Sisu User Guide

Autogenerated from HTML pages with pandoc

09/02/2015

Contents

Sisu User Guide

Revised for Sisu phase 2.
9.9. 2014

1. Introduction

1.1 Sisu supercomputer

1.1.1 User policy

Sisu (sisu.csc.fi) is a Massively Parallel Processor (MPP) supercomputer managed by CSC - IT Center for Science. Sisu, like all the computing services of CSC, aim to enhance science and research in Finland. Thus, usage of Sisu is free of charge for the researchers working in the Finnish universities.

The Sisu supercomputer is intended for well-scaling parallel jobs. The taito.csc.fi supercluster should be used for serial and modestly scaling tasks. The partitions (i.e. the batch job queues) available in Sisu are listed in Table 1.1. In Sisu, the minimum size of parallel jobs is 72 compute cores and the maximum size is at most 19200 computer cores. The number of simultaneously running jobs is limited to 30 per user. Jobs that utilize 72-1008 cores (3-42 nodes) can be used in Sisu without scaling tests, but if you wish to use more than 1008 cores you should first demonstrate efficient usage of the resources with scaling tests. The instructions for scaling tests can be found from:

https://research.csc.fi/sisu-scalability-tests

Even if you use less than 1008 cores for a job you must make sure that you are using the resources efficiently i.e. that your code, with the used input, does scale to the selected number of cores. The rule of thumb is that when you double the number of cores, the job will need to run at least 1.5 times faster. If it doesn't, use less cores. If your task doesn't scale to at least 72 cores, use Taito and run your scaling tests again. Note that scaling depends on the input (model system) as well as the used code. If you are unsure, contact CSC Service Desk.

Researchers that want to use Sisu should fill the **Application form for computing services.** **(This application form is not available as electronic version)**. This form is used to open a computing project to Sisu supercomputer and other servers of CSC. One project typically includes 1-10 user accounts.

A computing project at CSC has a common computing quota that can be extended by application. Use of Sisu or any other server will consume the computing quota granted to the project. The jobs running in Sisu reserve the resources in nodes (i.e in chunks of 24 cores). One node hour consumes 48 billing units from the computing quota of the project.

Table 1.1 Batch job partitions in Sisu

Partition	Minimum number of nodes	Maximum number of nodes	Maximum number of cores	Maximum runni
test	1	24	576	30 minutes
test_large	1	800	19200	4 hours
small	3	24	576	12 hours
small_long	3	24	576	72 hours
large	24	400	9600	72 hours
gc				

1.1.2 Hardware

Sisu (sisu.csc.fi) is a Massively Parallel Processor (MPP) supercomputer produced by Cray Inc., belonging to the XC40 family. It consists of nine high-density water-cooled cabinets for the compute nodes, and one cabinet for login and management nodes. In November 2012 the first phase of Sisu reached the position #118 of the Top500 list of fastest supercomputers in the world, with a theoretical peak of 244.9 teraflop/s (TF). The system was updated in September 2014, with additional new cabinets and new processors for the whole system. After the update theoretical peak performance of the system is 1688 TF. With this performance Sisu is ranked as 37th in the top500 list of most efficient computers in the world (Novermber

2014 release of the Top500 list). More information on the Linpack test and how it is used to rank the fastest supercomputers of the world can be found on the TOP500 website (http://www.top500.org/).

Sisu is composed of 422 compute blades, each of which hosts 4 computing nodes. Therefore, there are 1688 compute nodes (called XC40 Scalar Compute nodes) in total. Each node has 2 sockets. In the whole system there are 40512 cores available for computing, provided by 3376 processors, 12-core Intel (Xeon) *Haswell* (E5-2690v3, 64bits). These processors operate at a clock rate of 2.6GHz. The Haswell processors are well suited for high-performance computing and they comprise several components: twelve cores with individual L1 and L2 caches, an integrated memory controller, three QPI links, and an L3 cache shared within the socket. The processor supports several instructions sets, most notably the *Advanced Vector Extensions 2* (AVX2) instruction set; however, older instructions sets are still supported. Each Haswell core has dedicated 32KB of L1 cache, and 768 KB of L2 cache. The L3 cache is shared among the processors, and its size is 30 MB. Each node has 8 slots of 8GB DDR4 DIMMs, operating at 2133 MHz, for a total of 64GB per compute node. This means that there are 2,67 GB of memory available per core. The compute nodes have no local storage, and run a lightweight linux kernel provided by Cray, called *Compute Node Linux* (CNL).

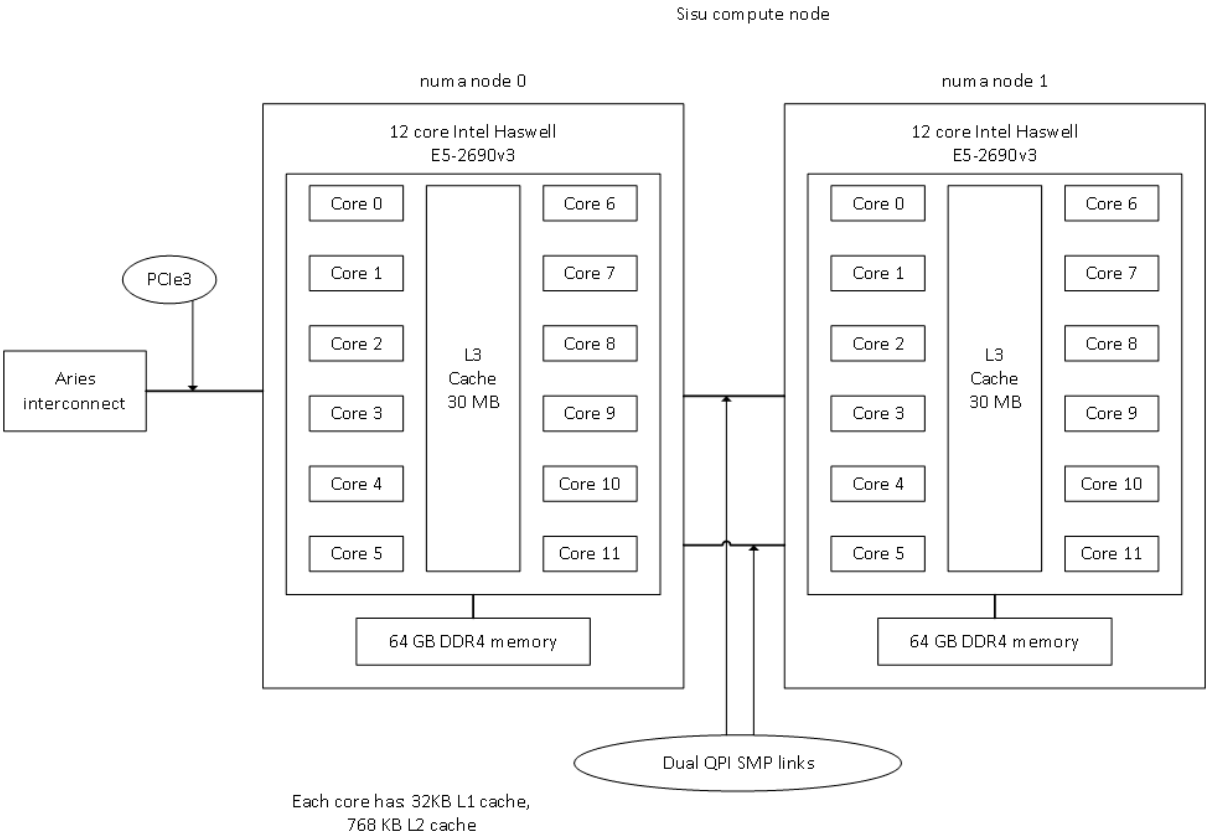

Figure 1. Configuration of a Sisu compute node.

In addition to the compute nodes, Sisu has 6 login nodes, used for logging into the system, submitting jobs, I/O, and service usage. Each login node has 2 *Intel Sandy Bridge* processors, unlike the compute nodes, and 256 GB of memory. The operating system is SUSE Linux Enterprise Server 11, installed on 2 TB of local storage. The system comprises also other servers for managing the supercomputer, the network, and the storage connections.

As said, each compute blade contains 4 compute nodes, but also one *Aries chip* for high-speed networking, connected to the computing nodes with PCIe x16 Gen3 links. Aries is a proprietary interconnect fabric, designed by Cray, using proprietary protocols. The topology of the network is called *dragonfly*, which is an "n-dimensional" torus, where rings in each dimension are replaced with an all-to-all connection among the nodes of that dimension. Dragonfly is considered a direct network topology, as it requires fewer optical links and no external top switches. The heart of the performance for massively parallel runs lies in this Aries interconnection network between the compute nodes.

Table 1.2 Configuration of the Sisu.csc.fi supercomputer. All nodes consist of two 12-core Intel Haswell

2.6 GHz processors. The aggregate performance of the system is 1688 TF.

Node type	Number of nodes	Number of cores/Node	Total number of cores	Memory/node
Login node	6	16	64	256 GB
Computing node	1688	24	40512	64 GB

The following commands can give some useful information from the whole Sisu system or from the current node a user is logged in.

To get a quick overview of all Sisu compute node characteristics use the following command:

```
sinfo -Nel
```

(print information in a compute node oriented format)

```
sinfo -el
```

(print information in a partition/queue oriented format)

For information about the disk systems one can use the following command:

```
df -h
```

Details about the available processors on the current node can be checked with:

```
cat /proc/cpuinfo
```

And details about the current memory usage on the node is shown with:

```
cat /proc/meminfo
```

1.2 Operating system and shell environment

Sisu is a Linux cluster. The login nodes of Sisu are based on the *SUSE Linux Enterprise Server 11.3* (SLES11.3) distribution. The system software set installed on the login nodes offers a wide selection of libraries and development packages to compile your own software.

