Best Practice mini-guide "JUROPA"

Jülich Research on Petaflop Architectures

Information provided by:
Forschungszentrum Jülich GmbH

Compiled from: http://www.fz-juelich.de

Editor: Maciej Szpindler
ICM University of Warsaw

March 2013

Table of Contents

1. Introduction

The supercomputer JUROPA is based on a cluster configuration of NovaScale servers from the French computer specialist Bull, and on blade servers from the American company Sun with Intel Nehalem processors. The system was designed by experts from the Jülich Supercomputing Centre and implemented together with the partner companies Bull, Sun, Intel, Mellanox and ParTec. It consists of 2208 computing nodes with a total computing capacity of 207 Tflop/s. JUROPA's operating system was developed by Forschungszentrum Jülich and ParTec. JUROPA is financed by Forschungszentrum Jülich within the framework of funding by the Helmholtz Association.

Figure 1. JUROPA compute racks, source: Forschungszentrum Jülich

2. System architecture and configuration

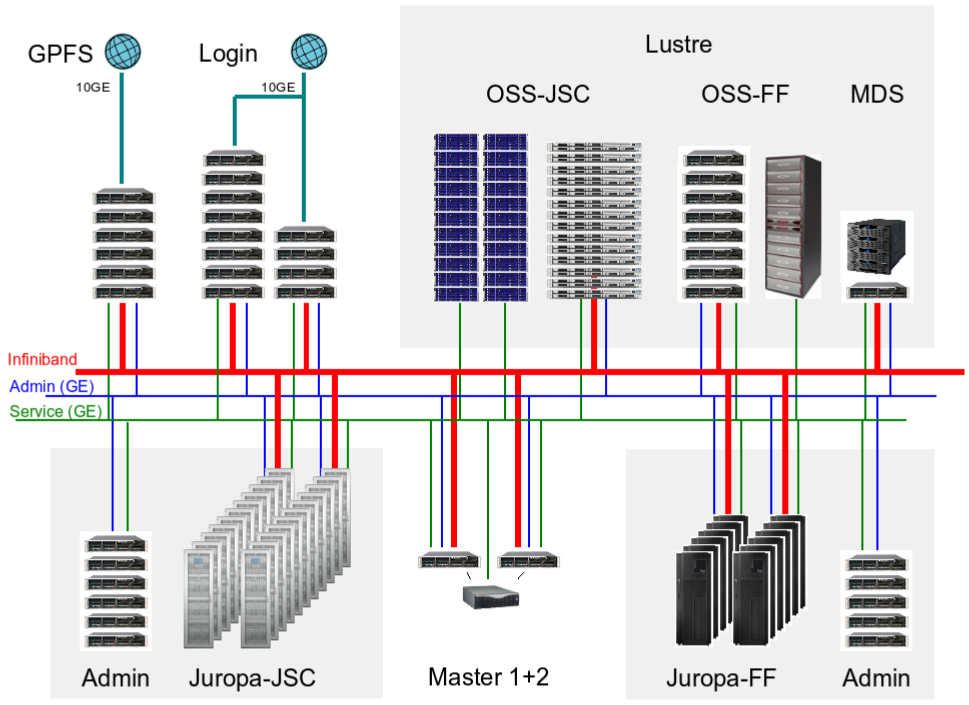
2.1. System configuration

JUROPA is FZJ/GCS production system based on the Sun Constellation System with a total peak performance of 207 Tflop/s. The cluster consists of compute nodes connected via high-speed InfiniBand QDR network and utilizes a Lustre storage subsytem. JUROPA has 2208 computing nodes with a following characteristics:

- Sun Blade 6048 system
- 2 Intel Xeon X5570 (Nehalem-EP) quad-core processors
- Processor clocked at 2.93 GHz with SMT (Simultaneous Multithreading also known as Intel Hyper-Threading)
- 24 GB of DDR3 physical memory
- InfiniBand QDR networking module

In total, JUROPA is composed of 2208 computing nodes with 17664 cores.

