Practical hints for the NAISS MPI course

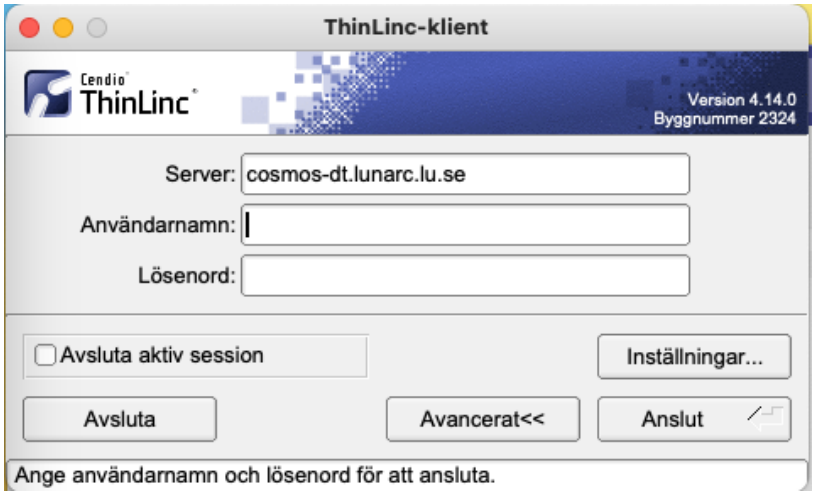
Connecting

COSMOS @ LUNARC

Option 1 (suggested)

Install the Thinlinc client for your OS (https://www.cendio.com/thinlinc/download)

Use the server name: **cosmos-dt.lunarc.lu.se**
and the username/password you got from LUNARC



A more detailed information can be found here: https://lunarc-documentation.readthedocs.io/en/latest/getting_started/using_hpc_desktop/

Option 2

Open a terminal (on some windows systems you might need an emulator) on your local system and connect via ssh. Use the server name: **cosmos.lunarc.lu.se**
and the username/password you got from LUNARC:

```
ssh cosmos.lunarc.lu.se -l username
```

Running and building MPI codes

For the following you should open a terminal window inside the HPC desktop and work at the command line prompt. A number of editors are available on COSMOS to edit code, scripts etc.

Setting up the environment

For the practical parts of the course we are using the GCC compiler suite and the OpenMPI MPI library. To get a modern GCC version of these, we recommend loading the foss/2021b module:

module load foss/2022b

When doing the course in Python, you also need to get access to Python with MPI4PY. In addition to foss/2022b, you need to additionally load the following:

module load Python/3.10.8
module load SciPy-bundle/2023.02
module load mpi4py/3.1.4

This will provide you with Python 3.10.8, including NumPy, SciPy, MPI4Py.

Getting the templates

We need to obtain the distribution

git clone https://github.com/MPI-course-collaboration/MPI-course.git
cd MPI-course/

If you need to refresh, e.g. due to updates, please do

git pull origin

Lecture nodes

The repository contains PDFs of the course's lectures. They are located in the directory Lectures of the repository. There is a separate directory for each day of teaching.

Compiling code

This section applies to Fortran and C/C++ users. When using Python you do not need to compile code.
To compile MPI code it is customary to utilise wrapper scripts. These call the underlying compiler and tell it where to look for MPI headers files and libraries.

cd Templates/Day_1/RunningCode/

Fortran compilation

With OpenMPI Fortran code is compiled with **mpifort**. Use the compiler options of the underlying Fortran compiler, gfortran in this case:

mpifort -O3 -march=native -o mpihello hello_mpi.f90

C/C++ code compilation

With OpenMPI, C code is compiled with **mpicc**. Use the compiler options of the underlying Fortran compiler, gfortran in this case:

mpicc -O3 -march=native -o mpihello hello_mpi.c

For C++ code the wrapper **mpiCC** can be utilised.