The computing nodes use a run-time environment provided by Cray, called *Compute Node Linux* (CNL). Compute Node Linux, is a run-time environment based on SLES as well, but it includes only drivers, libraries and compilers required to run high-performance computing applications and a minimal amount of user-level commands (see http://www.busybox.net/). In general, all libraries available on the computing nodes are also available on the login nodes. You can inspect which packages are installed on SLES based servers using the following command:

```
 rpm -qa
```

This command is also useful in finding out what is the version of an installed package. Other options can be given to the ***rpm*** command to inspect the system configuration. Alternatively, ***locate*** and ***find*** are also good tools for inspecting the software configuration of a system. Note that users can't use the *rpm* command to install software to Sisu.

During the lifetime of the cluster, CSC aims to keep the software packages up to date following the minor releases of the operating system, as long as this preserves the necessary compatibility with previous versions. The computing nodes have an identical software configuration. The same applies to the login nodes.

As the system packages will be updated without any previous notification, we suggest to use the *module environmnet* to load specific library versions and software supported by CSC, or to install your own version in $USERAPPL directory. In this way your software dependencies will be safely preserved.

As a general rule, ***x86-64*** binaries should be used for software installed on Sisu. *x86-64* is the 64-bit extension of the x86 instruction set.

The default and recommended command shell in Sisu is **bash**. Previously CSC has been using *tcsh* as the default command shell and you can still continue to use tcsh shell in Sisu too. The **chsh** command can be used to change the default shell.

When a user logs into Sisu, the bash startup script defines a set of CSC specific variables defining the location of the user specific directories: **$WRKDIR, $HOME, $TMPDIR** and **$USERAPPL**. Further, ***rm***, ***cp*** and ***mv*** commands are aliased so that by default they ask for permission before removing or overwriting existing files. Also the *clobber* options are set up so that output forwarding does not overwrite an existing file.

If you wish to add more settings or aliases that are automatically set up when you log in, you should add the corresponding Linux commands to the end of the bash set-up file *.bashrc* which is located in your home directory.

The Sisu system supports the UTF-8 character encoding, which makes it possible to represent every character in the Unicode character set. UTF-8 was not supported on older CSC systems, so care should be taken when sharing files with other systems.

User specific software should be compiled in the $TMPDIR, which resides on the login nodes, instead of $WRKDIR or $HOME which reside on the Lustre file system. Compilation in $TMPDIR is much faster as it runs on local login node's disk instead of Lustre.

1.3 Connecting to Sisu

To connect Sisu, use terminal programs like *ssh* (linux, MacOSX) or *PuTTY* (Windows), which provide secure connection to the server. If you are not able to use a suitable terminal program in your local computer you can use the *SSH console tool* in *Scientist's User Interface* (more details inCSC Computing Environment User's Guide chapter 1.3).

For example, when using the ssh command the connection can be opened in the following way:

```
ssh sisu.csc.fi -l username
```

The Sisu supercomputer has six login nodes named sisu-login1.csc.fi - sisu-login6.csc.fi. When you open a new terminal connection using the server name *sisu.csc.fi* you will end up to one of these login nodes. You can also open the connection to a specific login node if needed:

```
ssh sisu-login5.csc.fi -l username
```

1.4 Monitoring the load in Sisu

The load of Sisu can be monitored with command line tools or Scientists' User Interface.

In Scientists' User Interface the **Host Monitor** (Palvelinnäyttö) tool shows current CPU load, CPU load history and job count history of Sisu supercomputer and other CSC's servers. The basic version of the host monitor is open to everyone. After signing in to the Scientist's User Interface, you can use full version of the monitor with additional features such as details of all running batch jobs:

- Host Monitor (CSC account required, full access)
- Host Monitor (no authentication, limited functionality)

In Sisu, you can check the current load of the system or with commands:

```
squeue
```

and

```
sinfo
```

These commands show the jobs that are currently running of waiting in the batch job system.

1.5 Disk environment

Supercomputer sisu.csc.fi and supercluster taito.csc.fi have a common disk environment and directory structure that allows researchers to analyze and manage large datasets. A default CSC user account allows working with datasets that contain up to five terabytes of data. In Sisu you can store data to the personal disk areas listed in table 1.2 and figure 1.2. Knowing the basic features of different disk areas is essential if you wish to use the CSC computing and storage services effectively.

In Sisu, all directories use the same Lustre-based file server. Thus all directories are visible to both the front-end nodes and the computing nodes of Sisu. In addition to the local directories in Sisu, users have access to the *HPC archive* server, which is intended for long term data storage (see CSC Computing environment user's guide, Chapter 3.2).

Table 1.2 Standard user directories at CSC.

Directory or storage area	Intended use
$HOME	Initialization scripts, source codes, small data files. Not for running programs or research data.
$USERAPPL	Users' own application software.
$WRKDIR	Temporary data storage.
$TMPDIR	Temporary users' files.
project	Common storage for project members. A project can consist of one or more user accounts.
HPC Archive*	Long term storage.

*The Archive server is used through iRODS commands, and it is not mounted to Sisu as a directory.

The directories listed in the table above can be accessed by normal linux commands, excluding the HPC archive server, which is used through the *iRODS* software. The $HOME and $WRKDIR directories can also be accessed through the *MyFiles* tool of the Scientist's User Interface WWW service.

When you are working on command line, you can utilize automatically defined environment variables that contain the directory paths to different disk areas (excluding project disk for which there is no environment variable). So, if you would like to move to your work directory you could do that by writing:

```
cd $WRKDIR
```

Similarly, copying a file *data.txt* to your work directory could be done with command:

```
cp data.txt $WRKDIR/
```

In the following chapters you can find more detailed introductions to the usage and features of different user specific disk areas.

1.5.1 Home directory

When you log in to CSC your current directory will first be your home directory. Home directory should be used for initialization and configuration files and other frequently accessed small programs and files. The size of the home directory is rather limited, by default it is only 50 GB, since this directory is not intended for large datasets.

The files stored in the home directory will be preserved as long as the corresponding user account is valid. Home directory is also backed up regularly so that the data can be recovered in the case of disk failures. Sisu and Taito servers share the same home directory. Thus if you modify settings files like *.bashrc*, the modifications will affect in both servers.

Inside linux commands, the home directory can be indicated by the *tilde* character (~) or by using the environment variable, *$HOME*. Also the command ***cd*** without any argument will return the user to his/her home directory.

1.5.2 Work directory

The *work directory* is a place where you can store large datasets that are actively used. By default, you can temporarily store up to 5 terabytes of data in it. This user-specific directory is indicated by the environment variable, ***$WRKDIR***. The exact path of the work directory is **/wrk/*username***. The Sisu and Taito servers share the same $WRKDIR directory.

The $WRKDIR is NOT intended for long term data storage. Even though un-used files are not automatically removed at the moment, it is possible that in the future CSC will start using automatic cleaning procedures in this disk area. We will inform Sisu and Taito users, in case automatic cleaning procedures are taken into use.

Further, backup copies are not taken of the contents of the work directory. Thus, if some files are accidentally removed by the user or lost due to physical breaking of the disk, your data is irreversibly lost.

1.5.3 Software installation directory

Users of CSC servers are free to install their own application software on CSC's computing servers. The software may be developed locally or downloaded from the internet. The main limitation for the software installation is that the user must be able to do the installation without using the *root* user account. Further, the software must be installed on user's own private disk areas instead of the common application directories like */usr/bin*.

The *user application directory* ***$USERAPPL*** is intended for installing user's own software. This directory is visible also to the computing nodes of the server, so software installed there can be used in batch jobs.

Sisu and Taito servers have separate $USERAPPL directories. This is reasonable: if you wish to use the same software in both machines normally you need to compile separate versions of the software in each machine. The $USERAPPL directories reside in your home directory and are called: *appl_sisu* and *appl_taito*. These directories are actually visible to both Sisu and Taito servers. However, in Sisu the $USERAPPL variable points to $HOME/appl_sisu, and in Taito to $HOME/appl_taito.

