# CHEOPS

**Cologne High Efficient Operating Platform for Science**

# Application Software

(Version: 26.04.2023)

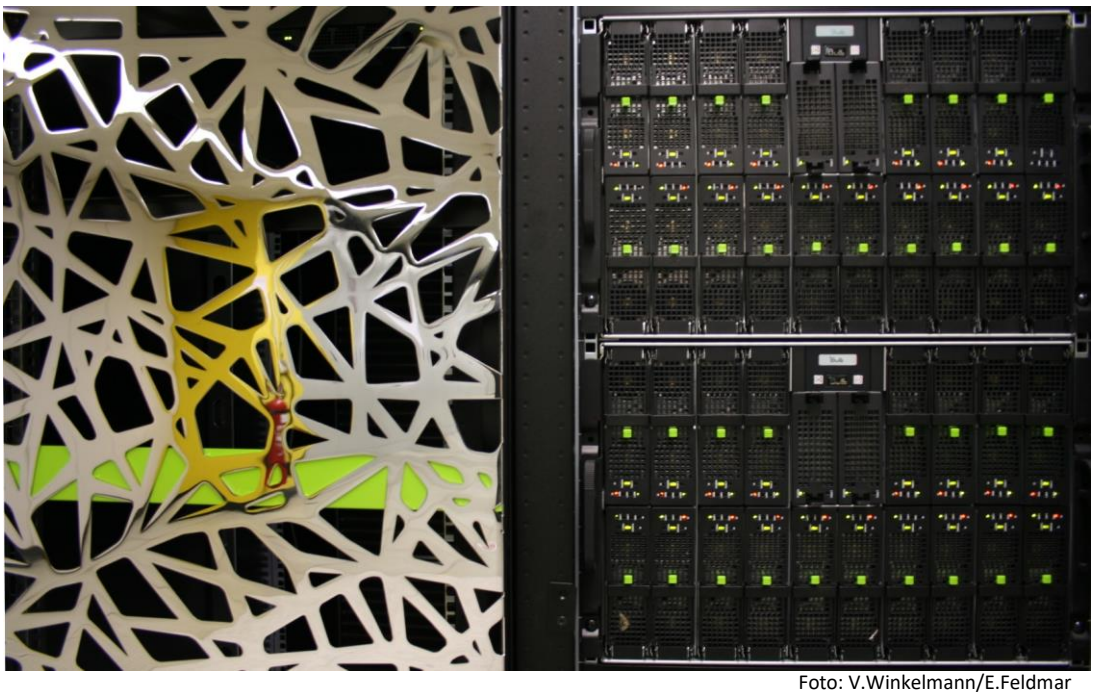


Foto: V.Winkelmann/E.Feldmar

Dr. Stefan Borowski
Lech Nieroda
Dr. Lars Packschies
Volker Winkelmann
Kevin Kaatz

EMail: wiss-anwendung@uni-koeln.de

# Content

# 1 About this document

This document gives an overview of the software applications available on RRZK's Cheops cluster. For more information about getting access to Cheops and its system software, please see the document CHEOPS Brief Instructions

http://ukoeln.de/JKMA6.

Submitting jobs on CHEOPS is straightforward given that environment modules are used. When a software module is loaded, all relevant execution path entries are automatically assigned, library paths are defined and important environment variables are set correctly. Consequently, the binaries and scripts you want to use with a specific application or program suite can be started without absolute paths. In addition, only your resource requests like maximum runtime (wall time), memory requirements and number of cores to use (and how they are distributed) have to be set in the batch job script.

You can list the available software modules with the command **`module avail`**. Please refer to the document CHEOPS Brief Instructions, Section 4 for the **`module`** command options.

For all calculations, it is strongly recommended to use the */scratch* file system. To make sure it exists, use the following command (-m 700 makes sure that only you have access to the data within the directory):

```
mkdir -m 700 /scratch/$USER
```

It is possible that your jobs could fail for several reasons, e.g.

- memory request exceeded: job runs out of memory
- time limit exceeded: job takes longer to finish
- node failure: job runs on node that fails during execution

Therefore, it is good idea to save the results of your calculations from time to time. To periodically save intermediate results during a run, please consider checkpointing for your application workflow as described in Chapter 6.

# 2 Chemistry applications

## 2.1 Gaussian

The following script describes a typical Gaussian job running on a single compute node:

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=10gb
#SBATCH --time=2:00:00
#SBATCH --job-name=g16_example
#SBATCH --output=g16_example_%j.output
#SBATCH --account=UniKoeln

module load gaussian/g16.C01
/usr/bin/time -p g16 < input.com >& output.log
```

This particular job script example requests 8 cores on a single node. The option **`--cpus-per-task`** is translated into **`%NProcShared`**. The memory request **`--mem`** of 10GB is translated into 9GB for **`%Mem`**, i.e. 90% of the SLURM request is given to Gaussian due to memory overhead. With last resource request **`--time`**, the runtime is expected not to exceed 2 hours. The names for **`--job-name`** and **`--output`** are arbitrary. In this case, you find the variable "**`%j`**" in the output setting - the resulting filename will contain the SLURM Job-ID for your convenience. For the option **`--account`**, you are kindly asked to set it to the appropriate argument (please refer to CHEOPS Brief Instructions, Section 5.3). The default value is ***`UniKoeln`***.
The **`module load`** command sets up the environment to run Gaussian jobs using version g16 rev. C01. The RRZK provides more than one version of the program (see **`module avail`**). Finally, you can start Gaussian using the **`g16`** command, as usual. Please fill in the correct input and output filenames to match your calculation setup.

To run a Gaussian calculation on more than one node, you only have to increase the number of nodes requested. The option **`--nodes`** is then translated into appropriate **`%LindaWorkers`**. Since the Gaussian modules convert SLURM resource requests into according Gaussian keywords, you should not use keywords like **`%LindaWorkers`**, **`%NProcShared`**, **`%Mem`** or the deprecated **`%NProcLinda`**, **`%NProc`** in your command files anymore.

## 2.2 Turbomole

The next example can be used to invoke a Turbomole 7.7 computation using **`jobex`**:

```
#!/bin/bash -l
#SBATCH --job-name=tm_example
```

```
#SBATCH --output=tm_example_%j.output
#SBATCH --mem=10GB
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --ntasks=8
#SBATCH --time 2:00:00
#SBATCH --account=UniKoeln

module load turbomole/7.7

time -p jobex -c 1000 >& out.out
```

The script uses similar values as the Gaussian single node example above. Turbomole, however, requires setting **`--ntasks`** additionally. The filename for Turbomole output (here: out.out) is arbitrary.
For a Turbomole calculation using two or more nodes, refer to the following script:

```
#!/bin/bash -l
#SBATCH --job-name=tm_example
#SBATCH --output=tm_example_%j.output
#SBATCH --mem=10GB
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --ntasks=16
#SBATCH --time 2:00:00
#SBATCH --account=UniKoeln

module load turbomole/7.7

time -p jobex -c 1000 >& out.out
```

## *2.3 ORCA*

ORCA is an MPI parallel quantum chemistry program for calculating electronic structures, molecular dynamics and spectroscopic properties with DFT, coupled cluster and multi-reference methods. Binaries of ORCA can be downloaded and used free of charge by academic institutions after registration. We provide version **`orca/5.0.3`** using our Open MPI version **`openmpi/4.1.1_mpirun`**. To use ORCA on CHEOPS, you have to register on the ORCA forum and forward the confirmation email together with your user name to hpc-esd@uni-koeln.de. We will then assign your account to the group **`orcauser`**, which enables execution of the ORCA binaries. A core based job example running ORCA on 16 cores from multiple nodes could look like:

```
#!/bin/bash -l
#SBATCH --ntasks=16
#SBATCH --mem-per-cpu=4000mb
```

```
#SBATCH --time=24:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module load orca/5.0.3

export workdir=/scratch/${USER}/${SLURM_JOB_ID}
mkdir -p -m 700 $workdir

cp myorca.inp $workdir
cd $workdir
${ORCADIR}/orca myorca.inp
cd -
cp -p ${workdir}/*.{gbw,loc,prop,xyz,trj,opt} .
```

Each task is hosted by one core and runs one MPI process. The batch system allocates the cores from any nodes with idle cores. Memory is allocated per core (option **`--mem-per-cpu`** meaning memory per core). The ORCA resource requests in the input file **`myorca.inp`**

```
%pal nprocs 16 end
%maxcore 3900
```

should correspond to the SLURM resource requests: The number of processes **`nprocs`** is equal to the number of requested tasks. The estimation of maximum memory used per process **`maxcore`** in MB should be below the requested memory per core. With the input file copied, the launcher **`orca`** is executed in a job specific working directory in **`/scratch`** for faster access to temporary files (e.g. integral tables) used by ORCA. Relevant output files are transferred back to the submission directory after the run has finished.

https://orcaforum.kofo.mpg.de/app.php/portal

## *2.4 NWChem*

NWChem is an application for quantum mechanics and molecular dynamics simulations. It comes with an MPI parallelization. A simple NWChem job script could be:

```
#!/bin/bash -l
#SBATCH --ntasks=16
#SBATCH --mem-per-cpu=4000mb
#SBATCH --time=24:00:00
#SBATCH --output==%x-%j.out
#SBATCH --account=UniKoeln

module load nwchem/7.0.2

srun -n $SLURM_NTASKS nwchem input.nw > output.out
```

https://nwchemgit.github.io/

## 2.5 Cfour

CFOUR is a quantum chemistry package with focus on high-level ab initio methods such as Møller-Plesset (MP) and Coupled Cluster (CC).

