ETH zürich

High Performance Computing for Science and Engineering II

P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Spring semester 2019

HW 1 - Setup and Algorithm Optimization

Issued: February 18, 2019
Due Date: March 4, 2019 1:00pm

1-Week Milestone: Finish Tasks 1 to 5 and come to the practice session (February 25).

Task 1: Getting started on Euler

Euler is the computing cluster of ETH Zurich. The cluster works with a queueing system: you submit your program with its parameters (called job) to a queue and wait for it to be finished. Your first task consists of the following steps:

1. **Creating an account**
   Euler accounts are created automatically when a user logs in for the first time. You will need to enter your nethz username and password. Euler is only accessible within the ETH Network.[1]
2. **Login**
   Login from within the ETH network on Euler via ssh:
   ```
   $ ssh username@euler.ethz.ch
   ```
   and insert your password when asked to.

   Congratulations! You are now on a login node of the Euler cluster. In this environment you can write code, compile and run small tests. Keep in mind that there are other people working on the same nodes, so be mindful of how you use them!

3. **Modules**
   The Euler environment is organized in modules, which are conceptually software packages that can be loaded and unloaded as needed. The basic commands to use the module system are:
   ```
   $ module load <modulename>
   ```
   Sets the environment variables related to the specified module.
   ```
   $ module unload <modulename>
   ```
   unsets the environment variables related to `<modulename>`.
   `$ module list`: lists all the modules currently loaded.
   `> module avail`: outputs a list of all the modules that can be loaded.

   In this course, we require that all submissions be compiled and tested using the modules:
   ```
   > module load new
   ```

[1] You may use VPN https://sslvpn.ethz.ch to connect to the ETH Network from home.

```
> module load gcc/6.3.0
> module load intel/2018.1
> module load mvapich2/2.3rc1
```

This configuration uses Intel 2018.1 (compatible with gcc 6.3.0) as main compiler, and mvapich2 as MPI library[2]. To compile, we simply run compilation of our program:

```
> $(CC)-O2 main.cpp -o program_name
```

4. **Submitting a job**
   Performance measurements and long computations should not be performed on the login nodes but rather they should be submitted to the queue.
   To submit a job to the queue, you can use the following command from the folder where your program is stored:

   ```
   > bsub -n 24 -W 08:00 -o output_file ./program_name program_args
   ```

   This command will submit a job for your executable `program_name` with arguments `program_args` by requesting 24 cores from a single node and a wall-clock time of 8 hours, after which, if the job is not already finished running, it will be terminated. The report of the job, along with the information that would usually appear on the terminal, will be appended in the file `output_file`, in the folder from where the job started.
   You can also allocate an interactive job for continuous development:

   ```
   > bsub -n 1 -W 01:00 -Is bash
   ```

   While your job is running you can always use the command:

   ```
   > bjobs
   ```

   to get the status and IDs of your jobs.
   In order to terminate a job you can use the command:

   ```
   > bkill <jobid>
   ```

Info: I/O performance and `$SCRATCH`:
Since your simulations might involve a lot of I/O (input/output), you must never run your simulations in your `$HOME` directory, but setup the runs in your `$SCRATCH` space. The disks associated with this space are designed for heavy loads[3]. Lastly, your quota in `$HOME` is much smaller compared to `$SCRATCH`. However, please note that the memory in `$SCRATCH` is temporary and **any files older than 15 days are deleted automatically** (see `$SCRATCH/__USAGE_RULES__`). Follow the links below for more information on the Euler cluster.
Use the following command to change to your scratch directory:

```
$ cd $SCRATCH
```

Additional information on the Euler cluster, its instruments and on how to use it can be found at:

[2]**Tip:** Save these commands in your $HOME/.bashrc file to automatically load these modules on login.
[3]However, `$SCRATCH` is not designed for frequent storing. If you are logging temporary results into a file, open the file once at the beginning, and do not flush e.g. more than one per second (or per minute). Related to it, note that `std::endl` not only prints the newline character `'\n'`, but also flushes the stream.

