Assignment 3: Task Farming

Practical information

Deadline: Saturday 24/2 23.59

Resources:

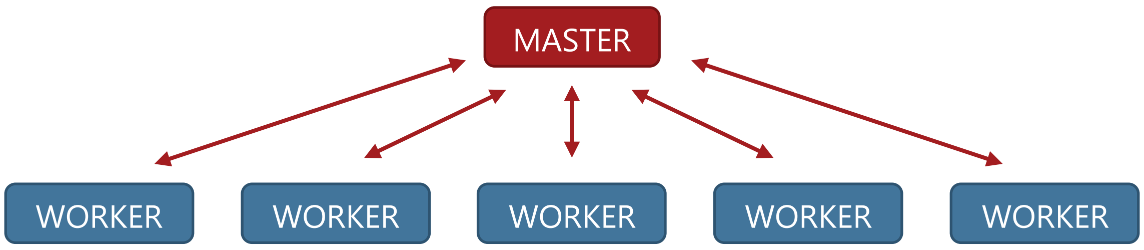
- ERDA for file storage
- Jupyter for the Terminal to access MODI
- Benchmarks through SLURM on MODI

Handin:

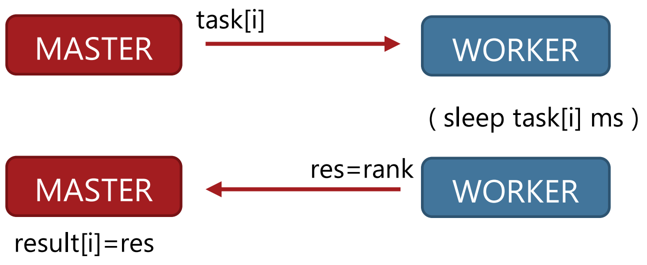
- Total assignment: a report of up to 3 pages in length (excluding the code)
- Part I: Your task farm code
- Use the template on Absalon to include your code in the report

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.



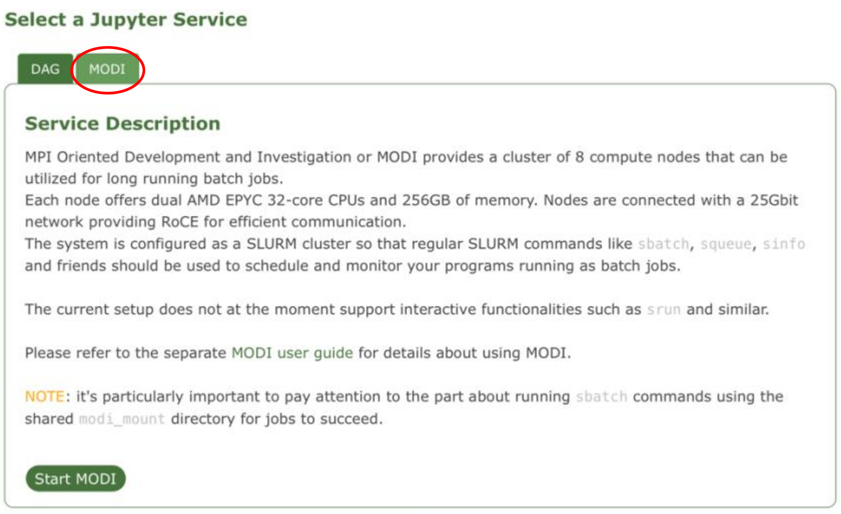
In this exercise you will implement a task farming algorithm in C++ using the message passing interface (MPI) to process data packages from high energy physics. The first step is to make a simplified version. In this version the master creates an array task[NTASKS] with random numbers. The master will distribute the tasks to the workers. Task no 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. 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 workunits (sum of tasks) were carried out by each worker.

MODI

For this assignment we need g++ with the MPI library. This can be obtained by using the so called wrapper command mpic++ It is provided on ERDA using MODI (not DAG!).



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.

