Assignment 2: Task Farming

Practical information

Deadline: Monday 6/10 23.59

Resources:

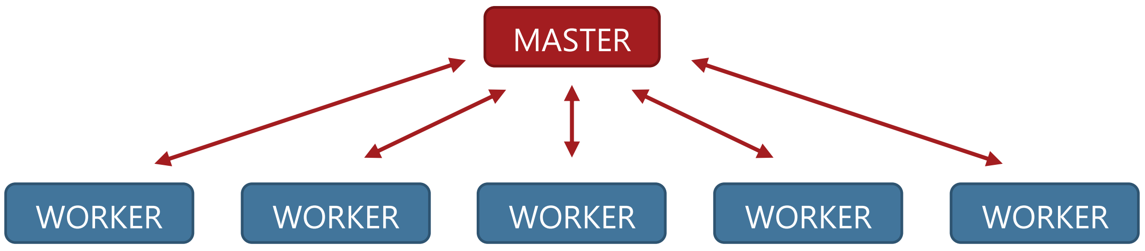
- ERDA for file storage
- Jupyter for Terminal to access MODI and code editor
- Development on DAG
- Benchmarks through SLURM on MODI

Handin:

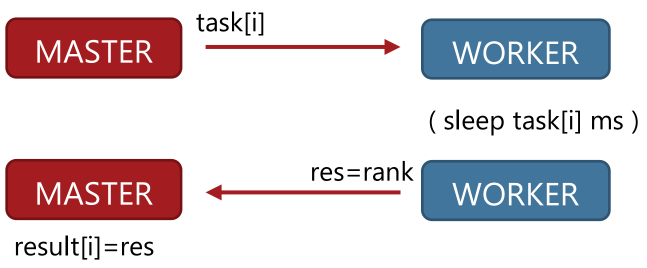
- Total assignment: a report of up to 3 pages in length (excluding the code)
- Your task farm codes for both part I and part II.
- Use the template on Absalon to include your code in the report

Introduction

Task farming is used in many branches of science and computing to process large amounts of individual tasks in an effective and rapid manner. The idea is to have one controlling process (the "master", "conductor", "farmer", "controller", "provider") and many executing processes ("workers", "agents", "consumers"). In older literature you will also see the terminology master-slaves.



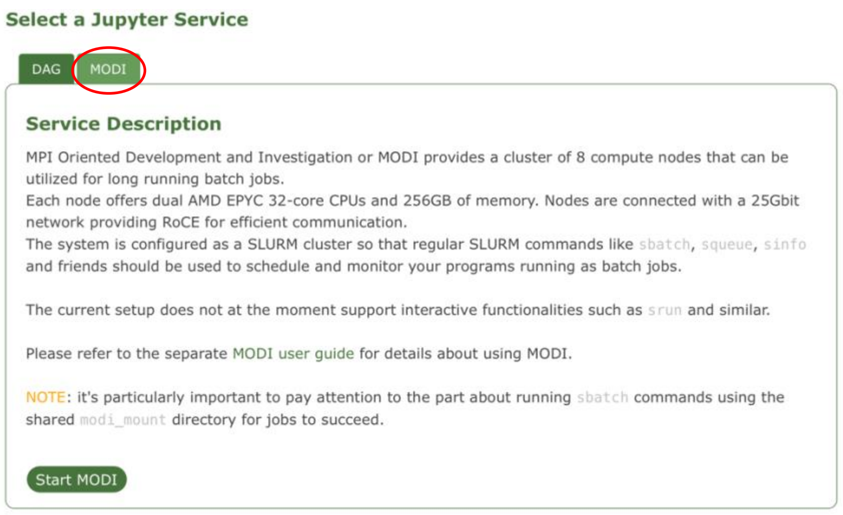
In this exercise you will implement a task farming algorithm using the message passing interface (MPI) first in a simplified version and then to process data packages from high energy physics. In the simple version the master creates an array task[NTASKS] with random numbers. The master will distribute the tasks to the workers. Task number i is just an integer task[i]. The "work" that has to be performed is that the worker sleeps (e.g. do nothing) for task[i] milliseconds. The "result" of carrying out the task is that the worker returns an integer which contains the rank of the worker. This rank has to be stored in the correct position, result[i], of a results array. The life cycle of carrying out a single task is sketched below



In the end the master loops over the result array and counts how many tasks and work units (sum of tasks) were carried out by each worker.

