation* of performance metrics and associated source code

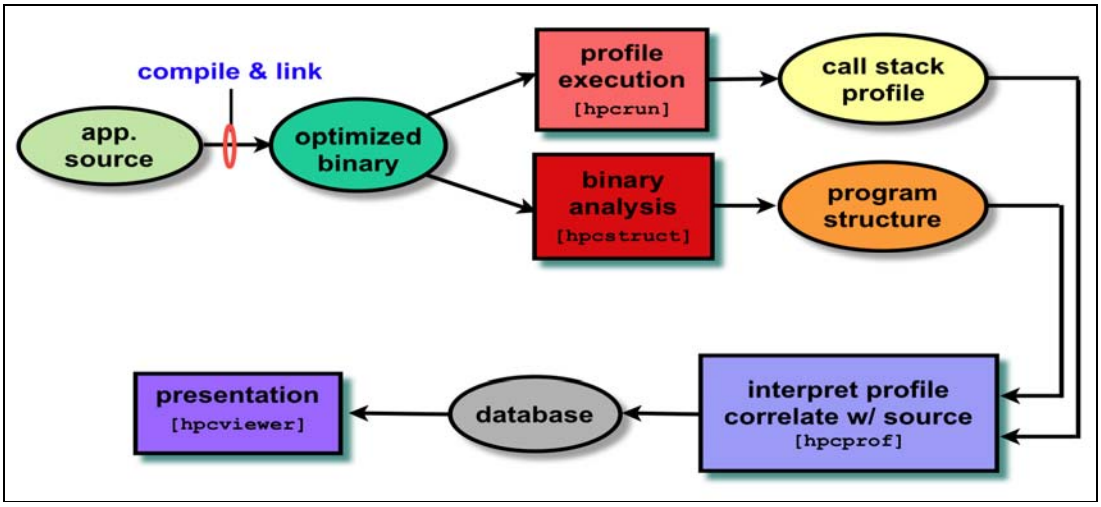


Figure 9-1. HPC Toolkit Workflow

As shown in the workflow diagram above, firstly, one compiles and links the application for a production run, using full optimization. Then, the application is launched with the **hpcrun** measurement tool; this uses statistical sampling to produce a performance profile. Thirdly, **hpcstruct** is invoked, this tool analyzes the application binaries to recover information about files, functions, loops, and inlined code. Fourthly, **hpcprof** is used to combine performance measurements with information about the program structure to produce a performance database. Finally, it is possible to examine the performance database with an interactive viewer, called **hpcviewer**.

9.2.2 HPC Toolkit Tools

The tools included in the **HPC Toolkit** are:

9.2.2.1 hpcrun

hpcrun uses event-based sampling to *measure* program performance. Sample events correspond to periodic interrupts induced by an interval timer, or overflow of hardware performance counters, measuring events such as cycles, instructions executed, cache misses, and memory bus transactions. During an interrupt, **hpcrun** attributes samples to calling contexts to form *call path profiles*. To accurately measure code from 'black box' vendor compilers, **hpcrun** uses on-the-fly binary analysis to enable stack unwinding of fully optimized code *without compiler support*, even code that lacks frame pointers and uses optimizations such as tail calls. **hpcrun** stores sample counts and their associated calling contexts in a *calling context tree* (CCT).

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

9.2.2.2 hpcstruct

hpcstruct *analyzes* the application binary to determine its static program structure. Its goal is to recover information about procedures, loop nests, and inlined code. For each procedure in the binary, **hpcstruct** parses its machine code, identifies branch instructions, builds a control flow graph, and then uses interval analysis to identify loop nests within the control flow. It combines this information with compiler generated line map information in a way that allows **HPC Toolkit** to correlate the samples associated with machine instructions to the program's procedures and loops. This correlation is possible even in the presence of optimizations such as inlining and loop transformations such as fusion, and compiler-generated loops from scalarization of **Fortran 90** array operations or array copies induced by Fortran 90's calling conventions.

9.2.2.3 hpcprof

hpcprof *correlates* the raw profiling measurements from **hpcrun** with the source code abstractions produced by **hpcstruct**. **hpcprof** generates high level metrics in the form of a performance database called the **Experiment** database, which uses the Experiment XML format for use with **hpcviewer**.

hpcprof-flat is the flat-view version of hpcprof and correlates measurements from **hpcrun-flat** with the program structure produced by **hpcstruct**.

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

9.2.2.4 hpcviewer

hpcviewer *presents* the Experiment database produced by **hpcprof** or **hpcprof-flat** so that the user can quickly and easily view the performance databases generated.

9.2.2.5 Display Counters

The **hpcrun** tool uses the hardware counters as parameters. To know which counters are available for your configuration, use the **papi_avail** command. The **hpcrun** and **hpcrun-flat** tools will also give this information.

```
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).

See
- For more information on the display counters, use the **papi_avail -d** command.
- The following chapters for more information on using **HPC Toolkit**.

9.3 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.

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

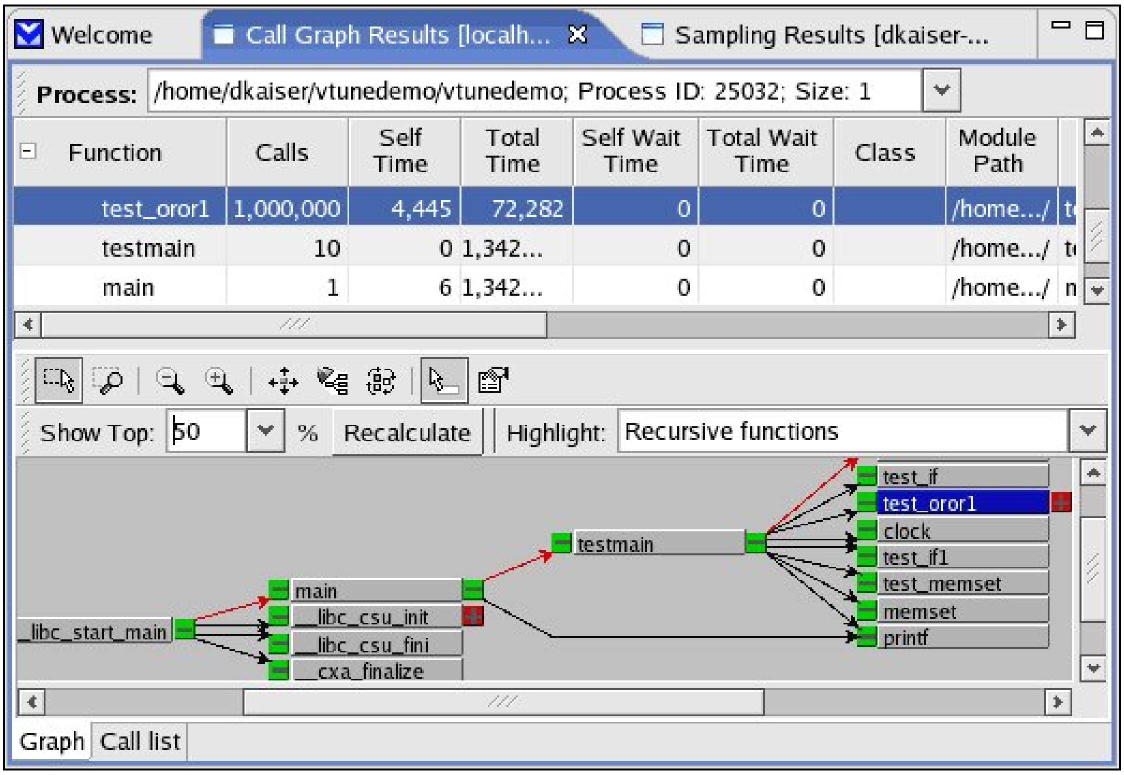
See http://www.intel.com/ for more details.

9.3.1 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.

9.3.2 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.



Process: /home/dkaiser/vtunedemo/vtunedemo; Process ID: 25032; Size: 1

Function	Calls	Self Time	Total Time	Self Wait Time	Total Wait Time	Class	Module Path	
test_oror1	1,000,000	4,445	72,282	0	0		/home.../	t
testmain	10	0	1,342...	0	0		/home.../	t
main	1	6	1,342...	0	0		/home.../	n

Show Top: 50 % Recalculate Highlight: Recursive functions

Figure 9-2. A Call Graph showing the critical path in red

Figure 9-2 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.

9.3.3 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.

9.3.4 Adapted to extreme computing clusters

Intel® VTune™ Performance Analyzer is adapted for extreme computing 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.

Chapter 10. Using HPC Toolkit

Prerequisites

- The executable must contain debugging information (if not, there will be no correspondence between the counters and code at source line level).
- The executable should be dynamically linked because **HPC Toolkit** overloads the default initialization functions to call **PAPI**. (If the executable is statically linked, a special linking script called **hpclink** must be run to link with the **hpcrun** components.)
- The executable must not use **ANSI libstdc++**. (If there is a static constructor in the **libstdc++**, the use of **HPC Toolkit** will produce a **SIGSEGV**).

Note HPC Toolkit provides the most complete performance information when working with fully optimized executables that include line map information within the object code. Most compilers provide this, which means that a special build process is not required.

Important

In order to produce complete results that allow you to view metrics and analyze performance, it is mandarory to run HPC Toolkit in one of the sequences below:

- hpcstruct, hpcrun, hpcprof, hpcviewer
- hpcstruct, hpcrun-flat, hpcprof-flat, hpcviewer
- hpcstruct, hpcrun-flat, hpcproftt

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

10.1 Step 1: Recovering the Program Structure with hpcstruct

hpcstruct analyzes an application binary or DSO <binary> and recovers the static program structure from the object code. **hpcstruct** writes a **XML** file (type=HPC ToolkitStructure) that maps the program's static source-level structure to its object code. By default, **hpcstruct** writes its results to the **basename(<binary>).hpcstruct** file. Normally, this file is then passed to HPC Toolkit's correlation tool hpcprof. It can also be used by the **hpcprof-flat** or **hpcproftt** tools.

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

Syntax

```
hpcstruct [options] <binary>
```

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 structure recovery for procedures matching the procedure glob <glob>

Structure Recovery Options

-I <path>, –include <path>

Use <path> when resolving source file names. For a recursive search, append a '*' after the last slash, e.g., **/mypath/*** (quote or escape to protect from the shell.) May pass multiple times.

–loop-intvl <yes|no>

Loop recovery heuristics assume an irreducible interval is a loop. {yes}

–loop-fwd-subst <yes|no>

Loop recovery heuristics assume forward substitution may occur. {yes}

-N <all|safe|none>, –normalize <all|safe|none>

Specify normalizations to apply to structure. {all}

all : apply all normalizations

safe : apply only safe normalizations

none : apply no normalizations

Example

```
hpcstruct LoopTest.exe
```

hpcstruct writes the structure tree for the **LoopTest.exe** program to the file *LoopTest.exe.hpcstruct.*

10.2 Step 2: Measuring Program Execution with hpcrun

hpcrun profiles the execution of an arbitrary command <command> using statistical sampling rather than instrumentation. It collects per-thread-call path profiles that represent the full calling context of sample points. Sample points may be generated from multiple simultaneous sampling sources. **hpcrun** profiles complex applications that use forks, execs, threads, and dynamic linking/unlinking. It may be used in conjunction with parallel process launchers such as SLURM's **srun**.

Example

A **< command>** executes and is monitored by **hpcrun**. After each instance of event **e** during period **p** a sample, containing information about the functioning of the command, is generated which is recorded by **hpcrun**.

When **<command>** terminates, **hpcrun** writes the profile measurement database to the *HPC Toolkit-<command>-measurements* directory:

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

Note Dynamically linked libraries can be run with the **hpcrun** command directly. However, for statically linked programs the **hpcrun** code must be linked to the application at build time. The **hpclink** tool performs this service by statically linking an application with the **hpcrun** profiling code. See *Section 10.7* for more information about **hpclink**.

Syntax

```
hpcrun [profiling-options] [--] <command> [command-arguments]
hpcrun [info-options]
```

Information Options

-l, -L –list-events	List events that are available; some may not be profilable.
-V, –version	Print version information.
-h, –help	Print help.

Profiling Options

-e <event>[@<period>], –event <event>[@<period>]

An event to profile and its corresponding sample period. <event> may be either a PAPI, native processor event or WALLCLOCK (microseconds). May run multiple times as implementations permit.

{WALLCLOCK@5000}.

Note WALLCLOCK and hardware events cannot be mixed.

-o <outpath>, –output <outpath>

Specifies a directory for the output data. {HPC Toolkit-<command>-measurements }

Notes

- Without an output option, multiple profile databases of the same <command> will be placed in the same directory.
- **hpcrun** uses preloaded shared libraries to initiate profiling. For this reason, **hpcrun** cannot be used to profile **setuid** programs. **hpcrun** may not be able to profile programs that themselves use preloading.

Examples

```
hpcrun -e PAPI_TOT_INS -e PAPI_TOT_CYC LoopTest.exe
```

The profiling database for the above command is written to the *HPC Toolkit-LoopTest.exe-measurements* ***directory***.

To retrieve the counters for 3000 events, enter:

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

10.2.1 Alternative Step 2: Measuring the Execution with Flat Sampling using hpcrun-flat

hpcrun-flat profiles the execution of an arbitrary command **<command>** using statistical sampling rather than instrumentation. It collects per-thread flat profiles, or IP (instruction pointer) histograms. Sample points may be generated from multiple simultaneous sampling sources. **hpcrun-flat** profiles complex applications that use forks, execs, and threads but not dynamic linking/unlinking. It may be used in conjunction with parallel process launchers, such as SLURM's **srun**.

A **< command>** executes and is monitored by **hpcrun-flat**. After each instance of event **e** during period **p** a sample, a counter associated with the current IP is incremented.

When **<command>** terminates, **hpcrun-flat** writes the per-thread profile into a file with the name *<command>*.hpcrun-flat.*<hostname>*.*<pid>*.*<tid>*. This file is known as a profile file and contains a histogram of counts for each module loaded.

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 for a long time or do not function correctly.

Syntax

```
hpcrun-flat [profiling-options] -- <command> [command-arguments]
hpcrun-flat [info-options]
```

Information Options

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

–L, –list-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

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}

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}

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.

Example

```
hpcrun-flat -e PAPI_TOT_INS -e PAPI_TOT_CYC -o flat.data
./LoopTest.exe
```

The *LoopTest.exe.hpcrun-flat.systemj.16701.0x0* profile file is written to the current directory.

10.3 Step 3: Correlating Call Path Profiling Metrics with hpcprof

hpcprof correlates the call path profiling metrics produced by **hpcrun** with the source code structure created by **hpcstruct**. It produces an **Experiment** database for use with the **hpcviewer** tool.

hpcprof produces the best results when the **-I** and **-S** options are used. The **-I** option provides paths for source code directories, and the **-S** option provides the source code structure from **hpcstruct.**

Syntax 1

```
hpcprof [options] <profile-dir-or-file> ...
```

By default, **hpcprof** 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** 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

--name <name>, –title <name>

Set the database's name (title) to <name>.

-I <path>, --include <path>

Use <path> when searching for source files. Use a * after the last slash indicates recursion, e.g. **/mypath/***, with a quote or escape to protect it from the shell. This option may be used multiple times. Source code files are copied into the Experiment database.

-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).

Special Options

--force   **hpcprof** currently allows a maximum of 32 profile files to prevent unmanageably large Experiment databases. The –force option removes that limit.

Output Options

-o <db-path>, --db <db-path>, –output <db-path>

Specify experiment database name <db-path> {./experiment-db}

Example

```
hpcprof -I . -S LoopTest.exe.hpcstruct HPC Toolkit-LoopTest.exe-
measurements
```

