InstitutsCluster II User Guide

Version 2.06

Steinbuch Centre for Computing, KIT

July 17, 2015

You are welcome to contact our hotline:
ic2-hotline@lists.kit.edu
In case you want to call us please dial: +49 721 608-48011

This user guide is available in PDF under
`http://www.scc.kit.edu/scc/docs/IC/ug/ugic2.pdf`.

Any comments and hints concerning this introduction are welcome. Please send corresponding e-mails to Hartmut.Haefner@kit.edu.

Revision history

Version	Date
First official version of the User Guide describing InstitutsCluster II Version 1.00	March 20, 2013
Minor changes in chapter “File Systems” Version 1.01	April 9, 2013
Minor changes regarding Intel MPI Version 1.02	April 11, 2013
Reconfiguration concerning the administration of accounts Version 2.00	Septembre 30, 2013
Short description of command rdata added in chapter “File Systems” Version 2.01	May 13, 2014
Minor changes in chapter “Parallel Programming” Version 2.02	June 5, 2014
Minor change concerning option `-perf-report` in chapter “Parallel Programming” Version 2.03	July 29, 2014
Change of hardware configuration - more thin nodes Version 2.04	Septembre 10, 2014
Minor change concerning OpenMPI in chapter “Parallel Programming” Version 2.05	Octobre 28, 2014
Minor change concerning mpirun-options in chapter “Parallel Programming” Version 2.06	July 17, 2015

Contents

1 Introduction

The Steinbuch Centre for Computing (SCC) at Karlsruhe Institute of Technology (KIT) operates the parallel InstitutsCluster II (abbreviated: ic2) as high performance computer and throughput computer for the institutes of KIT that have shared in the cluster with funds. Therefore only employees of these institutes can get accounts on InstitutsCluster II. The details how to get an account are available in section 3 or on the website of SCC Karlsruhe `http://www.scc.kit.edu/dienste/4948.php`.

InstitusCluster II (ic2) can fulfil the services of a parallel high performance compute server as well as the services of a traditional serial and throughput oriented compute server. This user guide is mainly written for those customers who want to use InstitutsCluster II (ic2) as parallel high performance computer system. But except for a few sections, it is also of interest to those users, who want to run serial programs only.

In order to limit the size of this guide, only the most important information about the use of InstitutsCluster II (ic2) has been collected here. This document is accompanied by many links to resources on the web, especially on the web server of SCC Karlsruhe where more detailed information is available:
`http://www.scc.kit.edu/dienste/hpc.php`.

2 Configuration of InstitutsCluster II

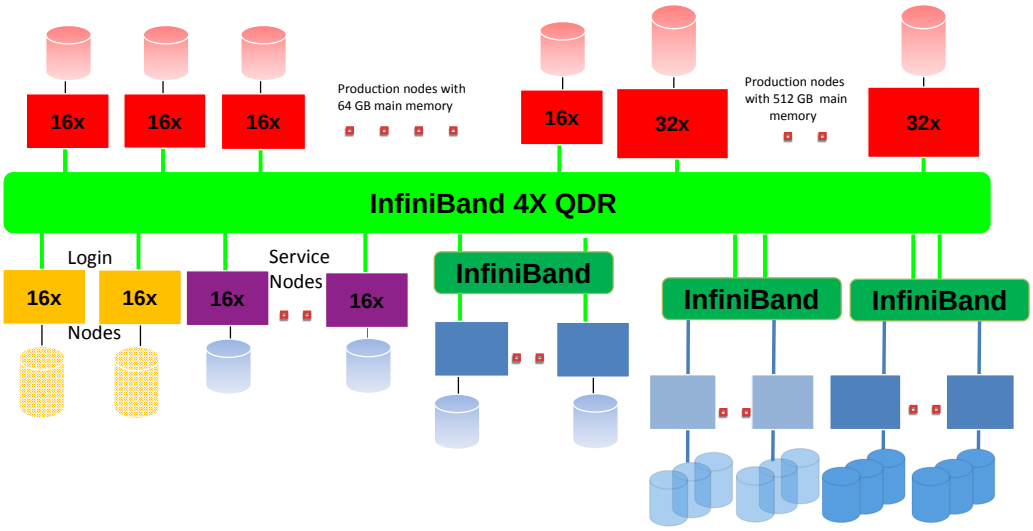

Figure 1: InstitutsCluster II at SCC of KIT

2.1 Architecture of InstitutsCluster II (ic2)

InstitutsCluster II is a distributed memory parallel computer where each node has sixteen Intel Xeon processors, local memory, disks and network adapters. All nodes are connected by a fast InfiniBand 4X QDR interconnect. In addition the file system Lustre, that is connected by coupling the InfiniBand of the file server with the InfiniBand switch of the compute cluster, is added to InstitutsCluster II to provide a fast and scalable parallel file system.

The basic operating system on each node is Suse Linux Enterprise (SLES) 11. On top of this operating system a set of open source software components like e.g. SLURM has been installled. Some of these components are of special interest to end users and are briefly discussed in this document. Others which are of greater importance to system administrators will not be covered by this document.

Nodes of InstitusCluster II may have different roles. According to the services supplied by the nodes, they are separated into disjoint groups. From an end users point of view the different groups of nodes are login nodes, compute nodes, file server nodes and administrative server nodes.

- Login Nodes
  The login nodes are the only nodes that are directly accessible by end users. These nodes are used for interactive login, file management, program development and interactive pre- and postprocessing. Two nodes are dedicated to this service but they are all accessible via one address and a DNS round-robin alias will distribute the login sessions to the different login nodes.

- Compute Node
  The majority of nodes are compute nodes which are managed by a batch system. Users will submit their jobs to the batch system JMS using SLURM as basic system and a job is executed depending on its priority, when the required resources become available.

- File Server Nodes
  The hardware of the parallel file system Lustre incorporates some file server nodes; the file system Lustre is connected by coupling the InfiniBand of the file server with the independant InfiniBand switch of the compute cluster. In addition to shared file space there is also local storage on the disks of each node (for details see chapter 4).

- Administrative Server Nodes
  Some other nodes are delivering additional services like resource management, external network connection, administration etc. These nodes can be accessed directly by system administrators only.

2.2 Components of InstitutsCluster II

InstitutsCluster II consists of

- 480 16-way Intel Xeon compute nodes. Each of these nodes contains two Octa-core Intel Xeon processors E5-2670 (Sandy Bridge) which run at a clock speed of 2.6 GHz and have 8x256 KB of level 2 cache and 20 MB level 3 cache. Each node has 64 GB of main memory, 2 local disks each with 1 TB and an adapter to connect to the InfiniBand 4X QDR interconnect.

- 5 32-way Intel Xeon compute nodes. Each of these nodes contains four Octa-core Intel Xeon processors E7-8837 (Westmere) which run at a clock speed of 2.67 GHz and have 8x256 KB of level 2 cache and 24 MB level 3 cache. Each node has 512 GB of main memory, 4 local disks each with 1 TB and an adapter to connect to the InfiniBand 4X QDR interconnect.

- 2 16-way Intel Xeon login nodes. Both nodes contain two Octa-core Intel Xeon processors E5-2670 (Sandy Bridge) which run at a clock speed of 2.6 GHz and have 8x256 KB of level 2 cache and 20 MB level 3 cache. Each node has 64 GB of main memory, 5 local disks each with 1 TB and an adapter to connect to the InfiniBand 4X QDR interconnect.

- 10 16-way Intel Xeon service nodes. Each of these nodes contains two Octa-core Intel Xeon processors E7-2670 (Sandy Bridge). Each node has 64 GB of main memory, one InfiniBand adapter and local disks.

- File server nodes that are part of the scalable, parallel file system Lustre that is tied to InstitutsCluster II via a separate InfiniBand network. The global shared storage of the file system is subdivided into a part used for home directories and a larger part for non permanent files. The directories in the larger part of the file system are often called work directories. The details are described in chapter 4.

An important component of InstitutsCluster II is the InfiniBand 4X QDR interconnect. All nodes are attached to this interconnect which is characterized by its very low latency of about 1 microsecond and a point to point bandwidth between two nodes of more than 3700 MB/s. Especially the very short latency makes the parallel system ideal for communication intensive applications and applications doing a lot of collective MPI communications.

With these types of nodes InstitutsCluster II can meet the requirements of a broad range of applications:

- applications that are parallelized by the message passing paradigm and use high numbers of processors will run on a subset of the 400 sixteen-way nodes and exchange messages over the InfiniBand interconnect,

- applications that are parallelized using shared memory either by OpenMP or explicit multithreading with Pthreads can run within the 16-way or 32-way nodes.

3 Access to InstitutsCluster II (ic2)

To login on InstitutsCluster II (ic2) an account is necessary. All employees of institutes - with shares in the cluster with funds - of KIT have to fill a form, that can be downloaded from the website `http://www.scc.kit.edu/hotline/3268.php`, and send it (per fax or mail) to SCC ServiceDesk.

3.1 Login

InstitutsCluster II (ic2) is a distributed memory parallel computer with two dedicated login nodes. These two login nodes are equipped with 16 cores, 64 GB main memory, local disks and network adapters. The Linux operating system Suse Linux Enterprose (SLES) 11 runs on all nodes, so that working on a single node of InstitutsCluster II (ic2) is comparable with working on a workstation.

3.2 Login on a Login Node

2 login nodes are available. The selection of the login node is done automatically. If you are connecting another time to a login node, the sessions might run on a different login node of InstitutsCluster II (ic2). Only the secure shell `ssh` is allowed to login. Other commands like `telnet` or `rlogin` are not allowed for security reasons.

A connection to ic2 can be established by the command

`ssh` *`KIT-account`*`@ic2.scc.kit.edu`

Be aware that the password can not be changed directly on ic2! Please change it on the website: https://intra.kit.edu (Link: Meine Daten)

If you are using OpenSSH (usually installed on Linux based systems) and you want to use a GUI-based application on ic2 like e.g. the debugger DDT, you should use the command

`ssh -X` *`KIT-account`*`@ic2.scc.kit.edu`

with the option `-X`.

4 File Systems

On InstitutsCluster II (ic2) the parallel file system Lustre is used for globally visible user data. Lustre is open source and Lustre solutions and support are available from different vendors. Nowadays, most of the biggest HPC systems are using Lustre.

Initial directories on the Lustre file systems are created for each user, and environment variables `$HOME`, `$PFSWORK` and `$HC3WORK` point to these directories. On hc3 the environment variable `$WORK` is the same as `$HC3WORK` and on ic2 `$WORK` is the same as `$PFSWORK`. Within a batch job there is a further directory `$TMP`. Some of the characteristics of the file systems are shown in Table 1.

The physical location of the file systems is shown in Fig 2. The file systems `$HOME`, `$PFSWORK` and `$HC3WORK` are currently visible on hc3 and ic2.

Property	$TMP	$HOME	$HC3WORK	$PFSWORK
Visibility	local	global	global	global
Lifetime	batch job	permanent	> 7 days	> 7 days
Disk space	1.8|3.8|4.5 TB for thin|fat|login nodes	427 TiB	203 TiB	301 TiB
Quotas	no	if required	if required	if required
Backup	no	yes (default)	no	no
Read perf./node	280|593|416 MB/s for thin|fat|login nodes	1 GB/s	1 GB/s	1 GB/s
Write perf./node	270|733|615 MB/s for thin|fat|login nodes	1 GB/s	1 GB/s	1 GB/s
Total read perf.	n*280|593 MB/s	8 GB/s	4.5 GB/s	6 GB/s
Total write perf.	n*270|733 MB/s	8 GB/s	4.5 GB/s	6 GB/s
global : all nodes of different HPC systems (including hc3) access the same file system; local : each node of ic2 has its own file system; permanent: files are stored permanently; batch job: files are removed at end of the batch job.				

Table 1: File Systems and Environment Variables

4.1 $HOME

The home directories of InstitutsCluster II (ic2) users are located in the parallel file system Lustre. You have access to your home directory from all nodes of hc3 and all nodes of InstitutsCluster II (ic2). A regular backup of these directories to tape archive is automatically done.

The `$HOME` directories are used to hold those files that are permanently used like source codes, configuration files, executable programs etc. The `$HOME` directories are located on the PFS2 (Parallel File System 2), i.e. the `$HOME` directories of ic2 and hc3 are the same.

For each user group (i.e. one institute) a fixed amount of disk space for home directories is reserved. The disk space is not yet controlled by so-called quotas; but quotas will be introduced in near future. You can find out the disk usage of the users in your group with the command `less $HOME/../diskusage` and your current quotas with the command `lfs quota -u $(whoami) $HOME`.

4.2 $WORK

On InstitutsCluster II (ic2) there is additional file space that can be accessed using the environment variable `$WORK` or (alternatively) `$PFSWORK`.

The work directories are used for files that have to be available for a certain amount of time, e.g. a few days. These are typically restart files or output data that have to be postprocessed.

All users can create large temporary files. But in order to be fair to your colleagues who also want to use this file system, large files which are no longer needed should be removed. SCC automatically removes old files in this file system which are older than 28 days. However, the guaranteed lifetime for files on `$WORK` is only 1 week.

The file system used for `$WORK` directories is also the parallel file system Lustre. This file system is especially designed for parallel access and for a high throughput to large files. The work file systems show high data transfer rates of up to 6 GB/s write and read performance when the data are accessed in parallel. You can find out your disk usage of `$WORK` with the command `lfs quota -u $(whoami) $WORK`. Similar statements are valid for the file system `$HC3WORK`. However, `$HC3WORK` should only be used if `$PFSWORK` is not available or if you want to share your work file system with jobs running on hc3.