```
#!/bin/bash -l
#SBATCH --ntasks=16
#SBATCH --mem-per-cpu=4000mb
#SBATCH --time=24:00:00
#SBATCH --output==%x-%j.out
#SBATCH --account=UniKoeln

module load cfour/2.1-parallel

xcfour > cfour.out
```

When used with MPI parallelization, the executable cfour calls the MPI launcher itself. Therefore, an invocation of SLURM's MPI launcher is not necessary. The basis sets are provided by a link to the GENBAS file, which is generated automatically.

http://www.cfour.de/

## 2.6 Quantum Espresso

Quantum Espresso is an MPI parallel application that uses density-functional theory on plane wave basis sets and pseudopotentials for electronic-structure calculations. When the module is used a subdirectory **`qe/`** is created in the user's scratch directory, which contains large files used during the calculation.

```
#!/bin/bash -l
#SBATCH --ntasks=16
#SBATCH --mem-per-cpu=4000mb
#SBATCH --time=24:00:00
#SBATCH --output==%x-%j.out
#SBATCH --account=UniKoeln

module load qe/6.8.0

srun -n $SLURM_NTASKS pw.x -in input.in > output.out
```

https://www.quantum-espresso.org/

## 2.7 Gromacs

Gromacs is available in version **`gromacs/2020.6`** compiled in single precision. This should be OK for most calculations. Gromacs features hybrid parallelization with both MPI and OpenMP. On a small number of cores, running Gromacs in pure MPI mode is more efficient. The following core based job example runs Gromacs with MPI on 16 cores:

```
#!/bin/bash -l
#SBATCH --ntasks=16
#SBATCH --mem-per-cpu=512mb
#SBATCH --time=24:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module load gromacs/2020.6

gmx_mpi grompp -f grompp.mdp -c conf.gro -t traj.cpt \
               -p topol.top -o topol.tpr

srun -n $SLURM_NTASKS gmx_mpi mdrun -deffnm md_test
```

Each task is hosted by 1 core and runs 1 MPI process. The batch system allocates the cores from any nodes with idle cores. Memory is allocated per core (option **--mem-per-cpu** meaning memory per core). With the variable **SLURM_NTASKS**, you tell the MPI launcher **srun** to start as many MPI processes as tasks.
Alternatively, a node based job requests the same amount of resources referring to nodes:

```
#!/bin/bash -l
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --ntasks=16
#SBATCH --mem=4gb
#SBATCH --time=24:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module load gromacs/2020.6

gmx_mpi grompp -f grompp.mdp -c conf.gro -t traj.cpt \
               -p topol.top -o topol.tpr

srun -n $SLURM_NTASKS gmx_mpi mdrun -deffnm md_test
```

The batch system allocates 2 nodes with 8 cores each. The 8 cores host 8 tasks running 8 MPI processes on each of the 2 nodes. Memory is allocated per node. While core based MPI jobs are scheduled earlier, node based MPI jobs run more efficiently with less wall time.
Starting Gromacs in hybrid mode (MPI processes with multiple OpenMP threads each) is only worth when employing many nodes exclusively in huge jobs. Here, the number of tasks per node gives the number of MPI processes per node as before. However, a task now requires more than one core for the OpenMP threads to run on (option **--cpu-per-task** meaning cores per task). Hybrid runs on our INCA nodes (with two sockets) usually use one or MPI processes per node. Correspondingly, each task should take all cores of the node or half of them. A job exclusively running on 64 INCA nodes with 12 cores could look like:

```
#!/bin/bash -l
#SBATCH --nodes=64
#SBATCH --ntasks-per-node=2
```

```
#SBATCH --ntasks=128
#SBATCH --cpus-per-task=6
#SBATCH --mem=1gb
#SBATCH --time=24:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module load gromacs/2020.6

gmx_mpi grompp -f grompp.mdp -c conf.gro -t traj.cpt \
               -p topol.top -o topol.tpr

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun -n $SLURM_NTASKS gmx_mpi mdrun -deffnm md_test
```

The nodes are completely occupied by 2 tasks with 6 cores each to run 2 MPI processes with 6 OpenMP threads each. The variable **SLURM_CPUS_PER_TASK** (meaning cores per task) gives the number of OpenMP threads **OMP_NUM_THREADS** to use.

http://www.gromacs.org
http://manual.gromacs.org/documentation

## *2.8 LAMMPS*

Another molecular dynamics application is LAMMPS, which is available with version **lammps/20220623u3.** Although LAMMPS also features hybrid parallelization, running it in pure MPI mode is more efficient for common job sizes. The following core based job example runs LAMMPS with MPI on 16 cores:

```
#!/bin/bash -l
#SBATCH --ntasks=16
#SBATCH --mem-per-cpu=1024mb
#SBATCH --time=24:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module load lammps/20220623u3

srun -n $SLURM_NTASKS lmp -in in.rhodo -log log.rhodo
```

Each task is hosted by 1 core and runs 1 MPI process. The batch system allocates the cores from any nodes with idle cores. Memory is allocated per core (option **--mem-per-cpu** meaning memory per core). With the variable **SLURM_NTASKS**, you tell the MPI launcher **srun** to start as many MPI processes as tasks.

# 3 Economy applications

## *3.1 CPLEX*

IBM ILOG CPLEX Optimization Studio (often informally referred to simply as CPLEX) is an optimization software package. The IBM ILOG CPLEX Optimizer solves integer programming problems, very large linear programming problems using either primal or dual variants of the simplex method or the barrier interior point method, convex and non-convex quadratic programming problems, and convex quadratically constrained problems (solved via second-order cone programming, or SOCP).

CPLEX Optimizer is parallelized and can run on more than one core of a single node to compute results. The following job example runs CPLEX on 8 cores of one node:

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=22GB
#SBATCH --time=04:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module use -a /opt/rrzk/modules/experimental/
module load cplex

MyProjekt=$HOME/Projekt_X

cd $MyProjekt

time oplrun -p . # don't miss the dot
```

CPLEX is started as a single task on a single node, requesting 8 cores to solve the optimization problem. In the above specified directory **`$HOME/Projekt_X`**, CPLEX model input files must be provided, such as e.g. **`.mod`** file, **`.dat`** file, **`.project`** file, and the **`.oplproject`** file.
Remark: If transferring project files from Windows to Cheops, keep in mind that CPLEX functions for reading/writing Windows specific file formats might not be available. In case of working with Excel formatted files, you must convert input data files to CSV format and change CPLEX program code to read/write CSV format prior to using CPLEX on Cheops.

Since CPLEX is a commercial product from IBM, which academic persons may use free under the academic license, we can only grant you access to CPLEX, if you have successfully registered at the IBM Academic Initiative under https://www.ibm.com/academic/home and forwarded the confirmation together with your user name to hpc-esd@uni-koeln.de. We will then assign your account to the group **`cplexuser`**, which enables execution of the CPLEX programs. For further information on IBM ILOG CPLEX see
http://www.ibm.com/analytics/cplex-optimizer.

# 4 High-level languages for numerical computations

On CHEOPS, RRZK provides several software packages with high-level languages for numerical computations, e.g. MATLAB, Scilab and R. All packages can be started within batch jobs, all batch scripts are similar to shell scripts, which might already be used on users' local workstations. For slight differences, see the comments on each product.

