Imperial College London Department of Aeronautics
High Performance Computing - Coursework Assignment Report
Name: Kin Pui Tam CID: 01338789 Date of Submission: 24/03/2021

1 Introduction

This report captures the results from the smoothed particle hydrodynamics (SPH) formulation of the Navier-Stokes equations written in C++ with external libraries including MPI, LAPACK, BLAS and Boost. The objective of this coursework is to create a parallel numerical code that uses an SPH model to simulate particle interactions. A serial code was first developed to test the model, which is then optimised for greater performance. Next, the code is parallelised such that it can run on multiple processes and finally it is further optimised. This program consists of five `.cpp` files and one `.h` header file as listed below, these files are then compiled using `Makefile` to create a single executable. Detailed documentation of all functions, classes and variables can be found within the following scripts.

1. `cwMain.cpp` initialises MPI and boost program options, while also constructs the class SPH with parameters either specified by the user or by default.
2. `sph.h` is the header file for the class SPH which contains the class headers, the headers for its member functions and the definition of constants and arrays.
3. `sph.cpp` contains the class constructor and the member functions for simulating the physics of the model.
4. `sphInit.cpp` initialises the position and velocity of each particle before the simulation.
5. `sphPara.cpp` contains the member functions necessary for implementing parallelisation.
6. `sphDisp.cpp` contains the member functions for printing vectors and matrices on the console.

The general workflow of the simulation is such that a user can execute the built program by specifying the case in the terminal, this sets the number of particles `N` which are placed evenly throughout a predetermined area within the domain[1]. MPI distributes the particles as evenly as possible to each process, each having assigned `n` particles. The initial conditions are set up by calculating the densities to rescale the mass `m` before the simulation starts. For each time step until the simulation time reaches the end time, 6 procedures are involved, including calculating density, pressure, pressure force, viscous force, gravity force and time integration. The term procedure is explicitly used throughout this report to avoid confusion with steps as in time steps or processes as in computing processes. The physical boundary conditions of the domain are enforced by reversing a particle's velocity and stepped down by a factor of `e` when it is within a distance `h` from the boundary and having an original velocity that points towards the boundary. Finally, the kinetic, potential and total energies are calculated before advancing to the next time step.

2 Energy Plots

The three test cases, dam break, block drop and droplet, were simulated by the program. Their energy plots comprise kinetic energy, potential energy and total energy, which are presented in this section.