```
msg: STRUCTURE: /usr/hpc/looptests/LoopTest.exe
msg: Line map : /usr/lib/HPC Toolkit/ext-libs/libmonitor.so.0.0.0
msg: Copying source files reached by PATH option to
   /usr/hpc//looptests/HPC Toolkit-LoopTest.exe-database
```

The experiment.xml Experiment database file and the source files are written to the *HPC Toolkit-LoopTest.exe-database* directory.

10.3.1 Step 3 Alternative A: Correlating Flat Metrics with Program Structure using hpcprof-flat

hpcprof-flat correlates flat profiling metrics with a static source code structure and, by default, generates an Experiment database for use with **hpcviewer**. **hpcprof-flat** is invoked in one of two ways. Firstly, correlation options are specified on the command line along with a list of flat profile files. Secondly, these options along with derived metrics are specified in the <**config-file**> configuration file. The first method is generally sufficient because derived metrics can be computed in **hpcviewer**. However, to facilitate batch processing for the second method, when run with the first method, a sample configuration file (**config.xml**) is generated within the **Experiment** database.

Syntax 1

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

The inputs to the usage of **hpcprof-flat** are (1) the source structure file created by the **hpcstruct** tool and (2) the profile files created by the **hpcrun-flat** tool. If the structure file is not provided, **hpcprof-flat** will default to a 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 the second form 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

–**name <name>**, –**title <name>**

Set the database name (title) to <name).

-I **<path>**, - -**include <path>**

Use <path> when searching for source files. Use a * after the last slash indicates recursion, e.g. **/mypath/***, with a quote or escape to protect it from the shell. 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> {./HPC Toolkit database}

–src [yes|no], --source[yes|no]

Indicates if source code files should be copied into the **Experiment** database. {yes}. By default, **hpcprof-flat** copies source files with performance metrics, resulting in a self-contained dataset that does not rely on an external source code repository. If copying is suppressed, the database is no longer self-contained. Note that only those source files reachable by PATH/REPLACE statements are copied.

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}. When **--csv** is specified, the **--src** option is set to **no**.

Syntax 1 Examples

```
hpcprof-flat -I . -S LoopTest.exe.hpcstruct flat.data/*
```

```
msg: Copying source files reached by PATH/REPLACE options to HPC
Toolkit-database
msg: Writing final scope tree (in XML) to experiment.xml
msg: Writing configuration file to HPC Toolkit-database/config.xml
```

```
ls -l HPC Toolkit-database
```

```
total 16
-rw-r--r-- 1 hpctk users  452 2009-11-10 20:33 config.xml
-rw-r--r-- 1 hpctk users 7296 2009-11-10 20:33 experiment.xml
drwxr-xr-x 3 hpctk users 4096 2009-11-10 20:33 src
```

Syntax 2

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

The correlation options are contained in the configuration file and cannot be specified on the command line for **Syntax 1** above. **<config-file>** is a configuration file generated by previous **hpcprof-flat** activity, and may be edited by the user. The configuration file syntax is briefly described in the **hpcviewer** section below.

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 LoopTest.exe.hpcstruct --config HPC Toolkit-
database/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.

10.3.2 Step 3 Alternative B: Correlating Flat Metrics with Program Structure using hpcproftt

hpcproftt provides an alternative to **hpcprof-flat** and **hpcviewer**. **hpcproftt** correlates profile metrics with either the *source code structure* (the first and default mode) or *object code* (second mode) and generates text output for a terminal. In both modes, **hpcproftt** uses a list of flat profile files as input.

hpcproftt also supports a third mode, in which it generates textual dumps of profile files. In this mode, the profile list may contain either flat or call path profile files.

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 the metrics specified by the **--metric** option. All metrics are normalized to use the **events** unit instead of **samples**, because this enables meaningful comparisons and derived metrics. Because percentages facilitate rapid comprehension, compared to values, all raw metrics are displayed as percentages. Likewise, derived metrics default to percentages whenever possible.

hpcproftt then correlates the metrics to the program structure based on the **hpcstruct** program structure file specified by the **--structure** option. If a 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 metric summary compares a source structure element, such as a procedure, with all other elements of that type throughout the program. The desired elements are chosen by switches specified with the **--source** option. Structure elements include Program, Load Module, File, Procedure, Loop, and Statement. Note that loops are included only when the **--structure** option is used. For example, the procedure summary shows exclusive metric values for each procedure in the program. Structure elements are pruned if all corresponding metrics are zero. Summaries are rank-ordered by the first metric.

Optionally, **hpcproftt** will annotates source files with Statement (line) level metrics. It can only annotate those files found by combing debug information using the **--include** search paths.

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]

--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
sgm	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 ***

--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 to logical OR additional globs.

-M <metric>, –metric <metric>

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

sum	Show also **Mean**, **CoefVar**, **Min**, **Max**, **Sum**
sum-only	Show only **Mean**, **CoefVar**, **Min**, **Max**, **Sum**

-I <path>, --include <path>

Use **<path>** when searching for source files. Use a **`*`** after the last slash indicates recursion, e.g. **`/mypath/*`**. Use quote marks or escape to protect it from the shell. 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 flat.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):
----------------------------------------------------------------------
36001    12030
3.60e+10 1.20e+10
======================================================================
Load module summary:
----------------------------------------------------------------------
100.00%  100.00%  /opt/hpctk/csg/looptests/LoopTest.exe
======================================================================
File summary:
----------------------------------------------------------------------
 100.00%  100.00%  [LoopTest.exe]DoLoops.cpp
======================================================================
Procedure summary:
----------------------------------------------------------------------
 100.00%  100.00%  [LoopTest.exe]<DoLoops.cpp>DoLoops(int, int)
======================================================================
Loop summary (dependent on structure information):
----------------------------------------------------------------------

======================================================================
Statement summary:
----------------------------------------------------------------------
  99.96%   99.98%  [LoopTest.exe]<DoLoops.cpp>21
   0.03% 8.3e-03%  [LoopTest.exe]<DoLoops.cpp>18
   0.01% 8.3e-03%  [LoopTest.exe]<DoLoops.cpp>17
```

Syntax 2

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

In object mode, **hpcproftt** performs fine-grained correlation and generates annotated object code. Unlike source structure correlation mode, true summaries are not computed; instead **hpcproftt** generates annotated object code, i.e. procedures and instruction. Moreover, only raw metrics corresponding to native events in one profile file may be correlated to the object code affected by these metrics. Accordingly, hpcproftt creates raw metrics for each native event in one profile file. Metrics use the **samples** unit instead of **events** and default to percentages, alternatively absolute values can be displayed. Procedures are pruned from the output if no associated metric totals apply to the threshold.

Notes

- On ISAs with variable-sized instructions, histogram buckets of 4 byes in size may contain information for more than one instruction. In this case, multiple instructions will report counts for the same bucket.

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

Object Correlation Options

--object[=s]

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

s: intermingle source line info with object code

--objannot Annotate object procedures with unmangled names that match glob <glob>. Protect glob characters from the shell with single quotes or backslash. May pass multiple times to logical OR additional globs.

--obj-values Show raw metrics as values instead of percentages

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

Example of Object Code Correlation

```
hpcproftt --object=s flat.data/LoopTest.exe.hpcrun-
flat.chekov.9140.0x0
```

```
======================================================================
flat.data/LoopTest.exe.hpcrun-flat.systemj.9140.0x0
======================================================================
======================================================================
Load module: /lib64/ld-2.10.1.so
----------------------------------------------------------------------
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):
         0        0
...
======================================================================
Load module: LoopTest.exe
----------------------------------------------------------------------
Metric summary for load module (totals):
     36001    12030

Procedure: _Z7DoLoopsii (DoLoops(int, int))
------------------------------------------------------------

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):
     36001    12030
   100.00%  100.00%

Metric details for procedure (percents relative to procedure):
DoLoops.cpp:6
```

```
0x4006d0:                   push   %r15
0x4006d2:                   push   %r14
0x4006d4:                   mov    %edi,%r14d
0x4006d7:                   push   %r13
0x4006d9:                   mov    %esi,%r13d
0x4006dc:                   push   %r12
DoLoops.cpp:17
0x4006de:                   xor    %r12d,%r12d
DoLoops.cpp:6
0x4006e1:                   push   %rbp
DoLoops.cpp:17
0x4006e2:                   xor    %ebp,%ebp
DoLoops.cpp:6
0x4006e4:                   push   %rbx
0x4006e5:                   sub    $0x28,%rsp
```

...

Syntax 3

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

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

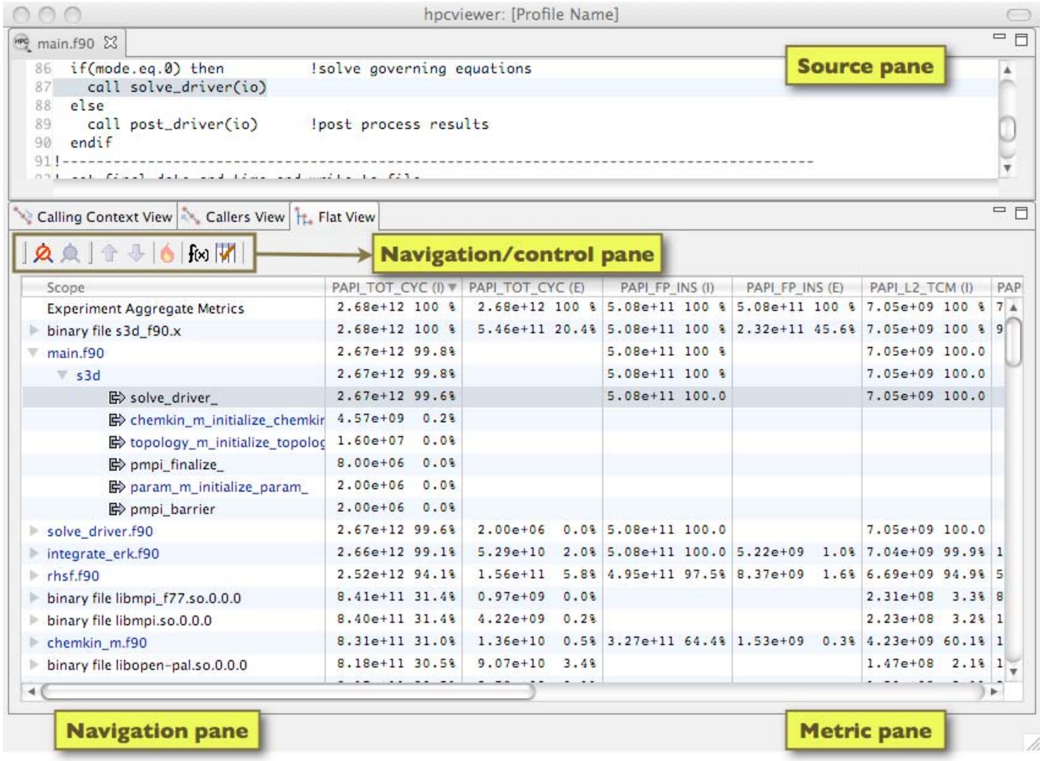
Example

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

```
======================================================================
flat.data/LoopTest.exe.hpcrun-flat.systemj.9140.0x0
======================================================================
--- ProfileData Dump ---
{ ProfileData: flat.data/LoopTest.exe.hpcrun-flat.systemj.9140.0x0 }
  { LM: /lib64/ld-2.10.1.so, loadAddr: 0x3971200000 }
    { EventData: PAPI_TOT_INS, period: 999999, outofrange: 0,
overflow: 0 }
    { EventData: PAPI_TOT_CYC, period: 999999, outofrange: 0,
overflow: 0 }
...
  { LM: LoopTest.exe, loadAddr: 0x400000 }
    { EventData: PAPI_TOT_INS, period: 999999, outofrange: 0,
overflow: 0 }
      { 0x400730: 1 }
      { 0x400734: 5 }
      { 0x400744: 5 }
      { 0x400750: 2 }
      { 0x400760: 18199 }
      { 0x400768: 17784 }
      { 0x40079c: 5 }
    { EventData: PAPI_TOT_CYC, period: 999999, outofrange: 0,
overflow: 0 }
      { 0x400734: 1 }
      { 0x400760: 5597 }
      { 0x400768: 6431 }
      { 0x40079c: 1 }
...
--- End ProfileData Dump ---...
```

10.4 Step 4: Checking the Results with hpcviewer

The **hpcviewer** tool allows the performance databases, produced by the previous steps, to be examined interactively. **hpcviewer** uses the Experiment database, specifically the XML file generated by **hpcprof** or **hpcprof-flat**.



```
  if(mode.eq.0) then           !solve governing equations
    call solve_driver(io)
  else
    call post_driver(io)       !post process results
  endif
!-----------------------------------------------------------------------------
```

Scope	PAPI_TOT_CYC (I) ▼	PAPI_TOT_CYC (E)	PAPI_FP_INS (I)	PAPI_FP_INS (E)	PAPI_L2_TCM (I)	PAP
Experiment Aggregate Metrics	2.68e+12 100 %	2.68e+12 100 %	5.08e+11 100 %	5.08e+11 100 %	7.05e+09 100 %	7
▶ binary file s3d_f90.x	2.68e+12 100 %	5.46e+11 20.4%	5.08e+11 100 %	2.32e+11 45.6%	7.05e+09 100 %	9
▼ main.f90	2.67e+12 99.8%		5.08e+11 100 %		7.05e+09 100.0	
▼ s3d	2.67e+12 99.8%		5.08e+11 100 %		7.05e+09 100.0	
solve_driver_	2.67e+12 99.6%		5.08e+11 100.0		7.05e+09 100.0	
chemkin_m_initialize_chemkir	4.57e+09 0.2%					
topology_m_initialize_topolog	1.60e+07 0.0%					
pmpi_finalize_	8.00e+06 0.0%					
param_m_initialize_param_	2.00e+06 0.0%					
pmpi_barrier	2.00e+06 0.0%					
▶ solve_driver.f90	2.67e+12 99.6%	2.00e+06 0.0%	5.08e+11 100.0		7.05e+09 100.0	
▶ integrate_erk.f90	2.66e+12 99.1%	5.29e+10 2.0%	5.08e+11 100.0	5.22e+09 1.0%	7.04e+09 99.9%	1
▶ rhsf.f90	2.52e+12 94.1%	1.56e+11 5.8%	4.95e+11 97.5%	8.37e+09 1.6%	6.69e+09 94.9%	5
▶ binary file libmpi_f77.so.0.0.0	8.41e+11 31.4%	0.97e+09 0.0%			2.31e+08 3.3%	8
▶ binary file libmpi.so.0.0.0	8.40e+11 31.4%	4.22e+09 0.2%			2.23e+08 3.2%	1
▶ chemkin_m.f90	8.31e+11 31.0%	1.36e+10 0.5%	3.27e+11 64.4%	1.53e+09 0.3%	4.23e+09 60.1%	1
▶ binary file libopen-pal.so.0.0.0	8.18e+11 30.5%	9.07e+10 3.4%			1.47e+08 2.1%	1

Figure 10-1. hpcviewer screen

Syntax

```
hpcviewer [database-directory]
```

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

Menus

hpcviewer provides three main menus.

File Menu

This menu includes several menu items for checking basic viewer operations:

New window	Open a new **hpcviewer** window that is independent of the existing one.
Open database	Load a performance database into the current **hpcviewer** window.
Preferences	Display the settings dialog box.
Exit	Quit the **hpcviewer** application.

Help Menu

This menu displays information about **hpcviewer**:

About	Displays basic information about the **hpcviewer**, including plug-ins used and error log.
hpcviewer help	Displays online help for using **hpcviewer**.

Debug Menu

This menu is used to display HPC Toolkit's raw XML representation of the performance data. This menu is intended for the use of tool developers

10.4.1 hpcviewer views

hpcviewer displays performance data for an application in three ways:

- Top-down call path (Calling context) view
- Bottom-up call path (Caller's) view
- Flat view

Select the view by clicking the corresponding view control tab.

Calling context view

This top-down view represents dynamic calling contexts (call paths) in which costs were incurred. Using this view, one can explore the performance measurements for an application in a top-down fashion and analyze the costs incurred by procedure calls in a particular calling context. The term "cost" is used instead of "time", because **hpcviewer** can present a multiplicity of measurements, such as cycles, or cache misses, or derived metrics, such as cache miss rates or bandwidth consumed. These items are indicators of the true execution cost. A calling context for a procedure **f** consists of the stack of procedure frames active when the call was made to **f**. Using this view, one can readily see how much of the application's cost was incurred by f, when called from a particular calling context. If finer detail is of interest, one can explore how the costs incurred by a call to **f** in a particular context are divided between **f** itself and the procedures it calls. HPC Toolkit's call path profiler **hpcprof** and the **hpcviewer** user interface distinguish the calling context by the individual call sites; this means that if a procedure **g** contains calls to procedure **f** in different places, these represent separate calling contexts.