DAG / MODI: development – MODI: benchmarks

For this assignment we need Fortran / g++ or python with the MPI library. This can be obtained by using the so-called wrapper commands mpifort and mpic++, and in the case of Python by loading the package mpi4py. It is provided on ERDA using either DAG or MODI and the *SLURM notebook*. For example to use MODI select the correct tab for the Jupyter service:



Select a Jupyter Service

Service Description

MPI Oriented Development and Investigation or MODI provides a cluster of 8 compute nodes that can be utilized for long running batch jobs.
Each node offers dual AMD EPYC 32-core CPUs and 256GB of memory. Nodes are connected with a 25Gbit network providing RoCE for efficient communication.
The system is configured as a SLURM cluster so that regular SLURM commands like `sbatch`, `squeue`, `sinfo` and friends should be used to schedule and monitor your programs running as batch jobs.

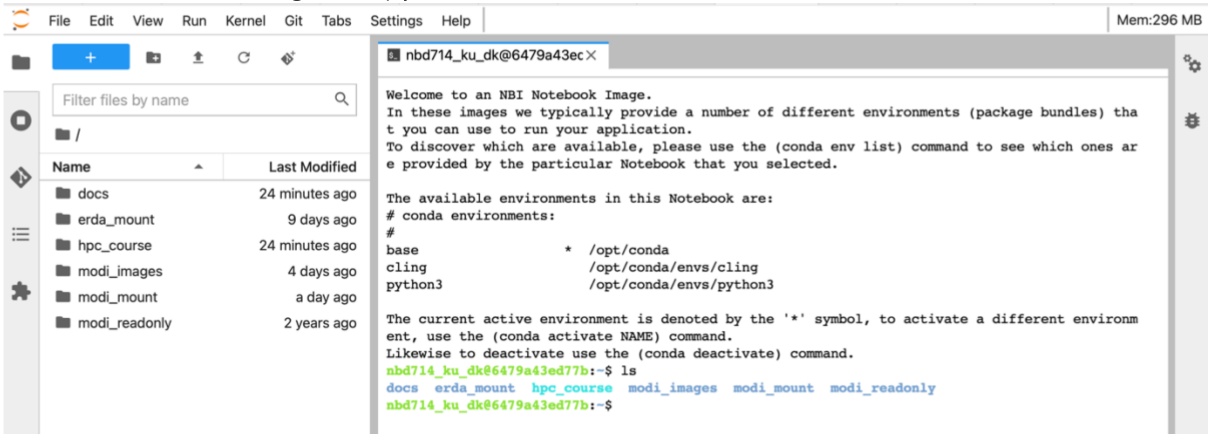
The current setup does not at the moment support interactive functionalities such as `srun` and similar.

Please refer to the separate MODI user guide for details about using MODI.

NOTE: it's particularly important to pay attention to the part about running `sbatch` commands using the shared `modi_mount` directory for jobs to succeed.

You can read more about MODI in the user guide: https://erda.dk/public/MODI-user-guide.pdf

Spin up a Jupyter session on MODI selecting the "HPC notebook" notebook image. In the terminal (or the folder view on the right side) you can see several folders.



Name	Last Modified
docs	24 minutes ago
erda_mount	9 days ago
hpc_course	24 minutes ago
modi_images	4 days ago
modi_mount	a day ago
modi_readonly	2 years ago

```
Welcome to an NBI Notebook Image.
In these images we typically provide a number of different environments (package bundles) tha
t you can use to run your application.
To discover which are available, please use the (conda env list) command to see which ones ar
e provided by the particular Notebook that you selected.

The available environments in this Notebook are:
# conda environments:
#
base                  *  /opt/conda
cling                    /opt/conda/envs/cling
python3                  /opt/conda/envs/python3

The current active environment is denoted by the '*' symbol, to activate a different environm
ent, use the (conda activate NAME) command.
Likewise to deactivate use the (conda deactivate) command.
nbd714_ku_dk@6479a43ed77b:~$ ls
docs  erda_mount  hpc_course  modi_images  modi_mount  modi_readonly
nbd714_ku_dk@6479a43ed77b:~$
```

The different folders contain:
`docs`**:** the MODI user guide.
`erda_mount`**:** your own files.
`hpc_course`: this folder can be ignored.
`modi_images`: images of virtual machines that can be used when submitting jobs.
`modi_mount`: this folder is the only one that can be seen from the cluster nodes. You need to copy any executables you use (this is called *staging*) to this filesystem before submitting a job.
`modi_readonly`: this folder can be ignored.

You can start the exercise using DAG, like you did in week 1, but selecting the *HPC notebook* image. This is relevant when developing your code, and you do it by selecting the DAG tab in the Jupyter service window. Then you can only see `erda_mount`, but that is also sufficient because you will only run MPI directly in the Jupyter instance.

