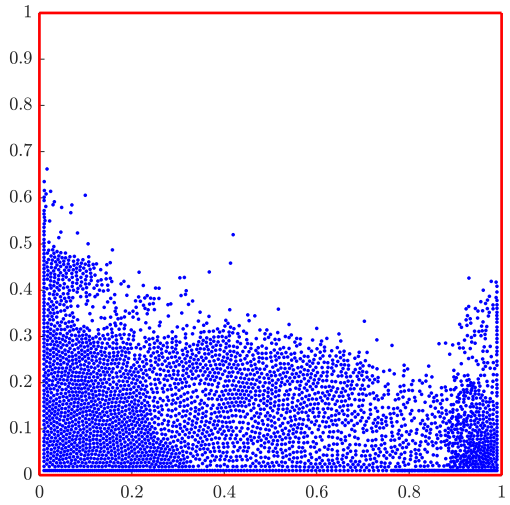
Imperial College London

High-Performance Computing:

Smoothed Particle Hydrodynamic Numerical Solution in a Unit Box

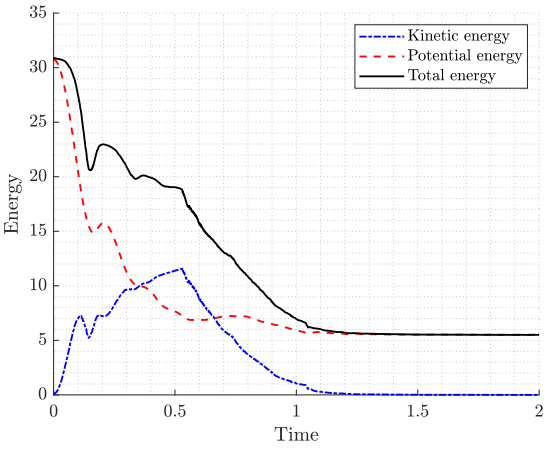


Academics responsible:	Dr Omar Bacarreza
	Dr Christopher Cantwell
Department:	Department of Aeronautics
Course:	H401 MEng Aeronautics
Module:	High-Performance Computing
Academic year:	2020/2021
Student:	Jonathan De Sousa
CID:	01485810
Personal tutor:	Dr. Oliver Buxton
Date:	24/03/2021

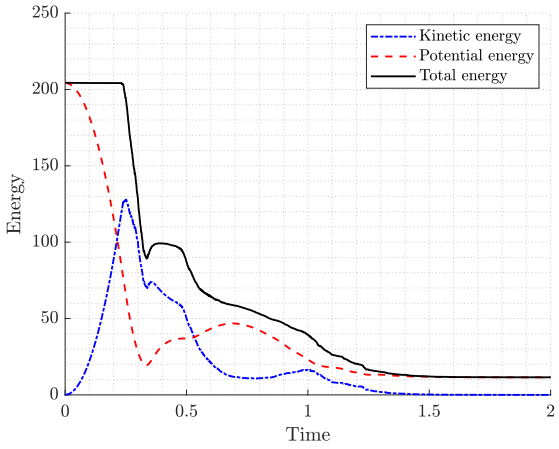
Department of Aeronautics
South Kensington Campus
Imperial College London
London SW7 2AZ
U.K.

1 Energy Results

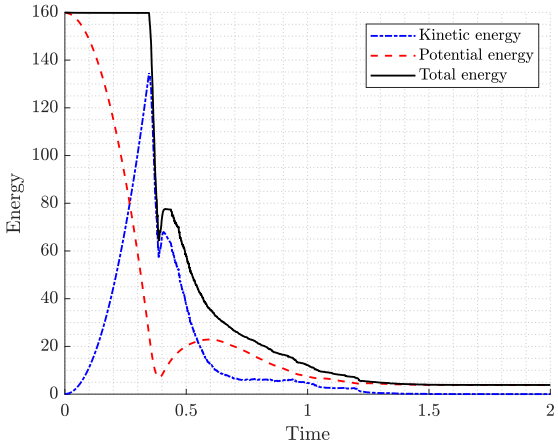
The two-dimensional smoothed particle hydrodynamic (SPH) formulation of the Navier-Stokes equations were initially implemented numerically in a basic unoptimised serial code. Using a basic code made debugging easier and more certain. From this code, the potential, kinetic and total energies of the particle system at each time-step were output for the three test cases specified in the coursework brief. The simulations were run for a time-step of 10^{-4} and a particle radius of influence of 0.01, with a computational domain occupying the region $[0, 1]^2$. The results were plotted in MATLAB, and are shown in Figure 1.