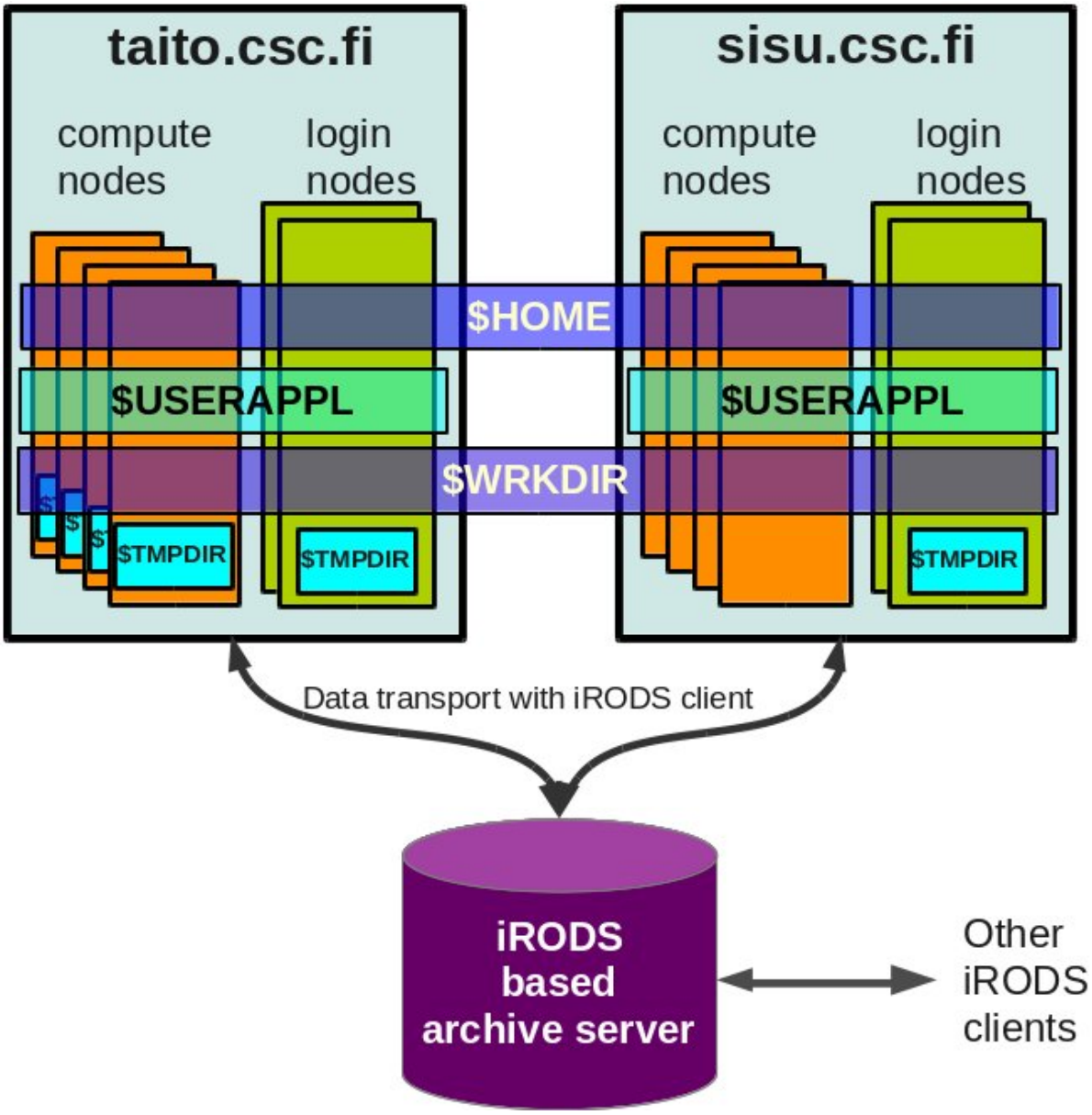

Figure 1.2 Storage environment in Sisu and Taito computers.

More information regarding *user application directory* ***$USERAPPL*** can be found in the CSC Computing environment user guide, especially in chapter 3.1.5, which includes practical examples of installing own software in ***$USERAPPL.***

CSC can help you with your own installation of the software/tool, please don't hesitate to contact the Service Desk.

1.5.4 Monitoring disk usage

The amount of data that can be stored to different disk areas is limited either by user specific quotas or by the amount of available free disk space. You can check your disk usage and quotas with the command:

```
quota
```

The *quota* command shows also your disk quotas on different areas. If the disk quota is exceeded, you cannot add more data to the directory. In some directories, the quota can be slightly exceeded temporarily, but after a so-called grace period, the disk usage must be returned to the accepted level.

When a disk area fills up, you should remove unnecessary files, compress existing files and/or move them to the HPC archive server. If you have well-justified reasons to use more disk space than what your quotas allow, you should send a request to the CSC resource manager (resource_at_csc.fi).

When some of your directories is approaching the quota limits, it is reasonable to check which files of folders require most space. To list the files in your current directory ordered by size, give command:

```
ls -lSrh
```

Note however, that this command does not tell how much disk space the files in the subdirectories use. Thus it is often more useful to use the command **du** (disk usage) instead. You can, for example, try the command:

```
du -sh ./*
```

This command returns the size of each file or the total disk usage of each subdirectory in your current directory. You can also combine *du* with ***sort*** to see what file or directory is the largest item in your current directory:

```
du -s ./* | sort -n
```

Note that as the *du* command checks all the files in your current directory and running the command may in some cases take several minutes.

2. Module environment

Environment modules provide a convenient way to dynamically change the user's environment so that different compiler suites and application versions can be used more easily. Modules system modifies the environment variables of the user's shell so that the correct versions of executables are in the path and the linker can find the correct versions of the libraries needed. Environment modules system can also define software specific environment variables if needed.

In Sisu, the program execution and development environment is managed by using the *environment modules* package developed by the Environment Modules Project (http://modules.sourceforge.net/). Note: Taito uses a different environment modules system called *Lmod*.

The basic syntax of the module commands is:

```
module sub-command module-file-name
```

The most commonly used module commands are listed in Table 2.1. Loading and unloading modules files is done with the module sub-commands ***load*** and ***unload***. For example, to use *Gromacs* software you should first execute the command:

```
module load gromacs
```

All available modules are listed with the command:

```
module avail
```

Currently loaded modules are listed with the command:

```
module list
```

The environment changes induced by a particular module as well as full path of modulefile(s) are shown with the command:

```
module show PrgEnv-cray
```

The example above displays the information about the default version of Cray programming environment.

Changing from the Cray programming environment to GNU programming environment, for example, is accomplished with the command:

```
module swap PrgEnv-cray PrgEnv-gnu
```

Table 2.1 Most frequently used Module sub-commands

Module command	Description
module load *module_name*	Load module into the shell environment.
module unload *module_name*	Remove module from the shell environment.
module show *module_name*	Display information about a modulefile.
module avail	List all available modulefiles.
module list	List loaded modules.
module purge	Unload all loaded modulefiles.
module help *module_name*	Prints information about a module or, if no argument is given, form the module sub-commands.
module swap *module1 module2*	Switch loaded modulef1 with module2.

3.Using the batch job environment

At CSC, batch job systems are used to execute computing tasks in clusters and supercomputers. The Sisu supercomputer uses **SLURM** (Simple Linux Utility for Resource Management System) batch job system, which is used in *Taito* cluster too. However, the SLURM setup differs significantly between Sisu and Taito. In Sisu, the SLURM definitions are used just to reserve computing resources. The parallel commands are launched using **ALPS** (Application Level Placement Scheduler) and the ALPS command ***aprun*** is used instead of the *srun* command which is normally used in SLURM batch jobs. The *aprun* command is also used to define many resource allocation details that in other severs are defined using

the SLURM commands.

Table 3.1 Batch job partitions in Sisu

Partition	Minimum number of nodes	Maximum number of nodes	Maximum number of cores	Maximum runni
test	1	24	576	30 minutes
test_large	1	800	19200	4 hours
small	3	24	576	12 hours
small_long	3	24	576	72 hours
large	24	400	9600	72 hours
gc				

3.1 Using SLURM commands to execute batch jobs

The basic SLURM commands for submitting batch jobs are ***sbatch***, which submits jobs to batch job system, and ***scancel***, which can used to stop and remove a queueing or running job. The basic syntax of the *sbatch* command is:

```
sbatch -options batch_job_file
```

Normally *sbatch* options are included in the batch job file, but you can use the options listed in table 3.3 also in the command line. For example:

```
sbatch -J test2 -t 00:05:00 batch_job_file.sh
```

If the same option is used both in the command line and in the batch job file the value defined in the command line overrides the value in the batch job file. When the job is successfully launched the command prints out a line, telling the ID number of the submitted job. For example:

```
Submitted batch job 6594
```

The job ID number can be used to monitor and control the job. For example, the job with ID 6594 could be cancelled with command:

```
scancel 6594
```

The progress of the submitted batch jobs can be followed with commands ***squeue***, ***apstat***, ***sinfo***, and ***sacct***. These commands can also be used to check the status and parameters of the SLURM environment. By default the *squeue* command lists all the jobs which are submitted to the scheduler. If you want to see the status of your own jobs, you can use the command:

```
squeue -l -u username
```

or

```
squeue -l -u $USER
```

You can also check the status of a specific job by defining the *jobid* with ***-j*** switch. The option *-p partition* will display only jobs on a specific SLURM partition. The partitions of the system can be checked with the command ***sinfo***, which shows information about SLURM nodes and partitions. *sinfo* shows, for example, which nodes are allocated and which are free:

```
[kkmattil@sisu-login5:~/.globus> sinfo -all
Tue Sep  9 14:58:49 2014
PARTITION AVAIL JOB_SIZE  TIMELIMIT   CPUS  S:C:T   NODES STATE      NODELIST
small     up    3-24       12:00:00     48 2:12:2       4 maint      nid0[1376-1379]
small     up    3-24       12:00:00     48 2:12:2       8 idle*      nid00[036-039,516-519]
small     up    3-24       12:00:00     48 2:12:2       1 down*      nid00933
small     up    3-24       12:00:00     48 2:12:2    1594 allocated  nid0[0016-0035,0040-0169,0200-
small     up    3-24       12:00:00     48 2:12:2      64 idle       nid00[170-190,211-253]
large     up    24-400   3-00:00:00     48 2:12:2       4 maint      nid0[1376-1379]
large     up    24-400   3-00:00:00     48 2:12:2       8 idle*      nid00[036-039,516-519]
large     up    24-400   3-00:00:00     48 2:12:2       1 down*      nid00933
large     up    24-400   3-00:00:00     48 2:12:2    1594 allocated  nid0[0016-0035,0040-0169,0200-
large     up    24-400   3-00:00:00     48 2:12:2      64 idle       nid00[170-190,211-253]
test_larg up    1-800       4:00:00     48 2:12:2       4 maint      nid0[1376-1379]
test_larg up    1-800       4:00:00     48 2:12:2       8 idle*      nid00[036-039,516-519]
test_larg up    1-800       4:00:00     48 2:12:2       1 down*      nid00933
test_larg up    1-800       4:00:00     48 2:12:2    1594 allocated  nid0[0016-0035,0040-0169,0200-
test_larg up    1-800       4:00:00     48 2:12:2      64 idle       nid00[170-190,211-253]
gc        up    24-800   1-00:00:00     48 2:12:2       4 maint      nid0[1376-1379]
gc        up    24-800   1-00:00:00     48 2:12:2       8 idle*      nid00[036-039,516-519]
gc        up    24-800   1-00:00:00     48 2:12:2       1 down*      nid00933
gc        up    24-800   1-00:00:00     48 2:12:2    1594 allocated  nid0[0016-0035,0040-0169,0200-
gc        up    24-800   1-00:00:00     48 2:12:2      64 idle       nid00[170-190,211-253]
test*     up    1-24           30:00     48 2:12:2       4 maint      nid0[1376-1379]
test*     up    1-24           30:00     48 2:12:2       8 idle*      nid00[036-039,516-519]
test*     up    1-24           30:00     48 2:12:2       1 down*      nid00933
test*     up    1-24           30:00     48 2:12:2    1594 allocated  nid0[0016-0035,0040-0169,0200-
test*     up    1-24           30:00     48 2:12:2      72 idle       nid0[0170-0190,0211-0253,1719-
```

