# Assignment 4: Inverse Problems

## Practical information

Deadline: Saturday 2/3 23.59

Resources:

- ERDA for file storage
- Jupyter for the Terminal to access MODI
- Benchmarks through SLURM on MODI (one node!)

Handin:

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

## Introduction

Inverse problems are problems where data from the laboratory, the observatory or the field are used to infer information about the internal structure or properties of an object. To solve an inverse problem, we need not only the data, but also the relation between data and the model parameters that represent the structure of the object. This relation is sometimes given by an explicit mathematical function that maps the model parameters into the data, but more often by an algorithm represented by a computer code.

Inverse problems where the mathematical relation is available are often linear, and this allows us to directly solve the problem through matrix algebra, which is a well-understood and relatively fast method. In many other cases, however, data are related to model parameters through a non-linear function that can only be represented by an algorithm. This is the case in, e.g., wave propagation problems where waveforms (seismic, electromagnetic, etc.) are used for reconstruction of objects through which the waves have propagated. Since we may not have direct access to the mathematical structure of the problem in such cases, we can only base our solution to the inverse problem on computational strategies that search for reasonable values of model parameters fitting the data within its observational uncertainty.

The algorithmic search for solutions consists of two components: (1) a method to compute the datafit from any set of model parameters, and (2) a strategy for proposing the next set of model parameters to be tested. The first component can be very time consuming, as it is often based on a simulation of the physical process considered in the problem.

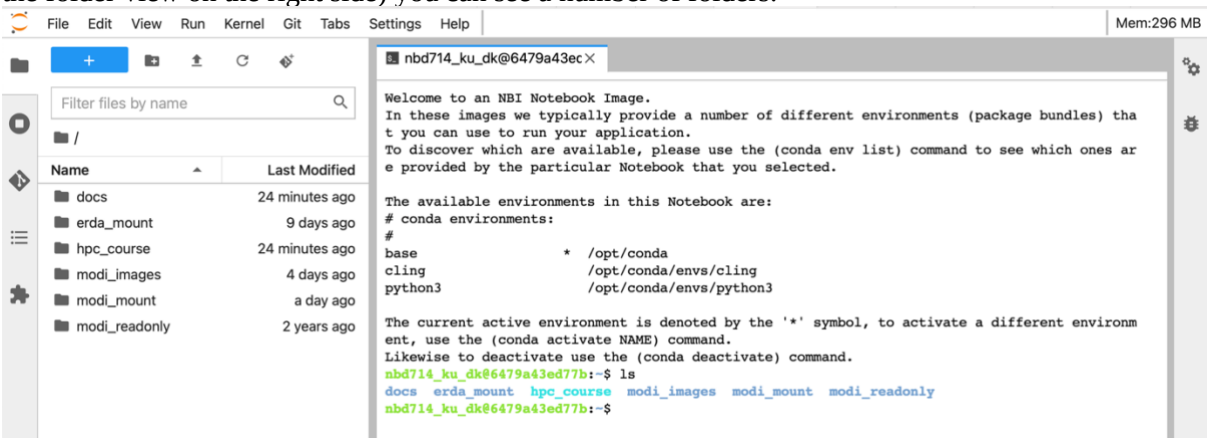
We shall here look at a simple wave propagation problem where plane waves propagate through a medium of plane-parallel layers. Our task is to simulate the wave propagation as fast as possible, thereby enabling a more efficient search for good solutions to the problem. The problem can be "linearized", but the solution to the resulting linear problem may, in some cases, deviate significantly from the correct (non-linear) solution. We will look at the physical difference between the complete formulation and the linearized formulation and try to understand why the fully non-linear formulation is better, but so much more computationally demanding.

## MODI

For this assignment we need g++ and maybe LLVM/clang++ to compile and MODI for running the benchmarks.


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




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.

## PREPARATIONS

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

```
cd erda_mount/HPPC
cp -a ~/hpc_course/module4 .
cd module4
ls
```

To be able to edit the files for the exercise navigate to the same folder in the file view. Here you can see seven files:

- Makefile
- seismogram_omp.cpp
- seismogram_seq.cpp
- job.sh
- velocity_data.txt
- density_data.txt
- wave_data.txt

Before you can run the code, you need to compile it. This can be done by running make in the terminal.

For the first task, one binary is produced: `mp`. The `seismogram_seq.cpp` code is there to give you a backup, a corresponding binary, `seq`, is produced. To run the code on the login machine of the MODI cluster using e.g. 4 threads you can do:

```
$ env OMP_NUM_THREADS=4 ./mp
```

## Code structure:

The code has two parts. It contains a main program which reads in the data files. From the main routine the `propagator` routine is called. The routine calculates the seismogram which is expected given the density profile, the wave-speed profile and the underground wave. To calculate the seismogram we need to go from real space to Fourier space, and therefore the program also contains two simple functions `fft` and `ifft` that uses divide and conquer to compute the (inverse) FFT.

