

GWDG – Kurs
Parallel Programming with MPI

MPI-Introduction

Exercises

Oswald Haan
oahan@gwdg.de

Outline

- 4 Steps to Start Parallel Computing
 - Access the GWDG Cluster
 - Copy the Source Code
 - Compile the MPI Program
 - Run the Executable
- Exercises
 - Hello World
 - Prime Number Test
 - Timing

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:
`~oahan/mpiexercises/f`
`~oahan/mpiexercises/c`
`~oahan/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 ~oahan/mpiexercises/[f,c,py]/* .`
- Edit, compile, link and start programs
on the frontend node **login-mdc.hpc.gwdg.de**

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:
 - > `source ../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:
 - > `source ../modules_intel.x`
- compile and link the MPI programm `hello_mpi`:
 - > `mpiifort -o hello_mpi.exe hello_mpi.f`
 - > `mpiicx -o hello_mpi.exe hello_mpi.c`or: use the makefile provided in directory `Start`
(!!set `FC=mpiifort` `rsp.` `CC=mpiicx` 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 py-mpi4py anaconda3 openmpi`or: Load modules by sourcing from the parent directory the script `modules.x` containing the `module load` commands :

```
> source ../modules.x
```

(anaconda3 provides the numpy-package, the order of loading the modules is important!)

- 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
```

Batch System **slurm**

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

There are a number of nodes reserved for this course

--reservation=<name>

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>

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
```

```
>
```

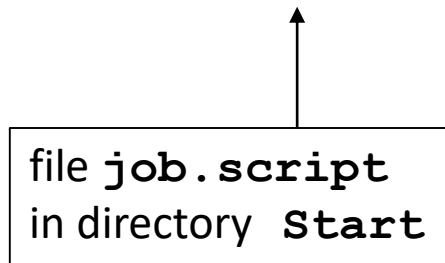
Starting a MPI-Job in Batch Mode

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

```
> sbatch -p medium
      -N 2 --ntasks-per-node 2
      --wrap='source ../modules_intel.x;
              mpirun ./hello_mpi.exe'
```

Batch-job on cluster with job script:

file **job.script**
in directory **Start**



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

source ../modules_intel.x
mpirun ./hello_mpi.exe
```

```
> sbatch job.script
```

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

Starting a MPI-Job interactively on a Cluster_Node

Allocate resources on the cluster with **salloc**

```
> salloc -p medium -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$ mpirun ./hello_mpi.exe
```

```
4 2 dmp043
```

```
4 3 dmp043
```

```
4 1 dmp042
```

```
4 0 dmp042
```

```
bash-4.2$ exit
```

```
...
```

```
>
```

Options for Slurm Commands

- **-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.
If **-n** and **--ntasks-per-node** is specified,
this options specifies the maximum number tasks per node.
- **-o <output>**
Output file for the job.
- **--reservation=<name>**
use reserved resources.

Useful Slurm Commands

- **sinfo**
shows partitions
- **squeue -u <username>**
shows submitted jobs
- **scontrol show job <jobID>**
shows particular job
- **scancel <jobID>**
cancels particular job

Solution for Exercises

If you have tried hard to perform the required exercises and the programs still don't work, you are allowed to look into the directories

`~ohaan/mpisolutions/f`

`~ohaan/mpisolutions/c`

`~ohaan/mpisolutions/py`

where you will find the completed programs for some exercises

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:

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 programs: `prime_mpi.f`, `prime_mpi.c`, `prime_mpi.py` contain code to find divisors of a given integer `nprime` :

Fortran version

```
nprime = 223456731
ntest = sqrt(real(nprime))
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

Python version

```
nprime = 223456731
ntest = int(math.sqrt(nprime))
nd = 0
for i in range(3,ntest):
    nr = nprime%i
    if nr == 0 :
        nd = nd+1
        print(rank, ' : ', i,int(nprime/i))
print ( ' found',nd,'divisors')
```

Exercise 3

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

This replicates the calculation $nproc$ times

Parallel Processing

Now let every process test a different set of numbers for possible divisors:

(Embarrassingly parallel distribution of workload)

Split the set $(3, \dots, ntest)$ of numbers to be tested into **$nproc$** different sets,
one for each task with taskid **$myid$**

Workload Distribution:

Distribute n items to np processes

n_l : local number of items on process $myid$

Goal: **Distribution of items as uniform as possible:**
Minimize the maximal local number n_l

define the largest integer smaller than n/np :

n_{la} = integer part of $(n+np-1)/np$

the local number n_l of strips for process $myid$:

```
if  $n-myid*n_{la} > 0$  :       $n_l = \min(n_{la}, n-myid*n_{la})$ 
else :                       $n_l = 0$ 
```

Distribute n items to np processes

Examples

```
n=11      np=3
nla=integer part of (11+3-1)/3 = 4
  myid    n-myid*nla    nl=min(nla, n-myid*nla)
  0       11            4
  1       7             4
  2       3             3
```

```
n=7       np=5
nla=integer part of ( 7+5-1)/5 = 2
  myid    n-myid*nla    nl=min(nla, n-myid*nla)
  0       7             2
  1       5             2
  2       3             2
  3       1             1
  4      -1             0
```

Exercise 3

divide `ntest` in `nproc` pieces of maximal size `nla` :
`nla` = integer part of $(n_{test} + n_{proc} - 1) / n_{proc}$
(`nla` * `nproc` >= `ntest`)

Size of the set of numbers to be tested on process `myid` :
`nl` = `min(nla, n - myid * nla)`

Task `myid` tests numbers `i` ranging :
from `max(3, myid * nla + 1)` to `myid * nla + nl`

`ntest = 13, nproc = 3, nla = 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

Fortran

```
do i = 3 , ntest
```

C

```
for( i = 3; i<=ntest; i = i+1 )
```

Python

```
for i in range(3,ntest,1)
```

by

Fortran

```
do i = 3+myid , ntest , nproc
```

C

```
for( i = 3+myid; i<=ntest; i = i+nproc )
```

Python

```
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

Fortran :

```
double precision t
t = MPI_WTIME()
  code segment to be timed
t = MPI_WTIME() - t
print*, "secs for code-segment:", t
```

C :

```
double t;
t = MPI_Wtime();
  code segment to be timed
t = MPI_Wtime() - t;
printf("secs for code-segment : %e \n", t);
```

Python :

```
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 in each task.

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

Set

`ntest = nprime - 1`

in order to get larger execution times

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 **10 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
```