Submission deadline: see sharepoint Coursework weighting: 100%

Assignment – N-body simulation

Parallel Scientific Computing (COMP3741)

Anne Reinarz* Christopher Marcotte† Gökberk Kabacaoğlu

This coursework consists of three components, one for each submodule covered by coursework. The components will be submitted individually via Gradescope.

- Parallel Computing: Deadline 21/01/2025
- Numerical Algorithms: Deadline 13/03/2025
- System Administration: Deadline 29/04/2025

Each section of this coursework consists of an implementational component and some questions. The questions should be answered in a short report (1 page maximum not counting figures per submodule).

You should use Durham's supercomputer Hamilton[1] and NCC to complete this assignment.

Hamilton uses a queuing system: when you submit a job to run on the machine these are queued. Since you share the resources with others, you will have to wait – potentially a few hours! – before they start. Make your runs as brief as possible to permit the scheduler to squeeze your job in as soon as a node is idle. For the speedup studies, a few minutes of wall time are typically sufficient.

Step 0: Verification

Before you start writing your own code, make sure you can run the non-vectorised, non-parallelised code (i.e. `step-0.cpp`) on Hamilton. Review the Python script which allows you to generate arbitrarily complex initial configurations of objects, `create_initial_conditions.py`. Ensure you can reliably invoke this script for your testing on Hamilton. When marking, we will use this script to generate test scenarios, with masses $m_1, \ldots, m_N$ normalised so that $m_i \in]0, 1]$.

*anne.k.reinarz@durham.ac.uk
†christopher.marcotte@durham.ac.uk
[1] https://www.dur.ac.uk/arc/hamilton/

Parallel Computing

In this section you will vectorise and parallelise a provided N-body solver. You can find the implementation in the `NBodySimulation` class.

In the template code in the `NBodySimulation` class, N bodies move freely in space. They are subject to gravitational attraction, but collisions are ignored.

Step 1: Vectorisation

In `step-1.cpp`, you will improve the performance of the code in the `NBodySimulation` class by exploiting vectorisation. To do this, add methods to the `NBodySimulationVectorised` class, which extends `NBodySimulation`.

Think carefully about the data structures the code is using, and assess the efficiency of your solution. Where appropriate, add OpenMP single-thread vectorisation pragmas. You might want to use performance analysis tools to find the appropriate places.

Report

Measure your vectorized code timing against your single-threaded timing, and explain the difference in results in terms of your vectorization approach.

Step 1 Marks

Marks will mainly be awarded for performance, but you have to ensure that you don't break the code semantics.

Criterion	**Marks**	**Comment**
Correctness	10	Same results as `step-0`.
Efficiency	20	Speed-up over the non-vectorised implementation.
Questions	10	Clarity and correctness of write-up. Experiments are performed where appropriate and sufficiently described in the write-up.
Total	40	

Step 2: Instruction-level parallelism

In `step-2.cpp`, you should extend the code in the `NBodySimulationVectorised` class by adding another level of parallelism. Parallelise your code with OpenMP multi-thread parallelisation pragmas. You can do this by adding methods to the `NBodySimulationParallelised` in `step-2.cpp`. This class inherits from the `NBodySimulationVectorised` class you used in the previous steps. You are allowed to change this in `step-2.cpp` and make the

`NBodySimulationParallelised` class extend `NBodySimulation` instead. You may wish to do so, for example, in the following cases:

- If you struggled with step 1 and your vectorised code does not work or is slower than the sequential code in `NBodySimulation`.
- When none of the methods you added to `NBodySimulationVectorised` can be reused in `NBodySimulationParallelised`. If that is the case, you may prefer to copy the code of the `NBodySimulationVectorised` class over to `step-1.cpp`, then rename the copied class `NBodySimulationParallelised`, and finally make any required modifications.

Ensure that you do not break the global statistics v_{max} and dx_{min} which are plotted after each step. This 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.

Report

In this section you will assess the performance of your code. You should use Durham's supercomputer Hamilton for this section.

- Provide some background information on the setup of your experiments (how many cores are available, what ISA extensions are being used, etc).
- Study the scaling of your code and compare it to a strong scaling model. Create a plot showing the scaling of your code with appropriate, transferable, quantities. Explain any unexpected features in your results.

These questions assess your understanding of how to design numerical and upscaling experiments, how to present findings and how to calibrate models. Marks will be awarded for completeness and correctness of data and presentation.