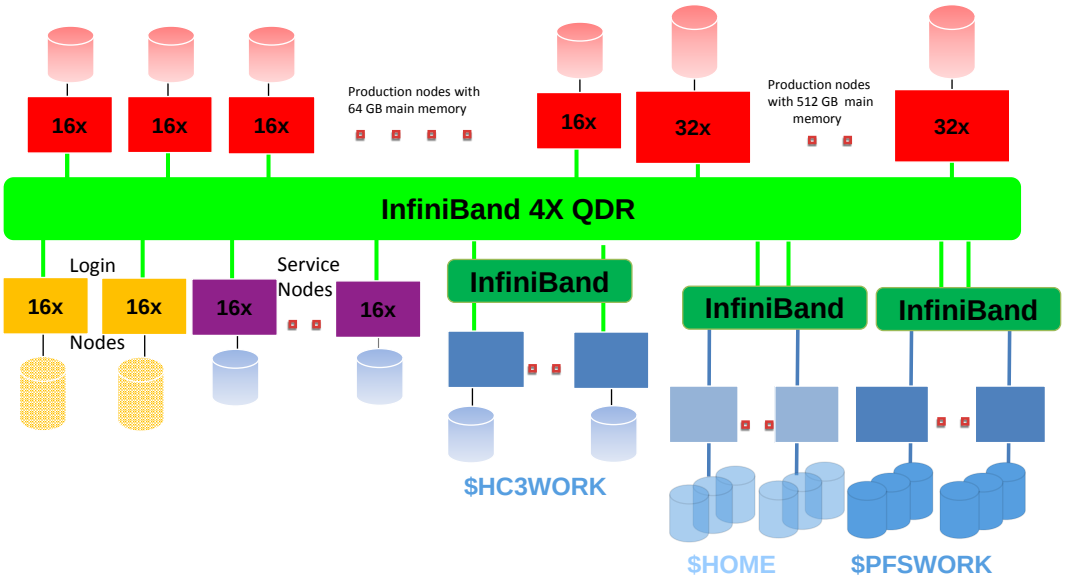

Figure 2: File systems on InstitutsCluster II (ic2)

4.3 Improving Performance on `$HOME` and `$WORK`

The following recommendations might help to improve throughput and metadata performance on Lustre file systems.

4.3.1 Improving Throughput Performance

Depending on your application some adaptions might be necessary if you want to reach the full bandwidth of the file systems. Parallel file systems typically stripe files over storage subsystems, i.e. large files are separated into stripes and distributed to different storage subsystems.

In Lustre, the size of these stripes (sometimes also mentioned as chunks) is called stripe size and the number of used storage subsystems is called stripe count.

When you are designing your application you should consider that the performance of parallel file systems is generally better if data is transferred in large blocks and stored in few large files. In more detail, to increase throughput performance of a parallel application following aspects should be considered:

- collect large chunks of data and write them sequentially at once,
- use moderate stripe count (not only stripe count 1) if only one task is doing IO in order to reach possible client bandwidth (usually limited by internal bus or network adapter),
- to exploit complete file system bandwith use several clients,
- avoid competitive file access or use blocks with boundaries at stripe size (default is 1MB),
- if many tasks use few huge files set stripe count to -1 in order to use all storage subsystems (see below for an example),
- if files are small enough for the local hard drives and are only used by one process store them on $TMP.

The storage subsystems of `$PFSWORK` are Infortrend RAID systems (Transtec PROVIGO 610). These RAID systems are attached to each server pair and the file system uses 6 software RAID6 volumes which vertically use all of these RAID systems. Each volume can reach up to 130 MB/s for writes and up to 150 MB/s for reads. The file system `$PFSWORK` uses 48 volumes. By default, files of `$PFSWORK` are striped across 2 volumes. The storage systems of `$HOME` are DDN SFA12K RAID systems. The file system `$HOME` uses 12 volumes. By default, files of `$HOME` are striped across 1 volume.

The storage system of `$HC3WORK` is one DDN S2A9900. The S2A9900 is configured with 28 stripe sets and each stripe set can reach about 300 MB/s for reads and writes. The default stripe count of `$HC3WORK` is 4.

However, you can change the stripe count of a directory and of newly created files. New files and directories inherit the stripe count from the parent directory. E.g. if you want to enhance throughput on a single file which is created in the directory `$WORK/my_output_dir` you can use the command
`lfs setstripe -c8 $WORK/my_output_dir`
to change the stripe count to 8. If the single file is accessed from one task it is not beneficial to further increase the stripe count because the local bus and the interconnect will become the bottleneck. If many tasks and nodes use the same output file you can further increase the throughput by using all available storage subsystems with the following command:
`lfs setstripe -c-1 $WORK/my_output_dir`

Note that the stripe count parameter -1 indicates that all available storage subsystems should be used. If all tasks write to the same file you should make sure that overlapping file parts are seldom used and that it is most beneficial if a single task uses blocks which are multiples of 1 MB (1 MB is the default stripe size).

If you change the stripe count of a directory the stripe count of existing files inside this directory is not changed. If you want to change the stripe count of existing files, change the stripe count of the parent directory, copy the files to new files, remove the old files and move the new files back to the old name. In order to check the stripe setting of the file `my_file` use `lfs getstripe my_file`.

Also note that changes on the striping parameters (e.g. stripe count) are not saved in the backup, i.e. if directories have to be recreated this information is lost and the default stripe count will be used. Therefore, you should annotate for which directories you made changes to the striping parameters so that you can repeat these changes if required.

4.3.2 Improving Metadata Performance

Metadata performance on parallel file systems is usually not as good as with local file systems. In addition, it is usually not scalable, i.e. a limited resource. Therefore, you should omit metadata operations whenever possible. For example, it is much better to have few large files than lots of small files.

In more detail, to increase metadata performance of a parallel application following aspects should be considered:

- avoid creating many small files,
- avoid competitive directory access, e.g. by creating files in separate subdirectories for each task,
- if lots of files are created use stripe count 1,
- if many small files are only used by one process store them on $TMP,
- change the default colorization setting of the command `ls` (see below).

On modern Linux systems, the GNU `ls` command often uses colorization by default to visually highlight the file type; this is especially true if the command is run within a terminal session. This is because the default shell profile initializations usually contain an alias directive similar to the following for the ls command:
`alias ls='ls -color=tty'`
However, running the `ls` command in this way for files on a Lustre file system requires a stat() call to be used to determine the file type. This can result in a performance overhead, because the stat() call always needs to determine the size of a file, and that in turn means that the client node must query the object size of all the backing objects that make up a file. As a result of the default colorization setting, running a simple `ls` command on a Lustre file system often takes as much time as running the `ls` command with the `-l` option (the same is true if the `-F`, `-p`, or the `-classify` option, or any other option that requires information from a stat() call, is used.). To avoid this performance overhead when using `ls` commands, add an `alias` directive similar to the following to your shell startup script:
`alias ls='ls -color=none'`

4.4 $TMP

While all tasks of a parallel application access the same $HOME and $WORK directory, the $TMP directory is local to each node on InstitutsCluster II (ic2). Different tasks of a parallel application use different directories when they do not utilize one node.

This directory should be used for temporary files being accessed by single tasks. On all compute nodes except of the 'fat nodes' the underlying hardware is two 1 TB disks per node. On the 'fat nodes' there are four 1 TB disks per node. Secondly this directory should be used for the installation of software packages. This means that the software package to be installed should be unpacked, compiled and linked in a subdirectory of $TMP. The real installation of the package (e.g. make install) should be made in(to) the Lustre file system.

Each time a batch job is started, a subdirectory is created on each node assigned to the job. $TMP is newly set; the name of the subdirectory contains the Job-id and the starting time so that the subdirectory name is unique for each job. This unique name is then assigned to the environment variable $TMP within the job. At the end of the job the subdirectory is removed.

4.5 Moving Files between Local Workstations and ic2

You should transfer files between ic2 and your workstation by using the command scp. You can transfer files in both directions. In special cases the passive ftp command (only from ic2 to your workstation) can be used.

scp has a similar syntax like rcp, i.e. files on a remote computer system are identified by prefixing the file name with the computer name and user-id. File name and computer name are separated by a colon, while user-id and computer name are separated by the sign @.

A small example is to copy the file mydata from user *xy1234* on the computer ws.institute.kit.edu into your $HOME directory on ic2. To accomplish this you may enter the following command on ic2:

scp *xy1234*@ws.institute.kit.edu:mydata $HOME/mydata

You will find further information on the scp command in the corresponding man page.

4.6 Access to the Filesystem bwFileStorage

Users of the filesystem bwFileStorage (http://www.scc.kit.edu/dienste/bwFileStorage.php) can furthermore transfer data to InstitutsCluster II via the tool rdata.
Therefore the environment variables $BWFILESTORAGE and $BWFS are set.

The command rdata executes the filesystem operations on special "data mover" nodes and distributes the load. Examples for the command are:

```
rdata "ls $BWFILESTORAGE/*.c"
rdata "cp foo $BWFS"
```

The command man rdata shows how to use the command rdata.

4.7 Backup and Archiving

There are regular backups of all data of the home directories,whereas ACLs and extended attributes will not be backuped or archived. With the following commands you can access the saved data:

The option -h shows how to use both commands.

Files of the directories $HOME and $WORK can be archived. With the following commands you can use the archive:

The option -h shows how to use the commands.

Command	Description
`tsm_q_backup`	shows one, multiple or all files stored in the backup device
`tsm_restore`	restores saved files

Table 2: Commands for Backup

Command	Description
`tsm_archiv`	archives files
`tsm_d_archiv`	deletes files from the archive
`tsm_q_archiv`	shows files in the archive
`tsm_retrieve`	retrieves archived files

Table 3: Commands for Archiving

More detailed information you can find on the following website:
`http://www.rz.uni-karlsruhe.de/rz/sw/tsm/xc`.

5 Modules

InstitutsCluster II (ic2) supports the use of Modules software to make it easier to configure and modify the user environment. Modules software enables dynamic modification of your environment by the use of modulefiles. A modulefile contains information to configure the shell for an application. Typically, a modulefile contains instructions that alter or set shell environment variables, such as PATH and MANPATH, to enable access to various installed software.

One of the key features of using modules is to allow multiple versions of the same software to be used in your environment in a controlled manner. For example, two different versions of the Intel C compiler can be installed on the system at the same time - the version used is based upon which Intel C compiler modulefile is loaded.

The software stack of ic2 provides a number of modulefiles. You can also create your own modulefiles. Modulefiles may be shared by many users on a system, and users may have their own collection of modulefiles to supplement or replace the shared modulefiles.

A modulefile does not provide configuration of your environment until it is explicitly loaded. That is, the specific modulefile for a software product or application must be loaded in your environment (with the `module load` command) before the configuration information in the modulefile is effective.

The modulefiles that are automatically loaded for you when you log in to the system can be displayed by the command `module list`. You only have to load further modulefiles, if you want to use additional software packages or to change the version of an already loaded software.

By default the modulefiles

`dot` adds the current directory to your environment variable PATH,

`intel` loads Intel C/C++ and Fortran90/95 compiler in a stable version,

`openmpi` loads OpenMPI in a stable version,

and - if necessary - further modulefiles will be loaded when logging in.

5.1 The most Important of Supplied Modulefiles

All the above mentioned software packages can be used by all users.

Modulefile	Description
dot	adds the current directory to your environment variable PATH
gcc	loads GNU C/C++ and Fortran90/95 compiler in a stable version
gcc/4.7.x	loads GNU C/C++ and Fortran90/95 compiler in version 4.7.x
intel	loads Intel C/C++ and Fortran90/95 compiler in a stable version
intel/13.x.x	loads Intel C/C++ and Fortran90/95 compiler in the stable version 13.x.x
openmpi	loads OpenMPI in an actual version
impi	loads Intel MPI in an actual version
ddt	loads graphical debugger in an actual version
mkl	loads Intel MKL for Intel compiler in an actual, stable version
itac	loads Intel trace collector and trace analyzer in an actual version

Table 4: Important Supplied Modulefiles

5.2 Viewing available Modulefiles

Available modulefiles are modulefiles that can be load by the user. A modulefile must be loaded before it provides changes to your environment, as described in the introduction to this section. You can view the modulefiles that are available on the system by issuing the `module avail[able]` command:

`module avail[able]`

5.3 Viewing loaded Modulefiles

A loaded modulefile is a modulefile that has been explicitly loaded in your environment by the module load command. To view the modulefiles that are currently loaded in your environment, issue the module list command:

`module list`

5.4 Loading and Unloading a Modulefile

You can load a modulefile in to your environment to enable easier access to software that you want to use by executing the `module load` command. You can load a modulefile for the current session, or you can set up your environment to load the modulefile whenever you log in to the system.

You can load a modulefile for your current login session as needed. To do this, issue the `module load` command as shown in the following example, which illustrates the DDT debugger modulefile being loaded:

`module load ddt` or `module add ddt`

Loading a modulefile in this manner affects your environment for the current session only.

If you frequently use one or more modulefiles that are not loaded when you log in to the system, you can set up your environment to automatically load those modulefiles for you. A method for doing this is to modify your shell startup script to include instructions to load the modulefile automatically.

For example, if you want to automatically load the DDT debugger modulefile when you log in, edit your shell startup script to include the following instructions. This example assumes that you use bash as your login shell. Edit the `$HOME/.bashrc` file as follows:

```
# if the 'module' command is defined, $MODULESHOME
# will be set
if [ -n "$MODULESHOME" ]; then
  module load ddt
fi
```

From now on, whenever you log in, the DDT debugger modulefile is automatically loaded in your environment.

In certain cases it may be necessary to unload a particular modulefile before you can load another modulefile in to your environment to avoid modulefile conflicts.

You can unload a modulefile by using the `module unload` or `module rm` command, as shown in the following example:

`module unload ddt` or `module rm ddt`

Unloading a modulefile that is loaded by default makes it inactive for the current session only - it will be reloaded the next time you log in.

5.5 Creating a Modulefile

If you download or install a software package into a private directory, you can create your own (private) modulefile for products that you install by using the following general steps:

1. create a private modulefiles directory,
2. copy an existing modulefile (as a template) or copy the corresponding default modulefile out of a subdirectory - if available - of the path `/software/all/modules/modulefiles` into the private modulefiles directory,
3. edit and modify the modulefile accordingly,
4. register the private directory with the `module use` command.

A user installing an arbitrary product or package should look at the manpages for modulefiles, examine the existing modulefiles, and create a new modulefile for the product being installed using existing modulefiles as a template. To view modules manpages, type:

`man module` or `man modulefile`

5.6 Further important Module Commands

The command `module help [modulefile...]` prints the usage of each sub-command.

The commands `module display modulefile [modulefile...]` or
`module show modulefile [modulefile...]` display information about a modulefile. The above mentioned commands will list the full path of the modulefile and all (or most) of the environment changes the modulefile will make if loaded. It will not display any environment changes found within conditional statements.

The command `module whatis [modulefile [modulefile...]]` displays the modulefile information set up by the module-whatis commands inside the specified modulefiles. If no modulefiles are specified all whatis information lines will be shown.

The command `module use [-a|-append] directory [directory...]` prepends directory [directory...] to the MODULEPATH environment variable. The –append flag will append the directory to MODULEPATH.

6 Compilers

On InstitutsCluster II (ic2) exist different compilers for Fortran (supporting the language standards of Fortran 77, Fortran 90, Fortran 95 and partially Fortran 2003), C and C++. There are two Fortran compiler families and two C/C++ compiler families. The Fortran compilers consist of the Intel compiler family and the GNU Fortran95 compiler. The C/C++ compilers consist of the Intel compiler family and the GNU C/C++ compilers in more than 3 versions. We recommend the latest versions of the C/C++ and Fortran compilers of the Intel compiler family.