(a) Dam break initial condition - particles initially occupied the region $[0, 0.2]^2$.



(b) Block drop initial condition - particles initially occupied the region $[0.1, 0.3] \times [0.3, 0.6]$



(c) Droplet initial condition - particles initially occupied a circle radius 0.1, centre $[0.5, 0.7]$.

Figure 1: Energy plots for the SPH-modelled system of particles for different initial conditions.

The plots illustrated physically realistic results, which further validated the correctness of the basic serial code along with the validation cases provided in the brief. From this basic code, optimisations and parallelisation were implemented, and are discussed in Sections 2 and 3 respectively.

2 Serial Performance and Optimisation

Before optimising the code, the serial performance of the basic unoptimised code was first analysed using the Oracle Developer Studio profiling tools. The performance analysis and optimisations were carried out

using the droplet test case for a total integration time of 2, a time-step of 10^{-4} and a radius of influence of 0.01. Compiler optimisations were turned off to test the true raw performance of the code.

Total CPU Time		Name
EXCLUSIVE sec.	INCLUSIVE sec.	
67.507	67.507	<Total>
40.318	48.784	SPH::solve_q()
8.836	9.226	SPH::solve_F()
7.545	7.815	SPH::solve_rho()
7.295	7.295	__sqrt_finite
1.711	1.711	<static>@0x76abc (<libm-2.28.so>)
1.171	1.171	sqrt

(a) Functions view showing the functions that cumulatively spent the longest time executing.

```
0.020   92.   for(int i = 0; i < N; i++) {
2.041   93.       for(int j = 0; j < N; j++) {
5.014   94.           rx[i * N + j] = x[i] - x[j];
5.464   95.           ry[i * N + j] = y[i] - y[j];
35.345  96.           q[i * N + j] = sqrt(rx[i * N + j] * rx[i * N + j] + ry[i * N + j] * ry[i * N + j]) / h;
```

(b) Source view showing the main "hot" region in the code - part of the `SPH::solve_q()` function.

Figure 2: Critical performance statistics for the unoptimised serial code.

As shown in Figure 2, the `SPH::solve_q()` takes up the largest proportion of the total execution time. The other SPH member functions also have significant CPU times, along with operations in the math library.

One method used to optimise the serial code was to replace as many division operations as possible with less computational-intensive multiplication operations. In sub-figure 2b, the main "hot" area of SPH.cpp, for example, `/h` was taken out of the nested for-loops and calculated as a separate variable to be multiplied by the rest of the code on line 96.

Other optimisation techniques utilised were pre-computation of repeated expressions which was used together with the technique of lifting out loop-independent terms. This was possible in several parts of the SPH.cpp and noise-addition part of the main.cpp codes because the pre-computed expressions were constants and therefore loop-independent. The pre-computations were mainly applied in `SPH::solve_SPH` and not locally within the intermediate member functions, which helped eliminate pre-computations within the time-advancement loop. Pre-computation of the normalised particle separation, q, for each particle was also done so that q values didn't need to be repeatedly computed for each part of the algorithm that required them, e.g. calculations density and forces.

Cache locality was also considered by making sure that the nested for-loops were coded such that when consecutive terms of a matrix were required they were on the same cache line, reducing the number of retrievals from main memory. Luckily, cache locality was inadvertently applied in the unoptimised serial code.

Having implemented the above-mentioned optimisations, the code was re-executed from the terminal using the `time` command for each of the test cases. the average increase in performance was 31.6%, as seen in Table 1.

It was appreciated that the `sqrt()` function in `SPH::solve_q()` was computationally expensive. Trying to replace it with the `pow()` function didn't improve performance - execution time increased to 2m 43.767s. Since computing q involved computing a 2-norm, the best method would have been to re-structure the whole code such that the highly optimised BLAS DNRM2 routine could be used. Since this was realised quite late, it, along with more intensive use of BLAS routines in general, should be considered for future improvements of the code. It should be noted however that BLAS routines were still used in the time-advanced part of the code to improve performance.