Submission script

To get performance out of an MPI code, it is crucial to have dedicated hardware. In a cluster environment this requires running on one or more backend nodes using a batch script. Within NAISS we use SLURM.
Sample scripts for the use on COSMOS are provided in the subdirectory named

- LUNARC

Running compiled code (C, Fortran, C++)

Copy the relevant sample script into the current directory

- **cp LUNARC/run_hello_lunarc.sh run_hello.sh**

and add the **accounting string** (mandatory) and **reservation** name to the script:

```
#SBATCH -A lu2023-7-75
```

To get a faster turnaround, we recommend using the relevant reservation name string for each day:

```
#SBATCH --reservation=mpi-course
#SBATCH --reservation=mpi-course-2day
#SBATCH --reservation=mpi-course-3day
#SBATCH --reservation=mpi-course-4day
```

You may use nano or another editor of your liking. It is best practise to set up the environment inside the script. The task count is set with the line

#SBATCH -n 4

(in the example above, for 4 tasks).

For MPI running the crucial line is the last row of the script:

mpirun mpihello

Submit

sbatch run_hello.sh

Monitor job progress with

squeue -u *userid*

If successful you get an output file of the form: **result_mpihello_10597613.out.** The number is the unique job number.

Running Python code with MPI4PY

Copy the relevant sample script into the current directory

- **cp LUNARC/run_hello_python_lunarc.sh run_hello_python.sh**

and add the **accounting string** and **reservation** name to the script. You may use nano or another editor of your liking. It is best practise to set up the environment inside the script. The task count is set with the line

#SBATCH -n 4

For MPI running the crucial line is the last row of the script:

srun python3 hello_mpi.py

Submit

sbatch run_hello_python.sh

Monitor job progress with

squeue -u *userid*

If successful you get an output file of the form: **result_mpihello_10597613.out.** The number is the unique job number.

Hello World code

Change directory

cd ../HelloWorld

Copy your submission script(s)

cp ../RunningCode/run_hello*.sh .

Write code in C, Fortran or Python to query the size of a communicator and the rank number

Compile the code in case of C and Fortran

Run the code in the batch system

Point-to-Point communication - calculate Pi

Change into the directory

cd ../CalculatingPi

Within the directories **C/**, **Fortran/** and Python/ we provide you with a working serial code. Choose a programming language and copy the relevant code, e.g.:

cp C/pi_square_serial.c pi_square.c
cp Fortran/pi_square_serial.f90 pi_square.f90
cp Python/pi_square_serial.py pi_square.py

Check that you can run the serial code and that it produces correct results. Parallelise the code using MPI.

Steps to be taken for the exercise:

- Initialize MPI
- Query the properties of the MPI ranks (size, and each process's rank)
- Divide up the work among the ranks
- Aggregate partial results
- Finalize the program

Message around a ring

Start in the folder with the material

cd MPI-course/Templates/Day_2/MessageAroundRing

We have template codes for C, Fortran and Python

Fortran demo

Go into the Fortran folder

cd Fortran

Copy the file: **ringsend.f90** and open the copy. Do not work in the original file.

Base communicator

The code has a subroutine **setup_mpi**, which

- Sets up base communicator **my_world**
- Returns a handle to the base communicator
- Returns the rank in base communicator
- Returns the size of base communicator

Search for "*Implement base communicator here*" and implement the functionality

Neighbours

Each task needs neighbours up- and down-stream. This is with periodic boundaries. The highest ranking task has rank zero as up neighbour, while rank zero has the highest ranking task as down neighbour.
Implement this after the comment "*Implement neighbours here*".

Data exchange

The actual exchange is to be implemented in the subroutine **exchange**. Implement it with either MPI_Ssend or MPI_Issend to make sure it is correct. Once it is correct change for MPI_Send or MPI_Isend.
Implement this after the comment "*Implement data exchange here*"
The data exchange needs to be executed more than once, for the messages to reach back their origin. Implement this in the main program at the comment "*Implement the travel around the ring here*"

Shutdown

The MPI library needs closing down. Implement this in the subroutine **shutdown_mpi**.
Implement this after the comment "*Implement shutdown of MPI library*"

C demo

Go into the C folder

cd C

Copy the file: **ringsend.c** and open the copy. Do not work in the original file.

Base communicator

The code has a function **setup_mpi**, which

- Sets up base communicator **my_world**
- Returns a handle to the base communicator
- Returns the rank in base communicator
- Returns the size of base communicator

Search for "*Implement base communicator here*" and implement the functionality

Neighbours

Each task needs neighbours up- and down-stream. This is with periodic boundaries. The highest ranking task has rank zero as up neighbour, while rank zero has the highest ranking task as down neighbour.
Implement this after the comment "*Implement neighbours here*".