6.1 Compiler Options

6.1.1 General Options

As on other Unix or Linux systems the compilers support the most common options:

`-c` compile only, do not link the object codes to create an executable program.

`-I`***path*** specify a directory, which is used to search for module files and include files, and add it to the include path.

`-g` include information for a symbolic debugger in the object code.

`-O [ level ]` create optimized source code. The optimization levels are 0, 1, 2, and 3. The option `-O` is identical to `-O2`. Increasing the optimization level will result in longer compile time, but will increase the performance of the code. In most cases at least optimization level `-O2` should be selected. The `-O2` option of the Intel compilers enables optimizations for speed, including global code scheduling, software pipelining, predication, and speculation. The GNU compilers additionally support the compiler options `-Os` optimizing the code for size and `-Ofast` as strongest optimization level disregarding strict standard compliance.

`-p` **or** `-pg` create code for profiling with the `gprof` utility. `-p` is not supported by the GNU compilers.

`-L`***path*** tell the linker to search for libraries in *path* before searching the standard directories

`-l`***library*** use the specified library to satisfy unresolved external references

`-o` ***name*** specify the name of the resulting executable program.

6.1.2 Important specific Options of Intel Compilers

All Intel C/C++ compilers can be called by the command `icc`. If you are using pure C++ code, you also can use the C++ compiler `icpc`. All Intel Fortran compilers can be called by the command `ifort`. Intel specific compiler options which are often needed are:

`-vec_report[0|1|2|3|4|5]` specifies the amount of vectorizer diagnostic information to report; valid suboptions are 0 (produces no diagnostic information) up to 5 (indicates non-vectorized loops and prohibiting data dependency information).

`-fp-model` ***keyword*** controls the semantics of floating-point calculations. *keyword* specifies the semantics to be used. See the possible values in the Intel Compiler User Guide.

`-parallel` tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel; to use this option, you must also specify `-O2` or `-O3`.

`-par_report[0|1|2|3]` controls the auto-parallelizer's level of diagnostic messages; valid suboptions are 0 (produces no diagnostic information) up to 3 (indicates diagnostics indicating loops successfully and unsuccessfully auto-parallelized and additional information about any proven or assumed dependencies inhibiting auto-parallelization).

`-openmp` enables the parallelizer to generate multithreaded code based on OpenMP directives.

`-openmp_report[0|1|2]` controls the OpenMP parallelizer's level of diagnostic messages; valid suboptions are 0 (produces no diagnostic information) up to 2 (displays diagnostics indicating loops, regions, and sections successfully parallelized and diagnostics indicating successful handling of MASTER constructs, SINGLE constructs, CRITICAL constructs, ORDERED constructs, ATOMIC directives, etc; suboption 1 is the default.

`-save` is only an Intel Fortran compiler option; it places variables, except those declared as AUTOMATIC, in static memory.

`-traceback` is only an Intel Fortran compiler option; it tells the compiler to generate extra information in the object file to allow the display of source file traceback information at run time when a severe error occurs.

The option `-fast` does not work on InstitutsCluster II because it includes the option `-static`.

6.1.3 Important specific Options of GNU Compilers

The GNU C/C++ compiler can be called by the command `gcc`. The GNU Fortran95/2003 compiler can be called by the command `gfortran`; the Fortran compilers supports all options supported by the C/C++ compiler. GNU specific compiler options which are often needed are:

`-funroll-loops` unrolls loops whose number of iterations can be determined at compile time or upon entry to the loop.

`-fprefetch-loop-arrays` generates instructions to prefetch memory to improve the performance of loops that access large arrays.

`-static` prevents linking with shared libraries.

6.2 Fortran Compilers

The standard Fortran compiler on ic2 is Intel's Fortran compiler. You can use different versions of this compiler. All versions of the Intel Fortran compiler support Fortran 77, Fortran 90 and Fortran 95 plus some features of the Fortran 2003 standard and other extensions. A detailed description is available in the different Intel Fortran User Guides and the different Intel Fortran Language References. All documents are available via the website: `http://www.scc.kit.edu/dienste/4983.php`

The Intel Fortran compiler may be invoked with several suffixes indicating the format of the source code, expected language standard and some other default options:

		default compiler option for	
command	file name suffix	source format	language level
`ifort`	`.f, .ftn, .for, .i`	`-fixed -72`	`-nostand`
`ifort`	`.F, .FTN, .FOR, .fpp, .FPP`	`-fixed -72 -fpp`	`-nostand`
`ifort`	`.f90, .i90`	`-free`	`-nostand`
`ifort`	`.F90`	`-free -fpp`	`-nostand`

Table 5: Fortran suffix names

Free format source codes should always be stored in files with file name extension `.f90`, `.i90` or `.F90` while files containing fixed format source code should have the file name extension `.f, .ftn, .for, .i, .F, .FTN, .FOR, .fpp, .FPP`.

To compile FORTRAN90/95 source code stored in file `my_prog.f90` the appropriate command is

```
ifort -c -O3 my_prog.f90
```

To compile an MPI program the basic compiler name must be substituted by the string `mpif90`. The parallel program `my_MPI_program.f90` therefore should be compiled with the command

```
mpif90 -c -O3 my_MPI_prog.f90
```

To compile multithreaded applications (i.e. OpenMP programs) the compiler option `-openmp` is added to the compiler name, i.e. the OpenMP program `my_OpenMP_program.f90` has to be compiled with the command

```
ifort -c -O3 -openmp my_OpenMP_prog.f90
```

When a FORTRAN90/95 program uses both parallelization paradigms (MPI and multi threading) then the compiler name must be substituted by the string `mpif90` and the compiler option `-openmp` must be used.

The GNU Fortran compiler supports the Fortran95 standard and some Fortran 2003 features. Again the above mentioned commands can be used with the GNU Fortran compiler, if the name of the Intel Fortran compiler `ifort` is substituted by the name of the GNU Fortran compiler `gfortran`.

6.3 C and C++ Compilers

The C and C++ compilers on ic2 are:

- latest Intel C/C++ compilers in versions 10.1, 11.1, 12.1 (default compiler) and 13.0;
- GNU project C/C++ compilers and PGI C/C++ compiler.

The C and C++ compilers on ic2 are invoked with commands `icc` ,`gcc` or `pgcc`. Details may be found in the appropriate man pages or in the compiler manuals (`http://www.scc.kit.edu/dienste/4983.php`).

To compile MPI programs the compiler scripts `mpicc` and `mpicxx` should be used for C and C++ programs.

To compile a C++ program `my_MPI_program.C` that calls MPI functions the appropriate command is therefore

```
mpicxx -c -O3 my_MPI_program.C
```

To compile an OpenMP program `my_OpenMP_program.C` written in C++ the following command should be used:

```
mpicxx -c -openmp -O3 my_OpenMP_program.C
```

6.4 Environment Variables

The environment variables `FFLAGS`, `FCFLAGS`, `F90FLAGS`, `CFLAGS` and `CXXFLAGS` are set for the Intel compiler and also for the GNU compiler suite. The environment variables are set so that your own code will be optimized safely when using the above mentioned environment variables instead of own compiler flags. For usage of aggressive optimization own compiler flags must be set!

7 Parallel Programming

Different programming concepts for writing parallel programs are used in high performance computing and are therefore supported on InstitutsCluster II. This includes concepts for programming for distributed memory systems as well as for shared memory systems.

A program parallelized for distributed memory systems consists of several tasks where each task has its own address space and the tasks exchange data explicitly or implicitly via messages. This type of parallelization is the most portable parallelization technique but may require a high programming effort. It is used on workstation clusters as well as on parallel systems like ic2.

In contrast to this, parallelization for shared memory systems is sometimes much easier but restricts the execution of the resulting program to a computer system which consists of several processors which share one global main memory. This type of parallelization can be used within a single node of ic2.

A lot of resources on these parallelization environments are available on the web. A starting address could be: `http://www.scc.kit.edu/dienste/4040.php`

7.1 Parallelization for Distributed Memory

For distributed memory systems most often explicit message passing is used, i.e. the programmer has to introduce calls to a communication library to transfer data from one task to another one. As a de facto standard for this type of parallel programming the Message Passing Interface (MPI) has been established during the past years. On ic2 MPI is part of the parallel environment.

7.1.1 Compiling and Linking MPI Programs

There are special compiler scripts to compile and link MPI programs. All these scripts start with the prefix `mpi`:

`mpicc` compile and link C programs;

`mpicxx` compile and link C++ programs;

`mpif77` **or** `mpif90` compile and link Fortran programs. Both variants work together with Intel and GNU compilers which means that it complies with the Fortran 90/95 language specification.

With these compiler scripts no additional MPI specific options for header files, libraries etc. are needed, but all the standard options of the serial compilers are still available.

Additional compiler command options for Intel MPI are:

Intel MPI	
Compiler Command Option	**Brief Explanation**
`-mt_mpi`	links the thread safe version of the Intel MPI Library
`-static_mpi`	links the Intel MPI library statically
`-ilp64`	enables partial ILP64 support, i.e. all integer arguments of the Intel MPI Library are treated as 64-bit values
`-echo`	displays everything that the command script does
`-show`	shows how the underlying compiler is invoked, without actually running it
`-{cc,cxx,fc,f77,f90}=`*`compiler`*	selects the underlying compiler

Further details on MPI may be found at `http://www.scc.kit.edu/dienste/4140.php`

7.1.2 Communication Modes

Communication between the tasks of a parallel application can be done in two different ways:

- data exchange using shared memory within a node,
- communication between nodes using the InfiniBand Switch.

On ic2 in general more than one task of a parallel application is executed on one node. In the operating mode - exclusice use of nodes - allocated nodes are used exclusively by up to sixteen MPI-processes or OpenMP-threads of a single batch job. In the operating mode - exclusice use of cores - allocated nodes can be used by different batch job (up to sixteen tasks with a summarized memory request of 64 or 512 GB). MPI-processes running on the same node use automatically the shared memory for the communication, i.e. they transfer messages by copying the data within the shared memory of the node. This results in a communication speed of about 8 GB/s for simple send/receive operations.

Tasks on different nodes communicate over the InfiniBand Switch. Data communication speed can reach more than 3700 MB/s.

So we can summarize:

- within a node always communication via shared memory is used,
- tasks running on different nodes communicate over the InfiniBand Switch.

7.1.3 Execution of Parallel Programs

Parallel programs can be started interactively or under control of the batch system.

Interactive parallel programs are launched with the command `mpirun`. They can only be executed on the node you are logged in, i.e. **launching the command `mpirun` interactively means that you cannot use another node than this one you are logged in. Especially the following restrictions hold:**

- maximum 4 MPI-processes,
- maximum 2 GB virtual memory per MPI-process and
- maximum 10 minutes CPU time per task are allowed.

Batch jobs are launched with the command `job_submit` and allow to start jobs in the development pool with a few nodes or in the production pool with many nodes. To start a parallel application as batch job the shellscript that is usually required by the command `job_submit` must contain the command `mpirun` with the application as input file.

The syntax to start a parallel application with OpenMPI (default MPI) is

`mpirun [` *`mpirun_options`* `]` *`program`*

or

`mpirun [` *`mpirun_options`* `] -f` *`appfile`*

both for interactive calls and calls within batch jobs or calls within shellscripts to execute batch jobs. The *mpirun_options* are the same for interactive calls and calls within batch jobs.

Important for the understanding: the option `-n #` or `-np #` is required calling `mpirun` interactively, but is ignored calling `mpirun` in batch jobs (the number of processors used in batch jobs is controlled by an option of the command `job_submit`). There is no option to specify the number of nodes you want to use, because calling `mpirun` interactively means to always use only one node and calling `mpirun` in a batch job means that the number of nodes is controlled automatically by the batch system.

Example:

```
#!/bin/bash
#
# This is an example for interactive parallel program execution.
#
# The program my_mpi_program will be run with 4 tasks.
#
mpirun -n 4 my_mpi_program
# A second version launching the same executable on the same number
# of processors is:
#
export MPIRUN_OPTIONS="-n 4"
mpirun my_mpi_program
```

7.1.4 `mpirun` Options

Calling

`mpirun -?` or `mpirun -h`

prints the usage of the command `mpirun`.

Calling

`mpirun -H`

prints the usage of the command `mpirun` with a brief explanation of the options.

The default MPI version is OpenMPI. Alternatively Intel MPI can be chosen by the command
`module add impi`

Subsequently all allowed mpirun options for OpenMPI and Intel MPI can be seen.

OpenMPI	
`mpirun` **Option**	**Brief Explanation**
`-n` # or `-np` #	MPI job is run on # processors (option is ignored in batch mode)
`-bycore|--bycore`	associate processes with successive cores
`-bysocket|--bysocket`	associate processes with successive processor sockets
`-bynode|--bynode`	launch processes one per node, cycling by node in a round-robin fashion
`--map-by {core|socket|node|...}` `[:PE=`*n*`]`	map processes to the specified object, defaults to socket. Many options are allowed, see http://www.open-mpi.org/doc/v1.8/man1/orterun.1.php. PE=*n* means that *n* processing elements (e.g. threads) are bound to each processor. (only version >= 1.8)
`-perf-report|--perf-report`	generate a performance report
`-{-}report-bindings`	report any bindings for launched processes
`-nobinding|--nobinding`	turns off the binding (don't use -bycore, -bysocket or -bynode)
`-V|--version`	prints version number
`-v|--verbose`	turns on verbose mode
`-d`	turns on debug mode
`--stdin` *rank*	MPI rank that is to receive stdin; the default is to forward stdin to rank=0, but this option can be used to forward stdin to any rank.
`-tag-output|--tag-output`	tag each line of output to stdout, stderr and stddiag with the rank that generated the output
`-{-}timestamp-output`	timestamp each line of output to stdout, stderr and stddiag
`-wdir` *directory*	change to *directory* before the user's program executes
`-f` *appfile*	allows to run different executables on different processors; the names of the executables must be stored in *appfile*; this option must always be the last option!

