

# Assignment 3: Task Farming

## Practical information

Deadline: Saturday 21/2 23.59

### Resources:

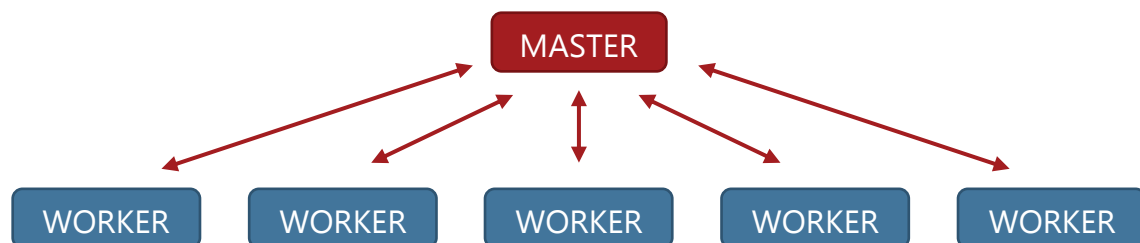
- ERDA for file storage
- Jupyter for Terminal to access MODI and code editor
- Development on DAG
- Benchmarks through SLURM on MODI

### Handin:

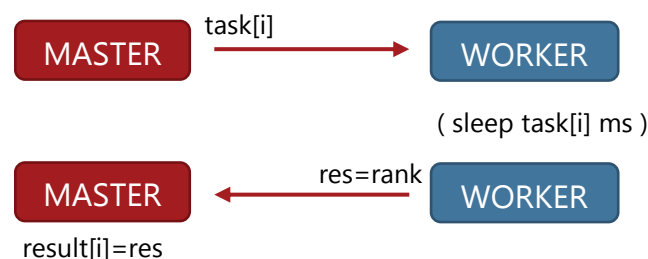
- Total assignment: a report of up to 3 pages in length (excluding the code)
- Your task farm codes for both part I and part II.
- An up to 3-minute video where one person from the group scrolls through the code and explains what you have done.
- Use the template on Absalon to include your code in the 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 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. 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 g++ with the MPI library. This can be obtained by using the so-called wrapper command `mpic++`. It is provided on ERDA using either DAG or MODI and the HPC notebook. To use MODI select the correct tab for the Jupyter service:

### Select a Jupyter Service

DAG MODI

#### 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.

Start MODI

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.

The screenshot shows a Jupyter Notebook interface. On the left, the file explorer displays a list of folders: `docs`, `erda_mount`, `hpc_course`, `modi_images`, `modi_mount`, and `modi_readonly`. The `modi_mount` folder is highlighted. On the right, the terminal window shows the output of the `ls` command, listing the same folders: `docs`, `erda_mount`, `hpc_course`, `modi_images`, `modi_mount`, and `modi_readonly`.

The different folders contain:

`docs`: the MODI user guide.

`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.

You can start the exercise using DAG, like you did in week 1 and 2. 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` and `hpc_course`, but that is also sufficient because you will only run MPI directly in the Jupyter instance.

## PREPARATIONS

Start by copying the exercise to your storage area and enter 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++23 -march=native -o task_farm
mpic++ task_farm_HEP.cpp -O3 -Wall -Wno-unused-const-variable -std=c++23 -march=native -o task_farm_HEP
mpic++ task_farm_HEP_seq.cpp -O3 -Wall -Wno-unused-const-variable -std=c++23 -march=native -o task_farm_HEP_seq
```

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
```

The advantage of doing this with DAG is that the virtual machine gets exclusive access to 8 threads / 4 cores, while on MODI everybody share access to the same machine (since it is intended to only be use for development), and therefore if the whole class uses MODI to test at the same time, it may become slow or run out of memory. Benchmarks should be carried out on MODI with access to up to 512 cores on 8 dedicated servers using SLURM (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 (points 4)

Use the template and our discussion Monday 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 up to 3 minutes video through Absalon. In the video, 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. Also discuss how you have validated the program gives the correct result.

## 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 cuts 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 SLURM queue in MODI to get exclusive access to a machine. 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 --threads-per-core=1
#SBATCH --exclusive

mpiexec aptainer 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 aptainer image is needed because each notebook image has a different set of software installed.

Subtask a) 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/511 workers. 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  $n_{\text{cut}}$ <sup>8</sup>. At a high number of ranks you may need to increase  $n_{\text{cut}}$  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. We can measure the average time it takes a single worker to update a task ( $T_{\text{task}}$ ), including any parallelization overheads, by measuring the total wall-clock time ( $T_{\text{wall}}$ ) it takes to carry out the analysis and then take in to account the number of workers ( $N_{\text{worker}}$ ) and the total number of tasks ( $N_{\text{task}}$ ):

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

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

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

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 on your easy-to-read graph from subtask a. Only consider the number of CPU cores dedicated to workers ( $N_{\text{worker}}$ ).

Subtask c) 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 them to employ for this analysis, and if you benchmark more than one version of the code, can you comment on how the changes in the performance is related to the changes in the code between different versions?

## Bonus Task 4: Assignment 3 in Python

You cannot gain any extra points, but for the interested, we have made a subfolder called “python\_bonus”, where all of assignment 3 is written in Python. For some of your own projects, it may make sense to use the Python package “mpi4py” to get MPI running python. You can use the “Python clone” of exercise 3 to try to write a non-trivial MPI program in Python, once you have finished the assignment.

It is completely optional, but you are most welcome to ask questions about how to do it during the week or the following week. Please refer to <https://mpi4py.readthedocs.io/en/stable/> for documentation of the mpi4py package.