Hardware Aware Scientifc Computing (WS 2023) **Exercise 0**

Prof. Dr. Peter Bastian, Santiago Ospina De Los Ríos
IWR, Universität Heidelberg

Submission date: 2023-10-30

Notes:

- Tutorials will take place on Tuesdays starting at 16:15 in the Mathematikon CIP Pool - Room 3.103 (3rd floor) - INF 205
- You need to show a *significant effort* in the exercises to be admissible for the exam.
- We will use the following system:
  - You need to submit your solutions into Moodle. Your solution won't be graded so it doesn't need to be a perfect solution or solve everything. This is to get a rough idea of what you have done.
  - However, your group has to report in Moodle the exercises for which you can present your solution in the tutorial. Your presentation should show that you put effort into solving the problem and tried to get it done.
  - Awarding points works the following way: You get all points for all exercises where your group volunteered for presenting. If you are actually picked to present an exercise but you cannot present anything, you won't get any points for this exercise sheet.
  - Every person should at least present once.
  - You need to reach 50% of the points to show a *significant effort*.
- People who can't attend the tutorial need to write an (justified) excuse if they will miss a tutorial.
- We will use Moodle to keep track of points. Please register here `https://moodle.uni-heidelberg.de/course/view.php?id=20117`.
- Please choose a group to present your exercise here `https://moodle.uni-heidelberg.de/mod/choicegroup/view.php?id=961346`.

Exercise 1 *What hardware am I using?*

Since we want to do hardware aware scientific computing it makes sense to get to know the hardware of your computer. Try to find out about the specifications of your hardware and take some notes as you might need those informations for the following exercises. The best way to do this will depend on your operating system.

You might be interested in the following: CPU model name, number of cores, clock rate, clock rate in turbo mode, cache sizes (note: find L1/L2 cache size per core), multithreading, memory bandwidth, vectorization (sse?, avx?).

(2 Points)

Exercise 2 *Getting the Code*

This is not a real exercise but a short explanation how to get the example code that will also be used in the exercises below. The code for this lecture can be found in a git repository hosted here on our gitlab[2].

You can get a copy of the code by calling

```
    git clone --recursive \
      https://parcomp-git.iwr.uni-heidelberg.de/Teaching/hasc-code.git
```

Have a look at the file `README.md` and try to build the examples on your machine. We assume that you either know or will figure out how to work with git. Some useful commands might be `git clone`, `git status`, `git log`, `git pull`, `git stash`, `git stash pop`, `git mergetool`.

(0 Points)

Exercise 3 *Compiler Flags*

Look at the examples in the git repository and get familiar with the `time_experiment.hh` header. Set up experiments to measure the following things:

1. An empty loop that just iterates from 0 to $n-1$ without doing anything.
2. A loop that accumulates someting in a global variable.
3. A loop that fills a vector of size n with the integers $0, \cdots, n-1$.

Compile this executable with different compiler flags (eg `O0`, `O1`, `O2`, `O3`, `-march=native`, `-funroll-loops`, `-ffast-math` and different combinations of these) and compare the runtimes. What can you observe?

Bonus:[3] Measuring high performance code is a tricky bussiness. Especially when you try to benchmark small functions it can be very difficult to get reasonable results. There exist specific libraries for creating so called micro benchmarks, where the library tries to make sure that the function is called often enough. An example for such a library is `https://github.com/google/benchmark` which can also be used online through `https://quick-bench.com/`.

Try using this to run the above examples and see if you get similar results. **(4 Points)**

Exercise 4 *Parallel* daxpy *Routine*

In the lecture you have seen a parallel version of the scalar product. Implement the blas `daxpy` routine

```
    y <-- alpha*x + y
```

in a similar way. Choose `std::vector<double>` as the type of `x` and `y` and `double` as type of `alpha`.

(4 Points)

[2] `https://parcomp-git.iwr.uni-heidelberg.de/Teaching/hasc-code`

[3] Bonus parts are really completely optional, maybe we will never talk about it in the exercises ;)