Intel MPI	
`mpirun` **Option**	**Brief Explanation**
Global options	
`-genvall`	propagates all environment variables to all MPI processes
`-genvnone`	suppresses propagation of any environment variables to any MPI processes
`-genvlist` *list_of_env_var_names*	passes a list of environment variables with their current values; a comma separated list of variables is expected
`-wdir` *directory*	specifies the working directory in which *executable* is run in the current arg-set
`-ppn` *#processes*	places the indicated number of consecutive MPI processes on every node in group round robin fashion
`-strace` or `-trace` *profiling_library*	profiles your MPI application using the indicated *profiling_library*; if the *profiling_library* is not mentioned, the default profiling library `libVT.so` is used
`-perf-report`	generate a performance report
`-binding` *"parameter=value[;parameter=...]"*	pins particular MPI process to a corresponding CPU and avoid undesired process migration; the quotes may be omitted for one-member list; the parameter list is printed in the Reference Manual of Intel MPI Library
`-rr`	places consecutive MPI processes onto different nodes in round robin fashion
`-scheck_mpi` or `-check_mpi` *checking_library*	checks your MPI application; the default checking library libVTmc.so is used, if *checking_library* is not mentioned
`-stune` or `-tune` *{directory,conf_file}*	optimizes the Intel MPI Library performance using data collected by the mpitune utility
`-V`	displays Intel MPI Library version information
`-verbose`	prints extra verbose information
`-ordered-output`	avoids intermingling of data output by the MPI processes; affects both standard output and standard error streams
`-print-rank-map`	prints rank mapping
`-print-all-exit-codes`	prints exit codes of all processes
Local options	
`-envall`	propagates all environment variables in the current environment
`-envnone`	suppresses propagation of any environment variables to the MPI processes in the current arg-set
`-envlist` *list_of_env_var_names*	passes a list of environment variables with their current values; a comma separated list of variables is expected
`-n` # or `-np` #	MPI job is run on # processors (option is ignored in batch mode)
`exe1:exe2:...`	allows to run different executables on different processors; the names of the executables are separated by colons. (this option must always be the last option!)

The last parameter of the command `mpirun` must be either the option `-f` *appfile* (only OpenMPI) or an executable program and shell script respectively or a colon separated list of executable programs (only Intel MPI).

Using an executable program and shellscript respectively as last parameter `mpirun` executes the programs in

Single Program Multiple Data (SPMD) mode, i.e. the same program is executed by all tasks of the parallel application. Sometimes parallel programs are designed in such a way that different programs are executed by the tasks of a parallel application. This is called Multiple Program Multiple Data (MPMD) mode which is also supported by `mpirun`. To use this mode the option `-f` *appfile* must be chosen as last parameter for OpenMPI and a colon separated list of executable programs must be chosen as last parameter for Intel MPI.

The format of the application file *appfile* is very simple. Running a master-slave model on 4 processors means that you have to create the following *appfile*:

```
-np 1  master
-np 3  slave
```

The master will run - as usual - on processor 0 and the slaves will run on the processors 1 up to 3.

If you want to set environment variables controlling OpenMPI or Intel MPI read the documentation on the MPIs reachable via `http://www.scc.kit.edu/dienste/4983.php`.

7.2 Programming for Shared Memory Systems

While MPI is a tool for distributed memory systems, OpenMP is targeted to shared memory systems, i.e. one node with several CPUs. OpenMP is an extension to Fortran and C and seems to become a de facto standard for parallel programming of shared memory systems. On ic2 the OpenMP specification is supported by the Intel Fortran and C compilers and also by the GNU compilers from version 4.2 on.

To compile programs for shared memory parallelism the Intel compiler option `-openmp` must be selected. The following options can be specified to control the behaviour of thread-parallelized programs:

`-openmp-report[0|1|2]` – controls the level of diagnostic messages regarding OpenMP;

`-openmp-stubs` – enables the compiler to generate sequential code; the OpenMP directives are ignored and a stub OpenMP library is linked;

`-par_report[0|1|2|3]` – controls the auto-parallelizer's level of diagnostic messages;

`-par_threshold[n]` – sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel; [n] is an integer from 0 to 100; the default value is 75;

`-parallel` – enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

For details see the Intel Compiler User Guide reachable via `http://www.scc.kit.edu/dienste/4983.php`

7.3 Distributed and Shared Memory Parallelism

For certain applications it might be convenient to combine the parallelization techniques for distributed memory and shared memory, i.e. parallelization within a node using shared memory parallelization with OpenMP and parallelization between nodes using message passing with MPI. In these cases the compilation must be started using one of the compiler scripts starting with prefix `mpi` and using the compiler option `-openmp`.

8 Debuggers

On InstitutsCluster II (ic2) the GUI based distributed debugging tool (ddt) may be used to debug serial as well as parallel applications. For serial applications also the GNU gdb or Intel idb debugger may be used. The Intel idb comes with the compiler and information on this tool is available together with the compiler documentation.

In order to debug your program it must be compiled and linked using the `-g` compiler option. This will force the compiler to add additional information to the object code which is used by the debugger at runtime.

8.1 Parallel Debugger ddt

ddt consists of a graphical frontend and a backend serial debugger which controls the application program. One instance of the serial debugger controls one MPI process. Via the frontend the user interacts with the debugger to select the program that will be debugged, to specify different options and to monitor the execution of the program. Debugging commands may be sent to one, all or a subset of the MPI processes.

Before the parallel debugger ddt can be used, it is necessary to load the corresponding module file:

```
module add ddt
```

Now ddt may be started with the command

`ddt` *`program`*

where *program* is the name of your program that you want to debug.

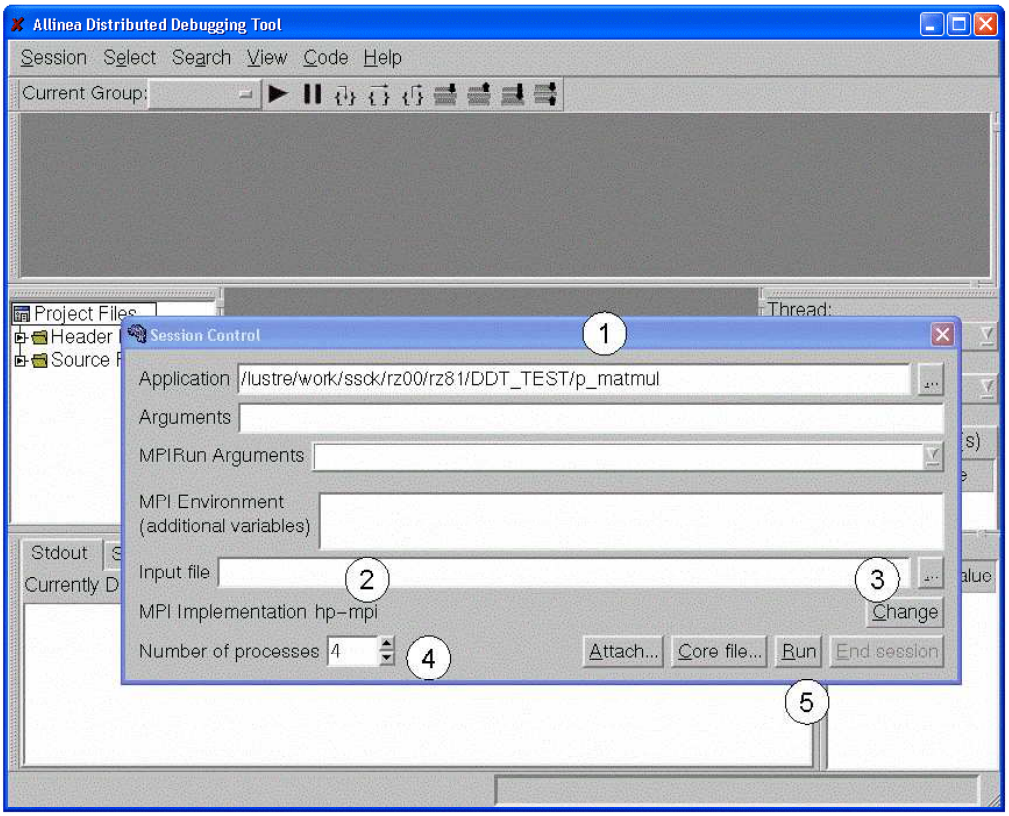

Figure 3: DDT startup window

Fig. 3 shows ddt's startup window. Before actually starting the debugging session you should check the contents of several fields in this window:

1. The top line shows the executable file that will be run under control of the debugger. In the following lines you may input some options that are passed to your program or to the MPI environment.
2. If your program reads data from stdin you can specify an input file in the startup window.
3. Before starting an MPI program you should check that 'openmpi' is the MPI implementation that has been selected. If this is not the case, you have to change this. Otherwise ddt may not be able to run your program.

   In order to debug serial programs, the selected MPI implementation should be 'none'

   You may also change the underlying serial debugger using the 'change' button. By default ddt uses its own serial debugger, but it may also use the Intel idb debugger.

4. Select the number of MPI processes that will be started by ddt. If you are using ddt within a batch job, replace `mpirun` by `ddt` in the command line of `job_submit` and make sure that the chosen number of MPI processes is identical to the number of MPI tasks (`-p` option) that you selected with the `job_submit` command. When you debug a serial program, select 1.

5. After you have checked all inputs in the ddt startup window, you can start the debugging session by pressing the 'run' button.

The ddt window now shows the source code of the program that is being debugged and breakpoints can be set by just pointing to the corresponding line and pressing the right mouse button. So you may step through your program, display the values of variables and arrays and look at the message queues.

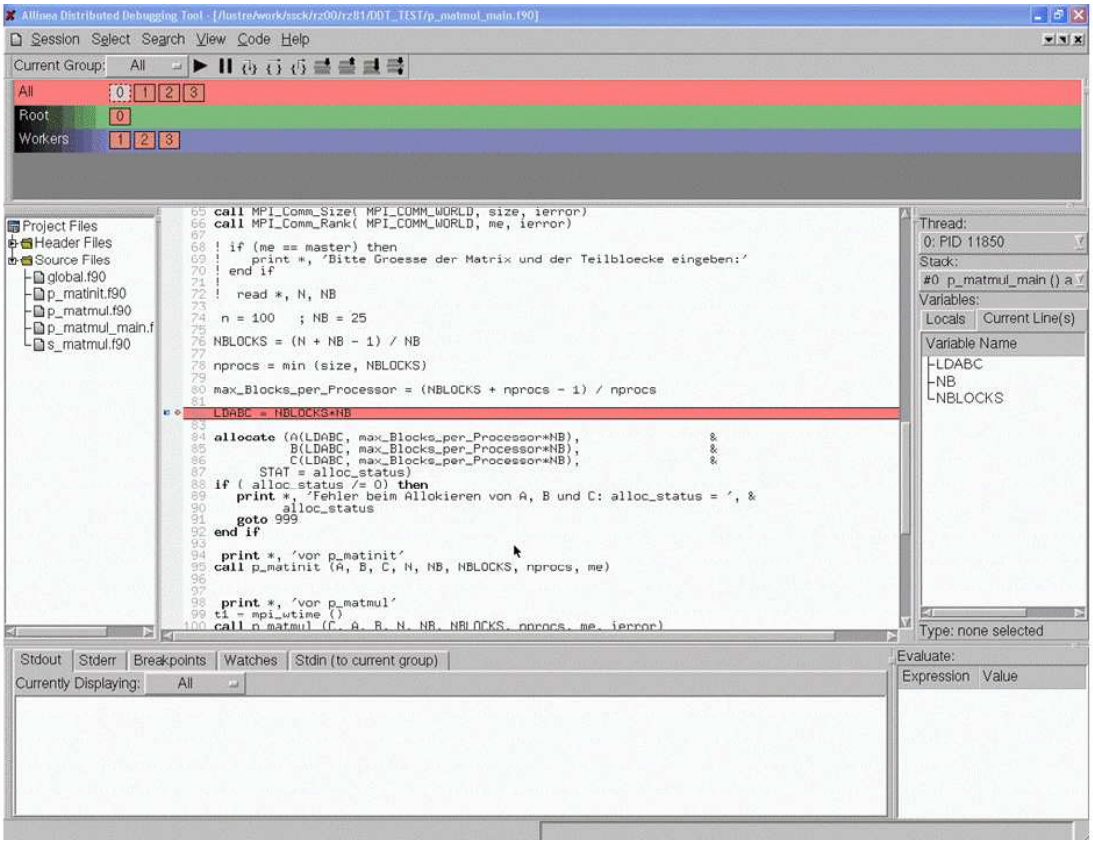

Figure 4: DDT window

9 Performance Analysis Tools

After installation and successful test of a program it is essential to analyze its performance. When performance bottlenecks have been detected they should be resolved by restructuring parts of the program, setting of specific compiler or runtime options or by replacing own code by optimized code from numerical libraries (cf. section 10). To get support and help to optimize your program you should contact the technical support staff at SCC (cf. section 13).

For most serial as well as parallel programs the processing power of the CPU is the limiting resource when running the program. This means that the performance analysis should concentrate on CPU usage. For parallel programs additionally the communication overhead has to be analyzed.

The most important questions in performance analysis are:

- Is the CPU used effectively, i.e. is there nearly no idle time?

- Is the communication organized in the right way so that no task is waiting for data to be sent from or to another task?
- Is the processor used most efficiently, i.e. what MFlops rate is achieved?

The performance analysis of a parallel program may therefore consist of the following steps:

1. Check the ratio of user CPU time, system CPU time and real (wall clock) time.
2. Analyze the communication behaviour of the program.
3. Find those parts of the program which consume the highest amount of CPU time.
4. Do a detailed analysis how the functional units of the processors are utilized.

For these steps different tools are supplied and will be described in the next sections.

When the most time consuming parts of the program and possible bottlenecks have been identified, then the next step is to restructure these parts of the program to improve the performance.

9.1 Timing of Programs and Subprograms

The simplest way to do a first timing analysis of a program is to use the `time` command to analyze a serial or multithreaded program.

9.1.1 Timing of Serial or Multithreaded Programs

To do a very first analysis of CPU usage of a serial or multithreaded application, just write the command time in front of the program name. i.e. in order to run the program `my_serial_program` under control of `time` enter the command

`time my_serial_program [ options ]`

After termination of the program you will get some additional lines of output which may look like:

```
real  2m9.051s
user  1m49.312s
sys   0m0.106s
```

In this example we see that the program used 1 min 49.312 sec CPU time in user mode and 0.106 sec in system mode. The system CPU time is the time which is consumed by operating system functions working for the application program. Most of this time is caused by input and output operations. The system CPU time should generally not exceed a few percent of the user CPU time. In those cases where the program has exclusive access to the resources of a node (e.g. in batch jobs running in the production class) the realtime should not be much higher than the sum of user and system CPU time. Otherwise the program seems to be waiting for completion of I/O operations.

In case of multithreaded programs the CPU time could be much higher than the real wall clock time. The CPU time is the sum of the times required by all threads of the program. The following example shows the measurement of a program running with two threads on a two way node:

```
real    3m24.255s
user    6m47.652s
sys     0m0.261s
```

As we see in this example the user time is nearly twice the real time. So the two threads of the program use the two CPUs without high overhead for I/O operations.