You can change the workload in the program by changing the number of frequencies computed.

```
// The number of frequencies sets the cost of the problem
const long nfreq=64*1024; // frequencies in spectrum
```

because of the very simple FFT algorithm, `nfreq` must be a power of two.

At the top of the code, a template class called `NUMA_Allocator` has been included. It allows to allocate a vector or array of elements that are spread out in memory according to the scheduling defined by the OpenMP pragma "`pragma omp parallel for schedule(static)`"" on line 43. It is enabled by swapping the typedef statements for `ComplexVector` and `DoubleVector` around line 85.

## OpenMP thread checker

In the lecture we discussed the concept of a thread checker. You can access the thread checker by changing the compilation flags. Open the Makefile and comment in the relevant compile options. This is a relatively new feature, which has been added to LLVM and then ported to gcc. Useful OpenMP support was only stable in recent versions of those compilers (like LLVM from version 12 and gcc from version 12 or 13). In older version, the thread checker would produce a lot of false positives.
Running with the thread checker options, assuming you have bugs, you will see a lot of output scrolling past on the screen. This can become too much, and it is better to redirect the screen output to a text file, like

```
$ env OMP_NUM_THREADS=4 ./mp >& out
```

The file can afterwards be inspected in the editor. If you want a summary, you can use the `grep` command to filter the output like this

```
$ grep SUMMARY out
```

This can also be done in one go

```
$ env OMP_NUM_THREADS=4 ./mp |& grep SUMMARY
```

Once you have removed the bugs, remember to change back to the fast compile settings with -O3, make clean, and make again, before you do benchmarks. A thread checker intercepts all memory accesses checking for race conditions or undefined behaviour and can slowdown the program by a factor of 10 to 100.
The thread checker available on MODI is unfortunately not very advanced. It will believe it has found problems, which are not really there. These are called *false positives*. You should therefore take the list of *possible* race condition with a grain of salt and evaluate them one by one, convincing yourself that there is no problem.
A much better tool, which unfortunately is not available on MODI, is the intel inspector – see chapter 7.9.2. It is part of the development tools made available for free as part of the Intel oneAPI toolkits for Linux and Windows. We can warmly recommend using intel inspector if you develop an OpenMP program with just a small level of complication; it *will* at some point find threading errors.

## Task 1: OpenMP parallelise the program (points 5)

Use openmp pragmas to parallelise the code. It is up to you if you only do single loop parallelisation, or you try to create a larger parallel region with several loops inside. If you make a larger region, you can use `omp single`. The task or section directives `omp task` and `omp section` could also be useful for some part of the code. To complete this task, besides your implementation submitted as the code and in pdf, you also submit a report through Absalon. In the report, you should explain how you have parallelised the program, and what was your strategy. Remember to test that you get the same checksum and include a line about it in the report.

## Task 2: Strong and weak scaling using SLURM (points 5)

When benchmarking the performance of your program, use the MODI servers. 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 1 core on one node is included. 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 singularity image is needed because each notebook image has a different set of software installed.
You should provide experimental results of running using 1, 2, 4, ..., 64 threads. The results should be presented as two easy to read graphs, which shows the strong and weak scaling of the parallel code measured as the wall clock time it takes to run the code as a function of the number of cores normalised

to the time it takes to run the code on one thread. It is recommended to run each benchmark experiment several times to get an error bar on the measurement.
Strong scaling: choose a large enough `nfreq` such that you have almost ideal scaling going from 1 to 2 cores (something larger than 65536 is probably needed, but it depends on how good your implementation is). Then increase the number of cores used while keeping the problem size fixed. Fit an Amdahl's law and measure the parallel fraction.

Weak scaling: a fixed amount of work per core can be obtained by adjusting `nfreq` according to the number of cores. Notice that the FFT routine is not linear in cost as a function of `nfreq`, but rather the cost of the FFT scales logarithmic as O(N logN) for N points. Therefore, the workload is not completely constant. You can either scale the cost individually of the FFT calls and subtract them out (how do they scale?), or just live with the slightly more realistic results. To add complication to getting a nice weak scaling, also observe that you cannot just set `nfreq` arbitrarily, but it has to be a power-of-two. Please write what you have done and how you think it changes your interpretation.

In the report it is expected that you interpret and discuss your scaling results. You should interpret them considering the code, the workload, and in the context of the shared memory architecture of ERDA.

Optionally, to test the impact of how CPU cores and memory controllers are distributed and interconnected on the AMD Naples CPUs you may use different binding options to place your threads either on different sockets, dies or close to each other on a single die.