HPC Coursework Report

Department: Department of Aeronautics
Student: Wee Zhao, Chua-Khoo
CID: 01117650
Date: March 25, 2021

1 Energy Plots

All simulations here are carried out using the *base* implementation, for a time period of $T = 5$, at a time-step of $\Delta t = 0.0001$, with $h = 0.01$. The energy time evolution in Figure 1, 2, 3 are sampled at an interval of 0.1 and truncated to $t = 0 - 2$ since steady-state has been reached.

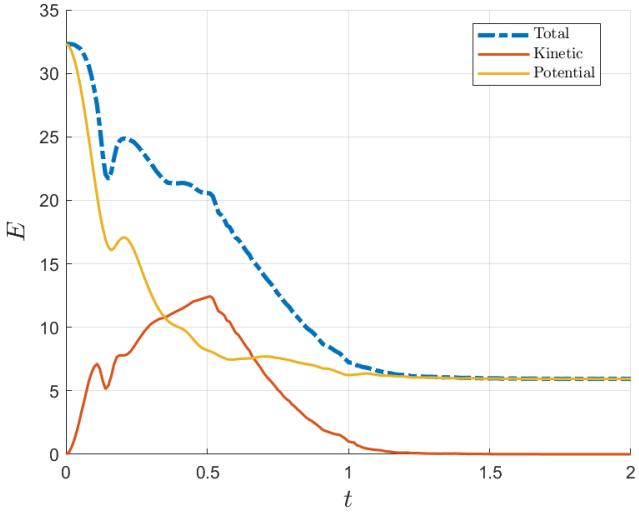

Figure 1: Dam-break energy evolution: 400 particles.

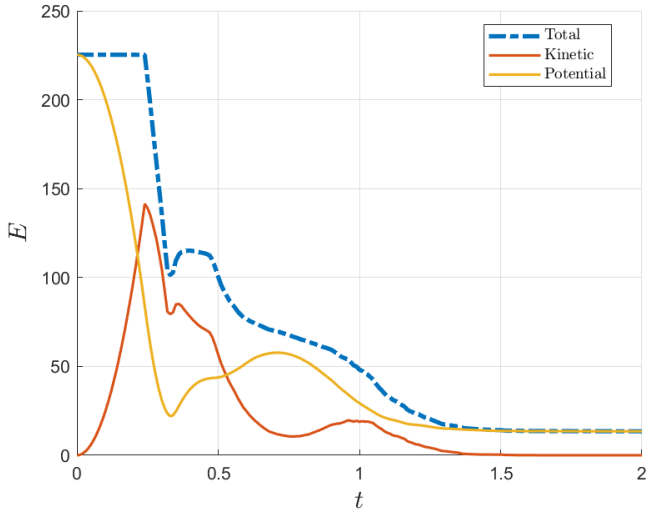

Figure 2: Block-drop energy evolution: 651 particles.

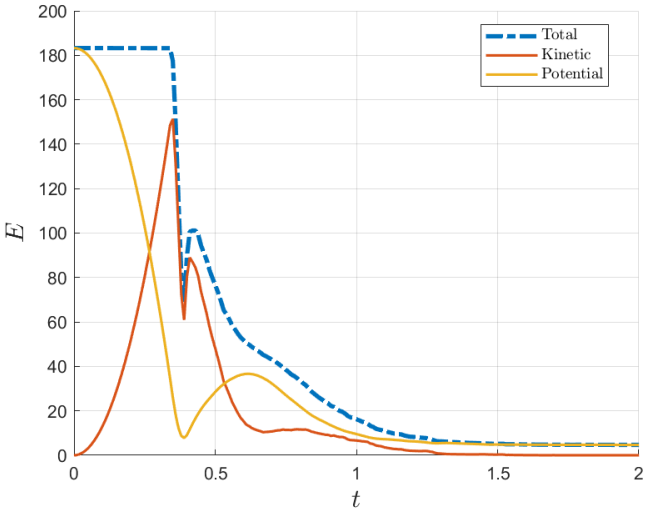

Figure 3: Droplet energy evolution: 340 particles.