9.1.2 Timing of Program Sections

A more detailed timing is the measurement of CPU time and real time for certain sections of the program. This may be accomplished by inclusion of some timing calls into the program. Fortran programmers could use the Fortran 95 subprograms `CPU_TIME` and `DATE_AND_TIME`. C and C++ programmers should use the appropriate system calls like `times` or `getrusage`.

In a parallel MPI program the MPI function `MPI_Wtime` may also be used to measure the wall clock time.

A simple Fortran example may look like:

```
SUBROUTINE timer (real_time, cp_time)
!
! timer computes :
!
! real_time: the real time in seconds since midnight
!
! cp_time  : the CPU time consumed by the program since program start
!
   REAL(KIND=4)                 :: real_time, cp_time
   INTEGER, DIMENSION(8)        :: values

   CALL CPU_TIME (cp_time)

   CALL DATE_AND_TIME (VALUES = values)

   real_time = ((values(5) * 60. ) + values(6) ) * 60. + values(7) + &
               values(8)/1000.

END SUBROUTINE timer

!------------------------------------------------------------------------

PROGRAM timer_example

 . . .

REAL(KIND=4)  :: real_time0, cp_time0
REAL(KIND=4)  :: real_time, cp_time

. . .

CALL timer (real_time0, cp_time0)

! The real time and CPU time used by subroutine compute
! will be measured.

CALL compute

CALL timer (real_time, cp_time)

real_time = real_time - real_time0
cp_time  = cp_time  - cp_time0

PRINT *, 'real time needed for compute : ', real_time, ' sec.'
PRINT *, 'CPU time needed for compute  : ', cp_time  , ' sec.'

. . .
```

```
END PROGRAM timer_example
```

9.2 Analysis of Communication Behaviour with MPI

To reach reasonable performance for parallel applications it is essential to have a good load balancing between all tasks (i.e. all tasks should do nearly the same amount of work). Additionally the communication operations should be organized in such a way that there is no need for any task to wait for a long time in order to communicate with other tasks.

To get a first overview on the performance of your parallel application you can use the option `-perf-report`. Using this option informations like how much time is spent in computation, communication, I/O and how much memory is needed is shown. The report gives a good overview how to proceed to optimize an application. The report is stored in the directory you started your *executable* from in HTML-format and can easily be displayed with a browser. E.g. you can enter
`firefox` *executable*`_.....html`

9.2.1 Analysis of MPI Communication with Intel Trace Collector / Trace Analyzer

The Intel Trace Collector (ITC) / Trace Analyzer (ITA) is a tool to analyze the runtime behaviour of MPI programs. It is GUI based, easy to use and gives many hints for program optimization. It only works when using Intel MPI.

- Intel Trace Collector is a library that uses the MPI profiling interface and which collects a lot of data about the MPI communication during program execution and
- Intel Trace Analyzer visualizes these data in various modes.

To use ITC/ITA follow these steps:

1. Load the corresponding module file using the command

   `module add itac`

   This will make available the default versions of ITC and ITA which are the latest Intel Trace Collector and Intel Trace Analyzer. For details see the information on the Web at
   `http://www.scc.kit.edu/dienste/7250.php`

2. Use the mpirun-option `-strace` when running the MPI-application, e.g. `mpirun -strace -np 4 myprog`

   When this program reaches the function `MPI_Finalize`, ITC writes a set of trace files. By default the master trace file has the name *program_name*`.stf`. In our example we get a trace file called `myprog.stf`. Through certain configuration options the operation of ITC can be adapted to special needs. For details see the User Guide at
   `http://www.scc.kit.edu/dienste/7250.php`

To analyze the trace file enter the command

`traceanalyzer` *trace_file*

Fig. 5 shows some of the most important displays of Intel Trace Analyzer. The upper window shows the timeline of the application.
The next lower window shows the event timeline; this window shows the dynamic behaviour of all MPI tasks and the dependencies via message transfers. Each horizontal bar represents one task of the application. The different colors represent different operations like computation (blue) and communication (red). The lines connecting these bars represent point-to-point communication operations.
The left lower window shows the function profile. You can choose a flat profile or bars like ”Load Balance”, ”Call Tree” or ”Call Graph”.

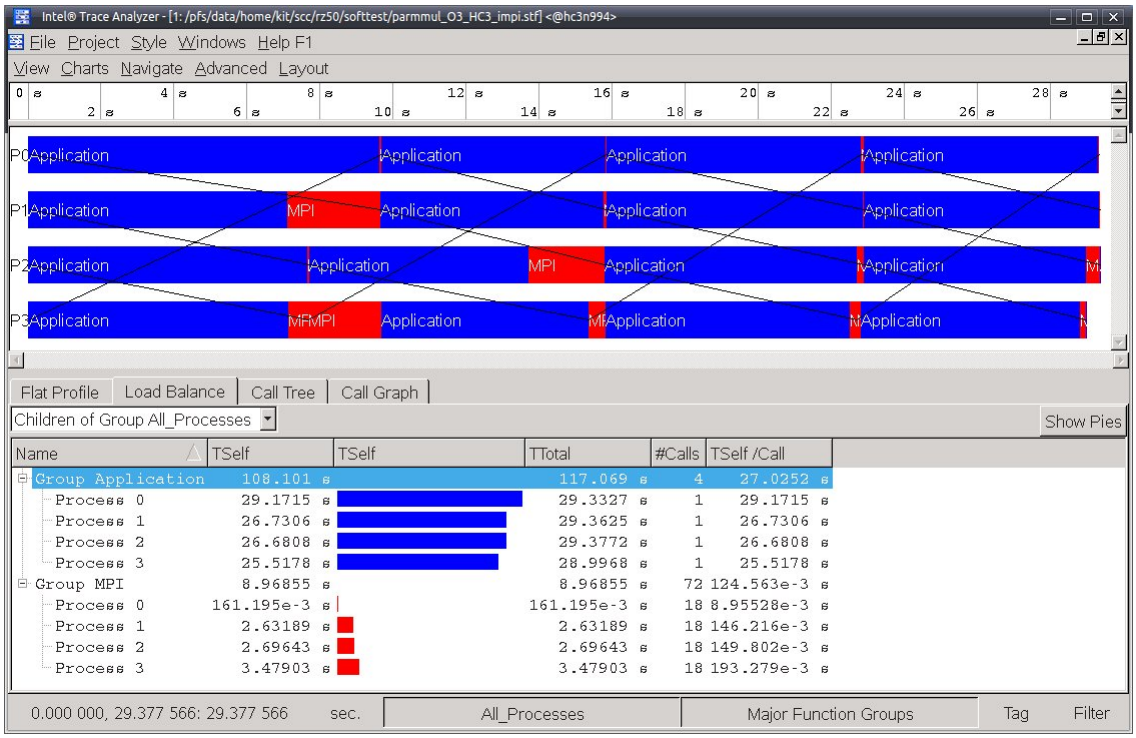

Figure 5: Some Intel Trace Analyzer (ITA) windows

The right lower window shows the message profile with different attributes like the average transfer rate between processes, the total time, the maximum or minimum time and so on.

From these graphs you can easily see if the communication operations are at the right places within the program.

9.3 Profiling

Profiling is used to identify those parts of a program that consume the highest amount of CPU time. In many cases more than 90% of the CPU time is used in less than 5% of the source code of the program. These most time consuming parts of the program should be optimized carefully. In some cases is is possible to replace own code by a call to some optimized functions or subprograms from highly tuned libraries (cf. section 10).

The profiling tool `gprof` is available on many Unix or Linux systems. The information you may get from this tools is:

- a flat profile with information on CPU usage by all subroutines and functions of the program and a
- a call graph profile which gives information not only on each function and subprogram, but also on its callees (number of calls, CPU time used by callee etc.).

To use the `gprof` utility the following steps are required:

1. Compile and link the program with `-pg` option.
2. Run the program as usual. When the program terminates a file `gmon.out` will be created. In case of a parallel program several output files `gmon.out.`i are written, where i is the task id.
3. To create the profiles, run

   `gprof` *`program`* `gmon.out*`

   where *program* is the name of your executable program. To create a profile for only one or a certain subset of tasks of a parallel application, you should replace the string `gmon.out*` by a list of file names.

10 Mathematical Libraries

Up to now the Intel Math Kernel Library (MKL) and the Library LINSOL has been installed as numerical library. Tuned implementations of well established open source libraries are part of MKL. The high-performance mathematical programming engine CPLEX is an optimization software package and is usually used by calling the CPLEX library.

10.1 Intel Math Kernel Library (MKL)

The Intel Math Kernel Library includes functions from following areas:

- Basic Linear Algebra Subprograms (BLAS - level 1, 2, and 3) and LAPACK linear algebra routines, offering vector, vector-matrix, and matrix-matrix operations;
- the PARDISO direct sparse solver, an iterative sparse solver, and supporting sparse BLAS (level 1, 2 and 3) routines for solving sparse systems of equations and Sparse BLAS (basic vector operations on sparse vectors);
- ScaLAPACK distributed processing linear algebra routines as well as the Basic Linear Algebra Communications Subprograms (BLACS) and the Parallel Basic Linear Algebra Subprograms (PBLAS);
- Fast Fourier transform (FFT) functions in one, two, or three dimensions with support for mixed radices (not limited to sizes that are powers of 2), as well as distributed versions of these functions;
- Vector Math Library (VML) routines for optimized mathematical operations on vectors;
- Vector Statistical Library (VSL) routines, which offer high-performance vectorized random number generators (RNG) for several probability distributions, convolution and correlation routines, and summary statistics functions;
- Data Fitting Library, which provides capabilities for spline-based approximation of functions, derivatives and integrals of functions, and search;
- Extended Eigensolver, a shared memory programming (SMP) version of an eigensolver based on the Feast Eigenvalue Solver.

Before linking a program with the Intel MK Libraries, a module must be loaded to set some environment variables:

```
module add mkl
```

Now the program may be linked using the MK Libraries: In the following examples we assume that a Fortran program `myprog.f90` and a C program `myprog.c` will be compiled and linked against the Intel MK Libraries. The appropriate options at link time can be identified by the Intel Math Kernel Library Link Line Advisor that can be found on the website `http://software.intel.com/sites/mkl/`.

More details on Intel MKL library are available in the User Guide of Intel MKL available via the website `http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation`.

10.2 Linear Solver Package (LINSOL)

LINSOL is a program package to solve large sparse linear systems. It has been developed at University of Karlsruhe Computing Center. For the simulation of many numerical problems the solution of large and sparse linear systems is required. For these problems the linear solver package LINSOL has been designed. Iterative techniques based on generalized conjugate gradient methods and beyond are implemented. Different polyalgorithms that select appropriate solvers from the whole variety of methods and direct solvers - the (Incomplete) Gauss algorithm for unsymmetric matrices and the (Incomplete) Cholesky algorithm for symmetric matrices - are available. Linsol is fully parallelized using MPI.

Before the usage of LINSOL the following module should be loaded to set some environment variables:

```
module add linsol
```

Then you can either use the LINSOL library or you can use the stand-alone interface that works on matrices in the Harwell-Boeing or LINSOL format.

In the directory `/software/all/linsol/examples` you find a file `exam01.f` that calls the LINSOL library by the Fortran90 interface `LSOLP`. It is compiled and linked here examplarily for arbitrary Fortran90/95 programs that call the LINSOL library:

- Calling LINSOL serially:

  ```
  ifort -O3 -o exam01 exam01.f -L/software/all/linsol/lib -llinsol -lnocomm -mkl
  ```

- Calling LINSOL on more than one processor:

  ```
  mpif90 -O3 -o exam01 exam01.f -L/software/all/linsol/lib -llinsol -lMPI -mkl
  ```

You can see the correct output of the executable `exam01` in the file `/software/all/linsol/examples/exam01.output`.

The stand-alone interface is used by the following command:

`linnox` *`parameter-file`*

For example you can solve the matrix `gre512.rua` stored in Harwell-Boeing format and the matrix `oilgen.lsol` stored in LINSOL format. Corresponding to the matrices there are parameter-files `gre512_param` and `oilgen_param` in the directory /software/all/linsol/examples. So the solution of the linear equation can be started by the command:

`linnox gre512_param` or `linnox oilgen_param`

You will find detailed informations on LINSOL on the website `http://www.scc.kit.edu/produkte/linsol`.

10.3 CPLEX

CPLEX is software to solve linear programming (LP) and related problems. More exactly, it solves linearly or quadratically constrained optimization problems where the objective to be optimized can be expressed as a linear function or a convex quadratic function. The variables in the model may be continuous variables or be restricted to integer values.

CPLEX provides the following features:

- Automatic and dynamic algorithm parameter control
- Fast, auromatic restarts from an advanced basis
- A variety of problem modification options
- A wide variety of input/output options
- Post solution information and analysis

CPLEX consists of

- the interactive optimizer `cplex`,
- the CPLEX Callable Library `/software/kit/CS/cplex/cplex121/lib/x86-64_debian4.0_4.1/static_pic/libcplex.a` , a C library providing the optimizer for integration into applications which allow to call C functions,

- the libraries
  `/software/kit/CS/cplex/cplex121/lib/x86-64_debian4.0_4.1/static_pic/libilocplex.a` ,
  `/software/kit/CS/cplex/concert29/lib/x86-64_debian4.0_4.1/static_pic/libconcert.a` ,
  and `/software/kit/CS/cplex/cplex121/lib/cplex.jar`
  offering an API that includes modeling facilities allowing a programmer to embed the CPLEX optimizers in applications written in C++ and Java (called "Concert Technology"), and
- interfaces to Python and MATLAB.

The Interactive Optimizer is able to read a problem interactively or from files in certain standard formats. After solving the problem, the solution can be displayed interactively or be written into text files. The program consists of the executable `cplex` which can be started after adding the module `cplex`.

The CPLEX Callable Library allows programmers to embed CPLEX optimizers in C, Fortran, or any other language that can call C functions. The directories where the Callable Library `libcplex.a` can be found is shown by the reference above which includes the path. The same holds for the libraries being part of the Concert Technology.

To use CPLEX with MATLAB, add the directory
`/software/kit/CS/cplex/cplex121/matlab/`
to your MATLAB path using the MATLAB command `addpath`.

Using CPLEX with Python is already set up by the module. With the module, examples can be executed directly, e.g.
`python /software/kit/CS/cplex/cplex121/examples/src/python/warehouse.py` .
Other examples may require specification of parameters.

The documentation is available by opening
`/software/kit/CS/cplex/cplex121/doc/html/en-US/documentation.html` ,
and a short overview by
`/software/kit/CS/cplex/cplex121/readme.html` .
The latter in addition informs on the directory structure under
`/software/kit/CS/cplex/cplex121/`.