- https://scicomp.ethz.ch/wiki/Euler
- https://scicomp.ethz.ch/wiki/User_documentation

Task 2: Cloning the Course's Repository

Once you have set up your local development environment on Euler, clone or download the contents of the course's repository using the following command:

```
$ git clone https://gitlab.ethz.ch/hpcse_fs19/fs2019.git hpcse2019
```

This creates a directory named `hpcse2019` inside your home folder. Its contents are those of the repository. If the repository gets updated, you can get the latest changes with the following command from within the repository's folder:

```
git pull
```

Further information about git:
Git Handbook by Github (10min)
Git cheat sheet by Github

Task 3: Compile and Run Heat2D

Throughout this semester, we will be working towards solving the n-Candle problem, as described in class. To solve this problem, we need to determine the values and uncertainties of all of its parameters. The mathematical model for this problem requires solving the steady state heat equation in two dimensions, subject to n external heat sources.

Uncertainty quantification methods requires us to run a vast number of evaluations of the computational model. In this homework assignment, you will explore the Multigrid method and apply algorithm optimization techniques to make the computational model converge to a solution as fast as possible. This step is perhaps the most important in achieving high-precision results given limited computational resources.

To help you in your endeavour (and facilitate grading), we provide you with a fully-functional code for the computational model, called *Heat2D*, that you can (and should!) use as a base for the project. You will find this code in the class repository (see Task 2), containing the following files:

- **heat2d.cpp**
- **Makefile** (Do not modify)
- **auxiliar/** (Folder – Do not modify)

Your tasks

a) First, compile Heat2D by running the following commands:

   ```
   > make clean; make
   ```

   Once compiled, you can run Heat2D by running the following command:
   `> bsub -I ./heat2d -p N`, where N is problem number.

Alternatively, for short runs you can use:

```
> ./heat2d -p N
```

We provide 3 different problems:

- **Problem 1:** Small Matrix size for quick tests (not graded).
- **Problem 2:** Big Matrix size for correctess and performance evaluation (graded).
- **Problem 3:** Another Big Matrix size problem (graded).

b) To make sure Heat2D is producing the correct solution, run a job on an Euler's compute node using the *–verify* flag:

```
> bsub -I ./heat2d -p N --verify
```

Note: To make sure your submission is correct, your code should pass this verification for Problems 2 and 3.

Task 4: The Power of the Multigrid

Recommended Read:

G. Strang - Multigrid Methods: http://math.mit.edu/classes/18.086/2006/am63.pdf

In this task, we evaluate the effect of using the Multigrid method on convergence speed. That is, how much time does Heat2D take to get close enough to the solution. First, take a look at *heat2d.cpp* and localize the following parameters:

- s.setGridCount(1);
- s.downRelaxations = 1;
- s.upRelaxations = 1;

Where: *setGridCount* indicates the number of grids to use, *downRelaxations* indicates the number of Jacobi steps down the V-cycle (after restriction), and *upRelaxations* indicates the number of Jacobi steps up the V-cycle (after prolongation).

We ask you to run the following steps:

a) You can see that, initially, we have set up all of these parameters to 1. This corresponds to a non-multigrid approach where we only apply the Jacobi method on the finest grid. We will use this case as baseline result to evaluate the effectiveness of the Multigrid approach. For now, run **Problem 2** and write down the resulting number of iterations and total running time. Remember to use the --**verify** flag to ensure results are correct.

b) Now, try adding a single coarse grid into the solver by setting *setGridCount(2);*. Continue increasing this parameter until you do not observe any new benefits. For each one of these runs, write down the number of iterations and total running time.

c) Once you have found the optimal value for *setGridCount*, use the same procedure to find the optimal values for *downRelaxations* and *upRelaxations*. Remember to write down your results and verify that the solution is correct.

After gathering all your results, answer the following questions in your report:

d) Write down your results, indicating for each the values of the 3 Multigrid parameters and the number of iterations and running time required by each. Clearly indicate which configuration provided the best result.

e) Explain in your own words the effects of using multiple grids and why it improves convergence speed as you add more levels.

f) Explore the effects of using more than one Jacobi steps going down and up the V-cycle and explain why that it improves convergence speed.

g) Optional: Our computational model only considers the V-cycle traversal of the grids. However, other types of cycles exist (e.g, W-cycle) that could provide faster convergence rates. Try to find and implement one that works better than the simple V.