The command **scontrol** allows to view SLURM configuration and state. To check when a job, waiting in the queue, is estimated to be executed, the command **scontrol show job** *jobid* can be used. A row "*StartTime=...*" gives an estimate on the job start-up time. It may happen that the job execution time can not be approximated, in which case the values is "*StartTime= Unknown*". The "*StartTime*" may change, i.e. be shortened, as the time goes.

Table 3.2 Most frequently used SLURM commands.

Command	Description
sbatch	Submit a job script to a queue.
scancel	Signal jobs or job steps that are under the control of SLURM (cancel jobs or job steps).
sinfo	View information about SLURM nodes and partitions.
squeue	View information about jobs located in the SLURM scheduling queue.
smap	Graphically view information about SLURM jobs, partitions, and set configurations parameters.
scontrol	View SLURM configuration and state.

3.2 Constructing a batch job file

The most common way to use the SLURM batch job system is to first create a *batch job file* which is submitted to the scheduler with the command *sbatch*. You can create batch job files with normal

text editors or you can use the *Batch Job Script Wizard* tool in the *Scientist's User Interface* web portal (https://sui.csc.fi/group/sui/batch-job-script-wizard). You can also submit your job scripts in the Scientist's User Interface by using its *My Files* tool (https://sui.csc.fi/group/sui/my-files) - select the batch job file, right-click on it and choose *Submit Batch Job*.

In the case of Sisu, many job-specific parameters are defined only with the ***aprun*** job submission command and not as a SLURM batch job definitions. At least you need to define only two SLURM parameters:

- *Partition* to be used (*test*, *test_large*, *small, small_long, large* or *gc*)
- The *amount of computing resources* (i.e. nodes). Normally it is useful also to define the computing time. The amount of computing resources to be used can be determined in two alternative ways:

1. You can reserve certain number of *nodes* (each having 24 computing cores) with the SLURM option ***-N***.

2. Alternatively you can define the *total number of cores* to be used with the option ***-n*** and then the distribution of the cores with the option: ***–ntasks-per-nodes***.

We recommend that you use full computing nodes for running jobs if possible. In this case reserving the resources by defining the number of nodes with *-N* is often more convenient.

The minimum size of a parallel job is 3 nodes (72 cores). By default the maximum size of a job is 42 nodes (1008 cores). If a user wishes to submit larger jobs (up to 400 nodes), the parallel performance of the software needs to be demonstarted first wih a scalability test.

- Sisu scalability test instructions

Below is shown an example of a SLURM batch job for Sisu:

```
#!/bin/bash -l
#SBATCH -J test_job
#SBATCH -o test_job%J.out
#SBATCH -e test_job%J.err
#SBATCH -t 05:30:00
#SBATCH -N 8
#SBATCH -p small

(( ncores = SLURM_NNODES * 24 ))
echo "Running namd with $SLURM_NNODES nodes containing total of $ncores cores"
module load namd
aprun -n $ncores namd2 namd.run
```

The first line of the batch job file (*#!/bin/bash -l*) defines that the *bash* shell will be used. The flag *-l* makes bash act as if it had been invoked as a login shell, and allows, e.g., to call the module command within the script, if needed. The following six lines contain information for the batch job scheduler. The syntax of the lines is

```
#SBATCH -sbatch_option argument
```

In the example above we use six sbatch options:

-J that defines a name for the batch job (test_job in this case)
-o defines file name for the standard output and
-e for the standard error
-t defines the maximum duration of the job, in this case 5 hours and 30 minutes

-N defines that the job will use 8 nodes (containing total of 8 x 24 = 192 computing cores)
-p defines the partition (queue) the job will be sent to, i.e., *small*

In the output and error file definitions notation *%J* is used to use the job id-number in the file name, so that if the same batch job file is used several times the old output and error files will not get overwritten.

After the batch job definitions, one inserts the commands that will be executed. In the example above, the script calculates the number of cores to be used (*$ncores*) so that changes in the number of nodes is automatically taken into account with the *aprun* command. Finally, command: *module load namd* sets up the *namd* molecular dynamics application and the *aprun* command launches the actual *namd* job. The job can be submitted to the batch job system with the command:

```
sbatch file_name.sh
```

The batch job file above includes only the most essential job definitions. However, it is often mandatory or useful to use several other *sbatch* options too. The options needed to run parallel jobs are discussed more in detail in the following chapters. Table 3.3 contains some of the most commonly used *sbatch* options. The full list of *sbatch* options can be listed with command:

```
sbatch -h
```

Table 3.3 Commonly used *sbatch* options applicable in Sisu supercomputer.

Slurm option	Description
–begin=time	Defer job until HH:MM MM/DD/YY
-d, –dependency=type:jobid	Defer job until condition on jobid is satisfied
-e, –error=err	File for batch script's standard error
-J, –job-name=jobname	Name of job.
–mail-type=type	Notify on state change: BEGIN, END, FAIL or ALL.
–mail-user=user	Who to send email notification for job state changes.
-N, –nodes=N	Number of nodes on which to run.
-o, –output=out	File for batch script's standard output.
-t, –time=minutes	Time limit in format hh:mm:ss.

List of batch job examples in this guide:

- An example of a SLURM batch job for Sisu.
- An exemplary script for running a parallel job.
- Hybrid parallel job example 1.
- A basic MPI batch job example.
- A memory intensive MPI batch job example.
- Hybrid MPI and OpenMP parallelization example.
- Hybrid MPI and OpenMP parallelization example for Intel compiler.
- Hyper-threading MPI batch job example 1

3.3 Using aprun to execute parallel processes

The command ***aprun*** is a Cray Linux Environment utility which launches the executable on compute nodes. This command is analogous to the SLURM command *srun*, which should not be used in Sisu. The following table lists the most important options for *aprun*. For a complete description of options see the manual page aprun(1).

```
man aprun
```

Table 3.4 Most important *aprun* options.

aprun option	Description
-n *PEs*	The number of processing elements (PEs, in Cray terminology), often same as number of cores needed by an application. In Sisu it defines the **number of MPI tasks**. The default is 1.
-N *PEs_per_node*	The number of PEs per node, at Sisu: **number of MPI tasks per compute node**. Not to be confused with *-N* option of *sbatch*, which has completely different meaning.
-m *size*	Specifies the memory per PE required, at Sisu: **memory required per MPI task**. Resident Set Size memory size in megabytes. K, M, and G suffixes are supported (16M = 16 megabytes, for example). Any truncated or full spelling of unlimited is recognized.
-d *depth*	The number of threads per PE, at Sisu: **number of OpenMP threads per MPI task**. The default is 1.
-j num_cpus	Specifies how many CPUs to use per compute unit for an ALPS job, at Sisu: **number of logical CPU cores per a physical CPU core**. In sisu, at most 2 logical cores are available per a single physical core. If the user does not want to use logical CPU cores, a setting of **-j 1** is recommended.
-L *node_list*	The *node_list* specifies the candidate nodes to constrain application placement. The syntax allows a comma-separated list of node IDs (node,node,...), a range of nodes (node_x- node_y), and a combination of both formats
-e *ENV_VAR=value*	Set an environment variable on the compute nodes, must use format *VARNAME=value*. To set multiple environment variables use multiple *-e* arguments.
-S *PEs_per_NUMA_node*	The number of PEs per NUMA node, at Sisu: **number of MPI tasks per NUMA node**. Each compute node in Sisu has two sockets and one socket has one 12-core processor. **Each socket is a NUMA node** that contains a 12-core processor and its local NUMA node memory.
-ss	Request strict memory containment per NUMA node, i.e. to allocate memory only from local NUMA node.
-cc *CPU_binding*	Controls how tasks are bound to cores and NUMA nodes. **-cc none** means that MPI affinity is not needed. **-cc numa_node** constrains MPI tasks and OpenMP threads to local NUMA node. Default is **-cc cpu** that is the typical use case for an MPI job (it binds each MPI task to a core).