Data exchange

The actual exchange is to be implemented in the function **exchange**. Implement it with either MPI_Ssend or MPI_Issend to make sure it is correct. Once it is correct change for MPI_Send or MPI_Isend.
Implement this after the comment "*Implement data exchange here*"
The data exchange needs to be executed more than once, for the messages to reach back their origin. Implement this in the main program at the comment "*Implement the travel around the ring here*"

Shutdown

The MPI library needs closing down. Implement this in the function **shutdown_mpi**.
Implement this after the comment "*Implement shutdown of MPI library*"

Python Demo

Go into the Python folder

cd Python

Copy the file: **ringsend.py** and open the copy. Do not work in the original file.

Base communicator

To set up the MPI communication, the code needs to

- Sets up base communicator **my_world**
- Provide a handle to the base communicator
- Query the rank in base communicator
- Query the size of base communicator

Search for "*Implement base communicator here*" and implement the functionality

Neighbours

Each task needs neighbours up- and down-stream. This is with periodic boundaries. The highest ranking task has rank zero as up neighbour, while rank zero has the highest ranking task as down neighbour.
Implement this after the comment "*Implement neighbours here*".

Data exchange

The actual exchange is to be implemented in the function **exchange**. Implement it with either MPI_Ssend or MPI_Issend to make sure it is correct. Once it is correct change for MPI_Send or MPI_Isend.
Implement this after the comment "*Implement data exchange here*"
The data exchange needs to be executed more than once, for the messages to reach back their origin. Implement this in the main program at the comment "*Implement the travel around the ring here*"

Shutdown

This is not required with MPI4PY

Collective communication

Templates for the collective calls are provided in the subdirectory:

MPI-course/Templates/Day_2/Collectives/Examples

and a 2D integration example can be found in:

MPI-course/Templates/Day_2/Collectives/Integration2D

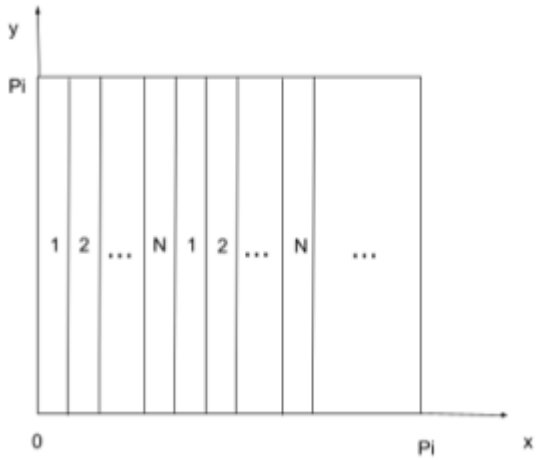
of your cloned repository with the C/Fortran examples. In the Readme.md file, you can find instructions on how to run the templates by using a Makefile.

2D Integration

In this example you will calculate the double integral:

$$\int_0^{\pi}\int_0^{\pi} sin(x + y)dxdy = 0$$

One way to parallelize this calculation is as follows. We start by discretizing the range in both x and y into bins of equal size. This will create a grid of *nxn* bins where *n* is an input parameter provided by the user. We can parallelize the code in one direction only, for instance in the “x” direction. In this case, the first bin in “x” will be computed by process 1, the second bin by process 2, and so on up to the process N. This procedure will be repeated until the last bin n is computed, see the figure below.



Notice that during the computation of each bin in “x” you can perform the regular (not parallel) integration in “y” over the entire range 0<y<π. As a first approximation, we can take the “volume” generated by each grid element (i,j) being equal to $\sin(x_i+y_j)\Delta x\Delta y$, with $\Delta x=/n$, $\Delta y=/n$ and x_i,y_j being the center of the grid element. Each process can accumulate a local value of the integral which can be “reduced” into a total integral value.

This problem is useful for two reasons, first we know the exact value of the integral (0) and second because it includes a double integral, the computation is heavier enough to detect the effects of a parallel implementation.