Figure 2. JUROPA architecture, source: Forschungszentrum Jülich



Processor architecture

JUROPA compute nodes are equipped with a two Intel Nehalem-EP (Xeon quad-core X5570) processors. The Intel Xeon X5570 processor is a 64-bit instruction set (x86_64) processor with the following architecture details:

- 4 physical cores (quad-core)
- 2-way SMT (hardware support for 8 concurrent threads)
- clock speed: 2.93 GHz
- caches:
  - L1: 64 KB per core
  - L2: 256 KB per core
  - L3: 8 MB shared by 4 cores
- memory bandwidth at the level of the processor: 32 GB/s

JUROPA is using login nodes for interactive work and job submission. Additionally GPFS nodes are in use for data exchange between the different HPC systems.

Operating system

The operating system used is Linux SUSE SLES 11 with ParaStation Cluster Management system.

Memory architecture

Each computing node contains a total of 24 GB DDR3 memory, clocked at 1066 MHz. With three processor memory channels the maximum possible memory bandwidth is 32 GB/s.

Network infrastructure

The JUROPA network uses a Fat Tree topology for InfiniBand inter-connect between the computing nodes and other system components. The network tree is composed of 92 blocks, 24 nodes each.

I/O subsystem

JUROPA is using a Lustre storage subsystem which is mounted on login nodes, compute nodes and GPFS nodes. Lustre Storage Pool has the following components:

- 4 Meta Data Servers (MDS)
  - 2 x Bull NovaScale R423-E2 (Nehalem-EP quad-core)
  - 2 x Bull NovaScale R423-E2 (Westmere-EP, 6-core)
  - 98 TB for meta data (2 x EMC² CX4-240)
- 14 Object Storage Servers (OSS) for the home file systems
  - Sun Fire X4170 Server
  - 500 TB user data (28 x Sun Storage J4400 Array)
- 8 Object Storage Servers (OSS) for the home file systems
  - 8 x Bull NovaScale R423-E2 (Nehalem-EP quad-core)
  - 500 TB user data (2 x DDN SFA10000 storage)
- 8 Object Storage Servers (OSS) for the scratch file system
  - 8 x Bull NovaScale R423-E2 (Westmere-EP, 6-core)
  - 834 TB user data (2 x DDN SFA10000 storage)
- Aggregated data rate ~50 GB/s
- Overall storage capacity: 1.8 PB

2.2. Filesystems

The Lustre Storage Pool is available on login, compute and GPFS nodes with the following file systems:

Table 1. Lustre filesystems on JUROPA

Filesystem	Capacity	Limits	Backup service	Usage
$WORK	800 TB	group quota: 3 TB, 2 million files	no backup, files older than 28 days will be deleted	recommended for large temporary files and high performance requirements
$HOME	29 TB per file system, distributed among user groups	group quota: 3 TB, 2 million files	daily backup	recommended for permanent program data with low performance requirements

Home directories reside in the Lustre filesystem. In order to hide the details of the home filesystem layout the full path to the home directory of each user is stored in the shell environment variable $HOME. References to files in the home directory should always be made through the $HOME environment variable. The initialization of $HOME will be performed during the login process.

The other important storage locations is the GPFS archive filesystem ($GPFSARCH).

Table 2. Archive filesystem on JUROPA

Filesystem	Limits	Usage
$GPFSARCH	group quota: 2 million files	Storage for large files that are used infrequently. Long-term storage is done on magnetic tape. Thus, the retrieval of data may incur a significant time-delay. Backup of all files located in this file system is performed on a daily basis.

Table 3. Lustre and GPFS filesystems visibility on JUROPA

Primary name	Mount point	Type	Access
$WORK	`/lustre/jwork`	Lustre	all nodes
$HOME	`/lustre/jhome1 .. 14`	Lustre	all nodes
	`/usr/local`	Lustre	all nodes
$GPFSARCH		GPFS	GPFS nodes only

It is highly recommended to access files always with help of these variables. The values of these variables are automatically set during login.