An exemplary script, *jobtest.sh*, for running a parallel job:

```
#!/bin/bash -l
#SBATCH -N 6
#SBATCH -J cputest
#SBATCH -p small
#SBATCH -o /wrk/username/%J.out
#SBATCH -e /wrk/username/%J.err
aprun -n 144 /wrk/username/my_program
```

Submitting jobtest.sh:

```
sbatch jobtest.sh
```

More information:

- aprun manual page

3.4 Hybrid parallel jobs

In some cases it is useful to use both MPI and threads-based parallelisation in the same task. ***cp2k*** is an example of a software that can use this kind of hybrid parallelization. Below is an example of a batch job file that can be used to launch a *cp2k* job. Note, that for *cp2k* it's usually best not to use hybrid parallelization.

Hybrid parallel job example 1.

```
!/bin/bash
#SBATCH -t 00:10:00
#SBATCH -J CP2K
#SBATCH -o ocp2k.%j
#SBATCH -e ecp2k.%j
#SBATCH -p test
#SBATCH -N 4
#SBATCH --no-requeue
# here we just ask SLURM for N full nodes, and tell below (to ALPS/aprun) how to use them

module load cp2k

export OMP_NUM_THREADS=12

aprun -n 8 -d 12 -N 2 -S 1 -ss -cc numa_node cp2k.psmp H2O-32.inp > H2O-32_n8d8.out

# -n 8 -> 8 mpi tasks
# -d 12 -> 12 threads per mpi task
#  (n*d must be equal total number of CORES requested. 8x12 = 4x24 in this case)
# -N 2 -> 2 mpi tasks per node
# -S 1 -> number of mpi tasks per NUMA node
# -ss -> let threads use memory only from their own CPU (numa node)
# -cc numa_node -> threads of one mpitask are assigned to the same physical CPU (numa node)
# you may also try -cc cpu (which is the default)
```

In this example, four nodes are reserved (*-N 4*) for ten minutes. The actual job consist of 8 MPI processes each using twelve threads, respectively. Note that you have to set both OMP_NUM_THREADS environment variable and the aprun *-d* variable. The total number of cores used in job will use is 8*12 = 96 cores, which must be equal to the allocated cores from SLURM (4*24). More information about compiling and runnig hybrid jobs can be found form in Chapter 4.4 Shared memory and hybrid parallelization. If you are uncertain of the correct options, contact CSC Service Desk. Note that for cp2k software used in this example, mpi-only parallelization is often more efficient than the hybrid parallelization above.

3.5 Interactive batch jobs

Interactive batch jobs can be used in Sisu for testing purposes or running graphical interfaces in some special cases. If you need to do that, contact servicedesk@csc.fi. Normally all jobs should be run through the queuing system.

3.6 Using Hyper-threading in batch jobs

In Sisu one can use *Hyper-threading* so that the number of simultaneous processes in a node can exceed the number of physical cores. In this approach, the operating system executes the program through

virtual cores. As each physical core can run two virtual cores, one Sisu node can have 48 virtual cores, when hyper-threading is used.

The efficiency of hyper-threading depends strongly of the program used. You should always first test if using hyper-threading really improves the execution times before launching larger hyper-threading utilizing jobs.

In practice, hyper-threading is taken in use by using *aprun* with option ***-j 2*** (assigning 2 compute units per core). You can also use *aprun* option ***-N 48*** to define that the job should be executed with 48 processing elements per node.

Hyper-threading batch job example 1.

```
!/bin/bash -l
### MPI parallel job script example, HyperTreading mode
#SBATCH -J HT_job
#SBATCH -e my_output_err_%j
#SBATCH -o my_output_%j
#SBATCH -t 00:11:00
#SBATCH -N 4
#SBATCH -p test

## 4x24=96 cores reserved
## run my MPI executable using hyperthreading (2x96=192 compute units)
aprun -j 2 -n 192 -N 48 ./test2
```

4. Compiling environment

4.1 Available Compilers

4.1.1 Compiler versions

In Sisu, all compilers are accessed through the Cray drivers (wrapper scripts) ***ftn***, ***cc*** and ***CC***. **ftn** will launch a Fortran compiler, **cc** will launch a C compiler and **CC** will launch a C++ compiler. No matter which vendor's compiler module is loaded, always use the *ftn*, *cc* and *CC* commands to invoke the compiler.

The Cray driver commands links in the libraries required and they produce code that can be executed on the *compute nodes*. For more information, see ***man ftn***, ***man cc*** or ***man CC***. The compiler arguments (options) to these drivers vary according to which compiler module is loaded. If *PrgEnv-cray* is loaded then Cray compiler options can be given (either on the command line or in your makefiles). These drivers have also some general options that are valid for all vendor compilers. See the above mentioned man pages for more information. If you compile and link in separate steps use the Cray driver commands also in the linking step and not the linker *ld* directly.

Sisu login nodes have the *Intel Sandy Brigde* microarchitecture whereas the compute nodes have the *Intel Haswell* microarchitecture. The compiler commands produce by default executables for the compute nodes, and in some cases the executables do not work in the login nodes. Please refer to section 4.1.3

on cross compilation if you experience problems in running executables after succesfull compilation or when using GNU autotools (*configure*; *make*; *make install*).

Table 4.1 Compiler suites installed on Sisu:

Compiler suite	Programming environment module	Man pages	User Guides
Cray compilers	PrgEnv-cray	man crayftn (Fortran) man craycc (C) man crayCC (C++)	Fortran reference C and C++ refence
Gnu Compiler Collection	PrgEnv-gnu	man gfortran (Fortran) man gcc (C/C++)	Gfortran manual Gcc manual
Intel Composer	PrgEnv-intel	man ifort (Fortran) man icc (C/C++)	Fortran User and reference C/C++ User and reference

Command line command ***module avail*** will show the compiler suite (module) currently loaded. By swapping the Programming environment module one can change from a compiler suite to another suite. For example:

```
module swap PrgEnv-cray PrgEnv-gnu
```

will change from the Cray compiler to the Gnu compiler collection.

Table 4.2 Cray driver commands.

Language (standard)	Compilercommand for Cray, Gnu and Intel compilers	MPI parallel compiler command	How to check the version of a loaded compller
Fortran 77, 95, 2003	ftn	ftn	ftn -V (for Cray and Intel) ftn –version (for Gnu)
C	cc	cc	cc -V (for Cray and Intel) cc –version (for Gnu)
C++	CC	CC	CC -V (for Cray and Intel) CC –version (for Gnu)

Any selected compiler suite might have several versions installed. In such a case it is possible to swap the version inside a selected programming environment. The Cray compiler versions have a modulefile naming style *cce/version_number*, the Gnu compiler collection versions *gcc/version_number* and the Intel compilers *intel/version_number*.

For example, swap from a Cray version to another Cray version is done in this way (PrgEnv-cray is the loaded Programming environment):

```
module swap cce/version_number cce/another_version_number
```

The command ***module avail cce*** will show all available Cray compiler versions (***module avail gcc*** will show all Gnu compiler versions and ***module avail intel*** will show all Intel versions).

When compiling a code, it is a good idea to write down all currently loaded modulefiles, i.e., give the command: ***module list*** and save the output in a file. This way you can check, what was the environment where your code worked well, if later it will not work properly (for example, after operating system upgrades, or the default version changes of libraries or compilers). If this happens you might load exactly the same modulefiles that you have saved (and see that updates might have not so good effects for your runs).

Table 4.3 Compilation examples

Command	Result
ftn f_prog.f95	Create a static executable: a.out
ftn -o my_fortran_prog f_routine_1.f95 f_routine_2.f95	Compile Fortran files and link them into a static executable: my_fortran_prog
CC -o my_c++_prog CC_routine.C	Will create a static executable: my_c++_prog
cc -c c_routine.c	Compile C file, create an object file: c_routine.o
ftn -o my_mixed_c_and fortran_prog f_routine_1.f95 c_routine.o	Compile Fortran file and link it with a C object file, create a static executable: my_mixed_c_and_fortran_prog
ftn -c f_routine_2.f95	Compile a Fortran file, create an object file: f_routine_2.o
ftn -o another_mixed_c_and fortran_prog c_routine.o f_routine2.o	Link the object files into a static executable: another_mixed_c_and fortran_prog

By default Cray drivers will build static executables. Dynamic builds can be made by the option *-dynamic*.

```
ftn -dynamic -o another_mixed_c_and_fortran_prog c_routine.o f_routine2.o
```

or by defining the environment variable CRAYPE_PE_LINK to have value *dynamic*.

```
export CRAYPE_LINK_TYPE=dynamic
ftn -o another_mixed_c_and_fortran_prog c_routine.o f_routine2.o
```

The command *ldd* will show the libraries that will be loaded dynamically

```
ldd another_mixed_c_and_fortran_prog
```

4.1.2 Selecting compiler options

The options available for Cray, Intel, and Gnu compilers can be found on man pages when the corresponding programming environment is loaded, or in the compiler manuals on the Web (see links above on this page).

In the table below some optimization flags are listed for the installed compilers. It is best to start from the safe level and then move up to intermediate or even aggressive, while making sure that the results are correct and the program has a better performance.

Table 4.4 Compiler options for Sisu

	Cray	GNU	Intel
Safe	-O1	-O2	-O2 -fp-model precise -fp-model source (Use all three options. One can also use the options -fp-model precise -fp-model source with intermediate and aggressive flags to improve the consistency and reproducibility of floating-point results)
Intermediate	O2 (this is the default)	-O3	-O2
Aggressive	-O3 (or) -O3,fp3	-O3 -ffast-math -funroll-loops -march=haswell -mtune=haswell -mavx2	-O3 -opt-prefetch -unroll-aggressive -no-prec-div -fp-model fast=2
Fused multiply-add (FMA)	-hfp1 (disabled) -hfp2 (default) -hfp3, fast	-mno-fma (default flag) -mfma (enabled)	-no-fma -fma, (enabled, default flag)

Sisu login nodes have the Intel Sandy Brigde microarchitecture whereas the compute nodes have the Intel Haswell microarchitecture. The compiler options for the Haswell architecture are enabled automatically (modulefile *craype-haswell* is loaded by default).