## *4.1 MATLAB*

MATLAB is provided as a software module which, when loaded, provides all relevant execution path entries and environment variables set correctly.
RRZK has obtained some licenses of the MATLAB Parallel Computing Toolbox enabling users to speed up their applications if parallel functions like **`parfor`**, **`matlabpool`** or **`createTask`** may be used to solve a problem with parallel code execution.
Even sequential jobs might benefit from MATLAB's ability to run multi-threaded functions on multi-core architectures. Since most CHEOPS nodes provide 8-12 cores, a number of MATLAB functions might show significant speed up. A list of relevant functions may be found under

http://www.mathworks.com/support/solutions/en/data/1-4PG4AN/?solution=1-4PG4AN

### 4.1.1 Sequential MATLAB batch jobs without internal multi-threading

A simple MATLAB batch job on CHEOPS using one core may look like this

```
#!/bin/bash -l

#SBATCH --job-name MyMatlabProg
#SBATCH --cpus-per-task=1
#SBATCH --mem=1G
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load matlab

MyMatlabProgram="$HOME/matlab/example1/myprog.m"

# start Matlab with my Matlab program
time matlab  -nodisplay -nodesktop -nosplash -nojvm \
             -singleCompThread -r "run $MyMatlabProgram"
```

where the variable **`MyMatlabProgram`** refers to the location of the MATLAB program to be started within the batch job. Notice the option **`-singleCompThread`** signaling that multi-threading of all internal MATLAB functions is switched off.

As already explained earlier in this document, the above script can be submitted to the batch system with the call

```
sbatch myprog.sh
```

if **myprog.sh** is the file name of the batch script.

### 4.1.2 Sequential MATLAB batch jobs with internal multi-threading

As mentioned earlier sequential MATLAB jobs may benefit from internal multi-threaded functions to speed up performance of a sequential program. Some MATLAB functions (see reference above) may use all cores of a compute node, so it is necessary to obtain all processors of a node.
A MATLAB batch job on CHEOPS using 8 cores of a node supporting MATLAB's internal multi-threading features may look like this

```
#!/bin/bash -l

#SBATCH --job-name MyMatlabProg
#SBATCH --cpus-per-task=8
#SBATCH --mem=4G
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load matlab

MyMatlabProgram="$HOME/matlab/example1/myprog.m"

# start Matlab with my Matlab program and required tasks

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

time matlab  -nodisplay -nodesktop -nosplash -nojvm \
             -r "run $MyMatlabProgram"
```

where the variable **MyMatlabProgram** refers to the location of the MATLAB program to be started within the batch job. Notice that the number of required processors has been set to 8, the variable **OMP_NUM_TREADS** has been set to the number of requested cores and that the option **-singleCompThread** has been omitted to enable internal multi-threading.
Note: While some MATLAB programs seem to speed up a little bit, in most cases it is more reasonable to check whether parts of the program like **for** loops can be run in parallel within MATLAB. For more information on performing parallel computations on multi-core computers, see

http://www.mathworks.com/products/parallel-computing/

### 4.1.3 Parallel MATLAB batch jobs with the Parallel Computing Toolbox™

RRZK has obtained several licenses of the Parallel Computing Toolbox™, which allows offloading work from one MATLAB® session to up to 12 MATLAB sessions running in parallel, called workers. When running batch jobs with functions from the Parallel Computing Toolbox™ in such a way, only one license of MATLAB itself, the Parallel Computing Toolbox™ and each further used toolbox are checked out from the license server.
Batch jobs on CHEOPS, which intend to use the Parallel Computing Toolbox™, may use the following script template:

```
#!/bin/bash -l

#SBATCH --job-name MyMatlabProg
#SBATCH --cpus-per-task=8
#SBATCH --mem=8G
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load matlab

MyMatlabProgram="$HOME/matlab/parallel/myparprog.m"

# start Matlab with my Matlab program
time matlab  -nodisplay -nodesktop -nosplash \
             -r "run $MyMatlabProgram"
```

For running parallel tasks within MATLAB, MATLAB's embedded JAVA virtual machine is used. Therefore, in opposite to sequential MATLAB jobs option **`-nojvm`** must be omitted. The JVM also needs additional memory, therefore for each requested worker/processor at least 1 GB of memory is recommended; see options in the header of the above batch script

```
#SBATCH --cpus-per-task=8
#SBATCH --mem=8G
```

In the case of multi-core jobs like a MATLAB job with parallel workers, the requested wall time (see sbatch option **`--time`**） is the expected runtime of the whole MATLAB job. You do **not** need to accumulate the individual runtime of each worker; essentially wall time should decrease if running the same numerical problem with more parallel workers.
On the other hand memory requirements will rise the more workers are used in parallel, thus the argument of sbatch option **`--mem`** must be increased**.**

For more information on performing parallel computations with MATLAB on multi-core computers, see

http://www.mathworks.com/products/parallel-computing/

or contact V.Winkelmann, RRZK.

### 4.1.4 Parallel MATLAB batch jobs using the MATLAB Parallel Server

While MATLAB batch jobs using the Parallel Computing Toolbox™ can only run on one node and may use all cores of that single node, the MATLAB Parallel Server™ offers the possibility to distribute parallel parts of a MATLAB program to more than one node using several cores of each node. The way to use the MATLAB Parallel Server is to start a MATLAB client on any computer, no matter whether being a remote personal desktop in the UKLAN, or an interactive node on the cluster itself: parallel computations can then be outsourced and sent as a separate batch job to the cluster, using the cluster as an "external MATLAB processor".
Anyhow, on the MATLAB client side the MATLAB environment for the MATLAB Parallel Server must be initialized prior to sending batch jobs to the cluster.
**Note: MATLAB Parallel Server is only available on Cheops1, starting with versions ≥ 2021a. The version number of the used MATLAB client must be identical to the MATLAB version on Cheops1.**

#### 4.1.4.1 Initializing the MATLAB client on any interactive Cluster Node

**First step required**

After logging into the cluster, configure MATLAB on the OS level to run parallel jobs on your cluster by calling the shell script **`configCluster.sh`**. **This only needs to be called once per version of MATLAB**.

```
$ module load matlab   # or different version ≥ 2021a
$ configCluster.sh
```

Later, MATLAB Jobs being setup by the MATLAB Parallel Server command **`batch`** will then default to the cluster rather than being submitted to the local node, where the MATLAB client was started.

**Starting the MATLAB Client on an interactive cluster node**

When being ready to use the MATLAB client in order to benefit from the MATLAB parallel server and submit MATLAB batch jobs to the cluster, please, **do not use the cluster frontend** to start the MATLAB client, but instead use a free node assigned to you by the SLURM command **`salloc`**:

```
salloc -n 1 -c 1 -t 02:00:00 --mem-per-cpu=4000 --x11
```

After allocation, the provided node can be entered by

```
srun -pty bash
```

Start the MATLAB client by

```
module load matlab/2021a    # at CHEOPS1 version must be >= 2021a
matlab -nodisplay           # or interactive client


…
```

Now follow the instruction in 4.1.4.3 for creating and submitting batch jobs to the cluster.

For creating batch jobs without the need of starting an interactive MATLAB client, but submitting a scripted batch job, see 4.1.4.4.

### 4.1.4.2 Initializing the MATLAB client on any personal remote desktop

#### Preliminary Work needed

Note: For submitting MATLAB batch jobs from a personal remote desktop to the cluster you must possess a valid HPC account an RRZK. For information on how to obtain an HPC account, see the access and use instructions on RRZK's home page.
Before calling the MATLAB Parallel Server from a remote desktop, the MATLAB Parallel Server support package must be installed on the remote desktop, which can be found on RRZK's technical details page of the cluster Cheops for the following operating systems:

- Windows
- Linux/MacOS

Download the appropriate archive file and start MATLAB. The archive file should be untarred/unzipped in the location returned by calling

```
>> userpath
```

#### Initialization MATLAB for Cluster Jobs

Initialize MATLAB to run parallel jobs on the cluster by calling **`configCluster`**. Script **`configCluster`** **only needs to be called once per version of MATLAB:**

```
>> configCluster
```

Submission to the remote cluster requires SSH credentials. You will be prompted for your SSH username and password on the cluster, or an identity file (private key) used to connect

to the cluster. The username and location of the private key will be stored in MATLAB for future sessions.

Later, MATLAB Jobs being setup by the MATLAB Parallel Server command `batch` will then default to the cluster rather than being submitted to the local machine, where the MATLAB client was started.

Now follow the instruction in 4.1.4.3 for creating and submitting batch jobs to the cluster.

