GWDG – Kurs
Parallel Programming with MPI

MPI-Introduction
Exercises

Oswald Haan
ohaan@gwdg.de

Accessing the GWDG-Cluster

GWDG's Scientific Compute Cluster can be accessed from one of its frontend nodes **login-fas.hpc.gwdg.de** and **login-mdc.hpc.gwdg.de**

For this course use frontend **login-fas.hpc.gwdg.de**
(old name, still valid, is: **gwdu103.gwdg.de**)

To open a shell on the frontend use the **ssh** client with ssh-key authentication, executing on your local terminal the command :
> ssh gwdu103.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 a front end from there.

Exercises and Examples

- Source code for all exercises and examples in the following directories:
  ~ohaan/Uebungen_f
  ~ohaan/Uebungen_c
  ~ohaan/Uebungen_py

- Copy the codes to your own directory, e.g. with the command
  cp -r ~ohaan/Uebungen_py/* .

- Edit, compile, link and start programs
  on the frontend node **gwdu103**

Batch System `slurm`

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

There are 8 nodes with 2x48 cores each, reserved for this course:

`--partition medium`

`--reservation pchpc-course`

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

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: **Uebungen_f/Start**
  Uebungen_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**

```
> make hello_mpi
```

Compile and Link for **intel-mpi**

- Source code in directory: **Uebungen_f/Start**
  Uebungen_c/Start

- Load modules:

```
> module load intel-oneapi-compilers
> module load 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: **`Uebungen_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 -p medium --reservation pchpc-course
                   -N 2 --ntasks-per-node 2
```

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

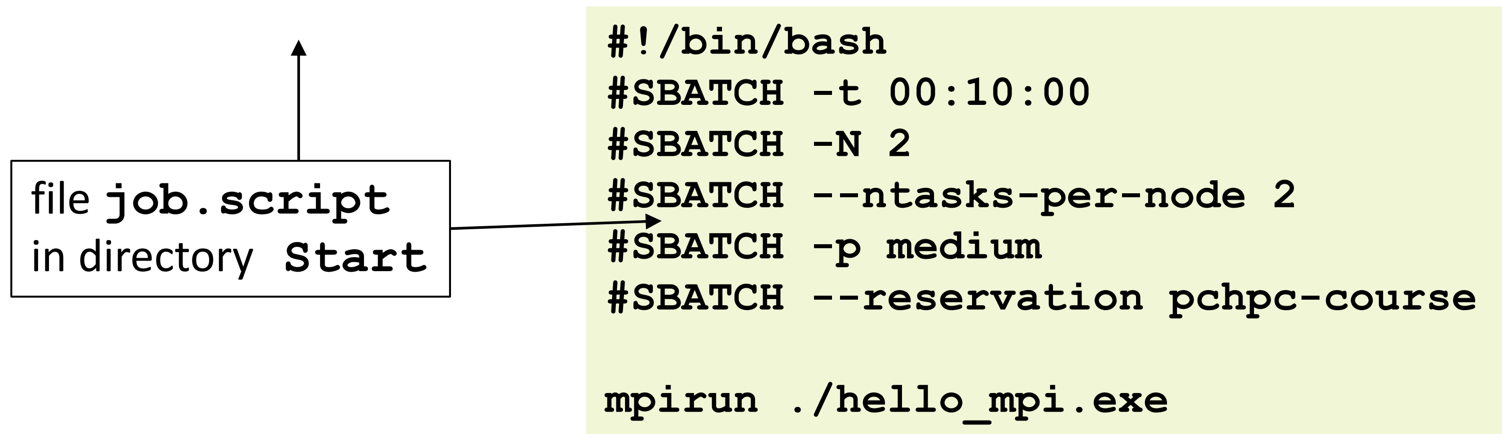
```
> . salloc.cmd
...
salloc: Nodes dmp[042-043] are ready for job
bash-4.2$ hostname
gwdu103
bash-4.2$ mpirun ./hello_mpi.exe
           4           2 dmp043
           4           3 dmp043
           4           1 dmp042
           4           0 dmp042
bash-4.2$ exit
...
>
```

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:




```
#!/bin/bash
#SBATCH -t 00:10:00
#SBATCH -N 2
#SBATCH --ntasks-per-node 2
#SBATCH -p medium
#SBATCH --reservation pchpc-course

mpirun ./hello_mpi.exe
```

```
> 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

scontroll show job <jobID> shows particular job

scancel <jobID> cancels particular job

Exercise - 1

Connect to the login node gwdu103

Copy the directories Uebungen_...

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 taks 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 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 = 118845
ntest = sqrt(dble(nprime))+1
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, … 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 (1,…,ntest) 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 **`(ntest+nproc-1)/nproc`**

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

```
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 **Uebungen_f/MV-seq**

Compile **dgemv.f** **time_dgemv.f** **using** **makefile**

Using optimizing-levels low **–O0** and high **–O3**

(**make time_dgemv_O0, make time_dgemv_O3**)

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 **`Uebungen_py/MV-seq`**

```
> python timing_mv.py
```