Haswell microarchitecture has support for fused multiply-add instructions (FMA). Whether using FMA instructios produce any speedup or not, is application and workload dependent.

4.1.3 Cross-compilation

The Haswell instruction set available on Sisu compute nodes is not fully backwards compatible with the Sandy Bridge instruction set available on the compute nodes. It is possible that binaries built to solely use the Haswell instruction set are not executable on the login nodes.

For instance, using an Intel compiler with the -xCORE-AVX2 compilation flag, an attempt to execute the compiled binary on a Sisu login node yields the following error message:

```
Please verify that both the operating system and the processor support Intel(R) MOVBE, F16C, FMA, B
```

Some build systems, such as GNU *Automake* and *CMake*, separate the configuration step in the compilation process. In the configuration step, a series of small tests is automatically performed on the system by compiling and running small test binaries in order to determine the correct build environment. Therefore, if Haswell instruction set is to be used, such build tools need to be invoked in a cross-compilation mode to avoid incorrect failures in the tests.

GNU Automake: Invoke configure on cross-compilation mode by using the *–host* specifier. For instance

```
./configure --host=x86_64-unknown-linux-gnu
```

For more detailed instructions, see GNU Automake documentation on cross-compilation. Another alternative is to switch to the module *craype-sandybridge* before running ./configure and then switch back to using the module *craype-haswell* before the actual compilation step.

CMake: Either use a Toolchain file or the command line to set CMAKE_SYSTEM_NAME and the compilers to be used in the compilation. For instance

```
cmake /path/to/src -DCMAKE_SYSTEM_NAME=xc30 -DCMAKE_C_COMPILER=cc
```

For more detailed instructions, see CMake documentation on cross-compilation.

4.1.4 Some Cray compiler options

The Cray compiler enables automatically the recognition of OpenMP directives and no specific compiler option is needed. Position independent code is generated by the options **-h pic** or **-h PIC**. To see listings showing which optimizations were performed where in the code, try **-r d** (Fortran compiler) **-h list=m** (C/C++ compiler). To see listings showing which potential optimizations were not performed, try **-h negmsgs**. These might be useful when trying to make changes to your source code. By default, no runtime checks are performed. It is possible to enable various runtime checks. For Fortran this is done with option **-R**, that has five sub-options (see more information on the man page). The C compiler has an option **-h bounds** that provides checking of pointer and array references at runtime. These checks might be useful for debugging purposes.

It is possible to enhance the Cray C/C++ compiler compatibility with the Gnu C/C++ Extensions by giving the option **-h gnu**.

By giving the option **-default64** to the *ftn* Fortran driver it passes **-i8** and **-r8** options to the Cray compiler. This option also links in the appropriate 64-bit MPI or SHMEM library. So with the Cray compiler MPI and SHMEM libraries have support for 32bit and 64bit integers. This functionality is not available for the Gnu or Intel compiler suites. When either Gnu or Intel suite is loaded only 32bit integers are accepted by MPI or SHMEM routines.

When the Cray compiler gives messages like error messages, use command **explain *message-ID*** to display explanations of the message. For example for Fortran compiler message ID: *ftn-100* :

```
explain ftn-100
```

And for C/C++ compiler, the message ID: *CC-17*:

```
explain CC-175
```

4.2 Mathematical libraries

Very short notes for mathematical libraries available in Sisu:

4.2.1 Cray LibSci

module:**cray-libsci** (module is loaded by default)

Available for *PrgEnv-cray*, *PrgEnv-gnu* and *PrgEnv-intel* (module is loaded by default).

Man page: **man intro_libsci**

The general components of Cray LibSci are:

- BLAS
- BLACS
- LAPACK
- ScaLAPACK
- FFT
- FFTW
- CRAFFT
- CASE
- IRT

Compiling and linking

statically linked executable:

```
ftn -o my_exe my_code.f95
```

dynamically linked executable:

```
ftn -o my_exe my_code.f95 -dynamic
```

When you run an MPI/OpenMP (threaded) application there are environment variables for threading control. Please read carefully chapter 4.4 Shared memory and hybrid parallelization (PrgEnv-intel examples will show how to use Intel specific thread control environment variables).

4.2.2 fftw3

module: **module load fftw/3.3.4.2**

Available for *PrgEnv-cray*, *PrgEnv-gnu* and *PrgEnv-intel*

Man page: **man intro_fftw3**

Compiling and linking

Statically linked executable:

```
ftn -o my_exe fftw3_code.f90
cc -o my_code fftw3_code.c
```

dynamically linked executable:

```
ftn -o my_exe fftw3_code.f90 -dynamic
cc -o my_code fftw3_code.c -dynamic
```

4.2.3 fftw2

module: **module load fftw/2.1.5.7**

Available for *PrgEnv-cray*, *PrgEnv-gnu* and *PrgEnv-intel*

Man page: **man intro_fftw2**

Compiling and linking

Statically linked executable:

```
ftn -o my_exe fftw2_code.f90
```

(User must specify which libraries to link against, see man intro_fftw2)

```
cc -o my_code fftw2_code.c
```

(User must specify which libraries to link against, see man intro_fftw2)

Dynamically linked executable:

```
ftn -o my_exe fftw2_code.f90 -dynamic
```

(User must specify which libraries to link against, see man intro_fftw2)

```
cc -o my_code fftw2_code.c  -dynamic
```

(User must specify which libraries to link against, see man intro_fftw2)

4.2.4 PETSc

module (for real data): **module load cray-petsc** (loads also *cray-tpsl*, that has libraries: MUMPS, SuperLU, SuperLU_dist, ParMETIS, HYPRE, SUNDIALS and Scotch). Please load also module **cray-libsci** if not loaded.

module (for complex data): **module load cray-petsc-complex.** Please load also module **cray-libsci** if not loaded.

Available for *PrgEnv-cray*, *PrgEnv-gnu* and *PrgEnv-intel*

Man page: **man intro_petsc**

Compiling and linking

Statically linked executable:

```
ftn -o my_exe petsc_code.F
cc -o my_exe  petsc_code.c
```

dynamically linked executable:

```
ftn -o my_exe petsc_code.F -dynamic
cc -o my_exe  petsc_code.c -dynamic
```

4.2.5 Trilinos

module: **module load cray-trilinos** (loads also *cray-tpsl*, that has libraries: MUMPS, SuperLU, SuperLU_dist, ParMETIS, HYPRE, SUNDIALS and Scotch). Please load also module **cray-libsci** if not loaded.

Available for *PrgEnv-cray*, *PrgEnv-gnu and PrgEnv-intel*.

man page: **man intro_trilinos**

Compiling and linking

statically linked executable:

```
CC -o my_exe trilinos_code.cpp
```

4.2.6 Intel MKL

Intel MKL library is not fully supported on Sisu (perhaps Cray will provide it later).

There is not any MKL module. Easiest way to use it is to load *PrgEnv-intel* and unload *cray-libsci*

User's Guide and **Reference manual**

Intel MKL is a mathematical library collection that is optimized for Intel processors. On Sisu, MKL can be used mainly with Intel compiler for Fortran and C/C++ programming. Please note that cluster libraries (BLACS, ScaLAPACK and Cluster FFT functions) are not compiled with Cray MPI libraries (but Cray MPI and IntelMPI both are MPICH2 libraries. IntelMPI is not available on Sisu).

MKL includes the following groups of routines:

- BLAS (Basic Linear Algebra Subprograms)
- Sparse BLAS
- LAPACK (Linear Algebra PACKage)
- PBLAS (Parallel Basic Linear Algebra Subprograms)

- BLACS (Basic Linear Algebra Communication Subprograms)
- ScaLAPACK (Scalable LAPACK)
- Sparse Solver routines (direct sparse solver PARDISO, direct sparse solver DSS, iterative sparse solvers RCI, preconditioners for iterative solution process)
- Vector Mathematical Functions (VML, arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding)
- Vector Statistical Functions (VSL, random numbers, convolution and correlation, statistical estimates)
- General Fast Fourier Transform (FFT) Functions
- Cluster FFT functions
- Partial Differential Equations (PDE) support tools (Trigonometric Transform routines, Poisson routines)
- Nonlinear least squares problem solver routines
- Data Fitting functions (spline-based)
- Support Functions (timing, thread control, memory management, error handling, numerical reproducibility)

MKL library has two integer interfaces 32-bit (they call it: LP64) and 64-bit integer (ILP64) interfaces. So if you work with data arrays that have more than 2^31-1 elements ILP64 interface is for you.

MKL supports sequential and threaded programming modes. Intel MKL is based on the OpenMP threading. See:Threaded MKL Functions and Problems for more information.

Compiling and linking

For quick linking Intel compiler supports variants of the *-mkl* compiler option. The compiler links your application using the LP64 interface and it does not use Intel MKL Fortran module 95 interfaces.