The manuals are available as PDF files from the directory `/software/kit/CS/cplex/cplex121/doc/pdf/` .

Examples can be found at
`/software/kit/CS/cplex/cplex121/examples/` ,
especially at the subdirectories of `src`. There can be found directories with examples for C, C++, Java, MATLAB and Python.

11 CAE Application Codes

InstitutsCluster II (ic2) is especially suited for solving physical problems.

- structural mechanics: ABAQUS, LS-Dyna, MD Nastran, Permas;
- fluid dynamics: ANSYS Fluent, ANSYS CFX, Star-CD, Star-CCM+, OpenFOAM.

All these codes are parallelized and should be started under control of the batch system requiring both the correct launching of the program and submitting it as a batch job. To facilitate this for the user, a command is provided which handles both tasks.

Other applications are

- COMSOL Multiphysics;
- Matlab;
- Pre/Postprocessing, Visualization: EnSight, HyperWorks, ICEM CFD.

11.1 ABAQUS

ABAQUS is a widely used program, based on the Finite Element technique, to solve problems covering the following features:

- linear and nonlinear stress/displacement problems,
- heat transfer and mass diffusion,
- acoustiscs,
- coupled problems (thermo-mechanical, thermo-electrical and more),
- all these problems may be static or dynamic (with implicit and explicit time integration),
- a large variety of material models are available,
- submodeling and substructuring,
- mesh adaption,
- design optimization,
- and much more.

A complete overview can be found in http://www.3ds.com/de/products.

The ABAQUS documentation can be accessed interactively by the command

```
abaqus doc
```

To start ABAQUS in batch modes the following command should be used:

```
abqjob -j ID -t TIME -m MEMORY [-c CLASS] [-T TIME] [-p PROCS] [-i FILE]
[-o OLD-JOB] [-f FILE] [-u USERSUB] [-s STRING] [-P PATH]
```

Parameters are:

-j *jobname*

-t *CPU-time in minutes*

-m *main memory in MByte*

-c *job-class* (p or d; default is p)

-T *real time in minutes* (optional)

-p *number of parallel tasks* (default is 1)

-i *inputfile* without .inp (optional)

-o *old jobname* on *RESTART and *POST OUTPUT (optional)

-f new or append

-u *user-subroutine*

-D *selection of the Direct Solver in ABAQus/Standard* (if p > 1): y or n (default is n)

-s *string* with further options

11.2 LS-DYNA

LS-DYNA is a general-purpose, implicit and explicit finite element program that is employed to analyze the nonlinear static and dynamic response of three-dimensional inelastic structures. Its fully automated contact analysis capabilities and error-checking features have enabled users worldwide to solve successfully many complex crash and forming problems.
In addition to LS-DYNA the tool LS-PREPOST for pre- and postprocessing is available. There are also well established interfaces to HyperWorks. More information about the program can be found in `http://www.dynamore.de`, where also manuals and tutorials are available.

The main applications are:

- Large Deformation Dynamics and Contact Simulations
- Large Deformation Dynamics and Contact Simulations
- Crashworthiness Simulation
- Occupant Safety Systems
- Metal Forming
- Metal, Glass, and Plastics Forming
- Multi-physics Coupling
- Failure Analysis

The submitting command is as follows:

`lsdynajob -j` *ID* `-t` *TIME* `-m` *MEMORY* `[-c` *CLASS*`] [-p` *PROCS*`] [-T` *TIME*`] [-s` *STRING*`]`

Parameters are:

-j *jobname*

-t *CPU-time in minutes*

-m *main memory in MByte*

-c *job-class* (p or d; default is p)

-p *number of parallel tasks* (default is 1)

-T *real time in minutes*

-s *string* with further options

11.3 MD Nastran

MD Nastran is also a finite element code to solve structural mechanics problems. A short description can also be found in `http://www.mscsoftware.com/products`. As pre- and postprocessors for Nastran models Patran and HyperMesh licenses are available.

The documentation is in PDF and can be found in the directory
/software/all/msc/nastran/md20081/doc/pdf_nastran/user/md_users_guide
on ic2.

The command is

`nastranjob -j` *ID* `-t` *TIME* `-m` *MEMORY* `[-c` *CLASS*`] [-p` *PROCS*`] [-e` *SCRATCHDIR*`]`
`[-T` *TIME*`] [-s` *STRING*`]`

Parameters are:

-j *jobname*

-t *CPU-time in minutes*

-m *main memory in MByte*

-c *job-class* (p or d; default is p)

-T *real time in minutes* (optional)

-p *number of parallel tasks* (default is 1)

-e *directory to store scratch files* ($WORK or $TMP; default is $WORK)

-s *string* with further options

The capabilities of the most CFD codes include

- stationary and instationary flow,
- laminary and turbulent flow,
- compressible and incompressible flow,
- multiphase and multiparticle flows,
- chemical reactions and combustion,
- newtonian and non-newtonian fluids,
- free surfaces,
- coupled heat transfer and convection,
- and many more.

11.4 PERMAS

PERMAS is also a general purpose finite element program, which the whole spectrum of functionalities of a widespread analysis code. A detailed description can be found in `http://www.intes.de`. Since PERMAS is a pure analysis code, model generation and results visualization must be performed by external programs. PERMAS offers a lot of interfaces to well-established pre- and postprocessors, such as MSC.Patran and HyperWorks. Currently the license is limited to one process with up to 8 parallel threads.

The documentation is online and can be accessed by input of the command `permasdoc`.

The PDF version is available in the directory
/software/all/intes/documentation/onldoc_v13.387/permas.

PERMAS is well parallelized in thread based mode. Therefore it cannot be processed on multiple nodes and the number of processors is limited to the number of cores of a single node.

PERMAS is invoked as a batch job by the command

`permasjob -j` *`ID`* `-t` *`time`* `-m` *`MEMORY`* `-c` *`CLASS`* `[-T` *`TIME`*`] [-p` *`PROCS`*`] [-e` *`SCRATCH`*`]`
`[-s` *`STRING`*`]`

Parameters are:

-j *projectname*

-t *CPU-time in minutes*

-m *main memory in MByte*

-c *job-class* (p or d; default is p)

-T *real time in minutes* (optional)

-p *number of parallel tasks* ($p \leq 8$, default is 1)

-e *specify the environment variable for scratch files* ($WORK or $TMP; default is $TMP)

-s *string* with further options (optional)

11.5 ANSYS Fluent

At the moment ANSYS Fluent version 14.5 is installed. The preprocessor 'Design-Modeler' and the Meshing-Tools of ANSYS (the Mesher in the Workbench and the ICEM_CFD) are available for Windows and several Linux distributions. For a graphical representation with ANSYS Fluent, the ANSYS Fluent code itself can be used or an installation of any visualisation code like e.g. EnSight are suitable. The documentation is provided interactively in the ANSYS Fluent GUI after pushing the Help button. General information is presented under
`http://www.ansys.com/Products/Simulation+Technology/Fluid+Dynamics/Fluid+Dynamics+Products/ANSYS+Fluent`.

The batch command is

`fluentjob -j` *ID* `-v` *VERSION* `-t` *CPU-time* `-m` *MEMORY* `[-c` *CLASS*`] [-T` *TIME*`] [-p` *PROCS*`]`

Parameters are:

-j *jobname*

-t *CPU-time in minutes*

-m *main memory in MByte*

-c *job-class* (p or d; default is p)

-T *real time in minutes* (optional)

-p *number of parallel tasks* (default is 1)

-v 2d|3d|2ddp|3ddp for different FLUENT versions

More information can be found at `http://www.scc.kit.edu/produkte/6724.php`.

11.6 ANSYS CFX

ANSYS CFX constists of 3 modules:

- CFX-Pre to import the mesh and formulate the model,
- CFX-Solver to configure and star the solver,
- CFX-Post to postprocess the results.

The mesh can be generated by codes like ANSYS ICEM_CFD or ANSYS Workbench, which must be installed on local sites. CFX-Pre and CFX-Post can be used interactively on local installations or on the login nodes of ic2. The solver should be operated in batch mode:

`cfx5job -j` *IDENT* `-t` *TIME* `-m` *MEMORY* `[-c` *QUEUE*`] [-R` *NAME*`] [-p` *PROCS*`]`
`[-s` *STRING*`] [-T` *TIME*`]`

Parameters are:

-j *jobname* without .def

-t *CPU-time in minutes*

-m *main memory in MByte*

-c *job-class* (p or d; default is p)

-T *real time in minutes* (optional)

-p *number of parallel tasks* (default is 1)

-R *name* of the result file for restart

-s *string* with further options

The documentation is available by the online help system or as PDFs in the directory /software/all/ansys_inc145/v145/commonfiles/help/en-us/help. More information can be found under `http://www.ansys.com/Products/Simulation+Technology/Fluid+Dynamics/Fluid+Dynamics+Products/ANSYS+CFX`.

11.7 ANSYS Mechanical APDL

ANSYS Mechanical APDL offers a comprehensive product solution for structural linear/nonlinear and dynamics analysis. The product offers a complete set of elements behavior, material models and equation solvers for a wide range of engineering problems. In addition, ANSYS Mechanical software offers thermal analysis and coupled-physics capabilities involving acoustic, piezoelectric, thermal-structural and thermal-electric analysis. A complete overview can be found in `http://www.ansys.com/Products/Simulation+Technology/Structural+Mechanics` and `http://www.scc.kit.edu/produkte/3866.php`.

The ANSYS documentation can be accessed interactively by the command `anshelp145` .

To start ANSYS Mechanical APDL in batch modes the following command should be used:

`ans145job [-p` *`PATH`*`] [-c` *`FILE18`*`] [-T` *`TIME`*`] [-M` *`MEM`*`] [-q` *`class`*`] [-j` *`xxxx`*`]`

Parameters are:

-p *name of the PATH where the input file resides* (optional) - please use the $HOME and $WORK environment variable

-c *name* of the input-file (essential)

-T *real time* in minutes (essential)

-M *main memory* in MB (essential)

-q d|p means usage of the development-pool and production-pool resp. (essential)

-j *xxxx* optionally changes the default job-name *filenn*.dat to *xxxxnn*.dat (maximum 4 characters)

11.7.1 Parallel jobs with ANSYS Mechanical APDL

First you have to write a small shell-script:

```
#!/usr/bin/sh
unset $(printenv | sed -n 's/^\(.*MPI.*\)=.*$//p')
export MPI_REMSH="/jms/bin/job_rsh"
cd working-directory
export MACHINES=`/software/all/ansys_inc145/scc/machines.pl`
ansys145 -j lal -dis -b -machines < Ansys-Input-File
```

The working directory could start with $WORK or $HOME.

Now you can submit the script to the batch system:

```
job_submit -t 5 -m 8000 -d t -c p -p 4 shell-script
```

The Job now starts with a time frame of 5 minutes (`-t 5`) and a demand of 8 GB of main memory (`-m 8000`). It runs on 4 CPU-cores (`-p 4`) at the production pool on thin nodes(`-c p -d t`). Please note, that the Script will only work at the production pool.

11.8 Star-CD

The Star-CD suite contains the meshing and modeling modules pro-STAR and pro-am (the automatic mesher). The solver can be started from the GUI of these modules or as a batch job:

`starcdjob -j` *CASE* `-t` *TIME* `-m` *MEMORY* `[-c` *QUEUE*`] [-p` *PROCS*`] [-s` *STRING*`] [-T` *TIME*`]`

Parameters are:

-j *case-name*

-t *CPU-time in minutes*

-m *main memory in MByte*

-c *job-class* (p or d; default is p)

-T *real time in minutes* (optional)

-p *number of parallel tasks* (default is 1)

-s *string* with further options

The parallelisation is licensed for up to 124 processors. The documentation is online available as PDF. The product's web site is `http://www.cd-adapco.com`

11.9 STAR-CCM+

STAR-CCM+ is parallel development to CD-adapco's STAR-CD CFD code with similar functionality but a complete different user interface and workflow. An overview can be found on the web site `http://www.cd-adapco.com`. In interactive mode STAR-CCM+ can be started by the command

```
starccm+
```

Be sure to provide enough memory by opening a Xterm on an exclusive node via a `job_submit` command. A STAR-CCM+ model may be prepared, the solution process should be performed as a batch job:

`ccm+job -j` *IDENT* `-t` *TIME* `-m` *MEMORY* `[-p` *PROCS*`][-c` *QUEUE*`][-T` *TIME*`]`

Parameters are:

-j *name of a simulation file filename*`.sim`

-t *CPU-time in minutes*

-m *main memory in MBytes*

-c *job-class (p or d; default is p)*

-T *real time in minutes (optional)*

-p *number of parallel tasks (default is 1)*

The complete documentation is online and available as PDF.

11.10 OpenFOAM

OpenFOAM (Open Source Field Operation and Manipulation) is an Open Source CFD Toolbox based on C++. There are a lot of libraries, called applications, which are ready for use as solvers and utilities. The main problem area to solve is CFD based on Finite Volumes, but mechanical stress-strain problems are also solvable.

Structured meshes may be generated by a block mesh generator, unstructured meshes may be generated by preprocessors like HyperMesh or ANSYS ICEM_CFD, but also ANSYS Fluent mesh files can be converted in OpenFOAM format. Results can be postprocessed by the open source ParaView or EnSight, Fieldview, ANYSY Fluent and others.

The OpenFOAM datasets to be processed must be provided in a certain structure, a so called case structure. Mesh and model description as well as output requests are to be formulated in so called dictionaries, which are files with prescribed entries.

OpenFOAM operates in parallel mode. In batch mode, a job is started by e.g.

```
job_submit -c p -p 16 -t 1000 -m 6000 "foamJob -p icoFoam"
```

where the `-p` option starts the solver in parallel mode, icoFoam is the solver for incompressible, laminar, newtonian fluids.

More information can be found in `http://www.scc.kit.edu/produkte/7023.php` where also links to important pages, documentation and tutorials are provided.

11.11 COMSOL Multiphysics

COMSOL Multiphysics is an application for almost all engineering regions based on the Finite Element Method. It is able to couple all physical areas like structural mechanics, fluids, heat etc.

Basically, COMSOL Multiphysics is interactively oriented and an access to the program goes over a GUI. Nevertheless it is possible to run COMSOL in batch mode and thus under the JMS environment using the `job_submit` command.