Debugging code

Preparation of the environment

Start in the folder with the material

```
cd MPI-course/Templates/Day_3/Debugging/Demo
```

Set up the environment

Fortran/C/C++ programming

You can continue to use foss/2022b

```
module purge
module load foss/2022b
```

Python programming

The version of Forge on Kebnekaise does not support Python 3.9, which is available for foss/2021a. You have to use foss/2020b to make the debugger work on Python code.

```
module purge
module load foss/2022b
module load SciPy-bundle/2023.02
module load mpi4py/3.1.4
```

Load the debugger

For all programing languages 2e need to add the module for the Linaro Forge tools. Load the module:

```
module load linaro_forge/23.0.3
```

Submission scripts

Copy the relevant sample script into the current directory. For Fortran, C and C++

```
cp LUNARC/run_debug_lunarc.sh run_debug.sh
```

For Python use

```
cp LUNARC/run_debug_python_lunarc.sh run_debug_python.sh
```

Building Fortran/C code for debugging

You need to compile without optimisation and with the **-g** option.

Fortran example:

mpif90 -g -O0 -o mpihello hello_mpi.f90

C example:

mpicc -g -O0 -o mpihello hello_mpi.c

Debugging

Start the gui on the frontend

ddt &

The &-sign is important to keep control on the commandline.

Make sure you have a **mpihello** build from Fortran and submit the script.

sbatch run_debug.sh

Once it runs, it will connect to the gui. Accept the connection in gui.

Points to look for in Fortran

- Debugger starts after **MPI_Init**
- Focus on group vs process
- Variables have “mini-graphic”
- Click through the tasks to show the values
- Show working of the **MPI_Barrier**
- Show “Restart session” in File-pull-down. Saves loosing ones backend job!

Build a C version and run

mpicc -g -O0 -o ringsend ringsend_sol.c

modify the batch script for the new executable name and submit

sbatch run_debug_python.sh

Points to look for

- It starts in MPI_Init in a subroutine!
- Use “step-out” to get out of that routine
- step-on

- Use “step into”-button to get into “exchange”
- show data is actually received in “MPI_Wait”

Modify the python runscript to run: **pi_square_parallel_blocked.py**

When starting:
- Locate your script in “Stacks”
- Set breakpoint in executable statement the current line
- Run to breakpoint

To view variables
- Locals window
- Current Line(s) window
- You can add to Evaluations and Compare in these windows

To enter into a function
- “step into” button
- right click on function name and “add breakpoint in <name>”

Exercises

In the folder Exerices there are four codes that have problems. Use the debugger to investigate.

Splitting communicators

Sending derived data types in C and Fortran

Point to point communication

In the template section you find a C and Fortran template with a derived data type named **ri_pair**. The derived data type consists out of a double and a default integer number. The provided template initialises MPI and has the code to transfer a derived data type (Fortran/C) from rank 1 to rank 0.
We recommend copying the template before starting modifying. It is your task to create an MPI_Datatype suitable for transferring the data in a single call.

Advanced exercise: user reduction

If you succeeded in creating the MPI_Datatype, you can implement a user reduction, which

- Sums the double element
- Determines the maximum of the integer element

over all tasks involved in the collective.

Sample solutions

Sample solutions are provided for a code which does the point to point transfer as well as the reduction. The sample solution does not implement a pad and is not suitable to transfer arrays of **ri_pair** elements.

Sending NumPy-objects in Python

In the template section you find a Python template for transferring NumPy objects using MPI collectives. The purpose of the code is to calculate the dot product of two vectors using Scatterv and Reduce.

Performance lab

Exercise instructions, are on the slides.

Clues to the debugging exercises

Deadlock in Pi

When collecting the results, rank 0 tries to receive data from himself (rank 0 is hardcoded). It should loop over the rank numbers 1 to n-1.

Deadlock in Ring

The code is using blocking MPI_Recv. It should be using non-blocking MPI_Irecv and matching MPI_Wait

Strange 2D integration

This code has two issues:

1. When using MPI_Bcast to share the problem size, it uses rank 1 as root. However the input is read on rank 0. Hence rank 0 should be the root of the MPI_Bcast
2. When printing the results it wants to print from rank 1, thought the result is known on rank 0

Wrong results in Pi