-mkl or **-mkl=parallel** to link with threaded MKL
-mkl=sequential to link with sequential MKL
-mkl=cluster to link with cluster libraries that use IntelMPI (MPICH2, there is no guarantee that this cluster option will work on Sisu. BUT it can work very well)

dynamically linked executables (if *cray-libsci is loaded* please unload *cray-libsci* before compiling and linking):

```
cc -o my_exe mkl_code.c -mkl=sequential -dynamic
ftn -o my_exe mkl_code.95 -mkl=sequential -dynamic
cc -o my_exe mkl_code.c -mkl=parallel -dynamic
ftn -o my_exe mkl_code.95 -mkl=parallel -dynamic
```

Statically linked executable (if *cray-libsci is loaded* please unload *cray-libsci* before compiling and linking):

Because MKL applications are not designed to to be fully static you might get compiler/linker warnings with following examples.

```
cc -o my_exe mkl_code.c -mkl=sequential
ftn -o my_exe mkl_code.95 -mkl=sequential
cc -o my_exe mkl_code.c -mkl=parallel
ftn -o my_exe mkl_code.95 -mkl=parallel
```

Note that the -mkl option NEEDS TO APPEAR AT THE END OF THE LINE.
Because MKL includes many libraries and programming interfaces a link line can be a long list. To find a correct line for a case, specify your choices using *Intel Math Kernel Library (MKL) Link Line Advisor* tool:

http://software.intel.com/sites/products/mkl/

Copy the results (link line and compiler option results) into a Makefile. In command line compiling and linking case remove all brackets that the advisor gives (for example if there is a variable (MKLROOT) in brackets then remove the brackets.

When you run an MPI/OpenMP (threaded) application there are environment variables for threading control. Please read carefully chapter 4.4 Shared memory and hybrid parallelization (PrgEnv-intel examples will show how to use Intel specific thread control environment variables)

4.3 Using MPI

4.3.1 Message passing Interface

MPI (Message Passing Interface) is a standard specification for message passing libraries. It allows portable parallel programs in Fortran and C. MPI has become a de facto standard for communication among processes that create a parallel application running on a distributed memory system (like Sisu.csc.fi).

In message passing each process/task has an address space in memory that other processes/tasks cannot directly access. In parallel application these processes/task communicate with each other by message passing. On Sisu (Cray XC40 Supercomputer) message passing implementation is optimized to take advantage of high speed Aries interconnect. All Programming Environments (PrgEnv-cray, PrgEnv-gnu and PrgEnv-intel) can utilize the MPI library that is implemented by Cray. Current release implements the MPI-3.0 standard but dynamic process management is not supported, see more information from manual page mpi_intro (command: man mpi_intro). Manual page section NOTES describes the MPI commands that are not supported and section ENVIRONMENT VARIABLES describes values that mainly control the runtime behaviour of message passing. For better performance it might be sometimes useful to change some values of these variables.

4.3.2 Compiling and linking

All compilers are accessed through the Cray drivers (wrapper scripts) **ftn**, **cc** and **CC**. No matter which vendor's compiler module is loaded, always use **ftn**, **cc** and **CC** commands to invoke the compiler. **ftn** will launch a Fortran compiler, **cc** will launch a c compiler and CC will launch a C++ compiler. If you compile and link in separate steps, use the Cray driver commands also in the linking step and not the linker ld directly. No additional MPI library linking options are required with the Cray wrappers.

For example, MPI programs written in Fortran 90, C and C++ can be built as follows:

```
ftn my_fortran_mpi_code.f95
cc my_c_mpi_code.c
CC my_C_mpi_code.C
```

Compile fortran file:

```
ftn -c f_routine.f95
```

Compile c file:

```
cc -c c_routine.c
```

Link object files into an static executable:

```
ftn -o my_mpi_app c_routine.o f_routine.o
```

Link object files into an dynamically linked executable:

```
ftn -dynamic -o my_mpi_app c_routine.o f_routine.o
```

Chapter *4.1 Compiling Environment* has more more information about compiler flags.

4.3.3 Include files

In Fortran, a source code file containing MPI calls must include an include file. When using Fortran 77, source code should contain the line:

```
include 'mpif.h'
```

With Fortran 90 (or later) it is recommended to use the MPI module:

```
use mpi
```

In C/C++ one should use

```
#include <mpi.h>
```

There are name-space clashes between *stdio.h* and the MPI C++ binding. To avoid this conflict make sure your application includes the *mpi.h* header file before *stdio.h* or *iostream.h*

4.3.4 Running MPI batch job

A basic MPI batch job example.

```
!/bin/bash -l
## The number of compute nodes for a 144 mpi processes job (6*24=144)
#SBATCH --nodes 6
## It is recommended to allocate just the number of nodes.
## Each compute node has 24 cores (See more details in section Hardware on Sisu User Guide).
## Give the number mpi processes and other job launching details on aprun line
## (see the last line of this example)

## Choose a suitable queue <test,small,large>
## How to check queue limits: scontrol show part <queue name>
## for example: scontrol show part small
#SBATCH -p test

## Name of your job
#SBATCH -J jobname

## System message output file
#SBATCH -o jobname_%J.out

## System error message file
#SBATCH -e jobname_%J.err

## How long job takes, wallclock time hh:mm:ss
#SBATCH -t 00:11:00

## Run MPI executable on compute nodes
## option -n gives the number of processes (recommendation: multiplies of 24)
## Above we have allocated 6 compute nodes, so it is possible to run 6*24=144 mpi processes.
## Calculate the total number of cores and store it in variable ncores
(( ncores = SLURM_NNODES * 24 ))
aprun -n $ncores /wrk/$USER/mpi_executable
```

Each compute node has 24 cores and 64 GB memory (per core memory size is 2.67GB). Strong recommendation: Submit jobs where number of the allocated cores is divisible by 24. More information can be found on Chapter Using Batch Job Environment.

When running memory intensive jobs (jobs that need more than 2.67GB memory per MPI task) the application must use less than 24 cores per compute node. On next example the parallel job can have nearly 8 GB per core (per MPI task) by using 8 cores per compute node. Furthermore, one specifies that each socket has 4 MPI tasks (each compute node has two sockets and one socket has one 12-core processor and its local 32 GB memory). A socket is also a NUMA node (so each compute node has two NUMA nodes).

```
!/bin/bash -l
## memory intensive example (actually taito.csc.fi should be better for memory intensive jobs).
## 144 mpi processes will need 18 nodes if we use just 8 cores per node 144/8=18
## the number of compute nodes
#SBATCH --nodes 18
## Give the number mpi processes and other job launching details on aprun line
## (see the last line of this example)

## name of your job
#SBATCH -J jobname
## system message output file
#SBATCH -o jobname_%J.out
## system error message file
#SBATCH -e jobname_%J.err
## how long job takes, wallclock time hh:mm:ss
#SBATCH -t 11:01:00

## Choose a suitable queue <test,small,large>
## How to check queue limits: scontrol show part <queue name>
#SBATCH -p small

## option: -n (total number of mpi processes)
## option: -N (number of mpi processes per compute node)
## option: -S (number of mpi processes per NUMA node)
## option: -ss (allocate memory only from a local NUMA node)
## run the application on compute nodes
(( ncores = SLURM_NNODES * 8 ))
aprun -n $cores -N 8 -S 4 -ss ./mpi_executable
```

Please remember that taito.csc.fi has very good facilities for memory intensive jobs. On Taito each node has at least 64GB memory and it has also 16 compute nodes that has 256GB memory and two nodes that have 1,5 TB of memory. Thus, if an memory intensive application does not benefit from the Aries interconnect then taito.csc.fi is better choice.

4.3.5 Manual pages

More information can be found on the manual pages:

```
man mpi_intro
man sbatch
man aprun
```

4.4 Shared memory and hybrid parallelization

A short introduction to MPI/OpenMP hybrid programming on Sisu.

Each compute node on Sisu contains two 12-core processors (24 cores all together). Hence it is possible to run hybrid parallel (MPI/OpenMP) programs efficiently.

4.4.1 How to compile

All Programming environments (Cray, Gnu and Intel) support OpenMP. Use the following compiler flags to enable OpenMP support.

Table 4.5 Compiler flags to enable OpenMP support

Compiler	Flag
Cray	no flag needed (OpenMP support is on by default)
Gnu	-fopenmp
Intel	-openmp

Here are static compilation examples for OpenMP and mixed (i.e. hybrid) MPI/OpenMPI (first line: Cray compiler, second line: Gnu-compiler, third line: Intel Compiler).

```
ftn -o my_hybrid_exe my_hybrid.f95
ftn -fopenmp -o my_hybrid_exe my_hybrid.f95
ftn -openmp -o my_openmp_exe my_openmp.f95
```

See OpenMP web site for more information including standards and tutorials

4.4.2 Include files

For Fortran 77 use following line

```
include 'omp_lib.h'
```

For Fortran 90 (and later) use

```
use omp_lib
```

For C/C++ use

```
#include <omp.h>
```

4.4.3 Running hybrid MPI/OpenMP programs

In many cases it is beneficial to combine MPI and OpenMP parallelization. More precisely, the inter-node communication is handled with MPI and for communication within the nodes OpenMP is used. For example, consider an eight-node job in which there is one MPI task per node and each MPI task has 24 OpenMP threads, resulting in a total core (and thread) count of 192. That is, for a 8 x 24 job the following flags are used. When running/submitting dynamically compiled executables remember to load same Programming Environment (PrgEnv-cray, PrgEnv-gnu or PrgEnv-intel) that was loaded when compiling was done.

```
#!/bin/bash --login
## hybrid MPI/OpenMP example
## valid for PrgEnv-cray and PrgEnv-gnu

## The number of compute nodes for a 8 node mpi/openmp job (8*24=192)
## Job layout: one mpi process per compute node and 24 openmp threads per compute node
#SBATCH --nodes=8

## Choose a suitable queue <test,small,large>
## How to check queue limits: scontrol show part <queue name>
## for example: scontrol show part small
#SBATCH -p small

#SBATCH -J jobname
#SBATCH -o jobname_%J.out
#SBATCH -e jobname_%J.err
#SBATCH -t 01:01:00

## number of OpenMP threads
export OMP_NUM_THREADS=24

## option: -n ( total number of mpi tasks )
## option: -d ( number of OpenMP threads per mpi task )
## option: -j ( number of logical CPUs per physical CPU core )
## option: -N ( number of mpi tasks per compute node )
## run the application on compute nodes
aprun -n 8 -d 24 -N 1 -j 1 ./hybrid_executable
```