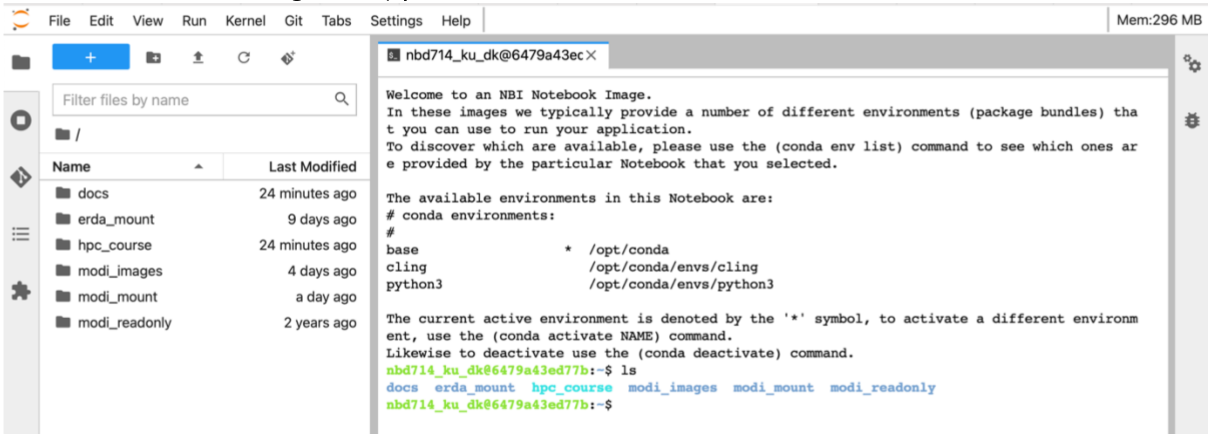
The current setup does not at the moment support interactive functionalities such as srun and similar.

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 a number of 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:
`erda_mount`: your own files.
`hpc_course`: course files.
`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.

PREPARATIONS

Start by copying the exercise to your storage area and enter into the folder. You can write '`ls`' to get a file listing of the folder.

```
cd erda_mount/HPPC
cp -a ~/hpc_course/module3 .
cd module3
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:

- Makefile
- task_farm_skeleton.cpp
- task_farm.cpp
- task_farm_HEP.cpp
- task_farm_HEP_seq.cpp
- job.sh
- mc_ggH_16_13TeV_Zee_EGAM1_calocells_16249871.csv

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++14 -march=native -o task_farm
mpic++ task_farm_HEP.cpp -O3 -Wall -Wno-unused-const-variable -std=c++14-march=native -o task_farm_HEP
mpic++ task_farm_HEP_seq.cpp -O3 -Wall -Wno-unused-const-variable -std=c++14 -march=native -o task_farm_HEP_seq
mpic++ task_farm.cpp -O3 -Wall -march=native -g -std=c++14 -o task_farm
```

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 on the login machine of the MODI cluster you need to use the mpiexec command. F.x. to run it with 8 processes you do:

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

Code structure:

The code is relatively simple. It 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.

Task 1: Write a functioning master-worker program (points 4)

Use the template and our discussion Monday (see updated slides that includes the solution) 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 "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.

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 both versions.

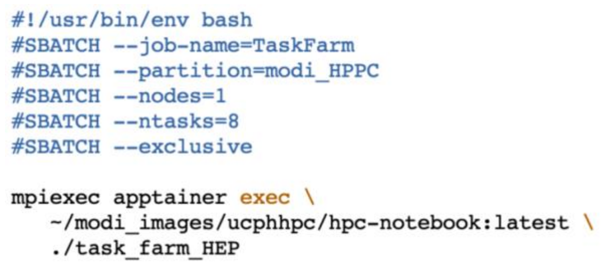
Task 2: Master-worker program for HEP data processing (points 2)

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 cut an accuracy must be computed. The master must distribute settings for the cuts to the workers. Each setting contains 8 double precision variables. 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. 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 MODI servers also through Jupyter. However, in order to get exclusive access to a machine you need to submit your run through SLURM. An example of a SLURM script, job.sh, that runs the parallel program on 8 cores on one node (1x8 cores) is:



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

mpiexec apptainer exec \
   ~/modi_images/ucphhpc/hpc-notebook:latest \
   ./task_farm_HEP
```

The "exclusive" flag means that there will only be one user on the nodes. "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.

Subtask a) provide experimental results of running using 1, 2, 4, ..., 64 MPI processes. 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 but 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. 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, which includes the absolute and relative performance of the parallel code measured as the wall clock time it takes to update a single task as a function of number of cores dedicated to workers. Given that the workload is fixed, this is what we call strong scaling. If you multiply the number by the number of worker CPU cores used, you obtain the CPU time spend per task. In the ideal case, this number should be fixed (e.g. a flat curve).
Subtask b) Estimate the serial and parallel fraction of the code in the context of Amdahl’s law. And plot a theoretical scaling curve for your results. Only consider the CPU cores dedicated to workers.
Subtask c) Discuss your results in the context of strong scaling. What is your recommendation to the high-energy physicists in terms of limits for how many CPU cores they can employ, and if you benchmark more than one version of the code, does the relative scaling make sense?