### 4.1.4.3 Configuring MATLAB Jobs for using the Parallel Server on the cluster

After MATLAB initialization for using the parallel server on the cluster, prior to submitting the job, we can specify various parameters to pass to our batch jobs, such as queue, e-mail, walltime, etc. Only attributes **`MemUsage`** and **`WallTime`** are required to submit MATLAB jobs.

```
>> % create a parallel cluster object and get a handle
>> c = parcluster;
```

**[REQUIRED]**

```
>> % Specify memory to use for MATLAB jobs, per core (MB)
>> c.AdditionalProperties.MemUsage = '4000';

>> % Specify the walltime (e.g., 5 hours)
>> c.AdditionalProperties.WallTime = '05:00:00';

>> % Specify an account to use for MATLAB jobs
>> c.AdditionalProperties.AccountName = 'account-name';
```

**[OPTIONAL]**

```
>> % Specify the constraints
>> c.AdditionalProperties.Constraints = 'constraint-name';

>> % Specify e-mail to receive notifications about your job
>> c.AdditionalProperties.EmailAddress= 'userId@uni-
koeln.de';

>> % Specify number of GPUs per node, using specified GPU
card
>> % currently only available for private nodes
```

```
>> c.AdditionalProperties.GpuCard = 'gpu-card';
>> c.AdditionalProperties.GpusPerNode = 1;


>> % Specify Local partition size (e.g., 2GB)
>> c.AdditionalProperties.LocalTmp = '2gb';


>> % Specify number of desired nodes (required if > 1, e.g.
2)
>> c.AdditionalProperties.Nodes = 2;


>> % Specify a queue to use for MATLAB jobs, if necessary
>> c.AdditionalProperties.QueueName = 'queue-name';
```

Save changes after modifying **AdditionalProperties** for the above changes to persist between MATLAB sessions.

```
>> c.saveProfile
```

To see the values of the current configuration options, display **AdditionalProperties** via

```
>> % To view current properties
>> c.AdditionalProperties
```

Unset a value when no longer needed, e.g.

```
>> %Turn off email notifications
>> c.AdditionalProperties.EmailAddress = '';
>> c.saveProfile
```

For now, the configuration for submitting a MATLAB batch job to the cluster has been done.

### A first independent batch job

Use the `batch` command to submit asynchronous jobs to the cluster. The `batch` command will return a job object which is used to access the output of the submitted job. See the MATLAB documentation for more help on `batch`.

```
>> % create a parallel cluster object and get a handle
>> c = parcluster;


>> % Submit job to query where MATLAB is running on the
cluster
```

```
>> job = c.batch(@pwd, 1, {}, …
       'CurrentFolder','.', 'AutoAddClientPath',false);

>> % Query job for state
>> job.State

>> % If state is finished, fetch the results
>> job.fetchOutputs{:}



>> % Delete the job after results are no longer needed
>> job.delete
```

To retrieve a list of currently running or completed jobs, call **parcluster** to retrieve the cluster object. The cluster object stores an array of jobs that have finished, are running, or are queued to run. This allows us to fetch the results of completed jobs. Retrieve and view the list of jobs as shown below, e.g.

```
>> c = parcluster;
>> jobs = c.Jobs;
```

Once we've identified the job we want, we can retrieve the results as we've done previously.

**fetchOutputs** is used to retrieve function output arguments; if calling **batch** with a script, use **load** instead. Data that has been written to files on the cluster needs be retrieved directly from the file system (e.g., via secure copy).

To view results of a previously completed job:

```
>> % Get a handle to position 2 in the job array
>> job2 = c.Jobs(2);
>> % Get a handle to the job with ID 6
>> job6 = c.findJob('ID',6);
```

NOTE: You can view a list of your jobs, as well as their IDs, using the above **c.Jobs** command.

```
>> % Fetch results for job from position 2
>> job2.fetchOutputs{:}
>> % Fetch results for job with ID 6
>> job6.fetchOutputs{:}
```

### Parallel Batch Jobs

Users can also submit parallel workflows with the `batch` command. Let's use the following example for a parallel job, which is saved as **`parallel_example.m`**.

```
function [t, A] = parallel_example(iter)

if nargin==0
    iter = 8;
end

disp('Start sim')

t0 = tic;
parfor idx = 1:iter
    A(idx) = idx;
    pause(2)
    idx
end
t = toc(t0);

disp('Sim completed')

save RESULTS A

end
```

This time when we use the **`batch`** command to run a parallel job, we'll also specify the size of the MATLAB Pool, here 4:

```
>> % create a parallel cluster object and get a handle
>> c = parcluster;

>> % Submit a batch pool job using 4 workers for
>> % 16 simulations
>> job = c.batch(@parallel_example, 1, {16}, 'Pool',4, …
       'CurrentFolder','.', 'AutoAddClientPath',false);

>> % View current job status
>> job.State
```

```
>> % Fetch the results after a finished state is retrieved
>> job.fetchOutputs{:}
ans =
       8.8872
```

The job ran in 8.89 seconds using four workers. **Note that these jobs will always request N+1 CPU cores, since one worker is required to manage the batch job and pool of workers.** For example, a job that needs eight workers will consume nine CPU cores.

We'll run the same simulation but increase the Pool size. This time, to retrieve the results later, we'll keep track of the job ID.

NOTE: For some applications, there will be a diminishing return when allocating too many workers, as the overhead may exceed computation time.

```
>> % create a parallel cluster object and get a handle
>> c = parcluster;

>> % Submit a batch pool job using 8 workers for 16
simulations
>> job = c.batch(@parallel_example, 1, {16}, 'Pool', 8, …
       'CurrentFolder','.', 'AutoAddClientPath',false);

>> % Get the job ID
>> id = job.ID
id =
       4
>> % Clear job from workspace (as though we quit MATLAB)
>> clear job
```

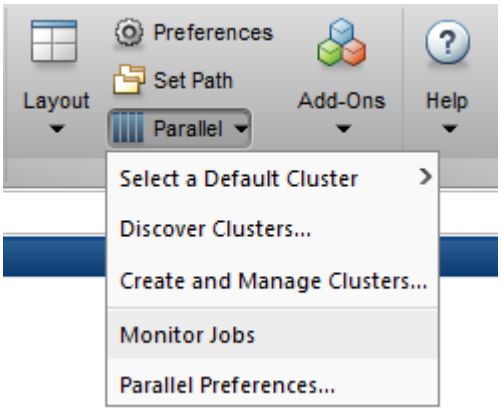
Once we have a handle to the cluster, we'll call the **`findJob`** method to search for the job with the specified job ID.

```
>> % create a parallel cluster object and get a handle
>> c = parcluster;

>> % Find the old job
>> job = c.findJob('ID', 4);

>> % Retrieve the state of the job
>> job.State
ans =
```

```
finished


>> % Fetch the results
>> job.fetchOutputs{:};
ans =
4.7270
```

The job now runs in 4.73 seconds using eight workers. Run code with different number of workers to determine the ideal number to use.

Alternatively, to retrieve job results via a graphical user interface, use the Job Monitor (Parallel > Monitor Jobs).




#### 4.1.4.4 Creating MATLAB Parallel Server Batch Jobs without an interactive MATLAB client on the Cluster

MATLAB Parallel Server is designed to be accessed through a running MATLAB client. If starting a MATLAB client is too time consuming, one can construct a nested shell script that starts the required MATLAB client in a small batch job running a MATLAB script containing all the necessary MATLAB commands as described in 4.1.4.3 needed to start the Parallel Server and execute the computations, and submit the computations in a second job to the cluster.
Example for a MATLAB Parallel Server batch job without interactively starting a MATLAB client, requesting for 2 nodes, and ordering a MATLAB pool of 8 workers, starting the parallel program shown in 4.1.4.3:

```
#!/bin/sh -l
#SBATCH --job-name=start_MATLAB-ParServJob
#SBATCH --output=start_MATLAB-ParServJob.%j.out
#SBATCH --nodes=1
#SBATCH --mem=4GB
#SBATCH --time=00:10:00
#SBATCH --account=UniKoeln

module load matlab/2021b  #initalize MATLAB
```

```
# location for MATLAB program using Parallel Server
export MyMatlabDir="$HOME/matlab/example1"
mkdir -p "$MyMatlabDir"
export MyMatlabProgram="$MyMatlabDir/myjob.m"

# create program using Parallel Server
cat > $MyMatlabProgram <<EOF
% Keep in mind that the following instructions are only for
% composing a MATLAB job that is then internally sent by
% sbatch to SLURM with the restrictions specified as
% properties for the parcluster object.
% This MATLAB job will start a parallel server job on 2 nodes
% and 8 cores executing a MATLAB program located in the file
% /scratch/$USER/parallel_example.m .

disp('Start 1. Batch Job')

username=getenv('USER');

% extend MATLABPATH by the location of the parallel program
% to be executed
P=matlabpath+":/scratch/"+username;
matlabpath(P);
% check whether the parallel program is found
which  parallel_example

% Create a parallel cluster object and the properties
% of the parallel batch job corresponding to the
% ressources MATLAB and SLURM will need

c = parcluster;
c.AdditionalProperties.MemUsage = '4000';
c.AdditionalProperties.WallTime = '01:00:00';
c.AdditionalProperties.AccountName = 'MySlurmAccountname';
%c.AdditionalProperties.EmailAddress = strcat(username,'@uni-
koeln.de');
c.AdditionalProperties.Nodes = 2;
c.AdditionalProperties.QueueName = 'devel-rh7';
c.saveProfile
% display summary of SLURM properties to be used
c.AdditionalProperties

% send the real application @parallel_example to the cluster
% and start the program

disp('Start 2. Batch Job')
job = c.batch(@parallel_example, 1, {16},
'Pool',8,'CurrentFolder','.', 'AutoAddClientPath',false);
fprintf("Started Job %d\n",job.ID);
job.State;

% Do NOT wait for the finish of the job, because otherwise
% this 1. Batch job would last as long as the submitted
% batch job, but doing nothing then waiting.
```

```
% some hints where to find the output
disp('To retrieve the output, start a new MATLAB session and
call');
disp('c=parcluster');
fprintf("c.findJob('ID', %d).fetchOutputs{:}\n",job.ID);

% Attention:
% Do not wait for the finish of the job, because otherwise
% this 1. Batch job would last as long as the submitted
% batch job, but doing nothing else then waiting.

disp('End of 1. Batch Job')
quit;
EOF

# start MATLAB with $MyMatlabProgram
time matlab -nodisplay  -r "run $MyMatlabProgram"

exit

# later, in a second MATLAB session you may retrieve the
# results with

#c=parcluster;
#jobs=c.Jobs % returns array of jobs started in parcluster c
#jobN = c.findJob('ID',N); % where N is the specific Job ID
#jobN.fetchOutputs{:}
```