Same example for Intel compiled hybrid program.

```
#!/bin/bash --login
## hybrid MPI/OpenMP example
## valid for PrgEnv-intel

## The number of compute nodes for a 8 node mpi/openmp job (8*24=192)
## Job layout: one mpi process per compute node and 24 openmp threads per compute node
#SBATCH --nodes=8

## Choose a suitable queue <test,small,large>
## How to check queue limits: scontrol show part <queue name>
## for example: scontrol show part small
#SBATCH -p small

#SBATCH -J jobname
#SBATCH -o jobname_%J.out
#SBATCH -e jobname_%J.err
#SBATCH -t 01:01:00

## number of OpenMP threads
export OMP_NUM_THREADS=24

## define intel thread affinity
export KMP_AFFINITY="granularity=fine,compact,1"

## option: -n ( total number of mpi tasks )
## option: -d ( number of OpenMP threads per mpi task )
## option: -j ( number of logical CPUs per physical CPU core )
## option: -N ( number of mpi tasks per compute node )
## option: -cc none ( mpi affinity not needed, this is a must in a Intel case )
 ## run the application on compute nodes
aprun -n 8 -d 24 -N 1 -j 1 -cc none ./hybrid_intel_executable
```

If you find out that OpenMP sections of your code do not give run-to-run numerical stability try (with Intel compiled code) to set the variable KMP_DETERMINISTIC_REDUCTION=yes.

Because each compute node on Sisu contains two 12-core processors it might be useful to try hybrid MPI/OpenMP job that has 2 mpi processes per compute node. In the next batch job example one MPI process is allocated per socket (each compute node has two sockets and one socket has one 12-core processor). Once again total core (and thread) count is 192 (16 mpi process and each mpi process has 12 OpenMP threads). Each socket in Sisu is a NUMA node that contains a 12-core processor and its local NUMA node memory (each core in a compute node also has access to remote NUMA node memory but references to remote NUMA memory are not that optimal like local references) .

```
#!/bin/bash --login
## hybrid MPI/OpenMP example
## valid for PrgEnv-cray, PrgEnv-gnu and PrgEnv-intel

## The number of compute nodes for a 8 node mpi/openmp job (16*12=192)
## Job layout: two mpi process per compute node and 12 openmp threads per compute node
## And more precisely: one MPI process per socket
#SBATCH --nodes=8

## Choose a suitable queue <test,small,large>
## How to check queue limits: scontrol show part <queue name>
## for example: scontrol show part small
#SBATCH -p small

#SBATCH -J jobname
#SBATCH -o jobname_%J.out
#SBATCH -e jobname_%J.err
#SBATCH -t 01:01:00
## number of OpenMP threads
export OMP_NUM_THREADS=12

## with PrgEnv-intel next line is a must (please remove comment characters, ##)
## export KMP_AFFINITY="compact,1"
## ( when PrgEnv-cray or PrgEnv-gnu do NOT use above line)

## option: -n ( total number of mpi tasks )
## option: -d ( number of OpenMP threads per mpi task )
## option: -S ( number of mpi tasks per NUMA node)
## option: -ss ( allocate memory only from local NUMA node)
## option: -j ( number of logical CPUs per physical CPU core )
## option: -N ( number of mpi tasks per compute node )
## -cc numa_node ( mpi tasks and OpenMP threads are constrained to the local NUMA node)
## run the application on compute nodes
aprun -n 16 -d 12 -S 1 -ss -j 1 -N 2 -cc numa_node ./hybrid_exe
```

4.5 Debugging Parallel Applications

4.5.1 TotalView debugger

TotalView is a debugger with graphical user interface (GUI) for debugging parallel applications. With TotalView you can:

- run an application under TotalView control
- attach to a running application
- examine a core file

Compile the application to be debugged, for example Fortran, c or C++ program. The compiler option *-g* is generating the debug information.

```
ftn -g -o mpi_prog mpi_prog.f95
cc -openmp -g -o hybrid_intel_compiled hybrid_mpi_openmp.c
CC -g -o myprog mycode.C
```

Sisu has compute nodes that either are included in the SLURM partitions or interactive nodes (which are not included in the SLURM partitions). Currently there are eight interactive nodes and jobs can be

launched on them without submitting a batch job. So launching a debugging session can be done on interactive nodes or by submitting a batch job.

Debugging on interactive nodes

First example will launch a basic statically linked mpi code debugging session. Second example will launch a Intel compiled and statically linked hybrid mpi-openmp debugging session. If a code needs some runtime environment variables please define those using aprun option -e (see the example, option -e will set an environment variable on the interactive compute nodes). The environment variables that are defined on login nodes are not visible to the interactive jobs. That basically means that do not run or debug dynamically linked codes on interactive nodes unless the code has been compiled using the default module environment.

```
totalview aprun -a -n 16 ./mpi_prog
totalview aprun -a -e OMP_NUM_THREADS=8 -e KMP_AFFINITY="compact,1" -n 4 -d 8 -S 1 -ss -N 2 -cc num
```

Debugging on compute nodes (SLURM partitions)

This method will need a special setup but it will work also for dynamically linked codes. First step, write a setup file called debug_environment.sh. It has to be in the same directory where the Totalview is launched. Here is an example script that changes the programming environment and sets the number of threads for a OpenMP program. Please note: On Cray systems Totalview does not support debugging OpenMP sections of code.

```
# debug_environment.sh example file
# Switch to Intel environment>
module switch PrgEnv-cray PrgEnv-intel
# Execute the program with eight threads per task
export OMP_NUM_THREADS=8 export
export KMP_AFFINITY="compact,1"
```

An example for launching Totalview session using SLURM small partition for a dynamically linked Intel environment hybrid program is as follows. The SLURM salloc command can not be used directly. CSC has made an wrapper command **tvsalloc** for launching a Totalview debugging session. So after *tvsalloc* give the needed SLURM options (below: run time of the job allocation (-t 02:00:00), number of nodes (-N 4), SLURM partition (-p small). After *aprun -a* give the aprun options that your job will need (below: total numebr of processes (-n 8), number of threads per process (-d 8), number of process per NUMA node (-S 1), allocate memory only from local NUMA node (-ss), number of processes per compute node (-N 2), process and OpenMP threads are constrained to the local NUMA node (-cc numa_node).

```
tvsalloc -t 00:25:00 -p test -N 2 totalview aprun -a -n 32 ./mpi_prog
tvsalloc -t 02:00:00 N 4 -p small totalview aprun -a -n 8 -d 8 -S 1 -ss -N 2 -cc numa_node ./hybrid
```

REMARK: Statically linked codes do not need the setup file, *debug_environment.sh*, unless your job uses runtime environment variables. But *tvsalloc* is necessary on SLURM partitions (test, small, large). If your debugging session will last less than half an hour it is good idea to submit the job to the SLURM test partition.

Debugging session

When a debugging session starts *Totalview Startup Parameters* window may appear. Just click *Ok* button (in a basic case). TotalView Root and Process window appear. Click the GO button in the Totalview process window. A pop-up window appears, asking if you want to stop the job

```
Process srun is a parallel job.
Do you want to stop the job now
```

Select *Yes* in this pop-up window.

Very basic features of Totalview

The Process Window contains the code for the process or thread that you're debugging. This window is divided into panes of information. The Stack Trace Pane shows the call stack of routines. The Stack Frame Pane displays all of a routine's parameters, its local variables, and the registers for the selected stack frame.

The left margin of the Source Pane displays line numbers. An ARROW over the line number shows the current location of the program counter (PC) in the selected stack frame. One can place a breakpoint (left mouse click) at any line whose line number is contained within a box. After setting one or many breakpoints *Go* button executes your code to the next breakpoint. When one is placing a breakpoint on a line, TotalView places an icon over the line number. To remove a breakpoint just click the breakpoint icon one more time.

To examine or change a value of a variable right click the variable and select *Dive* from the pop up menu. To see the values of that variable on all processes select *Across Processes* from the pop up menu. A new window will show the values and other information from that variable. On that window one can edit the variable values.

Stepping commands *Go*, *Next*, *Step*, *Out* and *Run to* (on the top of Process window) are controlling the way one is executing the code. *Go* means go to the next breakpoint, if a breakpoint locates inside a loop the next breakpoint is the same until loop ends. *Next* executes the current line and the program counter (arrow) goes to next line. *Step* executes one line in your program and if the source line or instruction contains a subroutine or function call, TotalView steps into it. *Out* executes all statements within subroutine or function and exits. *Run To* executes all lines until the program counter reaches the selected line. A line is selected by clicking a code line (not the line number) and the background of that line turn grey.

Attach to a running application

Unfortunately this is not properly supported on Sisu.csc.fi environment.

Debugging a core file

Launch Totalview. Click *a core file* and on the Core File Session window enter the core file name and the program name. By default core files are not generated. To enable core files add

```
ulimit -c unlimited
```

in your batch job file.

4.5.2 LGDB debugger

lgdb is a GDB-based parallel debugger used to debug applications compiled with Cray Compiler Environment, GNU, and Intel Fortran, C and C++ compilers. It allows programmers to either launch an application or attach to an already running application that was launched with *aprun*. Additionally, it provides comparative debugging technology that enables programmers to compare data structures between two executing applications. Comparative debugging should be used in conjunction with the CCDB GUI tool accessed by loading the cray-ccdb module.