2.1 Dam Break

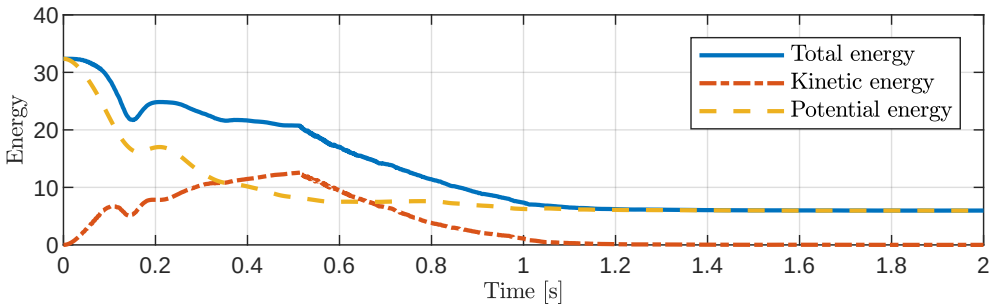

Figure 1: Energy plot for the dam break test case, `N = 20 x 20 = 400`

2.2 Block Drop

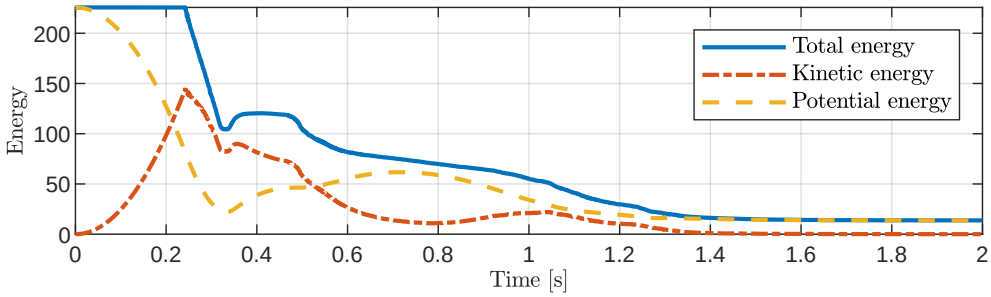

Figure 2: Energy plot for the block drop test case, `N = 21 x 31 = 651`

2.3 Droplet

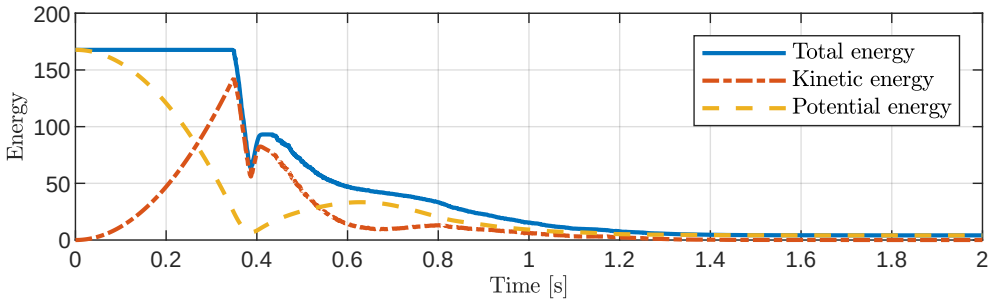

Figure 3: Energy plot for the droplet test case, `N = 311`

3 Serial Performance and Optimisation

A serial program was first created to provide a baseline reference to the expected behaviour of the model. This first version of the program cares little about performance and was used to provide later versions with model answers for each validation and test case. In this version, the code follows the exact routine as listed on the assignment handout, with one for-loop enclosing each procedure. This version of code is robust but is extremely time-consuming to run as it executes many repeated actions. As an example, by enclosing each procedure by its own for-loop, the value q cannot be passed to the next procedure, meaning that it has to be recalculated for pressure force and viscous force. After validating the results, an attempt to improve the baseline version was carried by storing the values of q in an array with indices that depend on both outer for-loop index `i` and inner for-loop index `j` for calculating densities. This means that the values of q for each pair of `ij` are only calculated once per time step when calculating densities. The array that stores the values of q can then be recalled by the procedures for calculating pressure and viscous forces, thereby reducing the number of times that q has to be repeatedly computed.

Further optimisation of the above improvement was to completely remove having to store the value of q at all. This is achieved by placing procedures within the same for-loop `i` or `j`. For example, density, pressure and gravity forces were placed inside the same outer loop `i`, pressure force and viscous force calculations were placed within the same inner loop `j` inside the outer loop `i`. This enables the values of q to be accessible to all the procedures within the loop `ij`. This approach, however, generates a problem for pressure force and viscous force calculations since they involve ρ_j and p_j which will only be calculated when `i = j`. Since the values of ρ are reset at the beginning of each time step, $\rho_j = 0$ for `i < j` and the density calculations yield incorrect results. To solve this issue, the densities and pressures are recalculated and stored after the mass `m` of each particle has been rescaled, such that when the while loop is executed, essentially giving a head start to the simulation. As a result, the for-loop `ij` technically calculates forces of this time step, while it also calculates densities and pressures for the next time step which are then stored for calculating forces at the next time step.

The final serial code is optimised with the above approach, together with other more common and straightforward techniques to enable a shorter run time. These include pre-calculating repeatedly used values and replacing division by multiplication, etc. Altogether, the serial code is able to run more than twice as fast as the original serial code that calculates each procedure after another.

4 Parallelisation and Design Decisions

The original thought on parallelising the program was to assign procedures to different processes. This is not an efficient way of splitting since there are only 6 procedures, meaning that if we assign more than 6 processes, the excess processes will either be idle or redundant. The workload is also unevenly distributed since some procedures are more computationally expensive, such as calculating pressure forces as compared to gravity force. Another possibility is space partitioning, which comes in several forms such as uniform grid partitioning and k-d tree partitioning[2]. The former splits the domain into uniform grids and the latter splits the domain along the median of the particle position along the chosen axis. Uniform partitioning creates a large number of cells, meaning that there is a high chance that a grid is empty and occupies unused memory. For most scenarios in our validation and test cases, the particles pile at the bottom of the domain, meaning for most of the run time, more than 90% of the whole domain is empty. The k-d tree also presents a steep learning curve since the domain has to be split at each time step and each particle may exist is multiple processes, making communication difficult.

The final attempt to parallelisation is done by splitting groups of particles to all assigned processes as evenly as possible. For example, if we have 13 particles to split to 4 processes, namely `rank` 0 to `rank` 3 with a total `size` of 4, since the remainder `rmd = N/size = 1`, for all processes `rank < rmd`, they will have `n = int(N/size)+1 = 4` particles, while the others have `n = int(N/size) = 3` particles. The even distribution aims at averaging the workload for each process as much as possible. Since the particles are now split into groups by processes, we have to find efficient means of communication between them. It is quite straightforward to spot the parameters that need passing around. Assuming we are now looking at `rank` 0 which has `n` out of `N` particles assigned to it. We can loop through its particles with an index `i` from `0` to `n-1`. These forces on this particle `i` is then calculated by investigating its interactions with all other particles `j` ranging from `0` to `n2-1`, where `n2` is the number of particles assigned to `rank d` out of `size`. Therefore, referring to the assignment handout, only ρ, p, **x** and **v** are passed in between the processes as they are the only variables that possess a subscript j. The original approach was to create a global array for each of these values such that all processes know the cross-process values of each particle at all time steps, for example, all ranks store an array of the position of all particles and these arrays are identical. This method becomes increasingly inefficient with the number of particles and processes since more memory is allocated, therefore the final optimised parallel program passes local information more frequently and global arrays only exist at `rank` 0 for console display and file output purposes.

5 Summary

The SPH program fulfils the requirements listed on the assignment handout, including accepting command lines, having a class named `SPH` to implement the simulation, generating an initial set of particles based on the command line and is parallelised using MPI, enabling it to run successfully on one or more processes. The build system Make is used to compile the code and git has been used for version control. Documentation and further descriptions of the program can be found within the corresponding scripts.

References

[1] C. Cantwell and O. Bacarreza, *High-performance Computing Coursework Assignment Handout*, Imperial College London, 2021.

[2] R. R. Kroiss, *Collision Detection Using Hierarchical Grid Spatial Partitioning on the GPU*, University of Colorado, 2013. [Online]. Available: `https://scholar.colorado.edu/concern/graduate_thesis_or_dissertations/vd66w0289` (visited on 10/03/2021).