Assignment 3: Inverse Problems

Practical information

Deadline: Friday 24/10 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 codes in the report

Note: exercise is only available in C++ and Fortran. Python does not have a good concept of shared memory parallelization. This may change in the future once the GIL is gone (see arXiv:2411.14887)

Introduction

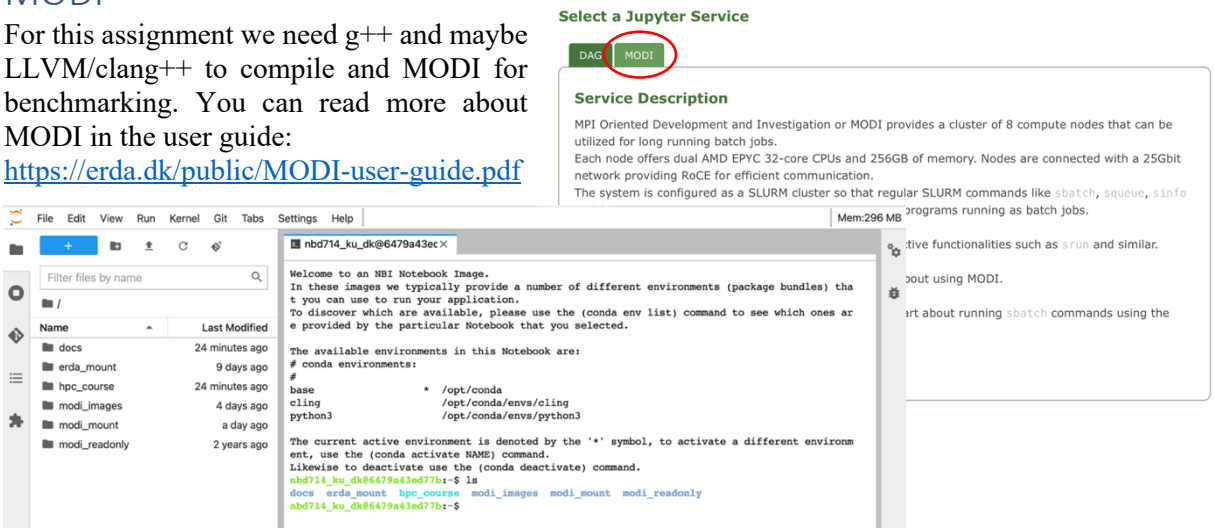
Inverse problems use laboratory or field data to infer information about the properties or internal structure of an object. To solve an inverse problem, we need not only the data but also a relation between data and model parameters that represent the structure of the object. This relation can be an explicit function that maps parameters to data but is more often 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 well-understood and relatively fast. In many other cases, however, data is related non-linearly to model parameters. 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 data-fit 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. The task is to simulate the wave propagation as fast as possible, thereby enabling a more efficient search for solutions to the problem. The problem can be "linearized", but the solution to the resulting linear problem may deviate from the correct non-linear solution. We will look at the physical difference between the complete 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 benchmarking. 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 "SLURM 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:
`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.

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. Steps below are for C++. Fortran is very similar.

```
cd erda_mount/ahpc
git pull
cd week3/cpp
```

or

```
cd week3/fortran
```

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 in the week3 and cpp/fortran subfolders:

- `Makefile`
- `seismogram_omp.cpp/f90`
- `seismogram_seq.cpp/f90`
- `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. To make it possible to change it in a script, this is done as a preprocessing step which is set in the Makefile with the option NFREQ. Because of the very simple FFT algorithm, `NFREQ` must be a power of two.

```
make NFREQ=65536
```

In Fortran (and C), when allocating memory it is only placed in memory when it is first accessed. Therefore, if first access is in an OpenMP region, memory pages will be distributed out according to where the calling threads are located in the system (so-called "first touch policy").
For C++ it is complicated to allocate a large block of memory such that it is local to each thread, and it requires a custom designed memory allocator. At the top of the code, a template class called `NUMA_Allocator` has been included that does exactly this. It allows you 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 53. It is enabled by setting a variable in the Makefile:

```
make NUMA_ALLOCATOR=y
```

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.
Until January 2025 Intel had a much better tool available, Intel Inspector (see chapter 7.9.2 in the book), but Intel has decided to discontinue it and instead invest all their effort and experience in the thread sanitizer included in LLVM. I therefore expect the quality of thread sanitizer to improve dramatically

in the newest version of the compilers. I *strongly* recommend using the thread sanitizer 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

Use OpenMP pragmas to parallelise the code. It is up to you, which strategies you would like to use. For each different strategy to OpenMP parallelise the code you can get a point. To get the point, besides your implementations submitted as code and in pdf, you should also include a section in your report where you list the different code versions, and briefly explain how they are different, and why you expect them to be improvements (even if they do not improve performance and/or scaling). Examples of different strategies could be replacing many parallel regions with a single region, removing explicit or implicit barriers, employing new types of pragmas, or using the pragmas in a different manner. Remember always to test that you get the same checksum and include a line about it in the report.

Task 2: Weak scaling using SLURM

When benchmarking the performance of your program, use the MODI servers. 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 five times 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 apptainer image is needed because each notebook image has a different set of software installed. To complete the task you have to

1. Make scaling plots: You should provide experimental results of running using 1, 2, 4, 8, 16, 32, and 64 threads. The results should be presented as three easy-to-read graphs, which shows the normalised time of (a) the program excluding the FFT (b) only the FFT, and (c) running the whole program. In all three cases, the time should be normalised to running with 1 thread. Run each benchmark experiment 5 times to get an error bar on the measurements. To have almost ideal scaling going from 1 to 2 threads, choose `nfreq=65536*ncore`.
2. Compute parallel fractions: Measure the parallel fraction for each of the different strategies by fitting for cases (a) and (b) an appropriate scaling law. 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. Please write how you account for this in your scaling laws.
3. Discuss and analyse: Interpret and discuss your results for the different strategies. You could interpret them considering the OpenMP strategies involved, the code, and the shared memory architecture of ERDA.