Further details

For more information related to Section 2, “System architecture and configuration” please refer to the FZJ website:

- JUROPA Configuration [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/Configuration/Configuration_node.html]

3. System Access

Login nodes

JUROPA is using a set of login nodes for accesing the system. Users should use ssh connections for accessing the system, using:

ssh [-X] user@juropa.fz-juelich.de

When using the generic name, a connection is established to one of the Login nodes in the node pool (`juropa01 ... juropa07`). Initiating two logins in sequence may lead to sessions on different nodes. In order to force the session to be started on the same nodes use the specific node names instead, for example when using the login node <code>juropa01</code> (accordingly for the other login nodes):

ssh [-X] user@juropa01.fz-juelich.de

Login procedure

Users can't login by suppling username/password credentials. Instead, login based on SSH key exchange is required.

The public/private ssh key pair has to be generated. On Linux or UNIX-based systems, the key pair can be generated by executing:

ssh-keygen -t [dsa|rsa]

It is not allowed to use ssh key without a passphrase! Please protect the ssh key with a non-trivial pass phrase to fulfill the FZJ security policy.

Intrusion control

Note that too many accesses (ssh or scp) within a short amount of time will be interpreted as an intrusion and will lead to automatically disabling the originating system at the FZJ firewall. For transferring multiple files in a single scp session, the -r option can be used, which allows to transfer whole directories.

If X11-based graphical tools are to be used on JUROPA, it is necessary to enable X11 forwarding in the file `/etc/ssh/config` or `$HOME/.ssh/config` on your workstation or to use the <code>-X</code> option when connecting to JUROPA.

Further details

For more information related to Section 3, “System Access” please refer to the FZJ website:

- JUROPA Access and Environment [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/UserInfo/Access.html]

4. User environment and programming

4.1. Generic x86 environment

JUROPA is using an Intel programming environment including Intel Professional Fortran, C/C++ Compiler, Intel Cluster Tools Intel Math Kernel Library and OpenMP Intra-Node Programming Model.

It is recommended to use the Intel compilers in order to get executables with highest possible performance for these platforms.

Following libraries are available on the JUROPA:

- Numerical libraries:
  - NAG Libraries, LAPACK, GSL, FFTW, GMP,
- Parallel libraries for distributed Memory Machines:
  - ScaLAPACK, PARPACK, PETSc, MUMPS, SPRNG, ParMETIS, hypre, sundials.

The common programming environment is maintained with the concept of software modules in directory `/usr/local`. The framework provides a set of installed libraries and applications (including multiple-version support) and an easy to use interface (module command) to set up the correct shell environment.

Modules environment

For more information on using `Modules` please refer to the PRACE Generic x86 Best Practice Guide [http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Generic-x86.pdf].

The persistent settings of the shell environment are governed by the content of `.bashrc`, `.profile` or scripts sourced from within these files. Please use these files for storing your personal settings.

4.2. System specific environment

4.2.1. Batch system

The batch system on JUROPA is Moab with the underlying resource manager Torque. The batch system is responsible for managing jobs on the machine, returning job outputs to the user and provides job control mechanism on the user's or system administrator's request.

Compute nodes are used exclusively by jobs of a single user; no node sharing between jobs is done. The smallest allocation unit is one node (8 processors). Users will be charged for the number of compute nodes multiplied with the wall-clock time used. Approximately 22 GB of memory per node are available for applications.