PREPARATIONS

Start by making a git pull to get the exercise and enter the `week2` folder. You can write '`ls`' to get a file listing of the folder.

```
cd erda_mount/ahpc
git pull
cd week2
ls *
```

To be able to edit the files for the exercise navigate to the same folder in the file view. Here you can open seven files (for the C++, similar for python and Fortran):

- Makefile
- task_farm_skeleton.cpp
- task_farm.cpp
- task_farm_HEP.cpp
- task_farm_HEP_seq.cpp
- job.sh

In the week2 main folder there is

- mc_ggH_16_13TeV_Zee_EGAM1_calocells_16249871.csv
- copy_to_modi_mount.sh

C++ or Fortran: before you can run the code, you need to compile it. This can be done with make. You should see something like this in the terminal:

```
$ make
mpic++ task_farm.cpp -O3 -Wall -Wno-unused-const-variable -std=c++17 -march=native -o task_farm
mpic++ task_farm_HEP.cpp -O3 -Wall -Wno-unused-const-variable -std=c++17 -march=native -o task_farm_HEP
mpic++ task_farm_HEP_seq.cpp -O3 -Wall -Wno-unused-const-variable -std=c++17 -march=native -o task_farm_HEP_seq
```

Python: to run in parallel you need to use the package `mpi4py`. It is only available on either DAG or MODI in a conda environment named `python3`. On DAG you have to activate it in your terminal:

```
conda activate python3
```

In a job script you need to specify it explicitly by giving the full path to the python executable: `/opt/conda/envs/python3/bin/python`. This has already been done in the job script.

For the first task, one binary is produced: `task_farm`. The `task_farm_skeleton.cpp` code is there to give you a backup. To run the code directly in the terminal using the mpiexec command you can either use DAG or MODI. F.x. to run it with 8 processes you do:

```
$ mpiexec -np 8 ./task_farm
```

Similarly, to run the python code you can do (after activating the python3 environment):

```
$ mpiexec -np 8 python ./task_farm.py
```

The advantage of doing this with DAG is that the virtual machine gets exclusive access to 8 cores, while on MODI everybody share access to the same machine (since it is intended to only be used for development), and therefore if everybody use MODI to test at the same time, it may become slow or run out of memory. Benchmarks can be carried out on MODI with access to up to 512 cores on 8 dedicated servers using SLURM, and you will use that for benchmarking your code (see Task 3).

Code structure:

The code contains a main program with the commands to start up MPI, get the rank of the process and the total number of processes. It then calls either master or worker, depending on the rank number.

The first task of the assignment is to use MPI message passing commands to implement communication between master and workers. You can use `MPI_Send` and `MPI_Recv`. For non-blocking you can use `MPI_Isend` and `MPI_Irecv`. You can read more about how they work in chapter 8.2 of the book.

Task 1: Write a functioning master-worker program

Use the template and our discussion as a starting point for implementing a working master-worker program. In this version the master creates an array task[NTASKS] with random numbers. The master will distribute the tasks to the workers. It is just an integer task[i]. The performed "work" is that the worker sleeps (e.g. do nothing) for task[i] milliseconds. The "result" of carrying out the task is that the worker returns an integer which contains the rank of the worker that is stored *in the correct order* in a result array. E.g. the result of task[i] is stored in result[i].
Besides your implementation submitted as the code and attached to the report, you also submit a report through Absalon. In the report, you should explain how you have parallelised the program. Remember to discuss when and what data you exchange between the MPI-processes and how that impacts the performance. It is up to you if you use blocking or non-blocking messages, or maybe a mixture.

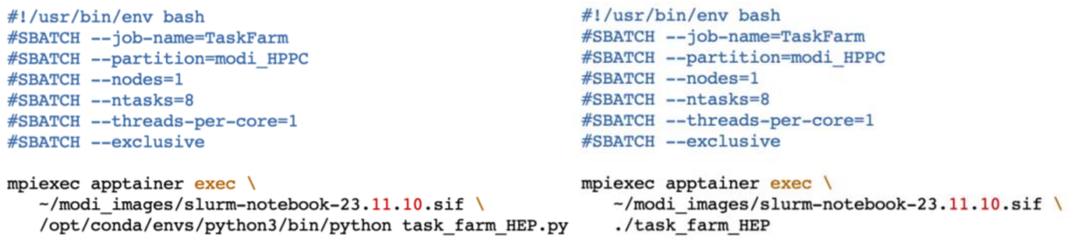
Task 2: Master-worker program for HEP data processing