Step 2 Marks

Criterion	**Marks**	**Comment**
Correctness	10	Same results as `step-0`.
Efficiency	30	Run-time improvements with increasing core count.
Questions	20	Clarity and correctness of write-up. Experiments are performed where appropriate and sufficiently described in the write-up.
Total	60	

Step 1 and step 2 to be submitted via Gradescope by 21/01/2025.

Numerical Algorithms

In this step we will consider the theory behind the implemented algorithm and extend the N-body solver.

Step 3: Basic N-body solver

Step 3.1: Setting a baseline

Before we begin extending the N-body code, examine the existing code carefully. Answer the following questions and make sure you understand the structure of the code.

Report

- Estimate the cost per step in N, the number of bodies at the beginning of the simulation. Explain your reasoning. Run the provided code for `step-0`, does it have the expected time complexity? Show evidence.
- Explain what optimizations could be used to reduce the complexity of the implementation. Note: you do not need to implement these improvements.
- Experimentally find the largest stable time step for a scenario of your choice. Is it what you would expect based on the theory? Explain why or why not.
- Estimate the convergence order of the implemented time-stepping scheme and plot both your estimate and the measured convergence results. Explain any differences between your estimate and the measured convergence.

Next we will try to improve the runtime of the code. Build on the results from step 2. If your parallelisation or vectorisation codes are slower than `step-0`, then build on `step-0`, instead. You will not lose marks in this part for issues with the previous part. Improve the runtime by adjusting the implementation. Make sure the results remain the same up to a reasonable tolerance. Consider both the data structures used for the forces and the force calculation itself.

Report

- Given what we have learned about floating point error propagation in the course, what is a reasonable tolerance in this context?
- What improvements have you made in the implementation and why? Have you changed the data structures? If not why are the existing data structures efficient?

Step 3.1 Marks

Criterion	Marks	Comment
Correctness	8	Implementation gives expected results up to a reasonable tolerance.
Speed	8	Implementation run-time especially for larger N.
Questions	24	Clarity and correctness of write-up. Experiments are performed where appropriate and sufficiently described in the write-up.
Total	40	

Step 3.2: Collisions

The aim of this step is to modify the code of the `NBodySimulation` class to make the objects merge upon collision. Create a new class called `NBodySimulationCollision` in which to save your modifications. Note that `step-3` will not compile without this class.

Two objects b_1 and b_2 collide if after a time step they end up in positions x_1 and x_2 such that

$$|x_1 - x_2| \leq C(m_1 + m_2), \qquad C = 10^{-2}/N,$$

where m_1 and m_2 are the masses of b_1 and b_2, respectively, and N is the number of bodies at the beginning of the simulation. It is not necessary to check the trajectory of the objects, you can simply use their position at the end of each step. This means that occasionally a collision might be missed. You should tweak the tolerance C in order to avoid this as much as possible.

The merger of the two objects has mass

$$m = m_1 + m_2.$$

Velocity and position should be the mass-weighted mean of the velocities and positions of the colliding objects: the new object should have position x and velocity v defined by

$$x = \frac{m_1 x_1 + m_2 x_2}{m_1 + m_2}, \qquad v = \frac{m_1 v_1 + m_2 v_2}{m_1 + m_2},$$

where v_1 and v_2 are the velocities of b_1 and b_2, respectively.

Report

- Compare your earlier convergence estimate and measurements to the case with collisions/mergers. Explain any differences in the measurements.

Step 3.2 Marks

Criterion	Marks	Comment
Collision & mergers	8	Collisions are correctly implemented. Implementation does not change results in scenarios without collisions.
Speed	8	Implementation speed in a scenario with many collisions.
Questions	4	Clarity and correctness of write-up. Experiments are performed where appropriate and sufficiently described in the write-up.
Total	20	

Step 4: Advanced Methods

In the lectures, we have seen several different potential improvements to N-body solvers. Choose one and implement it. This could be, for example, a high-order time-stepping scheme, or an adaptive time-step; or an improvement to the complexity of the force calculation, e.g. to $\mathcal{O}(N \log(N))$. It can also be something we haven't discussed in detail, such as improved visualisation or advanced collision detection.

Report

- Describe the improvement you have chosen and which effect it has had on your implementation. If applicable, explain if this what you would expect theoretically.

Step 4 Marks

Criterion	Marks	Comment
Correctness	10	Same results as `step-3.cpp`
Speed	10	For algorithmic improvements: The run-time has improved (at least for large simulations) over `step-0.cpp`. For additional features: The features are implemented efficiently.
Questions	20	Clarity and correctness of write-up. Experiments are performed where appropriate and sufficiently described in the write-up.
Total	40	

Step 3 and step 4 to be submitted via Gradescope by 13/03/2025.