Table 4. Job limits on JUROPA

	Resource	Limit
Interactive jobs	max. wallclock time	6 h
	default wallclock time	30 min
	max. number of nodes	8
	default number or nodes	1
	max. no. of running jobs (including batch jobs)	15
Batch jobs	max. wallclock time	24 h
	default wallclock time	30 min
	max. number of nodes	1024
	default number or nodes	1
	max. no. of running jobs	15

Submitting jobs with the Moab batch system

The Moab batch system is using batch scripts for job submission. The minimal template to be filled is:

Example 1. Template for job batch script

```
#!/bin/bash -x
#MSUB -l nodes=<no of nodes>:ppn=<no of procs /node>
#MSUB -l walltime=<hh:mm:ss>
#MSUB -e <full path for error file>
# if keyword omitted : default is submitting directory
#MSUB -o <full path for output file>
# if keyword omitted : default is submitting directory
#MSUB -v tpt=<no of threads per task>
# for OpenMP/hybrid jobs only

### start of jobscript
export OMP_NUM_THREADS=<no of threads/task>
# for OpenMP jobs only
cd $PBS_O_WORKDIR
echo "workdir: $PBS_O_WORKDIR"
NSLOTS=<nodes * ppn>
echo "running on $NSLOTS cpus ..."
mpiexec -np $NSLOTS [--exports=var1,...] <executable>
```

For parallel (MPI based) applications `mpiexec` together with application program (`executable`) must be used.

Note

The option `--exports` along with a comma separated list of environment variables ensures the export of all specified variables from the current job script to the processes spawned by the mpiexec command. This is necessary for instance if `OMP_NUM_THREADS` is defined for OpenMP. The use of `-x` for exporting of all environment variables is deprecated.

To submit the job defined with job script `jobscript` use the following command:

msub jobscript

On success, msub returns the job ID of the submitted job.

The following script is an example on how to define hybrid jobs (using MPI and OpenMP) for your application:

Example 2. Hybrid job (MPI and OpenMPI) allocating 8 nodes with 8 cores per node and starting 8 threads on each node

```
#!/bin/bash -x
#MSUB -l nodes=8:ppn=8
#MSUB -e /home/jhome3/test_user/my-error.txt
#MSUB -o /home/jhome3/test_user/my-out.txt
#MSUB -v tpt=8
### start of jobscript
export OMP_NUM_THREADS=8
cd $PBS_O_WORKDIR
echo "workdir: $PBS_O_WORKDIR"

# NSLOTS = nodes * ppn / tpt = 8 * 8 / 8 = 8
NSLOTS=8
mpiexec -np $NSLOTS --exports=OMP_NUM_THREADS ./hybrid_application
```

To start an interactive session for 30 minutes on two nodes with 8 processors each use the `-I` option for `msub`:

msub -I -l nodes=2:ppn=8,walltime=00:30:00

You will then get interactive access to a node and can start your applications right there.

The following table summarizes important `msub` command options:

Table 5.

Option	Suboption	Description
-l		set job limits (controlled by suboptions)
	nodes	number of compute nodes used by the job
	:ppn	processes per node
	:turbomode	enable CPU over-clocking
	walltime	wallclock timelimit for the job
-e		define file name of job's error output
-o		define file name of job's standard output
-j	oe	join standard output and error output into one file

Option	Suboption	Description
-v	tpt	define the number of threads per MPI task for an OpenMP job
-I		submit an interactive job
-M		define mail address to receive mail notification
-m		define when to send a mail notification
	n	never (default)
	b	at job begin
	e	at job end
	a	in case of job abort
-N		define the job's name
-W	depend	define job ID this job depends on
	afterok:	only start job, if previous job in the chain was ok

Other useful Moab commands are summarized below:

`showq [-r]`	show status of all (running) jobs
`canceljob jobid`	cancel a job
`mjobctl -q starttime jobid`	show estimated starttime of specified job
`mjobctl --help`	show detailed information about this command
`checkjob -v jobid`	get detailed information about a job

For further information please see also Moab documentation.

4.2.2. System specific compilers and libraries

MPI library implementation on JUROPA is ParTec MPI Message Passing Interface. Different versions of the MPI are available. Please use `module avail parastation` for more details.

4.2.3. Portability and compatibility with other x86 systems

The consistent and widely used processor architecture and programming environment from Intel provide a generic and portable x86 application environment.