2 Code Design and Implementation

The *.zip* file submitted contains the git repository with implementations located in respective branches. The implementations/branches are named *base*, *serial*, *parallel*.

2.1 Cursory View of the SPH Algorithm

The main performance bottleneck would be located at the computation of $\mathbf{r}_{ij}$, with a complexity scaling of $\mathcal{O}(n^2)$ for n particles. We can do better by considering anti-symmetry as $\mathbf{r}_{ij} = -\mathbf{r}_{ji}$. It still have the same complexity, but with a smaller scaling constant. In terms of parallelisation, the transfer of state data $\mathbf{x}$ and $\mathbf{v}$ through MPI for $\mathbf{Q}$, $\mathbf{r}_{ij}$ and $\mathbf{v}_{ij}$ computations would also be the main communication bottleneck.

After obtaining the data for $\mathbf{r}_{ij}$, $\mathbf{v}_{ij}$, q_{ij}, density and pressure states, we hold all the required data to proceed with the forcing and time-stepping computation in an isolated manner for each particle, and where parallel computations would speed up the simulation.

2.2 *base*

The *base* implementation contains the most naive serial implementation of the SPH algorithm. The main aim is to generate validation data, and acts as a basis for more complex algorithms. Although naive, various design choices were made taking code extensibility and efficiency into consideration.

The key design decision taken here is with the arrangement of the state vectors for particles. For cache coherency,

it is essential the data are stored in contiguous memory. For extensibility to higher spatial dimensions mentioned below, instead of splitting the component of each state vector into separate arrays, we want to store them in a single array. For parallelisation, we want the state vector components to be stored adhering to 'particle locality', avoiding extra computations to figure out where the components of a particle's state vector are spread across the array during communication. Hence, for a state vector $\mathbf{x}(i)$, the data is stored as $[\mathbf{x}(1), \mathbf{x}(2), ..., \mathbf{x}(n)]$, where n is the number of particles and N_{dim} the number of spatial dimensions. We call arrays storing vectors $2D$ arrays (a misnomer), and arrays storing scalars $1D$ arrays.

In terms of code design, the code base consist of a main class, *SPH*, containing the core SPH algorithm and data but also holds two utility class. Utility class *IO_Manager* deals with I/O by interfacing with the *boost_filesystem* library and *fstream*, while the *MPI_Manager* class coordinates massage passing between processes. For *base*, *MPI_Manager* is merely a placeholder. *linalg.h* is created to interface with the *Fortran BLAS* library. *BLAS*-Level 1 array scaling and addition operations are predominantly utilised by the SPH algorithm.

Another design choice made here is to consider code extensibility to arbitrary number of spatial dimensions. To allow for this, the header file, *const_param.h* contain the variable modification to switch between different number of spatial dimensions by changing *N_dim* and corresponding spatial domain boundaries. One main challenge which arose was during the initialisation of particle positions for the dam-break and block-drop initial condition. In general, we fill the initial box with a cubic lattice of evenly spaced particles. To do that, we generally use a nested *for-loop* to iterate across its respective spatial position, but now the loop depth is based on an user-defined *N_dim*. The solution is to use a recursive function to imitate each level of the nested loop, where the total recursion depth is parameterised by *N_dim*. The implementation is in the member function *SPH::RecursionBuild*. Array operations consisting of loops are parametrised by *N_dim* to ensure generalisability. So, if it turns out a three dimensional simulation is required the code can be a good starting point. Of course, the kernel functions of SPH are required to be modified for a different N_{dim}. Only $2D$ SPH kernel functions are implemented for this project.

Two implementation details to state: For $\mathbf{x} = [x_1, x_2, ..., x_{N_{dim}}]$, gravitational forcing is implemented to act on the N_{dim}-th dimension. For nested loop computations, care is taken so that the array memory are accessed contiguously and loop-independent expressions are lifted. Functions are not inlined for readability, and can be done via compiler optimisations.