The above shell script can be sent to SLURM on Cheosp1 with the SLURM **sbatch** command. The shell script creates a MATLAB program in **$HOME/matlab** that will at first modify the MATLAB environment, extending the MATLABPATH by **/scratch/$USER** in order to be able to find the desired parallel program **parallel_example.m** to be sent to the batch system. In the MATLAB program then a parallel cluster object is created, specifying its properties (= SLURM options) for a MATLAB Parallel Server job, which shall be used later by the MATLAB **batch** command. Finally, the shell script starts MATLAB running the previously created MATLAB program, which sends the desired parallel program to the cluster in a separate batch job (see MATLAB batch), respecting the specified attributes of the parallel cluster object, here e.g., obtaining 2 nodes, a maximum wall time of 1 hour, and being assigned a MATLAB pool of 8 workers (= 8 tasks). As mentioned before, MATLAB needs one additional worker for its pool management, therefore the MATLAB job will request 9 tasks. While the previously created MATLAB program has already ended, the parallel program in the second batch job might still keep running independently from the first creating batch job.

### 4.1.5 Batch Jobs without MATLAB Licenses - Using the MATLAB Compiler

For running many MATLAB batch jobs simultaneously on clusters, usually an equal number of licenses for the main MATLAB programs and equivalent licenses for MATLAB toolboxes are required. Please better compile the used MATLAB program prior to submitting the MATLAB batch job and start the compiled version instead of its source code version. In that

case, no licenses are required, neither for MATLAB itself nor for any toolbox. Such MATLAB batch jobs will therefore never abort due to exceeded license numbers.

At first, compile your MATLAB program interactively on the Cheops frontend.

```
module load matlab gnu

cd ${MYCODE}

mcc -m -R -nodisplay -R -nodesktop -R -nosplash \
    ${MyMatlabProgram}.m
```

where **${MYCODE}** keeps the name of the directory of the MATLAB program to be compiled, and **${MyMatlabProgram}.m** refers to the name of the specific MATLAB program.

**Note:** *The reason for compiling a MATLAB program on the cluster frontend is that after usage the compiler will be blocked for half an hour for the last user and its used compute node. Since the University of Cologne only owns one compiler license, even the last user would need luck to resubmit his/her job to the same node of the cluster used before. Using the cluster frontend bypasses this "license feature" of the MATLAB compiler.*
*Consequently, it might happen that, when trying to compile a MATLAB program, the compiler license is still blocked by another user who has called the compiler less than 30 minutes ago.*

After compilation a new compiled program **${MyMatlabProgram}** will exist, which may be used in a subsequent batch job. A run script for your program will be created, too, but is not required, because RRZK's module environment includes all specifications for the MATLAB runtime environment. An appropriate batch job using the compiled MATLAB program executable would look like this:

```
#!/bin/bash -l
#SBATCH --job-name MyMatlabCompilerTest
#SBATCH --mem=4G
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 1
#SBATCH --ntasks 1
#SBATCH --time 0:30:00

MYCODE=......                    # to be specified

MyMatlabProgram=.......          # to be specified


module load matlab gnu

cd $MYCODE

MyMatlabArgs=""      # optional to be specified

${MYCODE}/$MyMatlabProgram $MyMatlabArgs
```

### 4.1.6 NAG Toolbox for MATLAB

RRZK has licensed NAG Toolbox for MATLAB, a large and most comprehensive single numerical toolkit derived from the well-known NAG Numerical Library. The NAG Toolbox for MATLAB contains more than 1,400 functions that provide solutions to a vast range of mathematical and statistical problems and that both complements and enhances MATLAB.
NAG's collection of numerical functions is organized into more than 40 chapters, each devoted to a mathematical or statistical area, that ease the selection of required algorithms. Each function is accompanied by documentation delivered via MATLAB's native help system or via the web, along with advice on selection of the best algorithm and the interpretation of the results returned.
Each NAG function has an example program to demonstrate how to access it by solving a sample problem. This template can then be easily adapted to reflect the user's specific. NAG tested the validity of each function on each of the machine ranges for which the Toolbox is available.
Functions from the NAG Toolbox for MATLAB might be a lot faster than similar functions from original MATLAB toolboxes, but the NAG Toolbox for MATLAB currently is only available in a non-multi-threaded (sequential) version. Instead, use the MATLAB Parallel Computing Toolbox to further speeding up your MATLAB programs.
Due to a NAG campus license the use of the NAG Toolbox for MATLAB on Cheops is unrestricted, and MATLAB programs using the NAG Toolbox for MATLAB can also be developed on personal computers within the University of Cologne. For more information on obtaining the license key for the NAG Toolbox for MATLAB, contact V.Winkelmann.

https://www.mathworks.com/products/matlab-parallel-server.html
http://nag.com/numeric/mb/calling.asp

## *4.2 Scilab*

Scilab is an interactive platform for numerical computation providing a powerful computing environment for engineering and scientific applications using a language that is mostly compatible with MATLAB.
Scilab is provided as a software module which, when loaded, provides all relevant execution path entries and environment variables set correctly. Scilab is open source software with a GPL compatible license; therefore, there are no license limits on running several Scilab jobs in parallel. Since in the current version there is no Scilab feature for programming parallel tasks within a Scilab program, only sequential Scilab batch jobs are possible. On CHEOPS, a batch script for a sequential Scilab job may look like this:

```
#!/bin/bash -l
#
#SBATCH --job-name raxml-sequential
#SBATCH --output=scilab-sequential-%j.out
#SBATCH --cpus-per-task=1
#SBATCH --mem=16G
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln
module load scilab

MyScilabProgram="$HOME/scilab/example1/myprog.m"

# start Scilab with my Scilab program
time scilab -f $MyScilabProgram -nwni
```

where the variable **MyScilabProgram** refers to the location of the Scilab program to be started within the batch job.
As already explained earlier in this document, the above script can be submitted to the batch system with the call

```
sbatch myprog.sh
```

if **myprog.sh** is the file name of the batch script. For more information on how to use Scilab see

http://www.scilab.org

## *4.3 R for Statistical Computing*

R is a programing language and free software environment for statistical computing under the terms of the GNU General Public License (GPL). On CHEOPS special R versions are provided by modules. For example, loading the default R module **R/4.1.3_system** will provide the known **R** command with a few packages already installed:

```
$ module load R/4.1.3_system
 MODULE : R/4.1.3_system

This Version of R 4.1.3 is an unoptimized system build.
$ R
R version 4.1.3 (2022-03-10) -- "One Push-Up"
...
> installed.packages()
```

If you find all packages needed by your R program, you can start right away with your batch job. Otherwise, you should install missing packages in your home directory.

### 4.3.1 Installing additional packages

The Comprehensive R Archive Network (CRAN) hosts hundreds of additional packages, which are not immediately generated, packed and available with our R builds. Even worse, there might be CRAN packages interfering with each other by using same function names etc. They can be loaded by invoking **library()** within an interactive R session or an R program:

```
> library(package)
```

Should necessary packages be missing, it is possible to install additional packages by building up a private R library in your home directory. To create and fill such a library, you make an R library directory in your home directory (e.g. **$HOME/R/4.1.3**) and set the environment variable **R_LIBS_USER** to its path. Then start R interactively on a login node and invoke **install.packages()** to install a package from the CRAN repository for example:

```
$ mkdir -p $HOME/R/4.1.3
$ export R_LIBS_USER=$HOME/R/4.1.3
$ R

R version 4.1.3 (2022-03-10) -- "One Push-Up"
...

> install.packages("package", repos="https://cran.uni-muenster.de")
```

A lengthy installation protocol might occur with lots of messages including downloads of other required packages. Finally, the requested package and its dependencies are installed

to the desired path given by **R_LIBS_USER**. After loading the newly built package, **help()** provides an overview of its functionality:

```
> library(package)
> help("package")
```

Packages built for a specific R version are not compatible with other versions. When changing the R version used, you need to rebuild all packages of your R library.