Callers view

This bottom-up view enables one to look upward along the call paths. This view is particularly useful for understanding the performance of software components, or procedures, used in more than one context. For instance, a message-passing program may call **MPI_Wait** in many different calling contexts. The cost of any particular call will depend upon the structure of the parallelization in which the call is made. Serialization or load imbalance may cause long waits in some calling contexts while other parts of the program may have short waits because computation is balanced and communication is overlapped with computation.

Flat view

This view organizes performance measurement data according to the static structure of an application. All costs incurred in any calling context by a procedure are aggregated together in the flat view. This complements the calling context view, in which the costs incurred by a particular procedure are represented separately for each call to the procedure from a different calling context.

10.4.2 hpcviewer browser window

The **hpcviewer** browser window is divided into three panes: the **source pane**, the **navigation pane**, and the **metrics pane**.

Source pane

This displays the source code associated with the current entity selected in the navigation pane. When **hpcviewer** opens a performance database, the source pane is initially blank because no entity has been selected in the navigation pane. Selecting any entity in the navigation pane will cause the source pane to first load the corresponding file, and then scroll to and highlight the line corresponding to the selection. Selecting another entity in the navigation pane causes the source pane to display that entity's associated source file.

Navigation Window

This presents a hierarchical tree-based structure that is used to organize the presentation of an application's performance data. This tree can include load modules, source files, procedures, procedure activations, inlined code, loops, and source lines. Selecting any of these entities in the navigation pane causes its associated source code, if any, to be displayed in the source pane. Children in this hierarchy can be revealed or concealed by opening or closing any non-leaf entry in the view.

The type of entities in the navigation pane's tree structure depends on the view being displayed:

Calling context view
Entities in the navigation tree represent procedure activations, inlined code, loops, and source lines. Most entities link to a single location in the source code; procedure activations, however, link to two: the call site from which the procedure was called and the procedure itself.

Callers view
Entities in the navigation tree are procedure activations. Whereas procedure activations in the **calling context view** are paired with the called procedure, call sites in this view are paired with the calling procedure to facilitate the attribution of costs for a procedure that is called to different call sites and callers.

Flat view
Entities in the navigation pane correspond to source files, loops, source lines, and procedure call sites, which are rendered in the same way as procedure activations.

Navigation bar buttons

flatten unflatten

Flatten and unflatten the navigation hierarchy (flat view only). Clicking on the flatten button will replace each top-level scope with its children. If a scope has no children, i.e., it is a leaf, the node will remain in the view. This flattening operation is useful for changing the strict hierarchical view so that peers at the same level in the tree can be viewed and ranked together. For instance, this can be used to hide procedures in the flat view so that outer loops can be ranked and compared to one another. The inverse of the flatten operation is the unflatten operation, which makes an elided node in the tree visible again.

⇧ zoom-in ⇩ zoom-out

The up arrow zooms in to show the information for the selected line and its descendants only. The down arrow zooms out and reverses a zoom-in operation that has been carried out.

🔥 hot call path

This presents the cost of performance hot spots indicating the chain of responsibility for these costs. The costs are calculated by comparing parent and child values and showing the chain where the difference is greater than a threshold (default is 50%). The threshold value can be changed by using the **File** -> **Preferences** menu.

f(x) Derived metric

Creates a new metric from the mathematical formulas that are available. See the sections that follow for more information on derived metrics.

☑ Hide / Show metrics

Hides or shows metric columns. A dialog box appears, and the user selects the columns to be displayed or to hidden. See the sections that follow for more information.

Metric pane

This displays one or more performance metrics associated with the entities in the navigation pane. Entities in the tree view of the navigation pane are sorted at each level by the metric in the selected column. When **hpcviewer** is launched, the leftmost metric column is the default selection, and the navigation pane is sorted according to the values of that metric in descending order. Clicking on a column header changes the selected metric. The sort order can be reversed by clicking on the arrow at the head of the selected column.

Determining the relationship between the two metrics is easier when the metrics of interest are in adjacent columns of the **metric pane**. The order of columns can be changed by selecting the column header for a metric and then dragging it left or right to its desired position. The **metric pane** also includes scroll bars for horizontal scrolling (other metrics) and vertical scrolling (other scopes). Vertical scrolling of the **metric pane** and the **navigation pane** is synchronized.

10.5 Improving the Performance of hpcviewer

10.5.1 Source Pane

The source pane is used to display a copy of the program's source code or HPC Toolkit's performance data in XML format; it does not support editing of the pane's contents. Use a dedicated editor (not the one stored in HPC Toolkit's performance database) to edit your original copy of the source.

hpcviewer supports the following useful shortcuts and customizations.

Go to line: Scrolls the current source pane to a specific line number. **`<ctrl>-l`** opens a dialog box for the target line number to be entered.

Find: Searches for a string in the current source pane. **`<ctrl>-f`** opens dialog box for the target string to be entered.

Font: Change the font used for the metric table by using the *Preferences* dialog from the *File* menu. After opening the *Preferences* dialog, select *hpcviewer* preferences (the item at the bottom of the list in the column on the left side of the pane). The selected font will be used the next time **hpcviewer** is launched.

Minimize/Maximize window: Icons in the upper right corner of the window. Minimize or maximize the **hpcviewer** window.

10.5.2 Metric Pane

hpcviewer supports the following features:

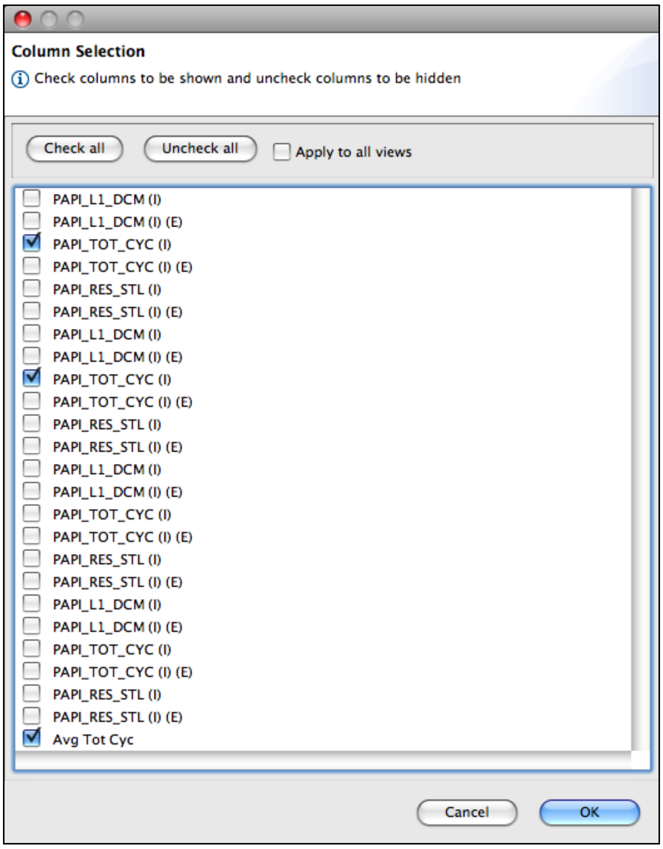
Maximizing a view. Expands the Source or Metric pane to fill the window. Double-click on the view tab to expand or to restore the view.

Sorting the metric pane contents by a column's values. Select the column to be sorted. If no triangle appears next to the metric, click again. A downward pointing triangle means that the rows in the metric pane are sorted in descending order according to the values in the column. Additional clicks on the header of the selected column will toggle back and forth between ascending and descending.

Changing column width. Place the cursor over the border (right or left) of the column's header field. The cursor changes into a vertical bar between a left and right arrow. Hold down the mouse button and drag the column border to the width desired.

Changing column order. Columns can be permuted as desired. Hold down the mouse button over the header of the column to be moved, and drag the column to its new position.

Hiding or showing metric columns. Use the ✓ button in the Navigation bar to bring up the column selection window, as shown below:



Column Selection

ⓘ Check columns to be shown and uncheck columns to be hidden

☐ Apply to all views

☐ PAPI_L1_DCM (I)
☐ PAPI_L1_DCM (I) (E)
☑ PAPI_TOT_CYC (I)
☐ PAPI_TOT_CYC (I) (E)
☐ PAPI_RES_STL (I)
☐ PAPI_RES_STL (I) (E)
☐ PAPI_L1_DCM (I)
☐ PAPI_L1_DCM (I) (E)
☑ PAPI_TOT_CYC (I)
☐ PAPI_TOT_CYC (I) (E)
☐ PAPI_RES_STL (I)
☐ PAPI_RES_STL (I) (E)
☐ PAPI_L1_DCM (I)
☐ PAPI_L1_DCM (I) (E)
☐ PAPI_TOT_CYC (I)
☐ PAPI_TOT_CYC (I) (E)
☐ PAPI_RES_STL (I)
☐ PAPI_RES_STL (I) (E)
☐ PAPI_L1_DCM (I)
☐ PAPI_L1_DCM (I) (E)
☐ PAPI_TOT_CYC (I)
☐ PAPI_TOT_CYC (I) (E)
☐ PAPI_RES_STL (I)
☐ PAPI_RES_STL (I) (E)
☑ Avg Tot Cyc

Figure 10-2. Hide\Show Columns Window

The **Column Selection** box contains the list of the metric columns, sorted according to their order in HPC Toolkit's performance database for the application. Each metric column is prefixed by a check box that indicates if the metric should be displayed (checked) or hidden (unchecked). Select the *Check all* button to display all the metric columns. A click on *Uncheck all* hides all the metric columns. If *Apply to all views* is checked the configuration will apply to all views; otherwise, the configuration applies to the current view only.

10.5.3 hpcviewer Limitations

Limited number of metrics. Although most **HPC Toolkit** components such as **hpcrun** and **hpcstruct** support a large number of metrics, it is not recommended to work with more than 100 metrics, including both exclusive and inclusive variants, with **hpcviewer**. Having a high number of metrics require a large amount of memory, and makes the viewer interface sluggish. If the application profiled has more than 50 processes and each process has its own metrics, then it is recommended to analyze a few representative processes only. To pinpoint scalability bottlenecks, we recommend an approach based on the differential analysis of a representative process using two executions at different scales.

See *Coarfa, C., Mellor-Crummey, J., Froyd, N., and Dotsenko, Y. 2007. Scalability analysis of SPMD codes using expectations* in the **Proceedings of the 21st Annual International Conference on Supercomputing** (Seattle, Washington, June 17 - 21, 2007) for more information.

Copying or printing metrics. **hpcviewer** does not currently support the copying or printing of metric values.

10.6 HPC Toolkit Metrics

hpcviewer allows the interactive examination of the performance databases that have been generated. **hpcviewer** uses the Experiment database, specifically the XML file generated by **hpcprof** or **hpcprof-flat**.

Exclusive costs are those incurred by scope itself; *inclusive* costs include costs incurred by any calls it makes. The data gathered by the profiler attributes the cost for each scope (a file, procedure, loop, or inlined function) exclusively. **hpcviewer** presents inclusive values for each cost metric associated with a program scope as well.

How are metrics computed?
Call path profile measurements collected by **hpcrun** correspond directly to the **Calling context view**. **hpcviewer** derives all other views from exclusive metric costs in the Calling context view. For the **Caller view**, **hpcviewer** collects the cost of all samples in each function and attributes that to a top-level entry in the caller view. Under each top-level function, **hpcviewer** can look up the call chain to identify all the contexts in which the function is called. For each function, **hpcviewer** apportions its cost according to the calling contexts that incur costs. **hpcviewer** computes the flat view by traversing the calling context tree and attributing the scope costs to the various parts of its static source code structure. The flat view presents a hierarchy of nested scopes from the load module, file, routine, loops, inlined code and statements.

Examples

file1.c

```
f () {
    g ();
}

// m is the main routine
m () {
    f ();
    g ();
}
```

file2.c

```
// g can be a recursive function
g () {
    if ( . . ) g ();
    if ( . . ) h ();
}

h () {
}
```

Figure 10-3. Source files

Figure 9-2 shows an example of a recursive program separated into two files: **file1.c** and **file2.c**. In this figure, numerical subscripts distinguish between different instances of the same procedure. In the other parts of this figure, alphabetic subscripts are used; there is no natural one-to-one correspondence between the instances in the different views. Routine **g** in *Figure 9-2* can behave as a recursive function depending on the value of the condition branch (lines 3-4).

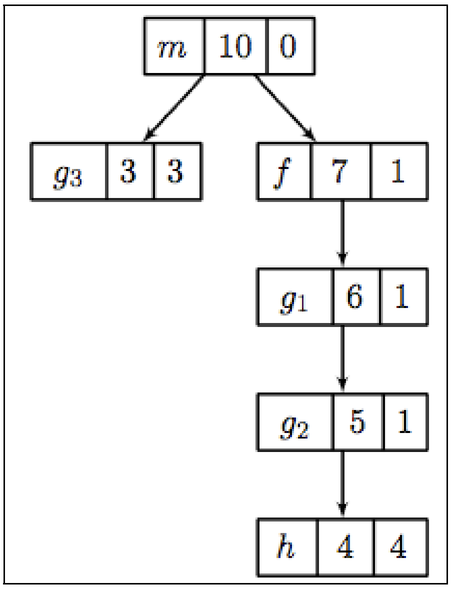


Figure 10-4. Calling Context view

Figure 9-3 shows an example of the call chain execution of the program, annotated with both inclusive and exclusive costs. The computation of the inclusive costs, from the exclusive costs in the **Calling Context view**, involves simply adding up all of the costs in the sub-tree below.

Note that on the right path of the routine **m**, routine **g**, instantiated in the diagram as **g1**, performed a recursive call **g2** before calling routine **h**. Although **g1**, **g2** and **g3** are all instances from the same **g** routine, each instance accrues a different cost. This separation of cost can be critical in identifying which instance of **g** has a performance problem.

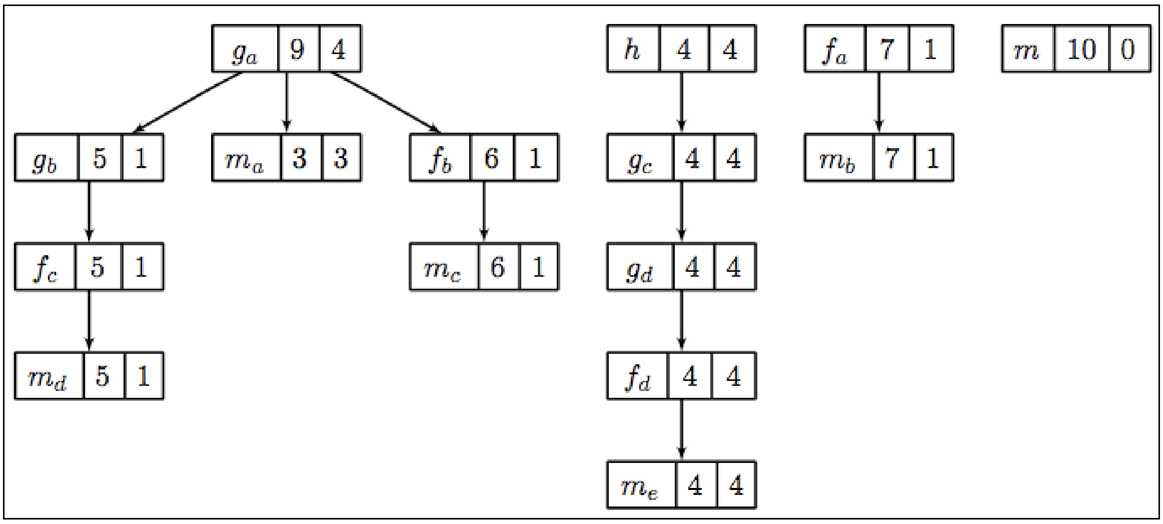


Figure 10-5. Caller view

Figure 9-4 shows the corresponding scope structure for the caller view and its computed costs for this recursive program. The procedure **g** shown as **ga**, a root node in the diagram, has different cost from **g** as a call-site indicated by the **gb**, **gc** and **gd** instances. For example, the inclusive cost of **ga** is 9 in the first tree of this figure, this is the sum of the highest cost for each branch in the Calling Context tree in *Figure 9-3*: the inclusive cost of **g3** (3) and **g1** (6). The cost of **g2** does not accrue here because it is a descendant of **g1**, in other words, the cost of **g2** is included in **g1**.