4.3. Known issues

The memory consumption for static inter-node (InfiniBand) communication buffers depends on the number of communicating MPI processes (tasks). Users should be aware of memory usage for application runs using a large number of tasks. For details please see section Section 5.2.3, “Reducing the memory consumption of MPI connections” of this guide.

Please refer to the section on application tunning for more information on how to address MPI connections memory consumption. The JUROPA Memory Logger (jumel) can be used to analyse the memory consumption of an application. Please see `module help jumel` for further information.

Further details

For more information related to Section 4, “User environment and programming” please refer to the FZJ website:

- JUROPA Quick Introduction [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/UserInfo/QuickIntroduction.html]
- JUROPA Compilers [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/UserInfo/CompilerDefaults.html]
- JUROPA Software [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/Software/Software_node.html]
- JUROPA Resource Limits [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/UserInfo/ResourceLimits.html]

5. Tuning applications and Performance analysis

5.1. General optimization for x86 architecture

For detailed information on code optimization on x86 architecture refer to the PRACE Generic x86 Best Practice Guide [http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Generic-x86.pdf].

5.2. System specific optimization

5.2.1. How to benefit from SMT

The Intel Nehalem processor offers the possibility of Simultaneous Multi-Threading (SMT) which Intel formerly also called Hyper-Threading (HT). In SMT mode each processor core can execute two threads or tasks (called processes in the following for simplicity) simultaneously, leading to an execution of at most 16 processes per JUROPA compute node. The two "slots" of each core for running processes are called Hardware Threads (HWT).

Each JUROPA compute node consists of two quad-core CPUs, located on socket 0 and 1, respectively. The cores are numbered 0 to 7 and the HWT are named 0 to 15 in a round-robin fashion.

In order to use SMT on JUROPA you need to set `ppn=16` in your Moab job script. The following examples show how to use SMT for pure MPI and MPI/OpenMP hybrid applications.

Example 3. Pure MPI code

```
#!/bin/bash -x
#MSUB -N SMT_MPI_64x1_job
#MSUB -l nodes=4:ppn=16
### start of jobscript ###

mpiexec -np 64 application.exe
```

This script will start application.exe on 4 nodes using 16 MPI tasks per node, where two MPI tasks will be executed on each core.

Example 4. Hybrid MPI/OpenMP code

```
#!/bin/bash -x
#MSUB -N SMT_hybrid_8x8_job
#MSUB -l nodes=4:ppn=16
#MSUB -v tpt=8
```

```
### start of jobscript ###

export OMP_NUM_THREADS=8
mpiexec -np 8 --exports=OMP_NUM_THREADS application.exe
```

This script will start application.exe on 4 nodes using 2 MPI tasks per node and 8 OpenMP threads per task.

Processes which are running on the same core will need to share the resources available to that particular core. Therefore, applications will profit most from SMT if processes are running on one core which are complementary in their usage of resources. For example, while one of the two processes is performing some kind of computation the second one is accessing the memory. This way the two processes do not compete for resources. If on the other hand the two processes would need to access the memory at the same time they would need to share the caches and memory bandwith and would therefore hamper each other. We recommend to test whether your code profits from SMT or not.

In order to check whether your application profits from SMT you should compare timings t of two runs on the same number of physical cores (i.e. the specification of nodes should be the same for both jobs), one run without SMT (two) and the second one with SMT (tw). If two/tw is greater than 1 your application profits using SMT, for two/tw less than 1 it does not.

According to experience you can expect a maximum speed-up of up to two/tw = 1.5 for codes profiting from SMT. However, applications may show a smaller benefit or might even slow down when using SMT. In such cases SMT should not be applied.

By default the processes for pure MPI applications are mapped to the cores of each JUROPA node in a round-robin fashion. This means that the first 8 processes are allocated on the HWT 0 to 7 with process 0 on HWT 0 and process 7 on HWT 7 and the next 8 processes are allocated on HWT 8 to HWT 15.