Use the template program `task_farm_HEP.cpp` and the results of task 1 to implement a master – worker program for analyzing high energy physics events data. In this version all ranks (both master and worker processes) reads in the data set. The master then creates a large set of possible cuts, that can be used to determine if an event was a background event or a real signal. For each set of cuts an accuracy must be computed. The master must distribute settings for the cuts to the workers. Each setting contains 8 double precision variables. In the case of C++ the settings are stored in a `std::array<double,8>` variable, but MPI functions need a pointer to the data. Therefore `settings[k].data()` (or equivalently `&settings[k][0]`), which is a pointer to the underlying data array, has to be passed to the MPI function that sends the data from Master. In the case of Fortran or Python you are already passing by references so everything should work without problems. Also be aware that you are now sending double precision variables (of MPI datatype MPI_DOUBLE) instead of the integers (MPI_INT) in task 1.

The "work" that must be performed is that the worker computes the accuracy, and that is returned as a result to the master. The results can be compared and validated to the output from the reference code produced by compiling and running `task_farm_HEP_seq.cpp`.

Task 3: Strong scaling of HEP processing using SLURM (points 4)

When benchmarking the performance of your program, use the SLURM queue in MODI to get exclusive access to a machine. An example of two SLURM scripts, job.sh, that run the parallel program on 8 cores on one node (1x8 cores) are (python to the left, Fortran or C++ to the right):



```
#!/usr/bin/env bash
#SBATCH --job-name=TaskFarm
#SBATCH --partition=modi_HPPC
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --threads-per-core=1
#SBATCH --exclusive

mpiexec apptainer exec \
   ~/modi_images/slurm-notebook-23.11.10.sif \
   /opt/conda/envs/python3/bin/python task_farm_HEP.py
```

```
#!/usr/bin/env bash
#SBATCH --job-name=TaskFarm
#SBATCH --partition=modi_HPPC
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --threads-per-core=1
#SBATCH --exclusive

mpiexec apptainer exec \
   ~/modi_images/slurm-notebook-23.11.10.sif \
   ./task_farm_HEP
```

The “exclusive” flag means that there will only be one user on the nodes. This is good for benchmarking but should never be used in production runs. “modi_HPPC” indicates the queue. Each node has 64 cores, so this is highly wasteful way to run, but it is a good way to get reliable benchmarks. The apptainer image is needed because each notebook image has a different set of software installed.

<u>Subtask a)</u> provide experimental results of running using 2, 4, ..., 64 MPI processes corresponding to 1, 3, …, 63 workers. You may also use a higher number of cores (and then more than 1 node is needed). The maximum is 512 cores. Notice that the “modi_HPPC” queue has a maximum runtime of 5 minutes and access to all 8 nodes, while the other queues (e.g. “modi_short”) only can access 6 of the 8 nodes. The number of settings that are explored depend on n_{cut}[8]. At a high number of ranks you may need to increase n_{cut} from the default 3 to 4 or even 5 in order to obtain reasonable run time and good performance. Then when you use a small amount of ranks you may need to run in the modi_short queue, since the run will take more than 5 minutes. How you choose to distribute the MPI-processes between the MODI compute nodes and their CPU-cores is up to you – remember to discuss briefly your decision in the report.
Present the result as an easy-to-read graph. We can measure the average time it takes a single worker to update a task (T_{task}), including any parallelization overheads, by measuring the total wall-clock time (T_{wall}) it takes to carry out the analysis and then take in to account the number of workers (N_{worker}) and the total number of tasks (N_{task}):

$$T_{task} = \frac{T_{wall} \times N_{worker}}{N_{task}}$$

Given that the workload is fixed (N_{task} is constant), what we are benchmarking is strong scaling. In the ideal case, T_{task} should be fixed (e.g. a flat curve, corresponding to no parallel overheads). The speedup of the code is then

$$Speedup[N\ workers] = \frac{T_{wall}[1\ worker]}{T_{wall}[N\ workers]} = \frac{T_{task}[1\ worker] \times N_{worker}}{T_{task}[N\ workers]}$$

<u>Subtask b)</u> Estimate the serial and parallel fraction of the code in the context of Amdahl’s law (see section 1.2 in the book). And plot a theoretical scaling curve for your results on your easy-to-read graph from subtask a. Only consider the number of CPU cores dedicated to workers (N_{worker}).

<u>Subtask c)</u> Discuss your results in the context of strong scaling; what do you think are the bottlenecks in your code and how does the hardware impact parallelization overheads when the number of workers change. What is your recommendation to the high-energy physicists in terms of limits for how many CPU cores it makes sense for they to employ for this analysis, and if you benchmark more than one version of the code, does the relative scaling make sense compared to the improvements between different versions?