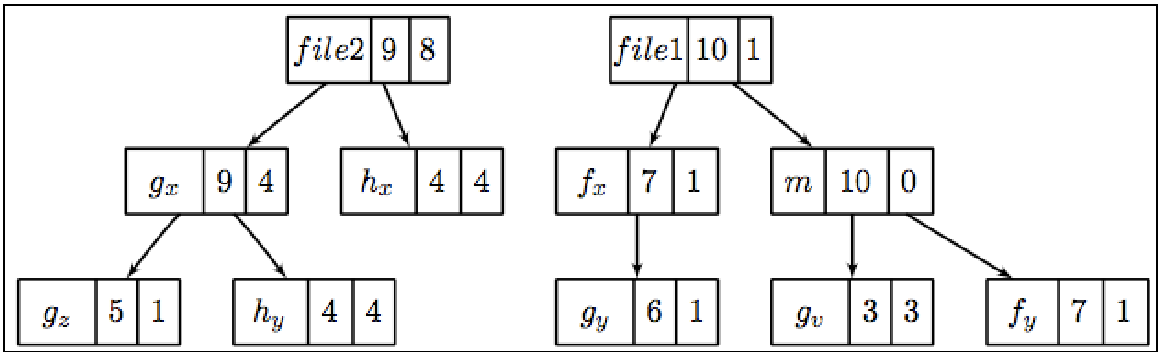


Figure 10-6. Flat view

Inclusive costs are computed similarly in the Flat view. The inclusive cost of a recursive routine is the sum of the highest cost for each branch in the Calling Context tree. For example, in *Figure 9-5* the inclusive cost of **gx**, defined as the total cost of all instances of **g**, is 9; this is consistent with the cost in the caller tree. The advantage of attributing different costs for each instance of **g** is that it enables the identification of the **g** call instances responsible for performance losses.

10.6.1 Derived Metrics

Often, the data only become useful when combined with other information such as the number of instructions executed or the total number of cache accesses. While users do not mind a bit of mental arithmetic and can compare values in different columns to see how they relate for a scope, doing this for many scopes is exhausting. To address this problem, **hpcviewer** provides a mechanism for defining metrics. A user-defined metric is called a **Derived metric**. A derived metric is defined by using a spreadsheet-like mathematical formula that refers to data in other metric table columns by using $n to refer to the value in the nth column.

Formula

The formula supported by **hpcviewer** is spreadsheet-like and can consist of any form of combined complex expressions. The syntax is very simple:

```
<expression> ::= <binary_op> | <function>
<binary_op> ::= <expression> <binary_operand> <expression>
<binary_operand> ::= + | - | * | /
```

Intrinsic Functions

Creating a new intrinsic function requires modifications to the source code. The source code for the **hpcviewer** is available from https://outreach.scidac.gov, or you can contact the HPC Toolkit team at hpc@rice.edu. The HPC Toolkit team can also provide a list of the intrinsic functions that are supported.

Derived Metric Dialog Box

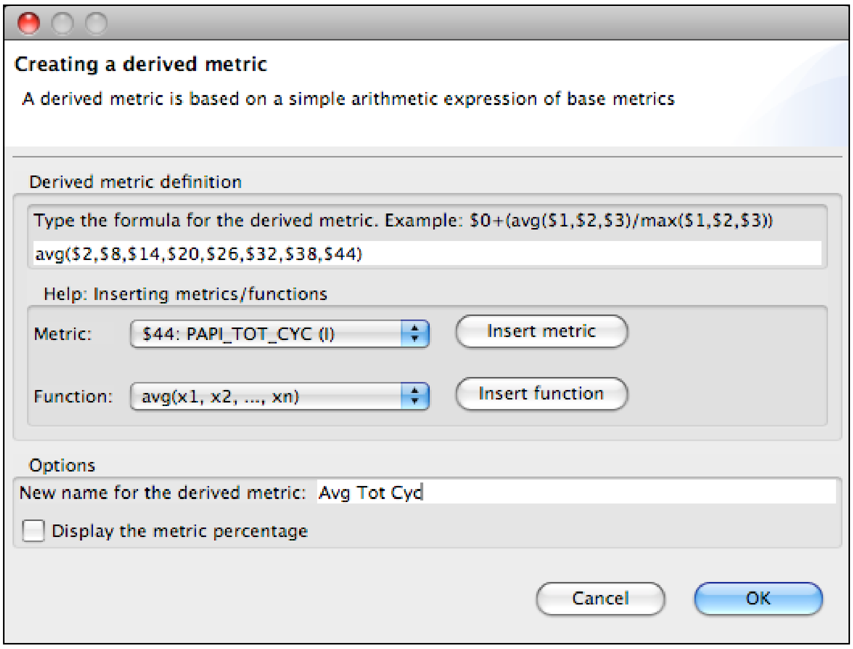
Consider a database containing information about five processes, each with two metrics.

1. Metrics 0, 2, 4, 6, 8 : Total number of cycles
2. Metrics 1, 3, 5, 7, 9 : Total number of floating point operations

To calculate the average number of cycles per floating point operation across all five processes, define a formula as follows:

avg($0, $2, $4, $6, $8) / avg($1, $3, $5, $7, $9)

A derived metric can be created by clicking the **Derived metric** button in the Navigation bar. A **Derived metric** window will appear as shown in *Figure 9-6* below:



Creating a derived metric

A derived metric is based on a simple arithmetic expression of base metrics

Derived metric definition

Type the formula for the derived metric. Example: $0+(avg($1,$2,$3)/max($1,$2,$3))

avg($2,$8,$14,$20,$26,$32,$38,$44)

Help: Inserting metrics/functions

Metric: $44: PAPI_TOT_CYC (I)

Function: avg(x1, x2, ..., xn)

Options

New name for the derived metric: Avg Tot Cyc

□ Display the metric percentage

Figure 10-7. Derived metric dialog box

The window has two main parts:

1. Derived metric definition, consisting of:

   Formula definition field: Field to define spreadsheet-like mathematical formulas.

   Metric: Used to insert metric ID. For example, in *Figure 9-6,* the metric PAPI_TOT_CYC has the ID of 44. Clicking the **Insert metric** button will insert the metric ID into the formula definition field.

   Function: Inserts functions in the **Formula definition field.** Some functions require only one metric as the argument, but others can have two or more arguments. The **avg()** function, which computes the average of some metrics, must have at least two arguments.

2. **Options**, offering two customizations:

   New name for the derived metric. Supply a string to be used as the column header for the derived metric. If one is not supplied, the derived metric will have no name.

   Display the metric percentage. When this option is selected, each scope's derived metric value will be augmented with a percentage value, which for scope **s** is computed as:

   100 * (Derived metric value of **s**) / (Derived metric value computed by applying the metric formula to the aggregate values of the input metrics).

   Such a computation can lead to nonsensical results for some derived metric formulae. For example, if the derived metric is computed as a ratio of two other metrics, comparing the scope's ratio with the ratio for the entire program will not yield a meaningful result. To avoid a confusing metric display, be careful when using this button to display the metric percentage.

10.6.2 Metric Syntax in the Configuration File

A configuration file is an XML document of the **`HPCVIEW`** type, and uses the following top-level elements:

<HPCVIEW>	Begin document
<TITLE name="*my-title*"/>	*my-title* indicates the Experiment database.
<PATH name="*path*"/>	A set of PATH directives specifying the path names to be search for the source files. *path* is a relative or absolute path containing the source code which is correlated with the performance data. 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 match another prefix occurring in profile data files or in a program structure file, when in operation. 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

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 for another 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).

10.6.3 Native or FILE Metrics

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

```
<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.

10.6.4 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** XML 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).

Example 1

The config.xml file produced by hpcprof-flat, contains the following elements, and includes native metrics only:

```
<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 config.new file produced by hpcprof-flat and subsequently edited by the user, contains the following elements, and includes 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>
```

10.7 Using HPC Toolkit with Statically Linked Programs

10.7.1 Introduction

Dynamically linked executables are the default on modern Linux systems. For these executables HPC Toolkit's **hpcrun** script uses library preloading to add HPC Toolkit's monitoring code into an application's address space.

However, sometimes one might want to build a statically linked executable, as:

1. These are generally faster in situations where the executable spends a significant amount of time calling library functions.
2. Currently, on scalable parallel systems, Compute Node kernels do not support the use of dynamically linked executables, and so statically linked executables have to be used.

For statically linked executables, preloading HPC Toolkit's monitoring code into an application's address space at program launch is not an option. Instead, monitoring code must be added at link time; HPC Toolkit's **hpclink** script is used for this purpose.

Adding HPC Toolkit's monitoring code into a statically linked application is simple. No source code modifications are required; a change to the build procedure is required. Object (**.o**) files are compiled exactly as before, but the final link step is modified so that **hpclink** loads HPC Toolkit's monitoring code into the executable.

10.7.2 Using hpclink

hpclink statically links an application to the **hpcrun** profiling code. Dynamically linked binaries can be run directly with the **hpcrun** command, but this does not work with statically linked programs. Instead, the **hpcrun** code must be linked to the application at build time.

This approach does not require source code to be modified. Object files are compiled as before. In the application's Makefile, locate the last step in the build, that is, the command that produces the final, statically linked binary. Edit this line to place the **hpclink** command before the command line.

The argument list passed to **hpclink** should be the same command line used to build the application natively, except that the binary may be renamed with the **-o** option.

Syntax

```
hpclink [options] compiler arg
```

Options

–verbose	Verbose output
-h, –help	Print help

-u, –undefined <symbol> Pass **<symbol>** to the linker as an undefined symbol. This option is rarely needed; however, if **hpclink** fails with an undefined reference, the **–undefined** option may enable the linker to link to the symbol correctly. For example, for a linker failure with an undefined reference to **__real_foo**, use the option **-u foo**. This may be used multiple times.

Example

```
hpclink gcc -o hello -g -O -static hello.c
```

This command compiles **hello.c** with **gcc** and links in the **hpcrun** code statically.

Example

```
hpclink gcc -o program1 -static main.o foo.o ... -lm
```

This command links an **hpcrun** enabled application with object files and the math library.

Note The command line passed to **hpclink** must produce a statically linked binary; otherwise, **hpclink** will fail.

For dynamically linked executables, **hpcrun** sets environment variables to pass information to the HPC Toolkit monitoring library. On standard Linux systems, statically linked executables can still be launched with **hpcrun**.

10.7.3 Troubleshooting hpclink

Some compilers require that interprocedural optimizations to be disabled before **hpclink** is used. To instrument your statically linked executable at link time, **hpclink** uses the **ld** option **--wrap** to interpose monitoring code between your application and various process, thread, and signal control operations, e.g. **fork**, **pthread create**, and **sigprocmask**. For some compilers, interprocedural optimizations interfere with the **--wrap** option and prevents **hpclink** from working properly. If this is the case, **hpclink** will generate error messages and fail. To use **hpclink** with these compilers, interprocedural optimizations must be disabled.

Note that interprocedural optimizations may be enabled implicitly when using a compiler optimization option such as **-fast**. In cases such as this, you can often specify **-fast** along with an option such as **-no-ipa**; this option combination will provide the benefit of all of optimizations for the **-fast** option but exclude interprocedural optimizations.

10.8 Using HPC Toolkit with MPI Programs

HPC Toolkit measurement tools collect data for each process and thread of a MPI program. HPC Toolkit can be used with pure MPI programs as well as hybrid programs that use OpenMP or pthreads for multi-threaded parallelism. HPC Toolkit supports C, C++ and Fortran MPI programs. It has been successfully tested with MPICH, MVAPICH and OpenMPI programs, and should work with almost all MPI implementations.

10.8.1 Running and Analyzing MPI Programs

Launching an MPI program with hpcrun

For dynamically linked binaries, use a command line similar to that below:

```
mpirun -np num hpcrun -e EVENT:count ... program arg ...
```

Note MPI launch commands (**mpirun**, **mpiexec**, etc.) appear first with their options, then **hpcrun** and its options, and finally the application program and its command line arguments.

Compiling and running a statically linked MPI program

On systems that run statically linked binaries on the Compute Nodes, use **hpclink** to build a statically linked version of your application with the **HPC Toolkit** library linked in. For example, **hpclink mpicc -o myprog file.o ... -l<lib> ...** Then, set the **HPCRUN EVENT LIST** environment variable in the launch script, before running the application.

```
export HPCRUN_EVENT_LIST="PAPI_TOT_CYC@4000000"
mpiexec -n 64 myprog arg ...
```

See *Section 10.7* for more information on using HPC Toolkit with Statically linked programs.

hpcrun files produced for an MPI program

In the example below, **s3d f90.x** is the **Fortran S3D** program compiled with **OpenMPI** and run with the command line “

```
mpiexec -n 4 hpcrun -e PAPI TOT CYC:2500000 ./s3d f90.x”.
```

For this example, 12 files were produced, as shown below:

```
krentel 1889240 Feb 18 s3d_f90.x-000000-000-72815673-21063.hpcrun
krentel 9848    Feb 18 s3d_f90.x-000000-001-72815673-21063.hpcrun
krentel 1914680 Feb 18 s3d_f90.x-000001-000-72815673-21064.hpcrun
krentel 9848    Feb 18 s3d_f90.x-000001-001-72815673-21064.hpcrun
krentel 1908030 Feb 18 s3d_f90.x-000002-000-72815673-21065.hpcrun
krentel 7974    Feb 18 s3d_f90.x-000002-001-72815673-21065.hpcrun
krentel 1912220 Feb 18 s3d_f90.x-000003-000-72815673-21066.hpcrun
krentel 9848    Feb 18 s3d_f90.x-000003-001-72815673-21066.hpcrun
krentel 147635  Feb 18 s3d_f90.x-72815673-21063.log
krentel 142777  Feb 18 s3d_f90.x-72815673-21064.log
krentel 161266  Feb 18 s3d_f90.x-72815673-21065.log
krentel 143335  Feb 18 s3d_f90.x-72815673-21066.log
```

Here, there are four processes and two threads per process. Looking at the file names, **s3d f90.x** is the name of the program binary, **000000-000** to **000003-001** are the MPI rank and thread numbers, and **21063** to **21066** are the process **Ids**. We can see from the file sizes that **OpenMPI** is spawning one helper thread per process. Technically, the smaller **.hpcrun** files imply a smaller calling-context tree (CCT), not necessarily fewer samples. Also, in this example, the helper threads are not doing much work.

Source code requirements

Only one change is required in the source program. Early in the program, preferably directly after **MPI Init()**, the program must call **MPI Comm rank()** with the **MPI COMM WORLD** communicator. Most **MPI** programs already do this. For example, a **C** program might begin with:

```
int main(int argc, char **argv)
{
int size, rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
}
```

Note The first call to **MPI Comm rank()** should use **MPI COMM WORLD**. This sets the process's MPI rank for **hpcrun**. Other communicators are allowed, but the first call should use **MPI COMM WORLD**. Also, the call to **MPI Comm rank()** must be unconditional, that is, all processes must make this call. The call to **MPI Comm size()** is not necessary for **hpcrun**, although most **MPI** programs normally call both **MPI Comm size()** and **MPI Comm rank()**.

10.8.2 Building and Installing HPC Toolkit for MPI Support

HPC Toolkit is designed to work with multiple **MPI** implementations at the same time. That is, multiple versions of **HPC Toolkit** are not required for multiple **MPI** implementations. Each MPI implementation uses a different value for **MPI COMM WORLD**; however, **hpcrun** (**libmonitor**) waits for the application to call **MPI Comm rank()** and uses the same communicator value that the application uses. This requires that the application call **MPI Comm rank()** with communicator **MPI COMM WORLD**, as previously noted.

10.9 More Information about HPC Toolkit

See www.hpctoolkit.org for more information regarding **HPC Toolkit,** including Troubleshooting HPC Toolkit. Refer also to the man pages for use tool and the command line help.

Chapter 11. Analyzing Program Performance with HPC Toolkit

Modern computer systems provide access to a rich set of hardware performance counters that can directly measure various aspects of a program's performance. Counters in the processor core and memory hierarchy enable the collection of measures of work (e.g. operations performed), resource consumption (e.g. cycles), and inefficiency (e.g. stall cycles). System timers indicate time consumption for the different operations.

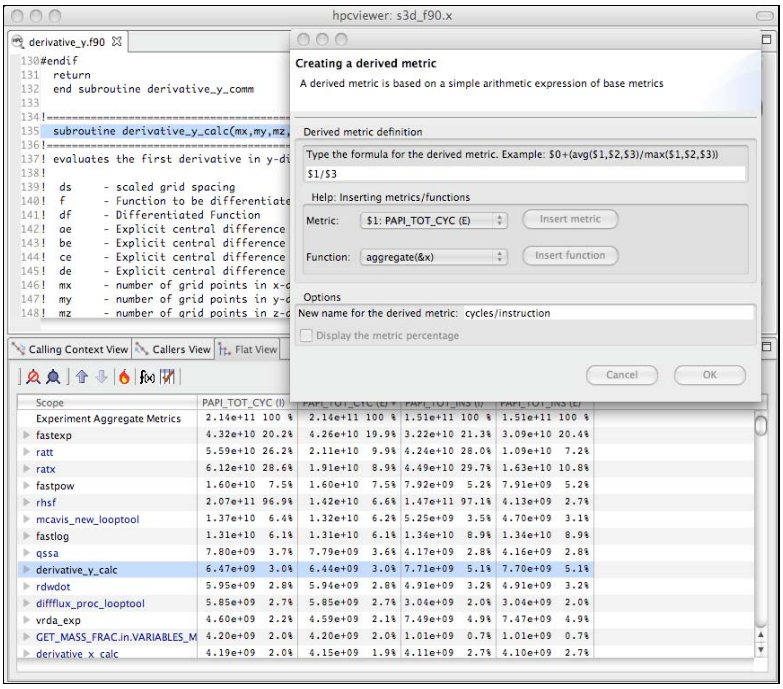
The values of individual metrics are of limited use by themselves. For instance, knowing the number of cache misses for a loop or routine is of little value by itself; only when it is combined with additional information, such as the number of instructions executed, or the total number of cache accesses does the data become informative. While a developer might not mind using mental arithmetic to evaluate the relationship between a pair of metrics for a particular program scope (e.g. a loop or a procedure), doing this for a number of program scopes is exhausting. To address this problem, **hpcviewer** supports the creation of derived metrics and provides an interface that enables spreadsheet-like formula to be used to calculate a derived metric for all program scopes.

11.1 Creating a New Derived Metric

Figure 11-1 shows how to use **hpcviewer** to create a cycles/instruction derived metric from the PAPI TOT CYC and PAPI TOT INS measured metrics; these metrics correspond to cycles and total instructions executed, measured with the PAPI hardware counter interface. Click the button marked **f(x)** above the metric window to open the derived metric window. Next, enter the formula for the metric of interest. When specifying a formula, existing metric data columns are referred to using a positional name, e.g. $n to refer to the nth column, where the first column is written as $0. The metric pane shows the formula $1/$3. Here, $1 refers to the data column representing the exclusive value for **PAPI TOT CYC**, and $3 refers to the data column representing the exclusive value for **PAPI TOT INS**.

Note An exclusive metric displays the metric value for the scope alone; an inclusive metric displays the scope metric value, including costs incurred for any functions it may call. In **hpcviewer**, inclusive metric columns are marked with **(I)**, and exclusive metric columns are marked with **(E)**.

Positional names for the metrics you use in your formula are defined by using the Metric drop down menu in the window. The metrics selected are inserted into the formula by using the insert metric button, or the positional name can be entered directly into the formula.



hpcviewer: s3d_f90.x

derivative_y.f90

```
#endif
  return
  end subroutine derivative_y_comm

!=====================================
  subroutine derivative_y_calc(mx,my,mz,
!=====================================
! evaluates the first derivative in y-di
!
!  ds      - scaled grid spacing
!  f       - Function to be differentiate
!  df      - Differentiated Function
!  ae      - Explicit central difference
!  be      - Explicit central difference
!  ce      - Explicit central difference
!  de      - Explicit central difference
!  mx      - number of grid points in x-d
!  my      - number of grid points in y-d
!  mz      - number of grid points in z-d
```

Creating a derived metric

A derived metric is based on a simple arithmetic expression of base metrics

Derived metric definition

Type the formula for the derived metric. Example: $0+(avg($1,$2,$3)/max($1,$2,$3))

$1/$3

Help: Inserting metrics/functions

Metric: $1: PAPI_TOT_CYC (E)

Function: aggregate(&x)

Options

New name for the derived metric: cycles/instruction

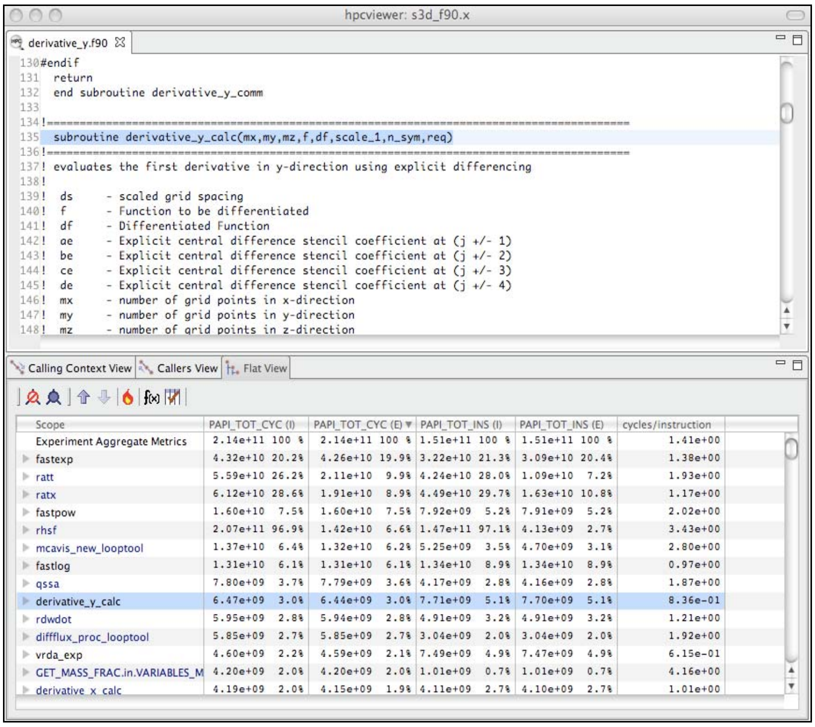
□ Display the metric percentage

Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▾	PAPI_TOT_INS (I)	PAPI_TOT_INS (E)
Experiment Aggregate Metrics	2.14e+11 100 %	2.14e+11 100 %	1.51e+11 100 %	1.51e+11 100 %
fastexp	4.32e+10 20.2%	4.26e+10 19.9%	3.22e+10 21.3%	3.09e+10 20.4%
ratt	5.59e+10 26.2%	2.11e+10 9.9%	4.24e+10 28.0%	1.09e+10 7.2%
ratx	6.12e+10 28.6%	1.91e+10 8.9%	4.49e+10 29.7%	1.63e+10 10.8%
fastpow	1.60e+10 7.5%	1.60e+10 7.5%	7.92e+09 5.2%	7.91e+09 5.2%
rhsf	2.07e+11 96.9%	1.42e+10 6.6%	1.47e+11 97.1%	4.13e+09 2.7%
mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	5.25e+09 3.5%	4.70e+09 3.1%
fastlog	1.31e+10 6.1%	1.31e+10 6.1%	1.34e+10 8.9%	1.34e+10 8.9%
qssa	7.80e+09 3.7%	7.79e+09 3.6%	4.17e+09 2.8%	4.16e+09 2.8%
derivative_y_calc	6.47e+09 3.0%	6.44e+09 3.0%	7.71e+09 5.1%	7.70e+09 5.1%
rdwdot	5.95e+09 2.8%	5.94e+09 2.8%	4.91e+09 3.2%	4.91e+09 3.2%
diffflux_proc_looptool	5.85e+09 2.7%	5.85e+09 2.7%	3.04e+09 2.0%	3.04e+09 2.0%
vrda_exp	4.60e+09 2.2%	4.59e+09 2.1%	7.49e+09 4.9%	7.47e+09 4.9%
GET_MASS_FRAC.in.VARIABLES_M	4.20e+09 2.0%	4.20e+09 2.0%	1.01e+09 0.7%	1.01e+09 0.7%
derivative_x_calc	4.19e+09 2.0%	4.15e+09 1.9%	4.11e+09 2.7%	4.10e+09 2.7%

Figure 11-1. Computing a derived metric (cycles per instruction) in **hpcviewer**

The name for the new metric is specified at the bottom of the derived metric window. The metric percentage (the percentage of the total the scopes value represents) can also be displayed by clicking the check box. For a metric that is a ratio, calculating the percentage of the total is not meaningful, so the box should be left unchecked.

After clicking **OK**, the derived metric window will disappear and the new metric will appear as the rightmost column in the metric window. If the metric pane is already filled with other metric columns, you may need to scroll right in the window to see the new metric. Alternatively, you can use the ☑ button on the navigation bar to hide some of the existing metrics so that there will be enough room on the screen to display the new metric. *Figure 11-2* shows the resulting **hpcviewer** display after clicking OK to add the derived metric.



```
#endif
  return
  end subroutine derivative_y_comm

!================================================================================
  subroutine derivative_y_calc(mx,my,mz,f,df,scale_1,n_sym,req)
!================================================================================
! evaluates the first derivative in y-direction using explicit differencing
!
!  ds     - scaled grid spacing
!  f      - Function to be differentiated
!  df     - Differentiated Function
!  ae     - Explicit central difference stencil coefficient at (j +/- 1)
!  be     - Explicit central difference stencil coefficient at (j +/- 2)
!  ce     - Explicit central difference stencil coefficient at (j +/- 3)
!  de     - Explicit central difference stencil coefficient at (j +/- 4)
!  mx     - number of grid points in x-direction
!  my     - number of grid points in y-direction
!  mz     - number of grid points in z-direction
```

Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▼	PAPI_TOT_INS (I)	PAPI_TOT_INS (E)	cycles/instruction
Experiment Aggregate Metrics	2.14e+11 100 %	2.14e+11 100 %	1.51e+11 100 %	1.51e+11 100 %	1.41e+00
fastexp	4.32e+10 20.2%	4.26e+10 19.9%	3.22e+10 21.3%	3.09e+10 20.4%	1.38e+00
ratt	5.59e+10 26.2%	2.11e+10 9.9%	4.24e+10 28.0%	1.09e+10 7.2%	1.93e+00
ratx	6.12e+10 28.6%	1.91e+10 8.9%	4.49e+10 29.7%	1.63e+10 10.8%	1.17e+00
fastpow	1.60e+10 7.5%	1.60e+10 7.5%	7.92e+09 5.2%	7.91e+09 5.2%	2.02e+00
rhsf	2.07e+11 96.9%	1.42e+10 6.6%	1.47e+11 97.1%	4.13e+09 2.7%	3.43e+00
mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	5.25e+09 3.5%	4.70e+09 3.1%	2.80e+00
fastlog	1.31e+10 6.1%	1.31e+10 6.1%	1.34e+10 8.9%	1.34e+10 8.9%	0.97e+00
qssa	7.80e+09 3.7%	7.79e+09 3.6%	4.17e+09 2.8%	4.16e+09 2.8%	1.87e+00
derivative_y_calc	6.47e+09 3.0%	6.44e+09 3.0%	7.71e+09 5.1%	7.70e+09 5.1%	8.36e-01
rdwdot	5.95e+09 2.8%	5.94e+09 2.8%	4.91e+09 3.2%	4.91e+09 3.2%	1.21e+00
diffflux_proc_looptool	5.85e+09 2.7%	5.85e+09 2.7%	3.04e+09 2.0%	3.04e+09 2.0%	1.92e+00
vrda_exp	4.60e+09 2.2%	4.59e+09 2.1%	7.49e+09 4.9%	7.47e+09 4.9%	6.15e-01
GET_MASS_FRAC.in.VARIABLES_M	4.20e+09 2.0%	4.20e+09 2.0%	1.01e+09 0.7%	1.01e+09 0.7%	4.16e+00
derivative_x_calc	4.19e+09 2.0%	4.15e+09 1.9%	4.11e+09 2.7%	4.10e+09 2.7%	1.01e+00

Figure 11-2. Displaying the new **cycles/instruction** derived metric in **hpcviewer**

The following sections describe several types of derived metrics that are of particular use to gain insight into performance bottlenecks and opportunities for tuning.

11.2 Using Derived Metrics to Improve Performance

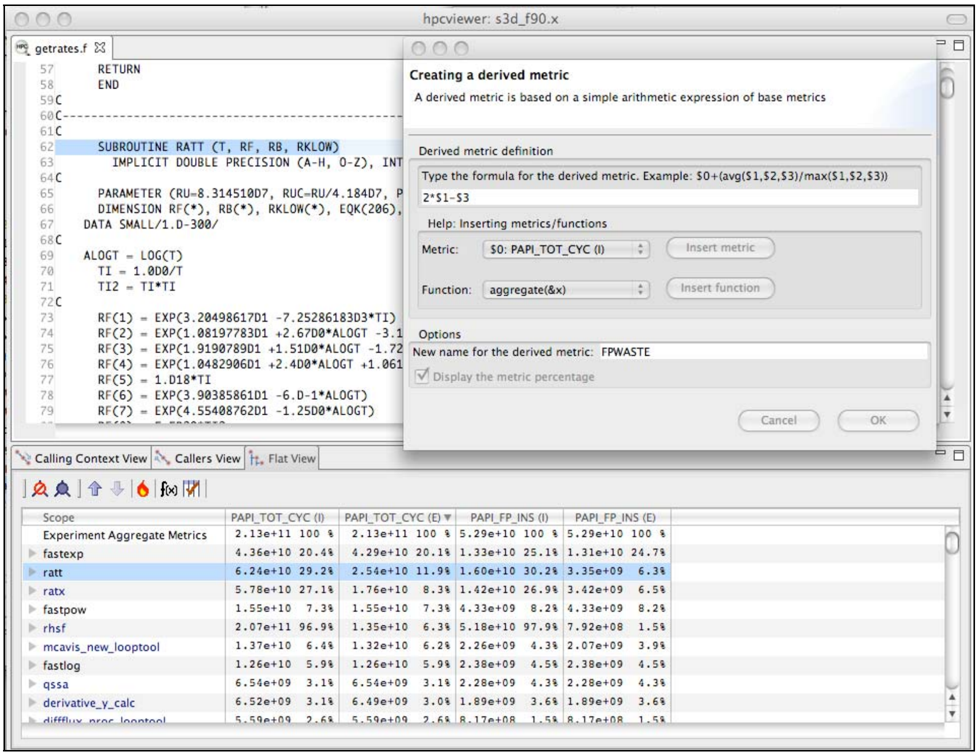
Knowing which program operations take most time or where most floating point operations occur is useful, however this is may not be sufficient to identify where performance can be improved. For program tuning, it is better to know where the resources are used inefficiently, than knowing the quantity of resources (e.g., time, instructions) consumed for program context.

To identify performance problems, it might appear useful to compute ratios to see how many events per cycle occur in each program context. For instance, one might compute ratios such as **FLOPs/cycle**, **instructions/cycle**, or the **cache miss** ratios. However, these ratios may be misleading, as there may be program contexts, e.g. loops, where computation is highly inefficient, e.g. with low operation counts per cycle. An additional consideration, thought, is that the inefficient contexts may not account for a significant amount of execution time, and just because a loop is inefficient does not mean that it is important for tuning.

The best tuning possibilities occur where the aggregate performance losses are greatest. For instance, consider a program with two loops. The first loop might account for 90% of the execution time and run at 50% of peak performance. The second loop might account for 10% of the execution time, but only achieve 12% of peak performance. In this case, the total performance loss for the first loop accounts for 50% of its execution time, which corresponds to 45% of the total program execution time. The 88% performance loss in the second loop would account for only 8.8% of the program's execution time. Therefore, tuning the first loop has a greater potential for improving the program performance, even though the second loop is less efficient.

A good way to focus on inefficiency directly is with a derived waste metric. These metrics are easy to compute, although, there is not a single universal measure of waste for all codes. Depending upon what one uses as the rate-limiting resource, e.g. floating- point computation, memory bandwidth, etc., one can define an appropriate waste metric, e.g., FLOP opportunities missed; bandwidth not consumed and apply that.

For instance, for a floating-point intensive code, one might consider keeping the floating-point pipeline full as a metric of success. One can pinpoint and quantify losses of this nature by computing a floating point waste metric, which is calculated as the difference between the potential number of calculations that could have been performed if the computation was running at its peak rate minus the actual number that were performed. To compute the number of calculations that could have been completed in each scope, multiply the total number of cycles used in the scope by the peak rate of operations per cycle. Using **hpcviewer,** one can specify a formula to compute such a derived metric and it will compute the value of the derived metric for all scopes. *Figure 11-3* shows the specification of this floating-point waste metric for a code.