### 4.3.2 Batch job running your R program

To run an R program on CHEOPS, you need to submit a job script to the batch system SLURM (see CHEOPS Brief Instructions). For example, your job script **myprog.sh** for a sequential R run using one core only could look like this:

```
#!/bin/bash -l
#SBATCH --ntasks=1
#SBATCH --mem=1gb
#SBATCH --time=01:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module load R/4.1.3_system
export R_LIBS_USER=$HOME/R/4.1.3

# R with my R program with command line arguments
R --vanilla -f myprog.R --args alg2 1000 1.0e-08
```

The requested task taking up to 1 GB of main memory may run up to 1 hour on the allocated core. After loading the R module needed, setting of **R_LIBS_USER** makes your additional packages available. Finally, R is executing your program **myprog.R** provided as argument to option **-f**. Option **--vanilla** takes care that no workspaces will be saved or restored unintentionally, nor will any user profiles or site profiles be loaded prior to execution. You may provide command line arguments to your R program following option **--args**. Please do not use the command processor **Rscript** in your R jobs because its environment differs from that of **R**.

### 4.3.3 Efficiency and parallelization

While R provides a wide variety of statistical methods with easily understandable code, it is not suitable for computationally intensive tasks. It can be more efficient executing those tasks in C++ and integrating them either with the R API or considerably easier with the Rcpp interface. As an introduction to Rcpp is not within the scope of this document, we refer you to its documentation.
Parallelization is another way to make the computation faster. R packages provide various methods with parallel workers, e.g. forking, socket communication or MPI. **Please do not use sockets for workers in your R program**. Such R jobs are not integrated with our batch system SLURM and would jam our Ethernet management network!

### 4.3.4 Multiple workers on single node

For R jobs running multiple workers on a single node, you should load the package parallel and use forking to set up the workers. Either you invoke multicore functions like **`mclapply()`** forking their workers in each call or you make a cluster of workers by forking:

```
library(parallel)
...
ntasks <- strtoi(Sys.getenv(c("SLURM_NTASKS")))
nworkers <- ntasks
cl = makeCluster(nworkers, type="FORK")
```

R reads the number of tasks allocated for the job from the SLURM environment variable **`SLURM_NTASKS`** and uses it to make the cluster of workers. To have multiple workers in your cluster, you have to increase the number of tasks requested in the job script:

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --mem=8gb
#SBATCH --time=01:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module load R/4.1.3_system
export R_LIBS_USER=$HOME/R/4.1.3

# R with my R program using forked workers on single node
R --vanilla -f myprog.R
```

Then the batch system will allocate an according number of cores on a single node to your job. As forking takes a complete copy of all data objects to the workers, you have to increase the requested memory per node with **`--mem`** as well.

### 4.3.5 Multiple workers on multiple nodes

For R jobs running multiple workers on multiple nodes, you have to install **`Rmpi`** first which relies on one of our MPI implementations, e.g. **`openmpi/4.1.1_mpirun`**:

```
module use /opt/rrzk/modules/special
module load openmpi/4.1.1_mpirun
module load R/4.1.3_system
R
…
install.packages("Rmpi",repos="https://cran.uni-
muenster.de",configure.args="--with-Rmpi-type=OPENMPI --with-
mpi= /opt/rrzk/lib/openmpi/4.1.1/icc ")
```

Afterwards you can build **snow** or **snowall** as these packages make using MPI easier. Finally, load the prebuilt package **parallel** followed by either **snow** or **snowfall** and make a cluster of MPI workers:

```
library(parallel)
...
ntasks <- strtoi(Sys.getenv(c("SLURM_NTASKS")))
nslaves <- ntasks-1
cl = makeCluster(nslaves, type="MPI")
```

As the MPI master occupies a task already, one worker less is available for the MPI cluster. In your job script, you now have to request more than one task and memory per worker with **--mem-per-cpu**. To initialize the MPI environment invoke the **R** command with the MPI launcher **mpirun**:

```
#!/bin/bash -l
#SBATCH --ntasks=64
#SBATCH --mem-per-cpu=1gb
#SBATCH --time=01:00:00
#SBATCH --output=%x-%j.out
#SBATCH --account=UniKoeln

module use /opt/rrzk/modules/special
module load openmpi/4.1.1_mpirun
module load R/4.1.3_system
export R_LIBS_USER=$HOME/R/4.1.3

# R with my R program with MPI workers on multiple nodes
mpirun -quiet -np 1 R --vanilla -f myprog.R
```

### 4.3.6 Multi-threading on single node

Some packages support parallelization with threads, e.g. when using threaded functions from MKL. In such case, parallelization is restricted to a single node again like with forking. However, now a single task utilizes more than one core by multiple threads. Therefore, you should request one task using multiple cores in your job script. Additionally, set the environment variable **OMP_NUM_THREADS** to the number of allocated cores per task:

```
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
...
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
```

Please note that you can run either thread or MPI parallel R programs. Using both in so-called hybrid runs is not supported since the MPI stack used is not thread-safe.

### 4.3.7 RStudio Server (Open Source Edition)

The RStudio Server provides a comfortable integrated development environment (IDE) with a browser based interface to an R session that runs remotely on an HPC Cluster node. The browser executes locally on a user's workstation without the necessity of X11 forwarding, thus reducing the amount of data to be transmitted and ensuring a relatively lag free work environment. The IDE includes a syntax-highlighting and auto-completing editor as well as a terminal console.

The appropriate module **rstudio/2022.02.3-492_server** can use either one of the currently two available R modules **R/4.0.2_system** or **R/4.1.3_system**, the latter being the default one. Simply load **R/4.0.2_system** before the **rstudio** module if you prefer this version.

In order to launch the RStudio server, first start an interactive job, then load the **rstudio** module, e.g.:

```
$ salloc -n 1 -c 4 -t 2:00:00 --mem 20G -p interactive-rh7
[…]
$ srun --pty bash
[…]
$ module load rstudio
 MODULE : loading required module R
 MODULE : R/4.1.3_system

This Version of R 4.1.3 is an unoptimized system build.
 MODULE : rstudio/2022.02.3-492_server

Start an interactive job on the interactive partition with
salloc/srun, then execute run-rstudio-server.sh on the
compute node and follow the instructions to establish a ssh
tunnel
```

As the module output suggests, it is necessary to launch the provided script **run-rstudio-server.sh** which will start the RStudio server and print further instructions for opening the ssh tunnel as well as copy pasting the right URL into the local browser:

```
$ run-rstudio-server.sh
Running on node cheops11801 with R=/usr/local/bin/R
TMPDIR=/tmp/nierodal.17181291 PORT=8787
DBFILE=/tmp/nierodal.17181291/database.conf
Run "ssh -nNT -L 8787:cheops11801:8787
nierodal@cheops1.rrz.uni-koeln.de" on your local pc
Access with "localhost:8787" on your local pc browser
```

```
TTY detected. Printing informational message about logging
configuration. Logging configuration loaded from
'/etc/rstudio/logging.conf'. Logging to
'/home/nierodal/.local/share/rstudio/log/rserver.log'.
```

Make sure to open the ssh tunnel in a terminal/powershell and the given URL in your browser as described above and the IDE should appear in the browser window.

Note that RStudio cannot use more than one node.

In order to install additional packages first define a directory in the file ~/.Renviron with the R_LIBS_USER variable, e.g.:

```
$ echo 'R_LIBS_USER=~/RSTUDIO_402' > ~/.Renviron
```

Then either use the console with the "**`install.packages`**(…)" command or the RStudio GUI (Tools -> Install Packages) to install them. There is a CRAN mirror on CHEOPS so in most cases access to internet is not required and an installation from within a compute node will work but some R packages need additional resources, e.g. files from github, and in these cases the console on the login node should be used. Do not start the RStudio Server on the login node as that would bind additional resources and affect the workflow of other users.

https://www.r-project.org
https://cran.r-project.org
https://www.r-project.org/nosvn/pandoc/Rcpp.html

# 5 Bioinformatics applications

On CHEOPS, RRZK provides several software packages for bioinformatics computations like phylogenetic analysis or sequence analysis. All packages can be started within batch jobs, all batch scripts are similar to shell scripts, which might already be used on users' local workstations. For slight differences, see the comments on each product.

## *5.1 RAxML*

RAxML (Randomized Axelerated Maximum Likelihood) is a program for sequential and parallel Maximum Likelihood based inference of large phylogenetic trees. It has originally been derived from fastDNAml, which in turn was derived from Joe Felsentein's dnaml being part of the PHYLIP package.
RaxML exists in four versions. One is a pure sequential program lacking of any parallel code. Another one is parallelized for the Message Passing Interface (MPI). A further one is a multi-threaded version that uses Pthreads to run RaxML in parallel on one cluster node. The last one is a hybrid version of the Pthread and MPI parallelized version, speeding up RaxML significantly in many cases.
RaxML is provided as a software module which, when loaded, provides all relevant execution path entries and environment variables being set correctly. See the following table for the names of the corresponding RaxML executables:

| Version | Program Name on Cheops |
|---|---|
| single-thread | **raxmlHPC** |
| multi-thread, one node | **raxmlHPC-PTHREADS** |
| MPI | **raxmlHPC-MPI** |
| Hybrid, multi-threaded, multi-nodes | **raxml-HYBRID** |

### 5.1.1 Sequential RAxML batch jobs without multi-threading

A simple (single threaded) RAxML batch job on CHEOPS using one processor core may look like this

```
#!/bin/bash -l

#SBATCH --job-name raxml-sequential
#SBATCH --output=raxml-sequential-%j.out
#SBATCH --cpus-per-task=1
#SBATCH --mem=200mb
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load raxml

RAXML=raxmlHPC

DATADIR=/opt/rrzk/software/raxml/RRZK/data
INPUT=$DATADIR/Cryothecomonas.phylip
OUTPUT=Cryothecomonas.sequential

time $RAXML -f a -x 12345 -p 12345 -N 100 -m GTRGAMMA \
            -s $INPUT -n $OUTPUT
```

where the variable **RAXML** refers to the location of the used RaxML executable (here the single threaded version) to be started within the batch job, **INPUT** specifies the input file to be analyzed and **OUTPUT** defines the base name of the generated output files.
As already explained earlier in this document, the above script can be submitted to the batch system with the call

```
sbatch raxml-sequential.sh
```

if **raxml-sequential** is the file name of the batch script.

### 5.1.2 Parallel RAxML batch jobs with multi-threading

A parallel RAxML batch job on CHEOPS using several processors on **one** computing node may look like this:

```
#!/bin/bash -l

#SBATCH --job-name=raxml-pthreads
#SBATCH --output=raxml-pthreads-%j.out
#SBATCH --cpus-per-task=4
#SBATCH --mem=800mb
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load raxml
```

```
RAXML=raxmlHPC-PTHREADS

DATADIR=/opt/rrzk/software/raxml/RRZK/data
INPUT=$DATADIR/Cryothecomonas.phylip
OUTPUT=Cryothecomonas.pthreads

time $RAXML -f a -x 12345 -p 12345 -N 100 -T 4 -m GTRGAMMA \
            -s $INPUT -n $OUTPUT
```

where in the example the batch option **`--cpus-per-task=4`** requests 4 processor cores, setting the shell variable **`RAXML=raxmlHPC-PTHREADS`** forces the script to use the multi-threaded version of RAxML, and finally the RaxML option **`-T 4`** instructs RaxML to use 4 parallel threads during execution.
In the case of multi-core jobs just like a batch job with parallel RaxML threads, the requested wall time is the expected runtime of the whole RAxML job (see sbatch option **`--time`**). You do **not** need to accumulate the individual runtime of each thread; essentially wall time should decrease if running the same job with more parallel threads.
On the other hand memory requirements will rise the more threads are used in parallel, thus the argument of the sbatch option **`--mem`** must be increased**.**
As already explained earlier in this document, the above script can be submitted to the batch system with the call

```
sbatch raxml-pthreads.sh
```

if **`raxml-pthreads.sh`** is the file name of the batch script.

### 5.1.3 MPI parallelized RAxML batch jobs on several computing nodes

A MPI-parallelized RAxML version exists which can run more than one RAxML process in parallel on several computing nodes via MPI (Message Passing Interface) and which allows performing parallel bootstraps, rapid parallel bootstraps, or multiple inferences on the original alignment. However, this version of RAxML is not multi-threaded and allows only distributing single threaded RAxML tasks over the requested number of nodes. This means that all communication is done via MPI, whose latency is larger than in the Pthreads case. Because RAxML also provides *hybrid* version merging MPI and Pthreads, we suggest using that hybrid version rather than the pure MPI version of RAxML.

### 5.1.4 Hybrid parallelized RAxML batch jobs on several computing nodes

A hybrid parallelized RAxML version exists which can run more than one RAxML process in parallel on several computing nodes via MPI (Message Passing Interface) and allows multi-threading within the cores of each requested node, thus reducing time for communication between the RAxML tasks within a single node.

A RAxML batch job on CHEOPS using 2 computing nodes with 8 cores each may look like this:

```
#!/bin/bash -l

#SBATCH --job-name=raxml-hybrid
#SBATCH --output=raxml-hybrid-%j.out
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=22GB
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load raxml

RAXML=raxmlHPC-HYBRID

DATADIR=/opt/rrzk/software/raxml/RRZK/data
INPUT=$DATADIR/Cryothecomonas.phylip
OUTPUT=Cryothecomonas.hybrid-2-1-8

time srun -n 2 \
          $RAXML -f a -x 12345 -p 12345 -N 100 -T 8  \
                 -m GTRGAMMA -s $INPUT -n $OUTPUT
```

where in the example the batch options

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8
```

ask for 2 computing nodes with 1 RAxML process started on each node, requesting 8 cores for each RAxML task (thus using 16 processors in total!). Setting the shell variable **`RAXML=raxmlHPC-HYBRID`** forces the script to use the hybrid parallelized version of RAxML. To benefit from the 2 requested nodes and their processors, RAxML is started via **`srun`**, taking care that only one RAxML process is started on each node and that each process uses 8 Pthreads within a node.

**A note from the RAxML authors:** *The current MPI-version only works properly on several computing nodes if you specify the option* ***`-N`*** *in the command line, since this option has been designed to do multiple inferences or rapid/standard BS searches in parallel! For all remaining options, the usage of this type of coarse-grained parallelism does not make much sense! The MPI-version is for executing large production runs (i.e. 100 or 1,000 bootstraps) on a LINUX cluster. You can also perform multiple inferences on larger datasets in parallel to find a best-known ML tree for your dataset. Finally, the novel rapid BS algorithm and the associated ML search have also been parallelized with MPI.*

In the case of multi-core jobs just like a batch job with parallel RaxML threads, the requested wall time is the expected runtime of the whole RAxML job (see sbatch option **`--time`**).

You do **not** need to accumulate the individual runtime of each thread; essentially wall time should decrease if running the same job with more parallel threads.
On the other hand memory requirements will rise the more threads are used in parallel, thus the argument of the sbatch option **`--mem`** must be increased**.**
As already explained earlier in this document, the above script can be submitted to the batch system with the call

```
sbatch raxml-hybrid.sh
```

if **`raxml-hybrid.sh`** is the file name of the batch script.

### 5.1.5 Hybrid parallelized RAxML batch jobs on several computing nodes (extd.)

The experienced user can run the hybrid-parallelized version of RAxML in an advanced manner if the special hardware architecture of the computing nodes of the cluster Cheops is respected. Each Cheops computing node consists of two Intel Nehalem quad core CPUs (that is 8 processors overall per node). Experiments show that if one starts one RAxML process with 4 threads on each quad core processor (that means 2 RAxML processes, each with 4 threads on one node, and not only one RAxML process with 8 threads as before), performance may increase up to 10 %. A corresponding batch job of the example above would read:

```
#!/bin/bash -l

#SBATCH --job-name=raxml-hybrid
#SBATCH --output=raxml-hybrid-%j.out
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=4
#SBATCH --mem=22GB
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load raxml

RAXML=raxmlHPC-HYBRID

DATADIR=/opt/rrzk/software/raxml/RRZK/data
INPUT=$DATADIR/Cryothecomonas.phylip
OUTPUT=Cryothecomonas.hybrid

time srun -n 4 \
          $RAXML -f a -x 12345 -p 12345 -N 100 -T 4 \
                 -m GTRGAMMA -s $INPUT -n $OUTPUT
```

where in the example the batch options

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=4
```

ask for 2 computing nodes with now **2** RAxML processes started on each node, requesting **4** processor cores for each RAxML task (thus using still 16 processors in total!). Setting the shell variable **`RAXML=raxmlHPC-HYBRID`** forces the script to use the hybrid parallalized version of RAxML. To benefit from the 2 requested nodes and their processors, RAxML is started via **`srun`**, taking care that now **2** RAxML processes are started on each node (in the whole **4**) and that each process now uses **4** Pthreads within a node.

https://cme.h-its.org/exelixis/web/software/raxml/
https://github.com/stamatak/standard-RAxML

## *5.2 MrBayes*

MrBayes is a program for the Bayesian estimation of phylogeny. Bayesian inference of phylogeny is based upon a quantity called the posterior probability distribution of trees, which is the probability of a tree conditioned on the observations. The conditioning is accomplished using Bayes's theorem. The posterior probability distribution of trees is impossible to calculate analytically; instead, MrBayes uses a simulation technique called Markov chain Monte Carlo (or MCMC) to approximate the posterior probabilities of trees.
On Cheops, MrBayes takes advantage of the fact that Metropolis coupling or heating is well suited for parallelization, and MrBayes uses MPI to distribute heated and cold chains among available processors. The maximum number of processors suitable for MrBayes corresponds to the product of the number of chains (**`nchains`**) and the number of simultaneous analyses (**`nruns`**) as specified in the MrBayes block at the end of the Nexus file.
MrBayes is provided as a software module which, when loaded, provides all relevant execution path entries and environment variables being set correctly.

A simple MrBayes batch job on CHEOPS using a standard configuration with 2 analyses of 4 chains each may look like this