System Administration

Please use NCC to implement and test all subsequent tasks. Use the baseline code (`step-0.cpp`) provided rather than building on previous steps.

Step 5: GPU Implementation

Instrument `NBodySimulation` with OpenMP's time measurement routines to identify the functions that require the most computation time. Neglect the time for I/O operations and the time for the *initial* memory allocation of the particles.

Next use OpenMP to develop a GPU implementation of `NBodySimulation`. Save your source code as file `step-5-gpu.cpp` and add a target to the `Makefile` that allows building your GPU code via a target `step-5-gpu` (i.e. by calling `make step-5-gpu` at the command line) that produces the executable `step-5-gpu`. Ensure that your code reproduces the correct results from `step-0`.

Report

- Justify the applied directives and clauses for parallelization and memory management,
- Justify any code restructuring and performance optimization techniques, and
- Determine the runtime of your GPU code for an input size of $N = 20$ and compare it to the runtime of the original, sequential CPU code for the same size.
- *Estimate* the speedup of your GPU code over the serial implementation in `step-0` for $N = 100000$ and discuss your estimate.

Step 5 Marks

Criterion	**Marks**	**Comment**
Instrumented code	10	Marks given per instrumented function
Correctness	10	GPU code produces the same results as `step-0`.
GPU implementation	30	Performance of the GPU code.
Questions	10	Clarity and correctness of write-up. Experiments are performed where appropriate and sufficiently described in the write-up.
Total	60	

Step 6: Cluster computing

Write an `sbatch` script `run_all.sh` that generates random initial conditions using `create_initial_conditions.py` for $N = 10, 12, 14$ and 16, and stores the inputs in a folder

called `data/`. Add functionality to your `sbatch` script which runs the corresponding test cases in the `data/` directory using the implementation from `step-0`.

`numactl` [2] is a commandline tool that comes in handy when analyzing the memory architecture of a compute node. For instance, to ensure data locality for processes or threads that operate on the same data. However, `numactl` is not necessarily installed on all clusters. Write a shell script `install_numactl.sh` that builds `numactl` from source and installs it locally in your home directory. Document the output of `numactl -H` when executed on the login-node of NCC in `numactl.out`.

Report

In the report, write a short text that explains how you could make an installation of the N-body solver available to all users on the *PVM*, the virtual machine cluster setup during practicals. Discuss the advantages and disadvantages of your chosen approach.

Step 6 Marks

Criterion	**Marks**	**Comment**
Batch script	10	Correctly stores data and runs all required test cases.
Numactl	10	The batch script correctly installs numactl and sample output is provided.
Questions	20	Clarity and correctness of write-up.
Total	40	

Step 5 and step 6 to be submitted via Gradescope by 29/04/2025.

[2] https://linux.die.net/man/8/numactl

Important submission requirements

- The code you submit must compile and run on Hamilton using the `Makefile` provided. No marks will be assigned for code that does not compile. You should use the code framework provided as your starting point. You may modify the `Makefile`.
- Your submission for each part should contain the following files.
  1. One PDF file named `report.pdf`, containing your answers to the questions above. This file should be no more than one page in length excluding figures.
  2. All source code required to compile and run your work on Hamilton (part 1 and 2) or NCC (part 3). We will expect to find files containing your solution for each step, e.g. `step-1.cpp` or `step-2.cpp`, as well as `Makefile`, `NBodySimulation.cpp`, `NbodySimulation.h`, `NBodySimulationVectorised.cpp`, and if needed any header or C++ source files that are required to compile these.

  Not following these submission guidelines, for example by writing a longer-than-one-page report, or not respecting the naming conventions, will result in marks being deducted.
- Make sure that the code does not produce any additional output in the terminal, i.e., do not alter any screen output. If you make the code accept any additional arguments, make sure that they are optional.
- In the report, write down what you observe. If you know that your implementation is wrong, tell us what you observe and what you would have expected instead: you may still obtain a substantial portion of the marks for that step.
- If you are having trouble visualising the output with ParaView, make sure that you have switched to "GaussianPoint" in the "properties" tab.

Collaboration policy

You can discuss your work with anyone, but you must avoid collusion and plagiarism.[3] Your work will be assessed for collusion and plagiarism through plagiarism detection tools.

[3] https://www.dur.ac.uk/learningandteaching.handbook/6/2/4/