```
      RETURN
      END
C
C-----------------------------------------------------
C
      SUBROUTINE RATT (T, RF, RB, RKLOW)
        IMPLICIT DOUBLE PRECISION (A-H, O-Z), INT
C
      PARAMETER (RU=8.314510D7, RUC=RU/4.184D7, P
      DIMENSION RF(*), RB(*), RKLOW(*), EQK(206),
    DATA SMALL/1.D-300/
C
    ALOGT = LOG(T)
      TI = 1.0D0/T
      TI2 = TI*TI
C
      RF(1) = EXP(3.20498617D1 -7.25286183D3*TI)
      RF(2) = EXP(1.08197783D1 +2.67D0*ALOGT -3.1
      RF(3) = EXP(1.9190789D1 +1.51D0*ALOGT -1.72
      RF(4) = EXP(1.0482906D1 +2.4D0*ALOGT +1.061
      RF(5) = 1.D18*TI
      RF(6) = EXP(3.90385861D1 -6.D-1*ALOGT)
      RF(7) = EXP(4.55408762D1 -1.25D0*ALOGT)
```

Creating a derived metric

A derived metric is based on a simple arithmetic expression of base metrics

Derived metric definition

Type the formula for the derived metric. Example: $0+(avg($1,$2,$3)/max($1,$2,$3))

2*$1-$3

Help: Inserting metrics/functions

Metric: $0: PAPI_TOT_CYC (I) [Insert metric]

Function: aggregate(&x) [Insert function]

Options

New name for the derived metric: FPWASTE

☑ Display the metric percentage

[Cancel] [OK]

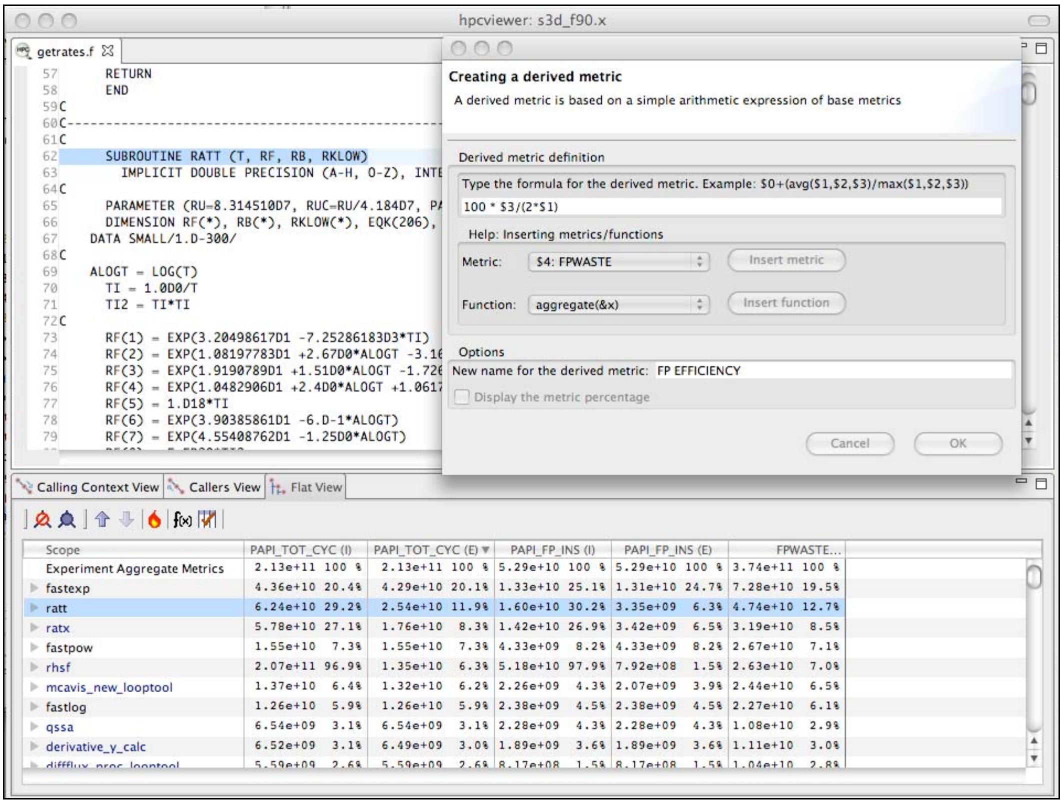
Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▼	PAPI_FP_INS (I)	PAPI_FP_INS (E)
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %
fastexp	4.36e+10 20.4%	4.29e+10 20.1%	1.33e+10 25.1%	1.31e+10 24.7%
ratt	6.24e+10 29.2%	2.54e+10 11.9%	1.60e+10 30.2%	3.35e+09 6.3%
ratx	5.78e+10 27.1%	1.76e+10 8.3%	1.42e+10 26.9%	3.42e+09 6.5%
fastpow	1.55e+10 7.3%	1.55e+10 7.3%	4.33e+09 8.2%	4.33e+09 8.2%
rhsf	2.07e+11 96.9%	1.35e+10 6.3%	5.18e+10 97.9%	7.92e+08 1.5%
mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	2.26e+09 4.3%	2.07e+09 3.9%
fastlog	1.26e+10 5.9%	1.26e+10 5.9%	2.38e+09 4.5%	2.38e+09 4.5%
qssa	6.54e+09 3.1%	6.54e+09 3.1%	2.28e+09 4.3%	2.28e+09 4.3%
derivative_y_calc	6.52e+09 3.1%	6.49e+09 3.0%	1.89e+09 3.6%	1.89e+09 3.6%
diffflux_proc_looptool	5.59e+09 2.6%	5.59e+09 2.6%	8.17e+08 1.5%	8.17e+08 1.5%

Figure 11-3. Computing a floating point waste metric in **hpcviewer**

Sorting by a waste metric will rank scopes according to the waste amounts. Those with the greatest waste will also be those that provide the best opportunities for improving overall program performance. A waste metric will typically highlight loops where:

- a lot of time is spent computing efficiently, but the aggregate inefficiencies accumulate,
- less time is spent computing, but the computation is rather inefficient, and
- scopes such as copy loops that contain no computation at all, which represent a complete waste according to a metric such as floating point waste.

In addition to a waste metric, one can compute a companion derived metric, relative efficiency metric, to pinpoint the possibilities for improving performance. A scope running efficiently will typically be much harder to tune than one running less efficiently. For our floating point waste metric, we one can compute the floating point efficiency metric by dividing FLOPs measured by potential peak FLOPS and multiplying the result by 100.



```
      RETURN
      END
C
C-----------------------------------------------
C
      SUBROUTINE RATT (T, RF, RB, RKLOW)
        IMPLICIT DOUBLE PRECISION (A-H, O-Z), INTE
C
      PARAMETER (RU=8.314510D7, RUC=RU/4.184D7, PA
      DIMENSION RF(*), RB(*), RKLOW(*), EQK(206),
    DATA SMALL/1.D-300/
C
    ALOGT = LOG(T)
      TI = 1.0D0/T
      TI2 = TI*TI
C
      RF(1) = EXP(3.20498617D1 -7.25286183D3*TI)
      RF(2) = EXP(1.08197783D1 +2.67D0*ALOGT -3.16
      RF(3) = EXP(1.9190789D1 +1.51D0*ALOGT -1.726
      RF(4) = EXP(1.0482906D1 +2.4D0*ALOGT +1.0617
      RF(5) = 1.D18*TI
      RF(6) = EXP(3.90385861D1 -6.D-1*ALOGT)
      RF(7) = EXP(4.55408762D1 -1.25D0*ALOGT)
```

Creating a derived metric

A derived metric is based on a simple arithmetic expression of base metrics

Derived metric definition

Type the formula for the derived metric. Example: $0+(avg($1,$2,$3)/max($1,$2,$3))

100 * $3/(2*$1)

Help: Inserting metrics/functions

Metric: $4: FPWASTE

Function: aggregate(&x)

Options

New name for the derived metric: FP EFFICIENCY

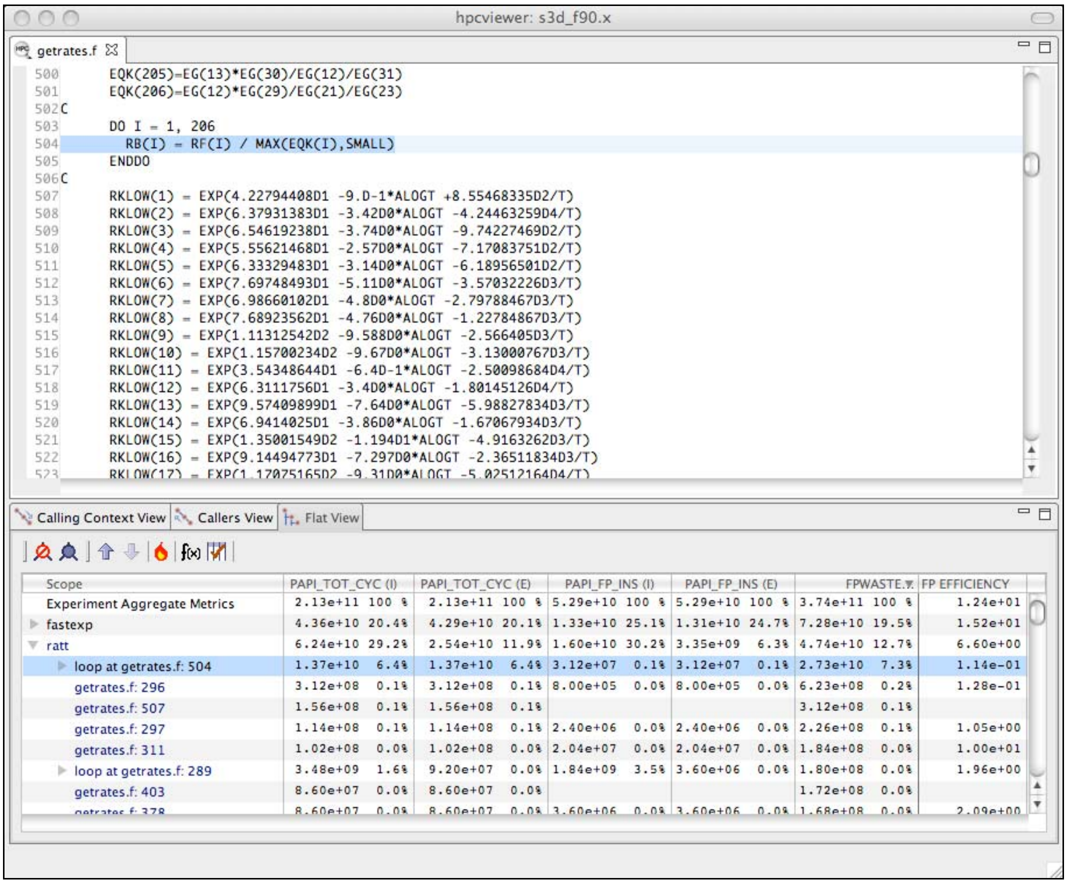
□ Display the metric percentage

Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E) ▼	PAPI_FP_INS (I)	PAPI_FP_INS (E)	FPWASTE...
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %	3.74e+11 100 %
fastexp	4.36e+10 20.4%	4.29e+10 20.1%	1.33e+10 25.1%	1.31e+10 24.7%	7.28e+10 19.5%
ratt	6.24e+10 29.2%	2.54e+10 11.9%	1.60e+10 30.2%	3.35e+09 6.3%	4.74e+10 12.7%
ratx	5.78e+10 27.1%	1.76e+10 8.3%	1.42e+10 26.9%	3.42e+09 6.5%	3.19e+10 8.5%
fastpow	1.55e+10 7.3%	1.55e+10 7.3%	4.33e+09 8.2%	4.33e+09 8.2%	2.67e+10 7.1%
rhsf	2.07e+11 96.9%	1.35e+10 6.3%	5.18e+10 97.9%	7.92e+08 1.5%	2.63e+10 7.0%
mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	2.26e+09 4.3%	2.07e+09 3.9%	2.44e+10 6.5%
fastlog	1.26e+10 5.9%	1.26e+10 5.9%	2.38e+09 4.5%	2.38e+09 4.5%	2.27e+10 6.1%
qssa	6.54e+09 3.1%	6.54e+09 3.1%	2.28e+09 4.3%	2.28e+09 4.3%	1.08e+10 2.9%
derivative_y_calc	6.52e+09 3.1%	6.49e+09 3.0%	1.89e+09 3.6%	1.89e+09 3.6%	1.11e+10 3.0%
diffflux_proc_looptool	5.59e+09 2.6%	5.59e+09 2.6%	8.17e+08 1.5%	8.17e+08 1.5%	1.04e+10 2.8%

Figure 11-4. Computing floating point efficiency in percent using **hpcviewer**

Scopes with a high waste metric ranking and a low relative efficiency metric often make the best targets for optimization. *Figure 11-5* shows the specification of a floating-point efficiency metric for a code. The top two routines in the **hpcviewer Flat View** window when combined account for 32.2% of the floating-point waste in a reactive turbulent combustion code. The second routine (**ratt**) is expanded to show the loops and statements within. While the overall floating point efficiency for **ratt** is 6.6% of peak (shown in scientific notation in the **hpcviewer** window below), the most costly loop in **ratt** that accounts for 7.3% of the floating point waste is executing at only .114% FP Efficiency.

Identifying the sources of inefficiency is the first step towards improving program performance via tuning.