Task 5: Memory / Cache Optimizations

Recommended Read:

Kowarschik, M. and Weiß, C., *An overview of cache optimization techniques and cache-aware numerical algorithms.* In Algorithms for Memory Hierarchies 2003.

You have found the best configuration for the Multigrid parameters. However, if you take a cursory look at the code, you will be able to find that the solver provided has many performance-related issues. Perhaps most notably, memory access patterns are extremely inefficient, causing many cache fails as they perform Multigrid operations and Jacobi relaxations. In this task, you will apply some of the techniques described in the paper by Kowarschik and Weiß to solve these problems and achieve optimal cache performance.

Please address the following items:

a) In section 3.1 *Data Access Optimizations*, the authors of the paper describe transformations to improve performance in algorithms that traverse a multi-dimensional array. Answer: How would you apply *Loop Interchange* and *Loop Fusion* to the code?. Apply those transformations to heat2d.cpp.

b) The allocation of rows for Grid structures (U, Un, Res, f) seem to be intertwined. Answer: How does this ordering affect cache efficiency?, and how would you solve this problem? Apply your proposed solution to heat2d.cpp.

c) Implement the *Loop Blocking* technique (as described in the paper above) for the Jacobi solver loops and experiment using different block sizes. Answer: Could you achieve any speed-up from it? If yes, write down the optimal block size. If not, explain why.

1-Week Milestone: At this point your code should be able to solve Problem 2 in < 14 seconds.

Task 6: Constant Manipulation

Recommended Read:

https://compileroptimizations.com/.

When employing the -O2 flag, we allow the compiler to apply a wide range of optimizations that vastly improve the performance of our code. Among them, constant folding and copy propagation are perhaps the most well-known techniques. The compiler is, however, limited to optimizations that do not violate the semantics of your code and therefore some of these potential optimizations are left unapplied.

a) Peruse the website provided above and try to find parts of your code that could be manually optimized. Hints:
   - The multiplication operation (*) is much faster than the division operation (/)
   - The pow() is very slow and must be taken out of inner loops wherever possible.

Write down the optimizations you applied in the report.

Task 7: Vectorization

Recommended Read:
https://software.intel.com/en-us/cpp-compiler-auto-vectorization-tutorial.

By now your version of heat2d.cpp may be running many times faster than the one we provided. That's good, but there are more opportunities to speed it up. One of them is to make use of the processor's vector operations. As you know, such operations apply the same instruction to multiple contiguous elements in memory at a time, increasing computational peak performance. It is your turn now to apply vectorization to your code and see how much faster it can converge to the solution.

a) Using the compilation flags indicated in the tutorial above, create a full vectorization report of your code. Identify the loops corresponding to Jacobi, Restriction, Prolongation, and Residual calculation functions. Please answer in your report: what are, if any, the problems that prevent the compiler to vectorize those operations?

b) Look further in the tutorial and Answer: what hints provided by the compiler exist to bypass these limitations and fully vectorize your code? Apply these hints to fully vectorize heat2d.cpp.

c) Answer: What is the difference between unaligned and aligned SIMD operations? Investigate the necessary steps to allow aligned vectorization and apply them to heat2d.cpp. Answer: Was there an observable difference in performance?

d) (Optional) Try to use Intel's intrinsics https://software.intel.com/sites/landingpage/IntrinsicsGuide/ to manually optimize these operations. Could you improve performance over that achieved by auto-vectorization?

HW1 Performance Goal: Your code should be able to solve Problem 2 in < 11 seconds.
Our Performance: Problem 2 in 8.897 seconds. Can you beat us? ;)

Guidelines for reports submissions:

- The recommended length of the reports is approximately 2 pages long.
- Login to your course Moodle site.
- Submit only two files: a pdf of your report, plus the solution in *heat2d.cpp*.