Create the model as usual via the GUI and save it as *filename*`.mph`. The form of a COMSOL job depends on the mode of parallelization. The COMSOL parallelization is thread based, which means one can specify the number of parallel tasks which reside on different nodes and communicate via MPI and each task tries to allocate as much as possible cores on its node for the threads. More information can be found in the COMSOL documentation and under `http://www.scc.kit.edu/produkte/3850.php`

The most comfortable and optimal way to start COMSOL jobs is to use the command comsoljob:

`comsoljob -i` *INPUT* `-o` *OUTPUT* `-t` *TIME* `-m` *MEMORY* `[-p` *PROCS*`] [-c` *CLASS*`] [-e` *SCRATCH*`] [-s` *STRING*`] [-T` *TIME*`]`

-i *Inputfile*

-o *Outputfile*

-t *CPU-time in minutes*

-m *main memory in MBytes*

-p *number of parallel tasks; (default is 1)*

-c *job class (p or d); (default is p)*

-e *specify the environment variable for scratchfiles ('$WORK' oder '$TMP'); (default is '$WORK')*

-s *string with further options (optional)*

-T *real time in minutes (optional)*

Examples:
`comsoljob -i filename.mph -o filename_out.mph -p` *nn* `-t 100 -m 6000`
results in a COMSOL job with *nn* tasks with 8 threads per task.

11.12 Matlab

Matlab is an extremely versatile program for problems covering the areas mathematics, engineering, biology, financial, statistics and a lot more. Matlab can be used interactively, but for large numerical problems it may be advisable to run it in batch mode. For this some feature should be deactivated and a m-File must be provided.

`matlab -nodesktop -nojvm -nosplash < filename.m`
or
`matlab -nodesktop -nosplash < filename.m`

runs a Matlab job without opening the usual desktop. Usually the Java Virtual Machine (JVM) should not be started, but sometimes it is required, so the option `-nojvm` must be omitted. The welcome screen is supressed. Any graphics from any commands in the m-File is also supressed. This command should be run under job_submit, especially if large memory and cpu times are needed and if the job should run multithreaded.

Further optimization of Matlab can be achieved

- by enabling the toolbox path cache: (File >> Preferences... >> General), check the boxes in the "Toolbox path caching" area and press the button;
- on computers or nodes with multiple processors Matlab will determine the maximum number of cores and will distribute threads on these by default. This should be considered by the parameter `-p 1/n` in the `job_submit` command. The number of threads can be specified explicitly by a command `maxNumCompThreads(n)` in the M-File; if multithreading should be prevented, the following option should be set as a Matlab startup option: `-singleCompThread`

The documentation is online available, the web site can be found in `http://www.mathworks.com/`.

11.13 Pre- and Postprocessors, Visualisation Tools

There are several tools for modeling, meshing and postprocessing. These are

- EnSight
- HyperWorks
- ANSYS

A detailed description is available on `http://www.scc.kit.edu/produkte` and links given there. There is no specific handling of these programs on ic2.

12 Batchjobs

As described in section 2 the majority of the nodes of InstitutsCuster II (ic2) is managed by the batch system.

Batch jobs are submitted using the command `job_submit`. The main purpose of the `job_submit` command is to specify the resources that are needed to run the job. `job_submit` will then queue the job into the input queue. The jobs are organized into different job classes like development or production. For each job class there are specific limits for the available resources (number of nodes, number of CPUs, maximum CPU time, maximum memory etc.). These limits may change from time to time. The current settings are listed with the command `job_info`. The command `job_queue [-l]` shows your queued jobs in standard or in long format. The command `job_wl` shows the workload (how many jobs are running - how many jobs are waiting) of InstitutsClusterII.

Important Batch commands	Brief Explanation
`job_submit`	submits an job and queues it in an input queue.
`job_cancel`	cancels an job from the input queue or a running job.
`job_info`	shows the different input queues and their specific limits for the available resources.
`job_queue`	shows your queued or running jobs in standard or long format.
`job_wl`	shows the workload of InstitutsClusterII.

For all the above mentioned commands there are manual pages are available; thus e.g. `man job_submit` can be called.

When the resources requested by a certain job become available and when no other job with higher priority is waiting for these resources, then the batch system will start this job.

12.1 The `job_submit` Command

The syntax of the `job_submit` command is available with

`job_submit -H`

The most important options are:

`job_submit -t` *time* `-m` *mem* `-c` *class[+]* `-p` *i*`[/`*j*`] [-T` *time*`] [-M` *mem*`] [-J "`*jobname*`"]`
`[-l af|aF|Af|AF] [-A` *account*`] [-N[s][b][c|C|e|E]] [-i` *file*`] [-o` *file*`]`
`[-e` *file*`|+] [-d[t|m|f]] [-x[+|-]]` *job*

`-t time:` maximum CPU time (minutes) on each CPU that is allocated to the job. The job will be terminated when one task exceeds its CPU time limit.

`-T time:` maximum elapsed time (minutes). The job will be terminated, when this time is exceeded. For many applications the elapsed time will not be much higher than the CPU time. Exceptions are I/O intensive applications which need a much higher elapsed time than CPU time. If this option is omitted, the default value is a function of the requested CPU time ($T = 1.01 * t + 1$; t is time from -t).

`-m mem:` maximum memory requirement per task in Mega Bytes. The 16-way nodes of ic2 are equipped with 64 GB or 512 GB of main memory.

`-M mem:` maximum virtual memory requirement per task in Mega Bytes. This option allows users to use the memory management of the operating system and can strongly downgrade the system performance, if it is not properly used. So this option is only available for special users.

`-J "jobname":` the job gets the name *jobname*. *jobname* is an arbitrary string of maximum 16 chars.

`-l af|aF|Af|AF:` the sign a or alternatively A means that account information is switched on or off; the sign f or alternatively F means that displaying of floating point exceptions is switched on or off. Default is `-l af`.

`-A account:` additional accounting information (only for special customers). *account* is a text string.

`-N[s][b][c|C|e|E][:mailaddress]:` this option allows the automatic sending of mails on the basis of events:

- **`s:`** submitting the job triggers the sending of a mail to *mailaddress*.
- **`b:`** starting the job triggers the sending of a mail to *mailaddress*.
- **`c:`** complete end of the job triggers the sending of a mail to *mailaddress*. Begin and end of STDOUT and STDERR will be sent by mail.
- **`C:`** complete end of the job triggers the sending of a mail to *mailaddress*. Complete STDOUT and STDERR will be sent by mail.
- **`e:`** Erraneous end of the job triggers the sending of a mail to *mailaddress*. Begin and end of STDOUT and STDERR will be sent by mail.

`E:` Erraneous end of the job triggers the sending of a mail to *mailaddress*. Complete STDOUT and STDERR will be sent by mail.

If the mailaddress is omitted the mailaddress bound to the userid will be chosen.

`-p i[/j]:` number of tasks (i) and threads per task (j)

default: $j = 1$

This option defines how many processors are required to run the job.

- If the job is single threaded, i.e. it is a serial or a pure MPI program without any usage of OpenMP or other multithreading techniques, then one CPU per task is needed. The format of this option is `-p` *i* where *i* is the number of tasks.
- When the program is an OpenMP parallelized program which does not contain any MPI calls, then the number of tasks is 1 and the number of threads must not exceed 16. The format of `-p` option in this case is `-p 1/`*j* where *j* is the number of threads.
- When both parallelization techniques (e.g. MPI and OpenMP) are used, then *i* is the number of MPI tasks and *j* is the number of threads per MPI task. The command `job_info` shows the valid combinations of *i, j* and *mem*.

`-c class[+]:` this option defines the job class. The sign + means higher priority (only available for special customers). Two job classes are available on Institutscluster II (ic2):

`d:` jobs in this class will start immediately, but do not have exclusive access to any resources of ic2. All cores of this class are operated in mode 'shared', i.e. multiple tasks can be executed simultaneously on a single core. Performance measurements are not reasonable in this class.

The class `d` is typically used for program development and test.

`p:` jobs in class production are distinguished by the exclusive access of nodes or cores. Thus two different operating modes can be chosen. In the first operating mode - exclusive use of nodes - always whole nodes, i.e. all 16 cores of one node, are accessed. This can lead to unused cores. In the second operating mode - exclusive use of cores - only several cores of one node are used exclusively. This implicitely means that unused cores of the node can be used by another program and thus the memory is not used exclusively.

`-i stdin_file:` when the option `-i` *stdin_file* has been selected the *job* will be executed as if *job* < *stdin_file* would have been specified (default: /dev/null).

`-o stdout_file:` the standard output of the job is written to the file named in this option. When the `-o` option is omitted the default output file is `Job_$JID.out` where `$JID` is a unique identification number of a job. It is created when `job_submit` is launched.

`-e stderr_file:` the error messages of the job are written to the selected file. If this option is omitted all error messages are written to a file `Job_$JID.err`. When a job does not generate any error messages, then the standard error file is deleted at job termination. Choosing `-e +` means to concatenate standard error with standard output, i.e. to write standard error into the standard output file.

`-d t|f:` this option should be used carefully. If `-d t` is chosen, thin nodes (i.e. nodes with 64 GB main memory) will be used. If `-d f` is chosen, fat nodes (i.e. nodes with 512 GB main memory) will be used. If this option is omitted the batch system decides if thin nodes or medium nodes or a singular fat node will be allocated to run the job. The batch system can also automatically migrate the job from thin nodes to a singular fat node and vice versa, if this is possible under the given conditions in terms of required processors and main memory.

`-x[+|-]:` this option should be used carefully. If `-x+` is chosen, exclusive access of nodes will be used. This means that no further executables will run on the requested nodes and that all nodes will completely charged onto your account. If `-x-` is chosen, exclusive use of nodes will be switched off. This means that further executables can run on the requested nodes (but not on the requested cores) and that only the processors which are used by your executable will be charged onto your account. If this option is omitted the batch system decides if exclusive use of nodes or exclusive use of cores will be chosen to run the job.

`job:` this is a parallel program call or a shell script to be executed on ic2. When the invocation of the job requires additional arguments, the parallel program call or the script with arguments may be enclosed in quotes or double quotes, but they can also be omitted.

Important remark: please read this paragraph before starting jobs in the production pool! First, the production pool contains nodes with 16 and 32 cores. Second, it always holds the equation: number_of_requested_processors = number_of_requested_tasks * number_of_requested_treads ($p = i * j$). If you are asking for more than 16 (32) cores the batch system must allocate more than one node. In this case exclusive use of nodes is chosen automatically (`-x+`). If you want to switch off exclusive use of cores you must choose the option `-x-`.
If you are asking for 16 (32) or less cores and for 64 (512) GB or less of main memory, then the batch system decides that your job will run within one node. If you are using $p < 16$ ($p < 32$) cores, then be aware that $16 - p$ ($32 - p$) cores are idling (and completely accounted)! In this case "shared" use of cores is chosen automatically (`-x-`). If you want to switch on exclusive use of the whole node you must choose the option `-x+`.

12.2 Environment Variables for Batch Jobs

Parameters can also be set by environment variables. The syntax is `export JMS_`*`parameter`*`=value`. Examples are: export JMS_t=10; export JM_o=stdout_file; export JMS_job="mpirun a.out". Parameters set in the command line overwrite parameters set by the environment. The command `job_submit` replaces chosen parameters by the appropriate environment variables and exports them to the user job.

Now some useful environment variables will be explained. You can get the complete list of environment variables by calling the shell command `set` in a batch job.

Environment Variable	Brief Explanation
`JMS_t`	contains the value of the CPU time limit which has been defined with option `-t`. This value can be used to compute the amount of CPU time within a program that is still available for the computation.
`JMS_T`	contains the maximum elapsed time as specified with the option `-T`.
`JMS_m`	contains the memory requirement as specified with the `-m` option.
`JMS_Nnodes`	contains the allocated number of nodes.
`JMS_p`	contains the requested number of processors.
`JMS_tasks`	contains the number of MPI tasks specified with the first value i in the `-p` option.
`JMS_threads`	contains the number of threads per task (process) as specified with the second value j in the `-p` option. After setting `JMS_tasks` the following assignment is done: `OMP_NUM_THREADS=$JMS_tasks`
`JMS_c`	defines the job class as specified with option `-c`.
`JMS_start_time`	gives the starting time of the job if it is in state running.
`JMS_submit_time`	gives the time at which the job has been submitted.
`JMS_submit_node`	contains the name of the node the job has been started on.
`JMS_node0`	contains the name of the first node.
`JMS_nodes`	lists all used node names. If e.g. 8 processors per node are used the used nodes are listed eight times.
`JMS_stdin`	contains the name of the input file of the job.
`JMS_stdout`	contains the name of the output file of the job.
`JMS_stderr`	contains the name of the standard error file of the job.
`JMS_pwd`	contains the name of the output directory of the job.
`JMS_user`	contains the userid of the user who has submitted the job.
`JMS_group`	contains the groupid of the user who has submitted the job.
`TMP and TEMP`	contain the working directory for temporary files of the job.

12.3 `job_submit` Examples

12.3.1 Serial Programs

1. To submit a serial job that runs the script `job.sh` and that requires 5000 MB of main memory, 3 hours of CPU time and 4 hours of wall clock time the command

   ```
   job_submit -t 180 -T 240 -m 5000 -p 1 -c p job.sh
   ```

   may be used. The high wall clock time (`-T 240`, i.e. 4 hours) is necessary when the program does a lot of I/O. In most other cases the wall clock time will only be slightly larger than the CPU time (`-t` option). Usually your job will be running on a thin node with 64 GB main memory, but it is also possible that it will be running on a fat node (512 GB main memory), if all thin nodes are just running jobs and fat nodes are idling.

2. Now we want to resubmit the same job, but a certain argument, e.g. `-n 100` has to be passed to the script, i.e. the command `job.sh -n 100`, has to be executed within the batch job. The appropriate `job_submit` command is now:

   ```
   job_submit -t 180 -T 240 -m 5000 -p 1 -c p "job.sh -n 100"
   ```
   or
   ```
   job_submit -t 180 -T 240 -m 5000 -p 1 -c p job.sh -n 100
   ```

12.3.2 Parallel MPI Programs

For your understanding you must know: **parallel programs (no shell scripts) must be launched by calling** `mpirun` *parallel program*; **shell scripts only run on the first processor!**

1. We want to run 4 tasks of the program `my_par_program` within a batch job in the job class development. Each task has a CPU time limit of 10 minutes and the memory requirement per task is 3000 MB. The wall clock time limit is set to 1 hour. This may be necessary when the nodes for the development class are heavily loaded and many other processes are using these nodes at the same time. The appropriate `job_submit` command is

   ```
   job_submit -t 10 -T 60 -m 3000 -p 4 -c d "mpirun my_par_program"
   ```
   or
   ```
   job_submit -t 10 -T 60 -m 3000 -p 4 -c d mpirun my_par_program
   ```

2. The same program will now be run in the production class on 16 processors. The maximum CPU time is 4 hours, the memory requirement per task is 7000 MB, thin nodes are chosen and exclusive use of nodes is switched off.