```
500        EQK(205)=EG(13)*EG(30)/EG(12)/EG(31)
501        EQK(206)=EG(12)*EG(29)/EG(21)/EG(23)
502C
503        DO I = 1, 206
504          RB(I) = RF(I) / MAX(EQK(I),SMALL)
505        ENDDO
506C
507        RKLOW(1) = EXP(4.22794408D1 -9.D-1*ALOGT +8.55468335D2/T)
508        RKLOW(2) = EXP(6.37931383D1 -3.42D0*ALOGT -4.24463259D4/T)
509        RKLOW(3) = EXP(6.54619238D1 -3.74D0*ALOGT -9.74227469D2/T)
510        RKLOW(4) = EXP(5.55621468D1 -2.57D0*ALOGT -7.17083751D2/T)
511        RKLOW(5) = EXP(6.33329483D1 -3.14D0*ALOGT -6.18956501D2/T)
512        RKLOW(6) = EXP(7.69748493D1 -5.11D0*ALOGT -3.57032226D3/T)
513        RKLOW(7) = EXP(6.98660102D1 -4.8D0*ALOGT -2.79788467D3/T)
514        RKLOW(8) = EXP(7.68923562D1 -4.76D0*ALOGT -1.22784867D3/T)
515        RKLOW(9) = EXP(1.11312542D2 -9.588D0*ALOGT -2.566405D3/T)
516        RKLOW(10) = EXP(1.15700234D2 -9.67D0*ALOGT -3.13000767D3/T)
517        RKLOW(11) = EXP(3.54348644D1 -6.4D-1*ALOGT -2.50098684D4/T)
518        RKLOW(12) = EXP(6.3111756D1 -3.4D0*ALOGT -1.80145126D4/T)
519        RKLOW(13) = EXP(9.57409899D1 -7.64D0*ALOGT -5.98827834D3/T)
520        RKLOW(14) = EXP(6.9414025D1 -3.86D0*ALOGT -1.67067934D3/T)
521        RKLOW(15) = EXP(1.35001549D2 -1.194D1*ALOGT -4.9163262D3/T)
522        RKLOW(16) = EXP(9.14494773D1 -7.297D0*ALOGT -2.36511834D3/T)
523        RKLOW(17) = EXP(1.17075165D2 -9.31D0*ALOGT -5.02512164D4/T)
```

Scope	PAPI_TOT_CYC (I)	PAPI_TOT_CYC (E)	PAPI_FP_INS (I)	PAPI_FP_INS (E)	FPWASTE	FP EFFICIENCY
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %	3.74e+11 100 %	1.24e+01
fastexp	4.36e+10 20.4%	4.29e+10 20.1%	1.33e+10 25.1%	1.31e+10 24.7%	7.28e+10 19.5%	1.52e+01
ratt	6.24e+10 29.2%	2.54e+10 11.9%	1.60e+10 30.2%	3.35e+09 6.3%	4.74e+10 12.7%	6.60e+00
loop at getrates.f: 504	1.37e+10 6.4%	1.37e+10 6.4%	3.12e+07 0.1%	3.12e+07 0.1%	2.73e+10 7.3%	1.14e-01
getrates.f: 296	3.12e+08 0.1%	3.12e+08 0.1%	8.00e+05 0.0%	8.00e+05 0.0%	6.23e+08 0.2%	1.28e-01
getrates.f: 507	1.56e+08 0.1%	1.56e+08 0.1%			3.12e+08 0.1%	
getrates.f: 297	1.14e+08 0.1%	1.14e+08 0.1%	2.40e+06 0.0%	2.40e+06 0.0%	2.26e+08 0.1%	1.05e+00
getrates.f: 311	1.02e+08 0.0%	1.02e+08 0.0%	2.04e+07 0.0%	2.04e+07 0.0%	1.84e+08 0.0%	1.00e+01
loop at getrates.f: 289	3.48e+09 1.6%	9.20e+07 0.0%	1.84e+09 3.5%	3.60e+06 0.0%	1.80e+08 0.0%	1.96e+00
getrates.f: 403	8.60e+07 0.0%	8.60e+07 0.0%			1.72e+08 0.0%	
getrates.f: 378	8.60e+07 0.0%	8.60e+07 0.0%	3.60e+06 0.0%	3.60e+06 0.0%	1.68e+08 0.0%	2.09e+00

Figure 11-5. Floating-point efficiency metric

11.3 Pinpointing and Quantifying Scalability Bottlenecks

On large-scale parallel systems, identifying impediments to scalability is of paramount importance. Two kinds of scalability are of particular interest for multicore processor systems:

- Scaling within nodes
- Scaling across the entire system

HPC Toolkit can be used to pinpoint and quantify bottlenecks for both types of scalability using call path profiles collected by **hpcrun**.

Use **differential profiling** to pinpoint scalability bottlenecks in parallel programs. Combinations of different execution profiles are compared.

See The *Differential profiling* paper written by P. E. McKenney and available from the *IEEE Computer Society* for more information.

Differentiating flat profiles can help to identify where in a program different costs are incurred for different executions.

Building upon McKenney's idea of differential profiling, call path profiles of parallel executions at different scales can be compared to pinpoint scalability bottlenecks. Differential analysis of call path profiles pinpoints not only differences between two executions (in this case scalability losses), but the contexts in which those differences occur.

Associating changes in cost according to calling contexts is particularly important for pinpointing context-dependent behavior for parallel programs. For instance, in message passing programs, the time spent by a call to **MPI Wait** depends upon the context in which it is called. Similarly, how the performance of a communication event scales as the number of processors in a parallel execution increases, depends upon a variety of factors such as whether the size of the data transferred increases or whether the communication is collective or not.

11.3.1 Scalability Analysis Using Expectations

Application developers have expectations about how the performance of their code should scale as the number of processors available for a parallel execution increases. Namely;

- When different numbers of processors are used to solve the same problem (strong scaling), one expects an execution's speedup to increase linearly with the number of processors employed;
- When different numbers of processors are used but the amount of computation per processor is held constant (weak scaling), one expects the total execution time with different numbers of processors to be the same.

In both these situations, a code developer can express their expectations for how performance will scale, as a formula that can be used to predict execution performance for different numbers of processors. One's expectations about how overall application performance should scale can be applied to each program context to pinpoint and quantify deviations from expected scaling. Specifically, one can scale and look at the performance of an application on different numbers of processors to pinpoint the contexts that are not scaling ideally.

To pinpoint and quantify scalability bottlenecks for a parallel application, we first use **hpcrun** to generate a call path profile for an application on two sets of processors.

Let E_p be an execution on p processors and E_q be an execution on q processors. Without loss of generality, assume that $q > p$.

In our analysis, we consider both *inclusive* and *exclusive* costs for **Calling Context Tree (CCT)** nodes. The inclusive cost at n represents the sum of all costs attributed to **n** and any of its descendants in the CCT, and is denoted by $I(n)$. The exclusive cost at n represents the sum of all costs strictly attributed to n, and we denote it by $E(n)$. If n is an interior node in a CCT, it represents an invocation of a procedure. If n is a leaf in a CCT, it represents a statement inside some procedure. For leaves, their inclusive and exclusive costs are equal.

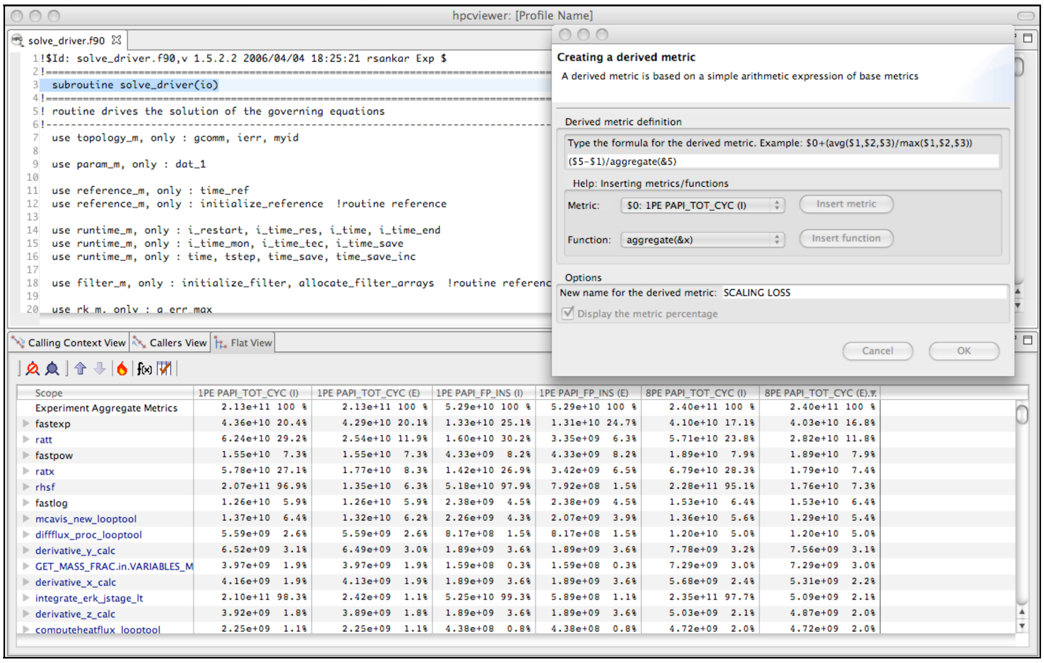
It is useful to perform scalability analysis for both inclusive and exclusive costs; if the loss of scalability attributed to the inclusive costs of a function's invocation is roughly equal to the loss of scalability due to its exclusive costs, then we know that the computation for that function's invocation does not scale. However, if the loss of scalability attributed to a function's invocation inclusive costs outweighs the loss of scalability accounted for by its exclusive costs, then we need to explore the scalability of the function's sub-functions and calls.

Given CCTs for an ensemble of executions, the next step for the scalability performance analysis is to define clearly our expectations. By looking at performance expectations for weak scaling and intuitive metrics, it is possible to see how much performance can deviate from our expectations.

See *Coarfa, C., Mellor-Crummey, J., Froyd, N., and Dotsenko, Y. 2007. Scalability analysis of SPMD codes using expectations* in the **Proceedings of the 21st Annual International Conference on Supercomputing** (Seattle, Washington, June 17 - 21, 2007) for more information on scalability analysis technique.

11.3.2 Weak Scaling

Consider two weak scaling experiments executed on p and q processors, respectively, $p < q$. *Figure 11-6* shows how a derived metric can be used to compute and attribute scalability losses. Here, we compute the difference in exclusive cycles used on one core of an 8-core run and one core in a single core run in a weak scaling experiment. If the code has perfect weak scaling, the time for the one core and the eight core executions would be identical. We compute the amount of excess work, by computing the difference between the eight-core run time minus the single core run time for each scope, and divide that by the total time spent by the eight core run. This formula tells us how much extra time we spent for the eight core run, and attributes differences to each scope. The fraction of excess work is a quantitative measure of scalability loss.



```
!$Id: solve_driver.f90,v 1.5.2.2 2006/04/04 18:25:21 rsankar Exp $
!===========================================================================================
 subroutine solve_driver(io)
!===========================================================================================
! routine drives the solution of the governing equations
!-------------------------------------------------------------------------------------------
 use topology_m, only : gcomm, ierr, myid

 use param_m, only : dat_1

 use reference_m, only : time_ref
 use reference_m, only : initialize_reference  !routine reference

 use runtime_m, only : i_restart, i_time_res, i_time, i_time_end
 use runtime_m, only : i_time_mon, i_time_tec, i_time_save
 use runtime_m, only : time, tstep, time_save, time_save_inc

 use filter_m, only : initialize_filter, allocate_filter_arrays  !routine referenc

 use rk_m, only : q_err_max
```

Creating a derived metric

A derived metric is based on a simple arithmetic expression of base metrics

Derived metric definition

Type the formula for the derived metric. Example: $0+(avg($1,$2,$3)/max($1,$2,$3))

($5-$1)/aggregate(&5)

Help: Inserting metrics/functions

Metric: $0: 1PE PAPI_TOT_CYC (I) [Insert metric]

Function: aggregate(&x) [Insert function]

Options

New name for the derived metric: SCALING LOSS

☑ Display the metric percentage

[Cancel] [OK]

Scope	1PE PAPI_TOT_CYC (I)	1PE PAPI_TOT_CYC (E)	1PE PAPI_FP_INS (I)	1PE PAPI_FP_INS (E)	8PE PAPI_TOT_CYC (I)	8PE PAPI_TOT_CYC (E)
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %	2.40e+11 100 %	2.40e+11 100 %
fastexp	4.36e+10 20.4%	4.29e+10 20.1%	1.33e+10 25.1%	1.31e+10 24.7%	4.10e+10 17.1%	4.03e+10 16.8%
ratt	6.24e+10 29.2%	2.54e+10 11.9%	1.60e+10 30.2%	3.35e+09 6.3%	5.71e+10 23.8%	2.82e+10 11.8%
fastpow	1.55e+10 7.3%	1.55e+10 7.3%	4.33e+09 8.2%	4.33e+09 8.2%	1.89e+10 7.9%	1.89e+10 7.9%
ratx	5.78e+10 27.1%	1.77e+10 8.3%	1.42e+10 26.9%	3.42e+09 6.5%	6.79e+10 28.3%	1.79e+10 7.4%
rhsf	2.07e+11 96.9%	1.35e+10 6.3%	5.18e+10 97.9%	7.92e+08 1.5%	2.28e+11 95.1%	1.76e+10 7.3%
fastlog	1.26e+10 5.9%	1.26e+10 5.9%	2.38e+09 4.5%	2.38e+09 4.5%	1.53e+10 6.4%	1.53e+10 6.4%
mcavis_new_looptool	1.37e+10 6.4%	1.32e+10 6.2%	2.26e+09 4.3%	2.07e+09 3.9%	1.36e+10 5.6%	1.29e+10 5.4%
diffflux_proc_looptool	5.59e+09 2.6%	5.59e+09 2.6%	8.17e+08 1.5%	8.17e+08 1.5%	1.20e+10 5.0%	1.20e+10 5.0%
derivative_y_calc	6.52e+09 3.1%	6.49e+09 3.0%	1.89e+09 3.6%	1.89e+09 3.6%	7.78e+09 3.2%	7.56e+09 3.1%
GET_MASS_FRAC.in.VARIABLES_M	3.97e+09 1.9%	3.97e+09 1.9%	1.59e+08 0.3%	1.59e+08 0.3%	7.29e+09 3.0%	7.29e+09 3.0%
derivative_x_calc	4.16e+09 1.9%	4.13e+09 1.9%	1.89e+09 3.6%	1.89e+09 3.6%	5.68e+09 2.4%	5.31e+09 2.2%
integrate_erk_jstage_lt	2.10e+11 98.3%	2.42e+09 1.1%	5.25e+10 99.3%	5.89e+08 1.1%	2.35e+11 97.7%	5.09e+09 2.1%
derivative_z_calc	3.92e+09 1.8%	3.89e+09 1.8%	1.89e+09 3.6%	1.89e+09 3.6%	5.03e+09 2.1%	4.87e+09 2.0%
computeheatflux_looptool	2.25e+09 1.1%	2.25e+09 1.1%	4.38e+08 0.8%	4.38e+08 0.8%	4.72e+09 2.0%	4.72e+09 2.0%

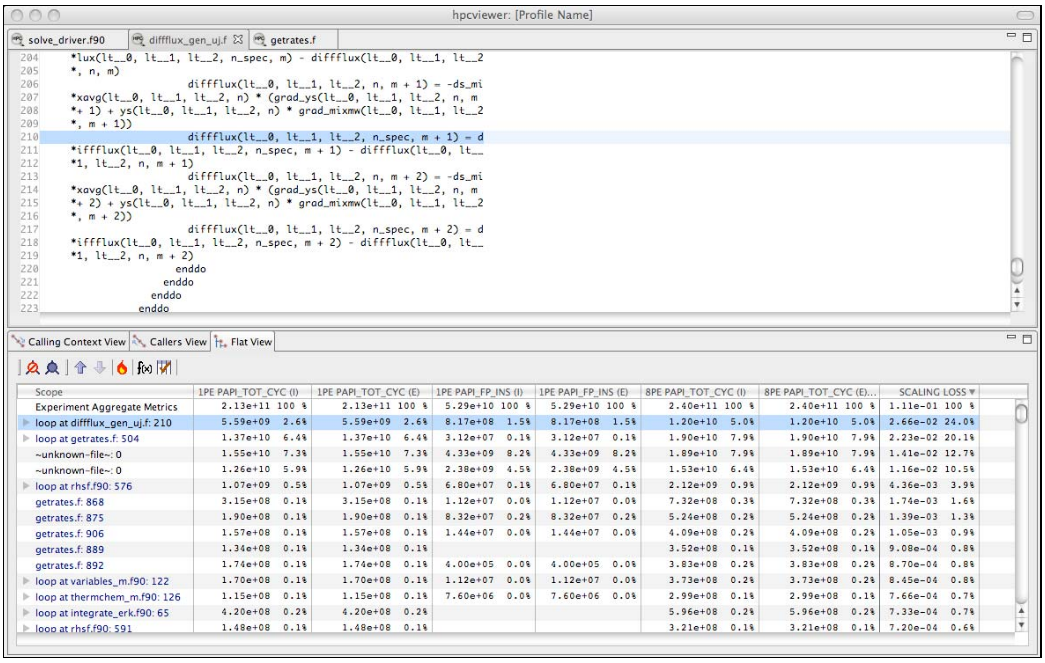
Figure 11-6. Scaling Loss Metric

By normalizing the total time spent for the eight-core run, we can attribute the fraction of the total execution excess time for each scope, when scaling from one to eight cores.

In **hpcviewer**, this metric for each scope ***s*** is computed by subtracting the exclusive time spent in **s** on one core (**$1**) from the time spent in **s** on eight cores (**$5**), and normalizing this quantity by the total aggregate time spent on 8 cores (**aggregate(&5)**).

This calculation pinpoints and quantifies scaling losses within a multicore node. A similar analysis can be applied to compute scaling losses between pairs of jobs scaled across an entire parallel system, and not just within a node.

Figure 11-7 shows an example of loop nests ranked by the **Scaling loss** metric. The source Window shows the loop nest responsible for the greatest scaling loss when scaling from one to eight cores. Unsurprisingly, the loop with the worst scaling loss is very memory intensive. Memory bandwidth is a precious commodity on multicore processors.



