

GWDG – Kurs  
Parallel Programming with MPI

# MPI-Introduction

## Exercises

Oswald Haan  
oahan@gwdg.de

# Accessing the GWDG-Cluster

GWDG's Scientific Compute Cluster can be accessed from one of its frontend nodes **gwdm101.gwdg.de**, **gwdm102.gwdg.de** .

Using the alias **login-mdc.hpc.gwdg.de** will connect you to one of these nodes.

To open a shell on the frontend use the **ssh** client with ssh-key authentication, executing on your local terminal the command :

```
> ssh login-mdc.hpc.gwdg.de -l <userid> -i <yourkey>
```

The generation of the ssh-key is described in detail [here](#).

If your local operating system does not include ssh, you can use PuTTY, a free implementation of ssh and Telnet for Windows and Unix platforms, which can be downloaded [here](#) .

Please note, that you can only connect to the cluster frontends from inside the campus network [GÖNET](#). If you are not inside GÖNET, you can either use a [VPN](#) or connect to **login.gwdg.de** first and ssh into the cluster frontend from there.

# Exercises and Examples

- Source code for all exercises and examples in the following directories:  
`~ohaan/mpiexercises/f`  
`~ohaan/mpiexercises/c`  
`~ohaan/mpiexercises/py`
- Copy the codes to your own directory with the command  
*(don't forget the dot (.) at the end of this command)*  
`cp -r ~ohaan/mpiexercises/[f,c,py]/* .`
- Edit, compile, link and start programs  
on the frontend node **login-mdc.hpc.gwdg.de**

# Batch System **slurm**

Submit programs to the GWDG-Cluster with **slurm**

There are a number of nodes reserved for this course

**--reservation=mpi-course**

More information on batch processing on GWDG's website for „High Performance Computing“ :

[https://info.gwdg.de/dokuwiki/doku.php?id=en:](https://info.gwdg.de/dokuwiki/doku.php?id=en:services:application_services:high_performance_computing:running_jobs_slurm)

[services:application\\_services:high\\_performance\\_computing:running\\_jobs\\_slurm](https://info.gwdg.de/dokuwiki/doku.php?id=en:services:application_services:high_performance_computing:running_jobs_slurm)

Links to slurm documentations :

<https://slurm.schedmd.com/pdfs/summary.pdf>

<https://slurm.schedmd.com/documentation.html>

# MPI Program hello\_mpi.f

```
program hello
  implicit none
  include 'mpif.h'
  integer nproc, myid, ierr, pnamelen
  character*(40) pname
c-----
c      start MPI
c-----

  call MPI_INIT( ierr )
  call MPI_COMM_SIZE( MPI_COMM_WORLD, nproc, ierr )
  call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
  call MPI_GET_PROCESSOR_NAME( pname, pnamelen, ierr )
  write(6,*)nproc, myid, pname
  call MPI_FINALIZE (ierr)

  stop
end
```

# MPI Program `hello_mpi.c`

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

int main(int argc, char **argv)
{
    int np, me, resultlen;
    char name[41];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Get_processor_name(name, &resultlen);

    printf("hello %i %i %s \n", np, me, name);

    MPI_Finalize();
    return 0;
}
```

# MPI Program `hello_mpi.py`

```
# hello_mpi
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
name = MPI.Get_processor_name()

print(size, rank, name)
```

# Compile and Link for openmpi

- Source code in directory: `mpiexercises/f/Start`  
`mpiexercises/c/Start`
- Load modules
  - > `module load openmpi`or: Load modules by sourcing from the parent directory the script `modules_openmpi.x` containing the `module load` commands:
  - > `. ../modules_openmpi.x`
- compile and link the MPI program `hello_mpi`:
  - > `mpifort -o hello_mpi.exe hello_mpi.f`
  - > `mpicc -o hello_mpi.exe hello_mpi.c`or: use the makefile provided in directory `Start`  
**(!!set `FC=mpifort` resp. `CC=mpicc` in the makefile !!)**
  - > `make hello_mpi`



# Compile and Link for intel-mpi

- Source code in directory: `mpisexercises/f/Start`  
`mpisexercises/c/Start`
- Load modules:
  - > `module load intel-oneapi-compilers intel-oneapi-mpi`or load modules by sourcing from the parent directory the script `modules_intel.x` containing the `module load` commands:
  - > `./modules_intel.x`
- compile and link the MPI programm `hello_mpi`:
  - > `mpiifort -o hello_mpi.exe hello_mpi.f`
  - > `mpiicc -o hello_mpi.exe hello_mpi.c`or: use the makefile provided in directory `Start`  
**(!!set `FC=mpiifort` `rsp.` `CC=mpiicc` in the makefile !!)**
  - > `make hello_mpi`

# Preparing the environment for mpi4py

- Source code in directory: **mpisexercises/py/Start**
- Load modules for openmpi and mpi4py :  
    > **module load openmpi python**  
or: Load modules by sourcing from the parent directory the script **modules.x** containing the **module load** commands :  
  
    > **. ../modules.x**
- No compile and link steps for python:  
    python is an interpreted language

# Starting a MPI-Job on Frontend

Fortran and C:

```
> mpirun -n 4 ./hello_mpi.exe
```

Python:

```
> mpirun -n 4 python ./hello_mpi.py
```

# Starting a MPI-Job interactively on Cluster

Allocate resources on the cluster with **salloc**

```
> salloc --reservation=pchpc-2023
        -N 2 --ntasks-per-node=2
```

Or source the command script **salloc.cmd** in directory **Start**

```
> . salloc.cmd
...
salloc: Nodes dmp[042-043] are ready for job
bash-4.2$ hostname
gwdu103
bash-4.2$ mpirun ./hello_mpi.exe
         4           2 dmp043
         4           3 dmp043
         4           1 dmp042
         4           0 dmp042
bash-4.2$ exit
...
>
```

# Submit a MPI-Job to the Cluster with `srun`

```
> srun -p medium -N 2 --tasks-per-node=2 ./hello_mpi.exe
```

```
...
```

```
4      3 amp067  
4      2 amp067  
4      1 amp049  
4      0 amp049
```

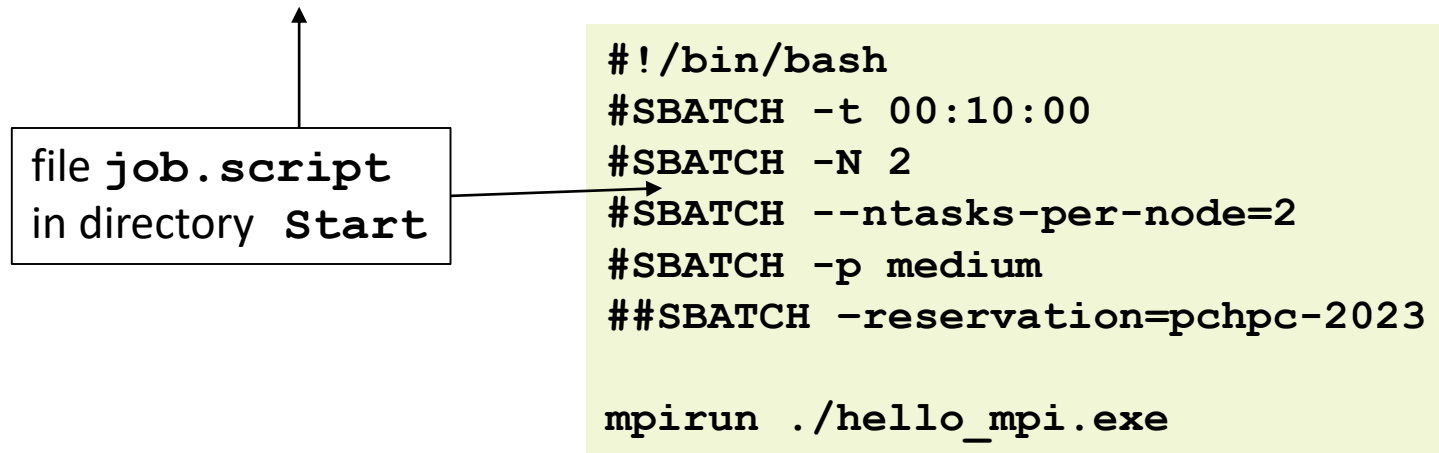
```
>
```

# Starting a MPI-Job in Batch Mode

Batch-job on cluster with slurm **sbatch** command:

```
> sbatch -p medium  
      -N 2 --ntasks-per-node 2  
      --wrap="mpirun ./hello_mpi.exe"
```

Batch-job on cluster with job script:



```
> sbatch job.script
```

Output in file **slurm-<job-id>.out**

# Some Slurm Commands

**sbatch** parameters:

- **-n <tasks>**

The number of tasks for this job. The default is one task per node.

- **-N <minNodes ,maxNodes>**

Minimum and maximum number of nodes for the job

- **--ntasks-per-node=<ntasks>**

Number of tasks per node.

- **-o <output>**

Output file for the job

**sinfo** shows partitions

**squeue -u <username>** shows submitted jobs

**scontrol show job <jobID>** shows particular job

**scancel <jobID>** cancels particular job

# Exercise 1

Connect to the login node **login-mdc.hpc.gwdg.de**

Copy the directories `mpiexercises/[f,c,py]/*`

Load the necessary module files

Compile the `hello_mpi` program

Run the `hello_mpi` executables

- locally
- interactively on the cluster using **salloc**
- as a batch job using **sbatch**

Try different numbers of nodes and tasks per node



## Exercise 2

Modify the hello\_mpi program:

Let the process with task number 0 inquire the MPI version of the implementation and write the result on standard output.

Use the MPI routine `MPI_GET_VERSION`

Syntax:

**Fortran** :