   ```
   job_submit -t 240 -m 7000 -p 16 -c p -d t -x- mpirun my_par_program
   ```

   As one node only contains 64 GB and the memory requirement for 16 tasks is 112 GB, only 8 tasks will be run on one node. Thus the job runs on 2 nodes. On the other 8 cores of each of the 2 nodes further jobs with the overall memory requirement of 8 GB per node can be run.

3. A third job sample includes the following functions:

   - create a subdirectory `Job_Output` within `$WORK`,
   - select `$WORK/Job_Output` as current working directory,
   - run 64 tasks of the program `my_par_program` (CPU time limit: 3 hours, memory requirement per task: 6000 MB). The program `my_par_program` is stored in `$HOME/bin`.

   In order to accomplish this, a shell script is needed. Let `job.sh` be the name of this script. Its content is:

   ```
   #!/bin/sh
   #
   cd $WORK
   ```

```
if [ ! -d "Job_Output" ]
then
  mkdir Job_Output
fi
cd Job_Output

mpirun $HOME/bin/my_par_program
```

To submit this job, use the commands

```
chmod u+rx job.sh
job_submit -c p -p 64 -t 180 -m 6000 job.sh
```

The `if` statement enables the user to run this job several times. It will not abort while trying to create a directory `Job_Output` that already exists.

When the script `job.sh` is executed, the master node of this job will run the shell commands like `cd` or `mkdir` and any other serial commands or programs. Only parallel programs launched by the command `mpirun` will be executed on all these processors that are requested by the `-p` option of the command `job_submit`. The job will run on 4 nodes (16 tasks per node) with exclusive use of nodes. The job can only run on thin nodes. As only jobs running on maximum 32 processors (`-p 32`) are allowed on fat nodes, the job can not run on this type of node. Adding the option `-x-` will take no effect because there are no idle cores on the requested nodes.

4. In order to run several parallel and serial programs within one batch job, again a shellscript is needed which contains the commands to start the programs.

   Let us assume that we want to run the two parallel programs `my_first_parallel_prog` and `my_second_parallel_prog`. Both programs are stored in the directory `$HOME/project/bin`. Before starting the second program we want to copy a file `results_1` from the current working directory, which is `$WORK`, into the `$HOME` directory. The job script `job_2.sh` may now look like this:

```
#!/bin/sh

cd $WORK

mpirun $HOME/project/bin/my_first_parallel_program

if [ "$?" = "0" ]
then
#    program terminated successfully, copy data and start next program

  cp results_1 $HOME/results_1

  if [ "$?" = "0" ]
  then
#    file result_1 has been copied successfully, next program may be started

    mpirun $HOME/project/bin/my_second_parallel_program
  else
    echo 'File results_1 could not be copied into $HOME directory'
    exit 1
  fi
else
  echo 'Program my_first_parallel_program terminated abnormally'
  exit 2
fi
```

To run this script on 32 cores (CPU time limit: 4 hours, memory limit: 14000 MB) use the `job_submit` command:

```
job_submit -c p -p 32 -t 240 -m 14000 -d t job_2.sh
```

Within this job first the `cd` command is executed on the master node. Then the program `my_first_parallel_prog` is executed with 32 MPI tasks on 8 nodes (8 x 4 cores per node) on thin nodes. The default mode - exclusive use of nodes - is chosen. So 12 cores per node are idling. When all tasks have been terminated, the master node copies the file `results_1` into the `$HOME` directory before the parallel execution of `my_second_parallel_prog` is initiated.

12.3.3 Multithreaded Programs

For programs based on OpenMP the OpenMP specification defines an environment variable `OMP_NUM_THREADS` to select the number of threads. For details on these variable see the documentation of Fortran and C compiler at `http://www.scc.kit.edu/dienste/4983.php`.

The variable `OMP_NUM_THREADS` is automatically initialized by `job_submit` to the value of j as specified by the option `-p` i`/`j.

The following examples illustrate the usage of `job_submit` command with multithreaded applications:

1. run the program `my_openmp_prog` with 2 threads, a CPU time limit of 4 hours per thread and a memory requirement of 2 GB:

   ```
   job_submit -c p -p 1/2 -t 240 -T 300 -m 2000 my_openmp_prog
   ```

   Because this program has not been parallelized with MPI, it consists of one single process that is split into several (in this case) 2 threads. This is described by the option `-p 1/2`.

   The operating system computes the CPU time on a per process basis, i.e. the CPU times of all threads of a process are added. To reflect this, the `job_submit` command multiplies the requested CPU time by the number of threads. In this case we have a limit of 8 hours. If there is a good load balance between the threads, each thread may consume approximately 4 hours of CPU time. Since the job will run on one node and the option `-x` is omitted, the default mode - exclusive use of cores - will be chosen.

   If there is a poor load balance among threads the available time for this job is limited by the wall clock time, i.e. five hours (300 minutes).

2. Now we want to run the same program with 6 threads and 3 hours of CPU time per thread and again 2 GB of main memory. In addition we want to select dynamic scheduling, i.e. the amount of work in parallelized loops is dynamically assigned to 6 threads. In a shell script the environment variable `OMP_SCHEDULE` is set to `dynamic` to inform the runtime system about dynamic scheduling.

   The script `openmp_job.sh` looks like:

   ```
   #!/bin/sh
   #
   export OMP_SCHEDULE="dynamic"
   #
   my_openmp_prog
   ```

   It is submitted to the batch system with the command

   ```
   job_submit -c p -p 1/6 -t 180 -T 300 -m 2000 openmp_job.sh
   ```

12.3.4 Programs using MPI and OpenMP

When running programs using distributed memory parallelism (e.g. MPI) as well as shared memory parallelism (e.g. OpenMP) both arguments (i and j) of the `-p` option of the `job_submit` command must be specified.

1. To run the program `my_parallel_program` on 32 16-way nodes with sixteen threads per node and 24 GB of main memory per MPI task, the `job_submit` command may look like:

   ```
   job_submit -c p -t 240 -T 300 -m 24000 -p 32/16 mpirun my_parallel_prog
   ```

2. To run a program with 2 MPI tasks and 4 threads and 24 GB of main memory per MPI task on one 16-way node, the `job_submit` command will be:

   ```
   job_submit -c p -t 240 -T 300 -m 24000 -p 2/4 mpirun my_parallel_prog
   ```

   The job can run on one thin node or one fat node! Choosing the option `-d t` means that the job will run on 1 thin node.

12.4 Commands for Job Management

There exist several commands to list, cancel or query jobs. To identify an individual job, a unique job-id which is determined by `job_submit` is associated with each job.

The `job_submit` command returns a message

`job_submit: Job` *job_id* `has been submitted.`

The *job_id* is the unique identification of this specific job and will be used in all job management commands to identify the job.

The job management commands to list, cancel or query jobs are `job_queue`, `job_cancel` and `job_info`.

`job_queue` [**-l**] The output of `job_queue` lists all own jobs, its job identification and its specific requirements. If you are using the option `-l`, further informations will be printed.

A sample output of `job_queue` is:

```
job-id  c  P     n/i/j     t     T     m    queued   s    start    end(t)
------------------------------------------------------------------------
   474  p  t      1/8/1    10    11  2000  22/15:57  r  22/15:57  22/16:07
```

- The first column shows the complete job identification. To select a job within the job management commands it is sufficient to specify the job-id (it is a numerical value).
- The second column displays the job class (column `c`). The job in this example belongs to class production (`p`). Jobs in class development will show a `d` in this column.
- The next column displays the partition (column `P`) the job runs in. The job in this example runs in the partition "thin nodes" (`t`). Another possible partition is `f` for "fat nodes".
- The next `n/i/j` shows how many nodes, MPI-processes and threads per node are requested for this job.
- The next two columns show the requested time in minutes.
- The column `m` shows the requested memory.
- The column `queued` and `start` and `end` show the times when the job is queued and when it has been started and when it will end at the latest time in the format day/wall clock time.
- The column `s` gives the status of the job which is `r` for running , `w` for waiting or `L` for looping in job chains (see next section).

`job_cancel` deletes a waiting job from the input queue or aborts a running job.

To delete the job with job-id 565 from the input queue, just enter the command

```
job_cancel 565
```

`job_info` lists the current settings for the job classes, i.e. the limits for CPU time, number of nodes, amount of memory, number of jobs per user etc.

These values may change from time to time reflecting the varying requirements and available resources.

12.5 Job Chains

The CPU time requirements of many applications exceed the limits of the job classes. In those situations it is recommended to solve the problem by a job chain. A job chain is a sequence of jobs where each job automatically starts its successor. To implement a job chain, the program must be prepared for restarting and the job script must contain some additional statements for file management and for starting of the next member in the chain.

A program that enables a restart functionality must write the intermediate results, that are needed to resume the computation, to an output file before a time limit is reached. This results in the following structure of the program:

```
if (first_run ) then
    initiate computation
else
   read restart_file
end if

main_loop: do

   compute next_step

   if (time_limit reached) then

      write new_restart_file
      terminate program

   end if

end do main_loop
```

The job script that implements a job chain contains a `job_submit` command to resubmit the same script again or to submit a different job script. The script must also save and rename the restart files. Care must be taken, that the chain can easily be restarted when it has been broken accidentally.

The following rules are especially important for job chains:

- The `job_submit` command to submit the next member of the job chain should always be activated at the end of a job. `job_submit` will check if the time interval from start of the job until submission of the next job exceeds a certain threshold value (30 seconds). If not, the submitted job will not run automatically. This feature will prevent job chains from looping, i.e. by mistake every few seconds a new job may be submitted without doing any reasonable work. Looping job chains are indicated by an `'L'` in the status column of the `job_queue` output.

12.5.1 A Job Chain Example

Within a batch job we want to run the parallel program `my_par_prog`. The program writes its intermediate results to a file named `restart`. This file is the input file for the next step in the job chain, i.e. the next invocation of `my_par_prog` will read this file. Each single job in the job chain will use 4 hours of CPU time. The job chain will terminate, when 20 jobs have been executed. The job script is `job_chain_1.bash`.

```
#!/bin/bash
#
# Sample job scripts job_shain_1.bash
# implements a simple job chain
#
# The chain will terminate after at most 20 runs.
```

```
#
MAX_JOBS=20
#
if [ "$JOB_COUNTER"  = "" ]
then
  echo "=========================================================="
  echo "=       Variable JOB_COUNTER not initialized             ="
  echo "=                job chain aborted!                      ="
  echo "=========================================================="
  exit 1
fi
#
# Run the program my_par_prog and save the return code in RETURN
#
mpirun my_par_prog
RETURN=$?
#
# Check the return code of my_par_prog
#
if [ "$RETURN" = "0" ]
then
#
#    program terminated successfully
#     - save restart file,
#
  cp restart Restart_Files/restart_$JOB_COUNTER
  RETURN_CP=$?

# Check return code of cp command

  if [ $RETURN_CP != "0" ]
  then
    echo "=========================================================="
    echo "=    Copy command failed, restart file not saved!        ="
    echo "=                job chain aborted!                      ="
    echo "=========================================================="
    exit 2
  fi
#
# Restart files may also be copied into tape archive using tsm_archiv command

# archive Restart_Files/restart_$JOB_COUNTER
# RETURN_ARCHIVE=$?

# check return code of archive command

#  if [ "$RETURN_ARCHIVE" != "0" ]
#  then
#    echo "=========================================================="
#    echo "= Archive command failed, restart file not archived! ="
#    echo "=                job chain aborted!                      ="
#    echo "=========================================================="
#    exit 3
#  fi

# Old restart files may be deleted here if they are no longer needed.
```

```
#  if [ $JOB_COUNTER -gt 1 ] ; then
#    rm Restart_Files/restart_‘expr $JOB_COUNTER - 1‘
#  fi
#
# - increment JOB_COUNTER
#
  JOB_COUNTER=‘expr $JOB_COUNTER + 1‘

# - submit next job
#
  if [ $JOB_COUNTER -lt $MAX_JOBS ]
  then
    job_submit
# The new job will be submitted with the same parameter as the last job
# using JMS_ environment variables!!!
  else
    echo "============================================================"
    echo "=        job chain completed successfully                  ="
    echo "============================================================"
  fi
  exit $?
else
#
# my_par_prog terminated with nonzero return code
#
  echo "============================================================"
  echo "=   my_par_prog terminated with return code $RETURN  ="
  echo "=               job chain terminated                       ="
  echo "============================================================"
fi

exit $RETURN
```

To initiate this job chain the following command must be entered:

```
export JOB_COUNTER=0
job_submit -c p -m 1000 -t 240 -p 64 job_chain_1.bash
```

In the above job script the file `restart` is copied to the directory `Restart_Files` and the value of `JOB_COUNTER`, i.e. the actual number of this job within the job chain, is appended to the file name. It must be checked carefully if this command has been completed successfully. All results computed so far are stored in restart files. These files should be saved from time to time.

When this job chain has been interrupted, it can easily be restarted manually. The latest restart file has to be copied from the directory `Restart_Files` to the working directory and must be renamed `restart`. Then the environment variable `JOB_COUNTER` must be set to the correct values, i.e. the number of jobs that have been completed successfully. Now the job chain can be started again with the `job_submit` command.

Instead of running a job a fixed number of times within a job chain, another method may be to terminate the chain when the program signals a successful completion of the whole computation by a nonzero return code.

12.5.2 Get remaining CPU Time

One problem with a job chain is to determine the amount of time that is still available for computation. The Fortran subroutine `time_left` will compute this value assuming that all the CPU time is spent in one program. An application program may call this subprogram and write the restart file and terminate execution, when the remaining time falls below a certain limit.

```
      SUBROUTINE time_left (time_remaining)

! time_left computes the difference between the environment variable
! $JMS_t and the CPU time consumed from start of the program.

      IMPLICIT NONE

      include 'mpif.h'

      REAL time_remaining

      REAL cputime, max_cpu_time

      CHARACTER*10 string_max_time

      INTEGER ierror, max_time

! Get CPU time consumed by each task and compute the maximum value

      CALL cpu_time (cputime)

      CALL MPI_Allreduce (cputime, max_cpu_time, 1, MPI_REAL,
     &                    MPI_MAX, MPI_COMM_WORLD, ierror)

! getenv delivers the value of environment variable JMS_t

      CALL getenv ('JMS_t', string_max_time)

! Convert this value into integer format

      READ (string_max_time, *) max_time

! Compute the remaining CPU time

      time_remaining = REAL(max_time)*60. - max_cpu_time

      END
```

13 Technical Contact to SCC at KIT

- **IC Hotline**
  Email: ic-hotline@lists.uni-karlsruhe.de
  Phone: +49 721 608-48011