```
     *lux(lt__0, lt__1, lt__2, n_spec, m) - diffflux(lt__0, lt__1, lt__2
     *, n, m)
                          diffflux(lt__0, lt__1, lt__2, n, m + 1) = -ds_mi
     *xavg(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m
     *+ 1) + ys(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2
     *, m + 1))
                          diffflux(lt__0, lt__1, lt__2, n_spec, m + 1) = d
     *iffflux(lt__0, lt__1, lt__2, n_spec, m + 1) - diffflux(lt__0, lt__
     *1, lt__2, n, m + 1)
                          diffflux(lt__0, lt__1, lt__2, n, m + 2) = -ds_mi
     *xavg(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m
     *+ 2) + ys(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2
     *, m + 2))
                          diffflux(lt__0, lt__1, lt__2, n_spec, m + 2) = d
     *iffflux(lt__0, lt__1, lt__2, n_spec, m + 2) - diffflux(lt__0, lt__
     *1, lt__2, n, m + 2)
                        enddo
                      enddo
                    enddo
                  enddo
```

Scope	1PE PAPI_TOT_CYC (I)	1PE PAPI_TOT_CYC (E)	1PE PAPI_FP_INS (I)	1PE PAPI_FP_INS (E)	8PE PAPI_TOT_CYC (I)	8PE PAPI_TOT_CYC (E)...	SCALING LOSS ▼
Experiment Aggregate Metrics	2.13e+11 100 %	2.13e+11 100 %	5.29e+10 100 %	5.29e+10 100 %	2.40e+11 100 %	2.40e+11 100 %	1.11e-01 100 %
▶ loop at diffflux_gen_uj.f: 210	5.59e+09 2.6%	5.59e+09 2.6%	8.17e+08 1.5%	8.17e+08 1.5%	1.20e+10 5.0%	1.20e+10 5.0%	2.66e-02 24.0%
▶ loop at getrates.f: 504	1.37e+10 6.4%	1.37e+10 6.4%	3.12e+07 0.1%	3.12e+07 0.1%	1.90e+10 7.9%	1.90e+10 7.9%	2.23e-02 20.1%
~unknown-file~: 0	1.55e+10 7.3%	1.55e+10 7.3%	4.33e+09 8.2%	4.33e+09 8.2%	1.89e+10 7.9%	1.89e+10 7.9%	1.41e-02 12.7%
~unknown-file~: 0	1.26e+10 5.9%	1.26e+10 5.9%	2.38e+09 4.5%	2.38e+09 4.5%	1.53e+10 6.4%	1.53e+10 6.4%	1.16e-02 10.5%
▶ loop at rhsf.f90: 576	1.07e+09 0.5%	1.07e+09 0.5%	6.80e+07 0.1%	6.80e+07 0.1%	2.12e+09 0.9%	2.12e+09 0.9%	4.36e-03 3.9%
getrates.f: 868	3.15e+08 0.1%	3.15e+08 0.1%	1.12e+07 0.0%	1.12e+07 0.0%	7.32e+08 0.3%	7.32e+08 0.3%	1.74e-03 1.6%
getrates.f: 875	1.90e+08 0.1%	1.90e+08 0.1%	8.32e+07 0.2%	8.32e+07 0.2%	5.24e+08 0.2%	5.24e+08 0.2%	1.39e-03 1.3%
getrates.f: 906	1.57e+08 0.1%	1.57e+08 0.1%	1.44e+07 0.0%	1.44e+07 0.0%	4.09e+08 0.2%	4.09e+08 0.2%	1.05e-03 0.9%
getrates.f: 889	1.34e+08 0.1%	1.34e+08 0.1%			3.52e+08 0.1%	3.52e+08 0.1%	9.08e-04 0.8%
getrates.f: 892	1.74e+08 0.1%	1.74e+08 0.1%	4.00e+05 0.0%	4.00e+05 0.0%	3.83e+08 0.2%	3.83e+08 0.2%	8.70e-04 0.8%
▶ loop at variables_m.f90: 122	1.70e+08 0.1%	1.70e+08 0.1%	1.12e+07 0.0%	1.12e+07 0.0%	3.73e+08 0.2%	3.73e+08 0.2%	8.45e-04 0.8%
▶ loop at thermchem_m.f90: 126	1.15e+08 0.1%	1.15e+08 0.1%	7.60e+06 0.0%	7.60e+06 0.0%	2.99e+08 0.1%	2.99e+08 0.1%	7.66e-04 0.7%
▶ loop at integrate_erk.f90: 65	4.20e+08 0.2%	4.20e+08 0.2%			5.96e+08 0.2%	5.96e+08 0.2%	7.33e-04 0.7%
▶ loop at rhsf.f90: 591	1.48e+08 0.1%	1.48e+08 0.1%			3.21e+08 0.1%	3.21e+08 0.1%	7.20e-04 0.6%

Figure 11-7. Loop nests ranked by Scaling loss

It is also possible to compute scaling loss where there is strong scaling, however the work on the larger number of processors has to have a corrective multiplier applied to account for the smaller fraction of work it receives.

11.3.3 Exploring Scaling Losses

Scaling losses can be explored in **hpcviewer** using its three views:

Calling context view

This top-down view represents dynamic calling contexts (call paths) where the costs were incurred.

Callers view

This bottom up view enables one to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are used in more than one context, such as communication library routines.

Flat view

This view organizes performance measurement data according to the static structure of an application. All costs incurred in any calling context by a procedure are aggregated together in one flat view.

Developers can use these views for CCTs, which are annotated with costs, to pinpoint performance bottlenecks quickly.

Typically, one begins analyzing an application's scalability and performance by using the top-down Calling context view. Using this view, one can readily see how costs and scalability losses are associated with different calling contexts. If costs or scalability losses are associated with only a few calling contexts, then this view suffices for identifying the bottlenecks. When scalability losses are spread among many calling contexts, e.g., among different invocations of MPI Wait, often it is useful to switch to the bottom-up Callers view to see which losses are due to the same underlying cause. In the bottom-up view, one can sort routines by their *exclusive* scalability losses, and then look upward to see how these losses accumulate from the different calling contexts, in which the routine was invoked.

Scaling loss based on excess work is intuitive; perfect scaling corresponds to a excess work value of 0, sublinear scaling yields positive values, and superlinear scaling yields negative values. Typically, CCTs for SPMD programs have similar structure. If CCTs for different executions diverge, using hpcviewer to compute and report excess work will highlight these program regions.

Inclusive excess work and exclusive excess work serve as useful measures of scalability associated with nodes in a CCT. By computing both metrics, one can determine if the application scales well or not for a CCT node and also pinpoint the cause if is scales poorly. If a node for a function in the CCT has comparable positive values for both inclusive excess work and exclusive excess work, then the loss of scaling is due to a computation problem in the function itself. However, if the inclusive excess work for the function outweighs that accounted for by its exclusive costs, then one should explore the scalability for it sub-functions and routine calls. To isolate code that is an impediment to scalable performance, one can use the hpcviewer hot call path button to trace a path down through the CCT to see where costs are incurred.

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

ABI
Application Binary Interface

ACL
Access Control List

ACT
Administration Configuration Tool

ANL
Argonne National Laboratory (MPICH2)

API
Application Programmer Interface

ARP
Address Resolution Protocol

ASIC
Application Specific Integrated Circuit

B

BAS
Bull Advanced Server

BIOS
Basic Input Output System

Blade
Thin server that is inserted in a blade chassis

BLACS
Basic Linear Algebra Communication Subprograms

BLAS
Basic Linear Algebra Subprograms

BMC
Baseboard Management Controller

BSBR
Bull System Backup Restore

BSM
Bull System Manager

C

CGI
Common Gateway Interface

CLI
Command Line Interface

ClusterDB
Cluster Database

CLM
Cluster Management

CMC
Chassis Management Controller

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

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.

CUBLAS
CUDA™ BLAS

CUDA™
Compute Unified Device Architecture

CUFFT
CUDA™ Fast Fourier Transform

CVS
Concurrent Versions System

Cygwin
A Linux-like environment for Windows. Bull cluster management tools use Cygwin to provide SSH support on a Windows system, enabling command mode access.

D

DDN
Data Direct Networks

DDR
Double Data Rate

DHCP
Dynamic Host Configuration Protocol

DLID
Destination Local Identifier

DNS
Domain Name Server:

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

DSO
Dynamic Shared Object

E

EBP
End Bad Packet Delimiter

ECT
Embedded Configuration Tool

EIP
Encapsulated IP

EPM
Errors per Million

EULA
End User License Agreement (Microsoft)

F

FDA
Fibre Disk Array

FFT
Fast Fourier Transform

FFTW
Fastest Fourier Transform in the West

FRU
Field Replaceable Unit

FTP
File Transfer Protocol

G

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

GCC

GNU C Compiler

GDB
Gnu Debugger

GFS
Global File System

GMP
GNU Multiprecision Library

GID
Group ID

GNU

GNU's Not Unix

GPL

General Public License

GPT

GUID Partition Table

Gratuitous ARP

A gratuitous ARP request is an Address Resolution Protocol request packet where the source and destination IP are both set to the IP of the machine issuing the packet and the destination MAC is the broadcast address `xx:xx:xx:xx:xx:xx`. Ordinarily, no reply packet will occur. Gratuitous ARP reply is a reply to which no request has been made.

GSL

GNU Scientific Library

GT/s

Giga transfers per second

GUI

Graphical User Interface

GUID

Globally Unique Identifier

H

HBA

Host Bus Adapter

HCA

Host Channel Adapter

HDD

Hard Disk Drive

HoQ

Head of Queue

HPC

High Performance Computing

Hyper-Threading

A technology that enables multi-threaded software applications to process threads in parallel, within each processor, resulting in increased utilization of processor resources.

I

IB

InfiniBand

IBTA

InfiniBand Trade Association

ICC

Intel C Compiler

IDE

Integrated Device Electronics

IFORT

Intel® Fortran Compiler

IMB

Intel MPI Benchmarks

INCA

Integrated Cluster Architecture:
Bull Blade platform

IOC

Input/Output Board Compact with 6 PCI Slots

IPMI

Intelligent Platform Management Interface

IPO

Interprocedural Optimization

IPoIB

Internet Protocol over InfiniBand

IPR

IP Router

iSM

Storage Manager (FDA storage systems)

ISV

Independent Software Vendor

K

KDC

Key Distribution Centre

KSIS

Utility for Image Building and Deployment

KVM

Keyboard Video Mouse (allows the keyboard, video monitor and mouse to be connected to the node)

L

LAN

Local Area Network

LAPACK

Linear Algebra PACKage

LDAP

Lightweight Directory Access Protocol

LDIF

LDAP Data Interchange Format:

A plain text data interchange format to represent LDAP directory contents and update requests. LDIF conveys directory content as a set of records, one record for each object (or entry). It represents update requests, such as Add, Modify, Delete, and Rename, as a set of records, one record for each update request.

LKCD

Linux Kernel Crash Dump:
A tool used to capture and analyze crash dumps.

LOV

Logical Object Volume

LSF

Load Sharing Facility

LUN

Logical Unit Number

LVM

Logical Volume Manager

LVS

Linux Virtual Server

M

MAC

Media Access Control (a unique identifier address attached to most forms of networking equipment).

MAD

Management Datagram

Managed Switch

A switch with no management interface and/or configuration options.

MDS

MetaData Server

MDT

MetaData Target

MFT

Mellanox Firmware Tools

MIB

Management Information Base

MKL

Maths Kernel Library

MPD

MPI Process Daemons

MPFR

C library for multiple-precision, floating-point computations

MPI

Message Passing Interface

MTBF

Mean Time Between Failures

MTU
Maximum Transmission Unit

N

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

NETCDF
Network Common Data Form

NFS
Network File System

NIC
Network Interface Card

NIS
Network Information Service

NS
NovaScale

NTP
Network Time Protocol

NUMA
Non Uniform Memory Access

NVRAM
Non Volatile Random Access Memory

O

OFA
Open Fabrics Alliance

OFED
Open Fabrics Enterprise Distribution

OpenCL
Open Computing Language

OPMA
Open Platform Management Architecture

OpenSM
Open Subnet Manager

OpenIB
Open InfiniBand

OpenSSH
Open Source implementation of the SSH protocol

OSC
Object Storage Client

OSS
Object Storage Server

OST
Object Storage Target

P

PAM
Platform Administration and Maintenance Software

PAPI
Performance Application Programming Interface

PBLAS
Parallel Basic Linear Algebra Subprograms

PBS
Portable Batch System

PCI
Peripheral Component Interconnect (Intel)

PDSH
Parallel Distributed Shell

PDU
Power Distribution Unit

PETSc
Portable, Extensible Toolkit for Scientific Computation

PGAPACK
Parallel Genetic Algorithm Package

PM

Performance Manager

Platform Management

PMI

Process Management Interface

PMU

Performance Monitoring Unit

pNETCDF

Parallel NetCDF (Network Common Data Form)

PVFS

Parallel Virtual File System

Q

QDR

Quad Data Rate

QoS

Quality of Service:
A set of rules which guarantee a defined level of quality in terms of transmission rates, error rates, and other characteristics for a network.

R

RAID

Redundant Array of Independent Disks

RDMA

Remote Direct Memory Access

ROM

Read Only Memory

RPC

Remote Procedure Call

RPM

RPM Package Manager

RSA

Rivest, Shamir and Adleman, the developers of the RSA public key cryptosystem

S

SA

Subnet Agent

SAFTE

SCSI Accessible Fault Tolerant Enclosures

SAN

Storage Area Network

SCALAPACK

SCALable Linear Algebra PACKage

SCSI

Small Computer System Interface

SCIPORT

Portable implementation of CRAY SCILIB

SDP

Socket Direct Protocol

SDPOIB

Sockets Direct Protocol over Infiniband

SDR

Sensor Data Record

Single Data Rate

SFP

Small Form-factor Pluggable transceiver - extractable optical or electrical transmitter/receiver module.

SEL

System Event Log

SIOH

Server Input/Output Hub

SIS

System Installation Suite

SL
Service Level

SL2VL
Service Level to Virtual Lane

SLURM
Simple Linux Utility for Resource Management – an open source, highly scalable cluster management and job scheduling system.

SM
Subnet Manager

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

SNMP
Simple Network Management Protocol

SOL
Serial Over LAN

SPOF
Single Point of Failure

SSH
Secure Shell

Syslog-ng
System Log New Generation

T

TCL
Tool Command Language

TCP
Transmission Control Protocol

TFTP
Trivial File Transfer Protocol

TGT
Ticket-Granting Ticket

U

UDP
User Datagram Protocol

UID
User ID

ULP
Upper Layer Protocol

USB
Universal Serial Bus

UTC
Coordinated Universal Time

V

VCRC
Variant Cyclic Redundancy Check

VDM
Voltaire Device Manager

VFM
Voltaire Fabric Manager

VGA
Video Graphic Adapter

VL
Virtual Lane

VLAN
Virtual Local Area Network

VNC
Virtual Network Computing:
Used to enable access to Windows systems and Windows applications from the Bull NovaScale cluster management system.

W

WWPN
World–Wide Port Name

X

XFS
eXtended File System

XHPC
Xeon High Performance Computing

XIB
Xeon InfiniBand

XRC
Extended Reliable Connection:
Included in Mellanox ConnectX HCAs for memory scalability

Index

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

REFERENCE
86 A2 22FA 04