For hybrid applications (MPI/OpenMP) the tasks and threads will be distributed over the cores in such a way that all threads belonging to one task will share the physical cores among themselves. Three scenarios for the distribution of tasks and threads for each node are handled by default:

- 2 MPI tasks with 8 OpenMP threads per task per node (2×8)
- 4 MPI tasks with 4 OpenMP threads per task per node (4×4)
- 8 MPI tasks with 2 OpenMP threads per task per node (8×2)

Important note

The mapping described here does not ensure that threads belonging to one task are pinned to a certain HWT. The mapping of tasks and threads just ensures that the *m* threads belonging to the certain task will be executed on the HWT assigned to this task.

You can customize the mapping of processes to the HWT by using the variable __PSI_CPUMAP (please note the two underscores "_" at the beginning of the variable). This variable should contain a comma-separated list of HWT, to which threads should be mapped. The threads will be distributed in the order of tasks.

Example 5. Hybrid MPI/OpenMP code (2x8, customized mapping)

```
#!/bin/bash -x
#MSUB -N SMT_hybrid_2x8_job
#MSUB -l nodes=1:ppn=16
#MSUB -v tpt=8
### start of jobscript ###

export OMP_NUM_THREADS=8
export __PSI_CPUMAP="0-3,12-15,4-11"
```

```
mpiexec -np 2 --exports=OMP_NUM_THREADS application.exe
```

This will allocate the threads of the first MPI task on the HWT 0-3 and 12-15 and the threads of the second MPI task on the HWT 4-11.

5.2.2. Optimising the available memory per core

The compute nodes of JUROPA have a NUMA (Non-Uniform Memory Architecture) design, i.e. on these systems the memory (24 GB/node) is divided into two memory nodes of 12 GB each, where each memory node is bound to one CPU socket containing 4 cores. Since latency and bandwidth of memory accesses are significantly worse, if the other socket is involved, processes are pinned to specific cores and bound to the local memory node. By default, the first 4 tasks are distributed across the first socket and further tasks are distributed across the second socket. If 3 GB of memory per core is not enough for your needs, then this ratio can be increased in several ways. Some examples are given in the table below:

Table 6. Task allocation parameters

ppn	tpp	nobind	tasks	mem	Explanation
8	8	---	1	12	Only one task is posed on a node, i.e. only the memory connected to the first socket is available.
8	8	export	1	24	If one task is posed on a node and the variable __PSI_NO_MEMBIND is set, the whole memory is available.
8	1	---	8	3	8 tasks are distributed across the node.
8	4	---	2	12	Four cores are reserved for each task, the first task is set on the first socket the second task is set on the second socket. Both tasks can use their local memory exclusively.
8	2	---	4	6	Two cores are reserved for each task, so 6 GB of memory are dedicated to each task.

Abbreviations in the above table:

`ppn`

Processes per node; this value is specified in your job script or `msub` command through the ppn option

`tpp`

Threads per process; this value is specified through the environment variable PSI_TPP in your job script (see text below)

`nobind`

No memory binding; this value is specified through the environment variable __PSI_NO_MEMBIND in your job script (see text below)

`tasks`

Lists the effective number of tasks per node resulting from the settings of ppn and tpp

`mem`

Lists the available memory per task in GB resulting from the settings of ppn and tpp

PSI_TPP and __PSI_NO_MEMBIND have to be exported within a batch script. Instead of exporting PSI_TPP it is also possible to set its value implicitly with the following Moab command within your batch script:

#MSUB -v tpt=1.

Then, of course, the value of tpt has to be set to your needs. Please be aware that, on the one hand, the access to the memory of a process from one socket to the other is assured through:

export __PSI_NO_MEMBIND=some_value

(provided that bash is used), but, on the other hand, the performance of this memory access might be reduced considerably, so it is a good idea to weigh up this option carefully.

5.2.3. Reducing the memory consumption of MPI connections

In some cases it might be necessary to reduce the amount of memory that has to be dedicated to MPI connections. This will apply in particular in cases where the job runs on many cores (>2000) and all-to-all communication is required. (As long as all-to-all communication is not needed, the use of on-demand connections will be beneficial; for this case see PSP_ONDEMAND below.)