2.3 *serial*

The optimisation done in this implementation is centred upon the anti-symmetry in $\mathbf{r}_{ij} = -\mathbf{r}_{ji}$. If we represent a matrix $\mathbf{Q}$ as $\mathbf{Q}(i,j) = q_{ij} = \dfrac{\|\mathbf{r}_{ij}\|}{h}$, we see that it is a $\mathbf{0}$ diagonal symmetric matrix. Furthermore, in a lot of cases, a particle is only surrounded by m particles, where $m \ll n$. Most of the information in $\mathbf{Q}$ are not utilised in the state update and time-stepping algorithm. $\mathbf{Q}$ can be considered sparse by discarding the inessential information. Hence, there are considerable memory and computational savings that could be optimised.

The optimisation done here is two-fold. The first aspect lies with the computation of $\mathbf{Q}(i,j)$. By exploiting symmetry, we can half the amount of computations of $\mathbf{r}_{ij}$ and $\mathbf{Q}(i,j)$. The second aspect lies with the storage of the interaction dataset of the (i,j)-particle pair (ie: $\mathbf{Q}(i,j)$, $\mathbf{r}_{ij}$, $\mathbf{v}_{ij}$). By only storing the interaction dataset where $\mathbf{Q}(i,j) < 1$, we reduce the required amount of data to parse through during the density, pressure and forcing computation loops.

Branching off from *base*, a struct called *interaction_set* is created in the *SPH* class. The struct is assigned to each particle, holding arrays of its interaction dataset. *Vector* containers are used due to the flexibility it provides by pre-allocating memory for array expansion for the varying number of neighbouring particles throughout simulation, and also improves code readability when compared the solution in *base*. The upper bound for the number of neighbours of a spherical shape (the kissing number) is used to allocate the vector's total memory capacity. Although we lose cache coherency when looping through particles (isolated struct for each particle), it is slightly alleviated by the fact that, for each particle i, the vectors $\mathbf{Q}(i,j)$, $\mathbf{r}_{ij}$, $\mathbf{v}_{ij}$ in the struct are arranged in contiguous memory, and accessed sequentially in the forcing computation loops. The changes are in *SPH::MapNeighbour*, *SPH::DensityPressureComputation* and *SPH::AccelComputation*. This implementation simplifies the code while also improving compute performance when compared to *base*. Approximately 40% savings in compute time is seen compared to *base* (see Table 1), and more than 65% if done with compiler optimisations and without I/O (Table 2).

2.4 *parallel*

In *parallel*, we address the parallelisation of state update and bring a divide and conquer approach to computing the interaction dataset. By splitting the particles evenly into k sets, the time complexity to compute the interaction dataset set is now $\mathcal{O}(n^2/k)$ for each process. Each process only computes the combinations of n/k particles in the local process with the n particles of the whole domain. k represents the number of processes in the *MPI* instance.

Building from the encapsulation of the interaction dataset in *serial*, we parallelise the code with *MPI*. Here, we make a distinction to the variable naming convention. A variable with $\{var\}\_G$ appended represents a property or data of the whole global simulation, while those without are properties local to the process. The set of particles each process received would be determined by evenly slicing the global array in a load balanced manner. Since particles will have inter-process interactions, after each leapfrog time step, a *MPI_Allgatherv* is used to distribute the updated $\mathbf{x}, \mathbf{v}, \boldsymbol{\rho}$ and $\mathbf{p}$ states to the $\{var\}\_G$ arrays for all processes.

Implementation based on the description above, minimal code refactoring are required to the core simulation functions. Communication calls via *MPI_Manager::Comm_Array_{}* manages to provides generality for different state array communication, while still having the *MPI_*$\{$*CollectiveOp*$\}v$ index and count parameters encapsulated in *MPI_Manager* itself. The symmetric properties of the interaction set is not exploited in this implementation. *Allgatherv* communication for every state update to compute the interaction set is also wasteful, as only states of neighbouring particles are needed. Exploiting them requires additional code complexity, and a better resolution is provided in *parallel-div*. Computational times of 2 parallel processes are similar to *serial*, as both have a $\mathcal{O}(n^2/2)$ scaling, but with more processes we start seeing better performance (see Table 1 and 2).