```
integer version, subversion, ierror  
MPI_GET_VERSION(version, subversion, ierror)
```

**C**:

```
MPI_Get_version(int *version, * subversion)
```

**mpi4py**:

```
version = MPI.Get_version()
```

## Exercise 3

The MPI program: `prime_mpi.f`, `prime_mpi.c`, `prime_mpi.py` contains the following code to determine if a given integer `nprime` is a prime number

```
nprime = 223456731
ntest = (nprime+1)/2
nd = 0
do i = 3 , ntest
    nr = mod(nprime,i)
    if (nr.eq.0) then
        nd = nd+1
        write(6,*)myid, ' : ', i
    end if
end do
write (6,*) ' found',nd,'divisors,
```

## Exercise 3

Run the code on  $nproc = 1, 2, 3, \dots$  processes

This replicates the calculation  $nproc$  times

Now let every process test a different set of possible divisors:  
(*Embarrassingly parallel distribution of workload*)

Split the set  $(3, \dots, nprime/2)$  into  **$nproc$**  different sets,  
one for each task with taskid  **$myid$**

## Exercise 3

divide `ntest` in `nproc` pieces of maximal size `nloc`  
`nloc` = integer part of  $(n_{\text{test}} + n_{\text{proc}} - 1) / n_{\text{proc}}$

Task `myid` tests divisors `i` ranging from  
`max(3, myid * nloc + 1)` to  
`min((myid + 1) * nloc, ntest)`

`ntest = 13, nproc = 3, nloc = 5`



Modify the code in: `prime_mpi.f`, `prime_mpi.c`, `prime_mpi.py`

## Exercise 3

Alternatively:

Every task tests divisors **nproc** numbers apart,  
starting from **3+myid**



replace

```
do i = 3 , ntest
  for( i = 3; i<=ntest; i = i+1 )
    for i in range(3,ntest,1)
```

by

```
do i = 3+myid , ntest , nproc
  for( i = 3+myid; i<=ntest; i = i+nproc )
    for i in range(3+myid,ntest,nproc)
```

Modify the code in: **prime\_mpi.f**, **prime\_mpi.c**, **prime\_mpi.py**

## Exercise 4: MPI\_WTIME

Real-time (elapsed time) in sec. : `MPI_WTIME()`  
Granularity of WTIME in sec : `MPI_WTICK()`

Timing a code

```
double precision t
t = MPI_WTIME()
  code segment to be timed
t = MPI_WTIME() - t
print*, 'secs for code-segment', t
```

## Exercise 4: Properties of MPI\_WTIME

### *Fortran, C*

compile `zeit_mpi.f`, `zeit_mpi.c` (using `make zeit_mpi`)

Run `zeit_mpi.exe` on one process :

```
> mpirun -n 1 ./zeit_mpi.exe
```

### *mpi4py:*

Run `zeit_mpi.py` on one process :

```
> mpirun -n 1 python ./zeit_mpi.py
```

## Exercise 5: Timing the prime-program

Use calls to `MPI_WTIME` to determine the time for testing the numbers in each task.

Compare the computing times for the case without distributing the tests to the two ways to distribute the tests



## Exercise-6 : Computing Speed of a Single Core

Speed of Matrix-Vector Multiplication

Source code in directory `mpisexercises/f/MV-seq`

Compile `dgemv.f` `time_dgemv.f` using `makefile`

Using optimizing-levels low `-O0` and high `-O3`

`(make time_dgemv_00, make time_dgemv_03)`

Use numerical library: MKL

`module load intel-oneapi-mkl`

`(make time_dgemv_mkl)`

Compare to peak performance of **20 Gflop/s** (2,5 GHz clockrate)

## Exercise-6 : Computing Speed of a Single Core

Speed of Matrix-Vector Multiplication

Source code in directory `mpisexercises/c/MV-seq`

Compile `dgemv.c` `time_dgemv.c` using `makefile`

Using optimizing-levels low `-O0` and high `-O3`

`(make time_dgemv_00, make time_dgemv_03)`

Use numerical library: MKL

`module load intel-oneapi-mkl`

`(make time_dgemv_mkl)`

Compare to peak performance of **20 Gflop/s** (2,5 GHz clockrate)

# Exercise-6 : Computing Speed of a Single Core

Computing Speed of a Single Core with python

Matrix-Vector Multiplication

in directory `mpisexercises/py/MV-seq`

```
> python timing_mv.py
```