In-lining of functions was considered but not implemented in the end because it provided a negligibly small improvement in performance at the cost of significantly reduced readability of code. For an integration time of 2 and time-step size of 10^{-4}, the time advancement loop involved 20000 iterations. With 5 functions being called in the loop, this leads to 100000 function calls in total, which is a relatively small number. This explains the negligible performance gain of in-lining.

Another notable optimisation was that used to allocate the dynamic memory for the particle positions in the droplet case in main.cpp. Instead of initially underestimating the required memory and having to constantly re-size the arrays for points in the square grid that fit in the circle (see Example 7.9 of the course notes), 79% of the memory size used for the square grid was allocated to the arrays for points in the circle. This is because the ratio of the areas of a circle inscribed in a square, is $\pi/4 \approx 78.5\%$.

3 Parallelisation

Once the optimised serial code was obtained, parallelisation was implemented. The initial method used to parallelise the code was to try and divide each and every step of the algorithm specified in the brief, equally between all processes. This was done to maximise the efficiency of the available computational resources.

This involved first obtaining the initial particle coordinates in main.cpp, then splitting up the particles to be worked on by each process as equally as possible. (Note that the method used (main.cpp: line 237-246) results in a reasonably equal split for a small number of processes, but as the number of processes increases, the split may result in the last process obtaining a significantly larger number of particles to operate on. This is an area of improvement for even greater performance.)

Once the particle coordinates have been split, each process then goes through each step of the algorithm, just like the serial code does but for a smaller number of particles. It calculates the normalised particle separation, q, computes the density, ρ, associated with each particle, computes the corresponding pressure, p, then computes the forces $\mathbf{F}^p$, $\mathbf{F}^v$, $\mathbf{F}^g$ and acceleration, $\mathbf{a}$ associated with each particle. The time evolution is then carried out, and the energies E_k and E_p are calculated, all locally as well. The process is then repeated over the full integration time. The parallelisation process was expected to increase performance but to further do so, instead of using MPI_Gatherv and MPI_Bcast in tandem, for which several areas of code required so, MPI_Allgatherv was used, which scales better as the number of processes increase.

Looking at Table 1 (fully parallelised), parallelisation does indeed lead to a significant increase in performance. However, realising that there were a very large number of MPI functions being called, and noting from the profiling process that the cumulative execution times of the BLAS routines were very small, the amount of parallelisation used in the program was relaxed for the calculation of the energies, such that the energies were calculated using full-particle system arrays, as opposed to locally and then having communication occur between processes to combine local contributions. This was done to reduce the overhead from calling multiple MPI functions for communication especially in the time-advancement loop, which iterates 20000 times for $T = 2$ and $dt = 10^{-4}$. As seen in Table 1 (partially parallelised), for a small number of processes the improvement is not clear, but as the number of processes increases, the increase in performance in seems to improve all-round between all test cases.

Table 1: Execution times for different levels of optimisation and parallelisation of the SPH program for different test cases, and the corresponding average increase in performance relative to the unoptimised serial program.

	Dam-break	Block-drop	Droplet	Average performance increase
Unoptimised, serial	2min 6.086s	4min 25.174s	1min 20.887s	—
Optimised, serial	1min 24.916s	3min 20.164s	50.529s	31.6%
Parallelised (fully, 2 processors)	1min 5.300s	1min 58.693s	29.246s	55.8%
Parallelised (partially, 2 processors)	1min 6.532s	1min 51.539s	29.497s	56.2%
Parallelised (fully, 20 processors)	11.473	21.959s	8.212s	90.8%
Parallelised (partially, 20 processors)	9.066s	17.366s	7.583s	92.3%

Knowing that the more relaxed method of parallelisation yielded better results for the case of this coursework, future improvements should consider more sophisticated approaches to parallelisation as opposed to naively splitting every calculation equally between all processes. This is especially important where a very large number of MPI function calls occur such as in the time-advancement loop. Better appreciation of the highly optimised BLAS routines should also be considered, in order to make trade-offs as to when to split computations then re-combine, and when not to.