In ParaStation MPI, roughly 0.55 MB of memory are needed for each MPI connection. If a program runs on 4000 cores and the communication pattern includes all-to-all communication, a total of 4000 times 3999 times 0.55 MB = 8797.8 GB of memory is needed just for the MPI connections, that is about 2.2 GB out of a total of 3 GB available per core.

ParaStation MPI uses 16 send buffers and 16 receive buffers per connection by default. Each buffer has a size of 16 KB. While the size of the buffers cannot be changed, the number of buffers can be modified via the environment variables PSP_OPENIB_SENDQ_SIZE and PSP_OPENIB_RECVQ_SIZE. If you want to reduce the number of these buffers you have to set the corresponding variables in your batch script:

export PSP_OPENIB_SENDQ_SIZE=3 (3 buffers for send)

export PSP_OPENIB_RECVQ_SIZE=3 (3 buffers for receive)

provided that bash is used. Both sizes might be modified independently.

The following table gives some figures for this purpose:

Table 7. Number of Q_SIZE buffers and MPI throughput

Q_SIZE	MB/connection
3	0.141
4	0.172
8	0.305

But please be aware that a reduced Q_SIZE might degrade the MPI's throughput and messaging rate. Furthermore, each Q_SIZE must be at least 3 in order to prevent deadlocks.

5.2.4. Using dynamic memory allocation for MPI connections

As described in the previous chapter, each MPI connection needs a certain amount of memory by default. The more processes are started, the more memory will be used for the MPI connections. As a rule of thumb, each MPI connection needs roughly 0.55 MB of memory. Besides the memory needed for your application, you have to take this fact into account.

On JUROPA, all MPI connections of your application will be established in the beginning of the run. This might lead to a memory shortage due to the reasons descibed above. If the memory allocation of your application plus the memory allocation for the MPI connections exceeds the memory capacity of the compute node, the job will fail.

The environment variable PSP_ONDEMAND influences the memory allocation for MPI connections. On JUROPA, the default of this variable is PSP_ONDEMAND=0, i.e. all needed memory will be allocated in the beginning of the run. If you alter the variable into PSP_ONDEMAND=1 within your batch script, then the memory allocation will take place dynamically. Dependent on the communication pattern of your application, this setting might circumvent your problem. But, if you do have all-to-all communication in your application, your program is likely to fail, since all MPI connections will have to be established when the all-to-all communication takes place.

5.2.5. Pinning OpenMP threads to processors

The Intel compiler's OpenMP runtime library provides the possibility to influence the binding of OpenMP threads to physical processing units. The behaviour is controlled by the environment variable KMP_AFFINITY. Information on this issue can be found on the Intel webpage.

On JUROPA, the default for the pinning is controlled by the PSI daemon. This means, the first four threads (0-3) are bound to the four cores of the first socket, all other threads (4-7) are bound to the cores of the second socket. If the user wants to change this setting, it is necessary to switch off the default setting explicitly by setting the environment variable __PSI_NO_PINPROC. Furthermore, the memory binding has to be switched off. The following example should clarify the general proceeding:

export OMP_NUM_THREADS=8

export KMP_AFFINITY=verbose,scatter

export __PSI_NO_PINPROC=1

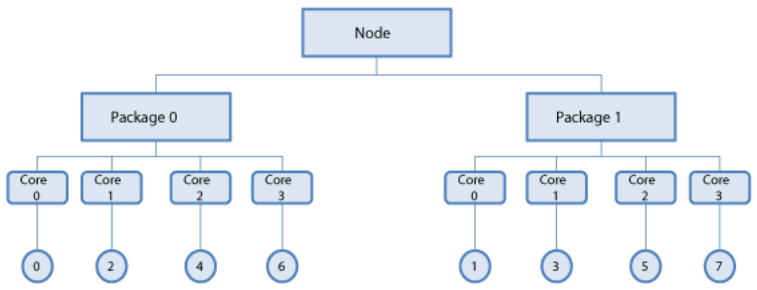
export __PSI_NO_MEMBIND=1

mpiexec --exports=OMP_NUM_THREADS,KMP_AFFINITY -n 1 ./application

Environment variables belonging to the PSI daemon need not to be exported explicitly in the mpiexec command.

