Durham University

Department of Computer Science

Coursework: Colliding suns

Parallel Scientific Computing I

Tobias Weinzierl

November 16, 2020

In this assignment, you will write a very simple N-body solver which runs efficiently on a single node of a supercomputer. It simulates N objects (bodies) in space—think of suns—which all move and fuse when they hit each other. I have broken down the assignment into steps of increasing difficulty. Each step's result is to be handed in separately (see instructions below), so I can mark them independently.

The assignment steps mirror content from the lecture. I strongly recommend that you revise the lecture script while you complete the coursework. Consult the script's appendix for hints and helpful material how to produce high-quality solutions. The challenges within the assignment cover both topics from Parallel Scientific Computing: numerical algorithms and parallelisation. **This assignment is to be completed and handed in via DUO. For deadlines, please consult the relvant level rubric on DUO (3rd Year Computing).**

Submission format

- You are given a code `assignment-code.cpp`. Use this one as starting point. It is slightly different than the demo code we have used throughout the lecture (it plots two additional lines).
- You submit exactly one zip file which contains files named `step-1.cpp`, `step-2.cpp`, `step-3.cpp` and `step-4.cpp`, plus one PDF file `report.pdf`. If you don't stick to the naming conventions, your submission will not be marked. There is a Python validation script on DUO. You can run it via `python3 validate.py my-zip` and it will tell you whether your zip file conforms to the submission guidelines.
- Do not make the code output any additional information to the terminal, i.e. do not alter any screen output.
- Do not make the code accept any additional arguments. No matter how you alter your code, it should still only read the parameters my template code accepts, too.
- I will translate your code on a Linux machine with the Intel compiler. My tests will use the compiler options `-fopenmp -O3 -xhost --std=c++0x`. On Hamilton, you can try out these flags with

```
module load intel/2019.5
icpc -fopenmp -O3 -xhost --std=c++0x assignment-code.cpp
```

- You can reorganise the code in whichever way you want, but all the implementation has to end up in the single `.cpp` file per step. Do not split up your code into multiple files per step!
- DUO provides a Python script that allows you to generate arbitrary complex, random initial configurations of particles. I'll use this same script to generate scenarios. My masses of choice for the suns m_i will be normalised, i.e. drawn from $m_i \in]0,1]$.

Step 1: Multiple free objects

This first step extends the given assignment code into a real N-body code. This is the baseline of the whole coursework. It is used in all follow-up steps.

In the template code given to you, one object (the first one specified on the command line) is free, while all the other bodies are fixed in space. Alter the code such that all objects move freely through space. Furthermore, make two objects fuse into one if they collide and ensure that the terminal output of the code (minimal object distances, maximum velocities, final number of objects and object's position) remains valid.

- Two objects collide if they end up in positions p_1 and p_2 with

  $$|p_1 - p_2| \leq C(m_1 + m_2) \qquad \text{with } C = 10^{-2}/N$$

  after a time step. N is the object count at the simulation startup. There is no need to check the actual trajectory. Simply use the final position after each step.
- The merger of two objects has a new (weighted) velocity of

  $$v = \frac{m_1 v_1}{m_1 + m_2} + \frac{m_2 v_2}{m_1 + m_2}.$$
- The merger has a new mass of $m_1 + m_2$.
- You can average the objects' positions. Use the weighted average here, too, i.e. use $(m_1 x_1 + m_2 x_2)(m_1 + m_2)^{-1}$.

Marks will be given for the correctness of the solution. The solution to this step is to be handed in as a single file called `step-1.cpp`. The step assesses your understanding of the test code that I used throughout the lecture. It touches on the principle of numerical imprecision due to time stepping. Finally, it establishes a baseline to assess workload imbalances for the next steps. I do recommend that you use some basic profiling to eliminate major runtime bottlenecks.

This step is worth 20 marks (correct movement: 6, fusion: 6, speed of implementation: 8).

Step 2: Vectorisation

This step extends the source code from step 1 and makes it exploit vectorisation.

Assess the vector efficiency of your solution. Where appropriate—you might want to use performance analysis tools—add OpenMP vectorisation pragmas. The step assesses your understanding of the vector paradigm and basic OpenMP vectorisation pragmas. Marks will preliminary be awarded for performance, but you have to ensure that you don't break the code semantics. The solution to this step is to be handed in as a single file called `step-2.cpp`.

This step is worth 20 marks (correctness: 5, obtained efficiency: 15).

Step 3: Higher order time integration

This step extends the source code from step 2, and adds a better numerical scheme. If you struggle with step 2, you are allowed to start with the solution of step 1.

Implement an RK(2) (Runge-Kutta 2) algorithm for your particle movement. This scheme is also called Runge's method (cmp. script).

The step assesses your understanding of numerical time integrators. Your lecture script discusses multiple variants of time stepping schemes with second-order accuracy. The most widely used scheme for N-body codes is the leapfrog scheme. I however want you to implement Runge-Kutta here to assess that you understand RK's data flow.

Marks will be awarded for the correctness of the implementation. The solution to this step is to be handed in as a single file called `step-3.cpp`.

This step is worth 20 marks.

Step 4: OpenMP

This step should extend the source code from step 3 and adds multicore parallelism. If you struggle with step 3, you are allowed to fall back to the solution from step 1 or step 2.

Parallelise your code with OpenMP vectorisation pragmas. Ensure that you do not break the global statistics `v_max` and `dx_min` which are plotted after each step.

The step assesses your understanding of the OpenMP parallelisation paradigm. Marks will be awarded for performance, but you have to ensure that you don't break the code semantics. The solution to this step is to be handed in as a single file called `step-4.cpp`.

This step is worth 20 marks (correctness: 5, obtained efficiency: 15).

Step 5: Report

This step assesses your ability to run numerical/HPC experiments. You should use your solution of step 4 for your experiments.

Study the scalability of your code and compare it to a strong scaling model. What are the involved constants? You can either use your own machine (if it is reasonably powerful) or use Durham's supercomputer Hamilton or use another Durham machine.

Furthermore, assess the convergence order of your time stepping scheme for both the scheme from step 1 and the scheme from step 3. For this, I recommend that you take a two-particle setup and you track where the particles collide. Rely solely on numerical data, i.e. do not try to use any analytical solution.

The step assesses your understanding of how to design numerical and upscaling experiments, how to present findings and how to calibrate models. Marks will be awarded for the data and the presentation. Besides some background information about the machine et al, your report should consist of two plots (one for the scaling and one for the convergence behaviour) and a brief paragraph that interprets the data. The solution to this step is to be handed in as a one-page PDF report `report.pdf`.

This step is worth 20 marks (data: 5, speedup plot: 5, convergence plot: 5, interpretation: 5)

Hamilton

If you plan to use Hamilton, please consider:

- Start early. If you submit tests to run on the machine, these tests are queued—you share the resources with others—so you have to expect a few hours until they start.
- Familiarise yourself with the machine early. To use a supercomputer, you have to write some startup scripts, you have to familiarise yourself with the modules environment, and so forth. It is not difficult, but you might have to invest some time.

- Make your supercomputer runs as brief as possible. This way, the scheduler on the supercomputer will squeeze them in whenever some idle time arises on a node. For the speedup studies, a few minutes of simulation time typically do the job.