```
#!/bin/bash -l

#SBATCH --job-name MyMrBayes
#SBATCH --output MrBayes-test1.nxs-%j.out
#SBATCH --mem=10G
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 8
#SBATCH --cpus-per-task 1
#SBATCH --time 1:00:00
#SBATCH --account=UniKoeln

module load mrbayes

DATADIR=/opt/rrzk/software/mrbayes/RRZK/data
INPUT=test1.nxs
OUTPUT=$INPUT.$SLURM_JOB_ID.log
time srun -n $SLURM_NTASKS mb $DATADIR/$INPUT > $OUTPUT
```

where the batch system is asked to provide 1 computing node for 8 tasks (8 processor cores) for this job (remember: we want to compute 2 analyses a 4 chains each); a memory limit of 10 GB over all cores is requested, and an expected runtime (walltime) of 1 hour.
If the whole number of chains, e.g. the product **nchains** by **nruns**, exceeds 8 you can increase the number of nodes in order to provide more processors to MrBayes and to speed up your program. Note that most nodes on Cheops only provide 8-12 processors; specifying more than 12 tasks per node will reduce your job's priority due to the less availability of nodes with more than 12 processors. MrBayes cannot use more than **nchains** x **nruns** processors, therefore do not request more nodes/tasks as needed!
In the case of multi-node jobs, the requested wall time is the expected runtime of the whole MrBayes job (see sbatch option **--time**).
You do **not** need to accumulate the individual runtime of each node; essentially wall time should decrease if running the same job and having enough chains to be distributed on the requested nodes/tasks.

In the above job, all output files are written to the working directory where the job was sent. Since MrBayes creates several output files, it is recommended to use a separate working directory for each MrBayes batch job to avoid mixing output of different jobs.
Finally, the MrBayes executable **mb** is started via the MPI command **srun** which takes care that **mb** is started in parallel, using **test1.nxs** as input file while the log of MrBayes is directed into the logfile **test1.nxs.<jobid>.log** of the current working directory.
As already explained earlier in this document, the above script can be submitted to the batch system with the call

```
sbatch myprog.sh
```

if **myprog.sh** is the file name of the batch script.

http://mrbayes.csit.fsu.edu/

## *5.3 PhyloBayes-MPI*

PhyloBayes-MPI is a Bayesian Markov chain Monte Carlo (MCMC) sampler for phylogenetic inference exploiting a message-passing-interface system for multi-core computing. The program will perform phylogenetic reconstruction using either nucleotide, protein, or codon sequence alignments. Compared to other phylogenetic MCMC samplers, the main distinguishing feature of PhyloBayes is the use of non-parametric methods for modelling among-site variation in nucleotide or amino acid propensities.
A run of the MCMC sampler program **pb_mpi** will produce a series of points drawn from the posterior distribution over the parameters of the model. Each point defines a detailed model configuration (tree topology, branch lengths, nucleotide or amino acid profiles of the mixture, etc.). The series of points defines a chain.
On Cheops, PhyloBayes is provided as a software module. Since **pb_mpi** is parallelized, several processor cores or even nodes can be used for running chains in order to speed up processing. A simple PhyloBayes batch job on CHEOPS may look like this:

```
#!/bin/bash -l

#SBATCH --job-name MyPhyloBayes
#SBATCH --output PhyloBayes-brpo-%j.out
#SBATCH --mem=4G
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 8
#SBATCH --time 01:00:00
#SBATCH --account=UniKoeln

module load phylobayes

DATADIR=$PBMPI_HOME/data/brpo
INPUT=brpo.ali
CHAINMAME=brpo.chain.$SLURM_JOB_ID

time srun -n $SLURM_NTASKS pb_mpi \
             -d $DATADIR/$INPUT -cat -gtr $CHAINMAME
```

where the batch system is asked to provide 1 computing node for 8 tasks (8 processor cores) for this job; a memory limit of 4 GB over all cores is requested, and an expected runtime (wall time) of 1 hour. In the case of multi-node jobs, the requested wall time is the expected runtime of the whole PhyloBayes job (see sbatch option **`--time`**).
For multi-node jobs, you do **not** need to accumulate the individual runtime of each node. Usually **`pbmpi`** runs until the specified wall time is exceeded and aborts thereafter. For estimating the required memory and runtime, please read the PhyloBayes manual.

In the above job, all output files are written to the working directory from where the job was sent to the batch system. Since PhyloBayes creates several output files, it is recommended to use a separate working directory for each PhyloBayes batch job to avoid mixing output of different jobs.
Finally, the PhyloBayes executable **`pbmpi`** is started via the MPI command **`srun`** that launches **`pbmpi`** in parallel using **`brpo.ali`** as input file.
As already explained earlier in this document, the above script can be submitted to the batch system with the call

```
sbatch myprog.sh
```

if **`myprog.sh`** is the file name of the batch script.

http://phylobayes.org

# 6 Checkpointing

## 6.1 What is checkpointing and why should I use it

If an application features checkpointing, it is able to save the state of your calculations periodically by storing intermediate results in a checkpointing file. When your job aborts for some reason (e.g. job running out of memory, node failure …), you can restart your job from the last checkpoint image file instead of starting over the calculation from the beginning. Checkpointing saves you computing time, provides you with results earlier and yields us a better cluster utilization.
Application-based checkpointing is the safest way of restarting your application if needed. If your application has checkpointing capabilities, please use them. The overhead of writing a checkpoint from time to time is quite low and saves you a lot of work if a restart is needed. If your application is not capable of checkpointing, there is the alternative of external tools, such as DMTCP.

## 6.2 External checkpointing by DMTCP

DMTCP (Distributed Multi-Threaded Checkpointing) is a tool that can checkpoint a whole application run including its data. Currently, our build **`dmtcp/2.6.1rc1_smp`** is able to checkpoint SMP applications only (jobs running on a single node without MPI parallelization). A job script example of a checkpointed run could look like

```
#!/bin/bash -l

#SBATCH --cpus-per-task=4
#SBATCH --mem=1024mb
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load myapp/1.0
module load dmtcp/2.6.1rc1_smp

dmtcp_launch --new-coordinator \
             --ckpt-open-files \
             --allow-file-overwrite myapp arg1 arg2
```

The launcher **`dmtcp_launch`** starts your application myapp and a DMTCP coordinator that takes care of your application by extracting and storing checkpoint data to a DMTCP checkpoint directory at every DMTCP checkpoint interval. Both, the interval and the directory are set automatically when loading the module **`dmtcp/2.6.1rc1_smp`** within your job script. The path of the checkpoint directory **`/scratch/${USER}/ckpt.${SLURM_JOB_ID}`** is referenced on standard error output to facilitate the convenient restart of your application.
If for some reason your job is aborted, you can restart it with the corresponding restart job script

```
#!/bin/bash -l

#SBATCH --cpus-per-task=4
#SBATCH --mem=1024mb
#SBATCH --time=01:00:00
#SBATCH --account=UniKoeln

module load myapp/1.0
module load dmtcp/2.6.1rc1_smp

dmtcp_restart --new-coordinator \
              last_checkpoint_dir/ckpt*.dmtcp
```

Resource requests for the restart should be identical to those of the previous run. For ***last_checkpoint_dir***, you should insert the checkpoint directory of the aborted job. **dmtcp_restart** will restart the whole **myapp** run from the checkpoint files including its environment. However, you should keep track of redirecting standard output to files because DMTCP is not aware of this.

https://dmtcp.sourceforge.io/docs/index.html