Taking into account the setting from the example above the binding is as follows:

Figure 3. OpenMP threads binding example, source: Forschungszentrum Jülich



The threads are represented by circles in the picture above. KMP_AFFINITY=scatter distributes the threads as evenly as possible across the node.

5.2.6. Requesting large amounts of memory on the front-end nodes

Pre- and post-processing of input and output data is typically done on one of JUROPA's front-end nodes.

While all login nodes and most of the GPFS nodes possess 24 GB of main memory, there are two GPFS nodes (juropagpfs04, juropagpfs05) that are equipped with 192 GB each. These nodes must be used, if interactive pre- or post-processing requires more than 24 GB of memory. Please note that the GPFS nodes are interactive front-end nodes, they cannot be used by batch jobs.

To access the GPFS nodes login with ssh:

ssh userid@juropagpfs04.fz-juelich.de or **ssh userid@juropagpfs05.fz-juelich.de**

5.2.7. I/O tuning

In order to optimize the I/O performance of programs on JUROPA and to avoid any disturbance of other users you should follow a few rules when writing your software for the system:

- Write your data in large blocks instead of small portions to disk.
- Use buffered I/O whenever possible and avoid to flush your output frequently.
- Avoid task-local file I/O (each MPI task reads/writes to its own file). Use parallel I/O, for example MPI I/O or dedicated libraries like SIONlib, HDF5, pnetCDF, etc.

Depending on the programming language used, there are some specific rules:

FORTRAN

When using the ifort compiler, make sure that the environment variable

```
FORT_BUFFERED = true
```

is set. This is the case in the default environment, when the module parastation is loaded; in case you unload this module make sure that you set the FORT_BUFFERED variable properly.

Alternatively, you can use the compiler option `-assume buffered_io` to switch on I/O buffering for FORTRAN programs. The Intel Fortran Compiler (`ifort`) offers the non-standard parameters BUFFERED, BLOCKSIZE and BUFFERCOUNT for the open statement in order to customize the buffering of the output. Here are typical values, recommended for JUROPA.

Table 8. Compiler parameters for buffered I/O management in Fortran

`BUFFERED=YES`	enables buffering of WRITE commands
`BLOCKSIZE=1048576`	sets the size of the buffer to 1 MB
`BUFFERCOUNT=n`	uses n buffers; in general n=1 is appropriate

C

Do not use fflush (except at the end of your output/program).

C++

Do not use flush (except at the end of your output/program). Do not use endl, because this performs an additional flush each time called. If you need a line break, please use \n instead.

5.3. Available generic x86 performance analysis tools

The following generic profiling and tracing tools are available on JUROPA:

- HPCToolkit sampling profiler
- mpiP MPI profiling library
- Tau performance analysis system
- VampirTrace tracing library
- Scalasca performance analyzer
- Paraver tracing tool

Further details

For more information related to Section 5, “Tuning applications and Performance analysis” please refer to the FZJ website:

- Memory Optimization on JUROPA [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/UserInfo/MemoryOptimisation.html]

- SMT Usage on JUROPA [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/UserInfo/SMT.html]
- I/O Tuning on JUROPA [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/UserInfo/IO_Tuning.html]
- Parallel Performance Analysis on JUROPA [http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/UserInfo/Performance.html]

References

[Moab] *Moab HPC Suite Documentation [http://docs.adaptivecomputing.com/]*. Copyright © Adaptive Computing.

[BPGx86] *PRACE Generic x86 Best Practice Guide [http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Generic-x86.pdf]*.