3 Further Improvements

3.1 *parallel-div*

Here, we can further exploit the observation that for n particles, the number of neighbours, m, is $m \ll n$. The main idea is to divide up the simulation domain into k sub-domains and $\mathbf{Q}$ would only check the sets of particles in the sub-domain, reducing the complexity to $\mathcal{O}\left((n/k)^2\right)$ if each sub-domain is a parallel process. It exploits the locality between particles, and enables the interaction set's symmetry to be exploited with ease. Communication at every state update would be reduced to only the sets of particles crossing sub-domain boundaries. For our SPH simulation, gravitational forcing is the most persistent forcing. A suspended blob of particles will fall in direction of gravity and spread across the domain boundary. The directions orthogonal to the gravity forcing would be the directions with the bigger variance in particle position. Hence, the domain division is split it in regard to the orthogonal gravitational direction for simplicity. By tabulating and sorting particle position in each remaining dimension, we split the domain into intervals based on every (n/k_{dim})-th particle in each direction. To address the particle interactions between sub-domain boundaries, for each sub-domain, we have two sizes. An actual physical sub-domain, where all particles in it are updated by the corresponding process, and a ghost-domain which is Δh bigger. Hence, interactions of these ghost particles belonging to neighbouring sub-domains are accounted for. Particles which lie exactly on the boundary are considered to belong to the sub-domain in the -ve direction. Then for $\mathbf{Q}$ computations in each sub-domain, we can then exploit the variance in the gravitational direction. We employ an *early-break* strategy. For every evaluation of the interaction criteria at $\|\mathbf{x}_{ij}\| \geq h$, an earlier sufficient condition is $\|x_{N_{dim}}(i) - x_{N_{dim}}(j)\| \geq h$, where we can further narrow the interacting pairs without the full $\|\mathbf{x}_{ij}\|$ computations. Due to time constraints, *parallel-div* is not completed in time, however implementation details are described in subsequent sections.

3.2 I/O and Shared-Memory Paradigms

There are a variety of optimisation strategies that could be applied from n-body problem simulations for interaction computations. However, here, we discuss about I/O and the message passing overhead. To gain temporal information of the simulation, output of data is required at fine time intervals. Writing files in text format is not ideal. We encounter the translation overhead between *floats* and text data, and loses on numerical precision. The *HDF5* format would be a good starting point to address these issues. There is also the aspect of parallel I/O to consider. In terms of parallelisation, *MPI* communication contributes to significant overhead. For a local node, each process shares the same physical memory. We can avoid the message passing overhead of *MPI* and achieve concurrency by utilising the shared-memory paradigm of *OpenMP*. For clusters, hybrid parallelisation would be the ideal way.

4 Performance Benchmark

Branch	No. of Process	Dam-break	Block-drop	Droplet
base	1	495 s	1310 s	378 s
serial	1	284 s	673 s	214 s
parallel	2	275 s	661 s	200 s
	4	177 s	432 s	133 s
	6	149 s	354 s	113 s
parallel-div	2	$-$ s	$-$ s	$-$ s
	4	$-$ s	$-$ s	$-$ s
	6	$-$ s	$-$ s	$-$ s
No. of Particles		400	651	340

Table 1: Computational time, in seconds, of serial and parallel implementations with number of processes listed. Simulations are carried out with $T = 5.0$, $\Delta t = 0.0001$, and $h = 0.01$, and without complier optimisations applied. Energy and particle states are written to files at intervals of 0.1 simulation time. The simulation was ran on a system with an Intel Core i7-10750H with 16 GB of RAM.

Taking away the disk write latency and the communication bottleneck from I/O and then applying the $-O3$ compiler optimisation, we perform another set of performance benchmark in Table 2.

Branch	No. of Process	Block-drop	Droplet
base	1	106 s	32 s
serial	1	35 s	11 s
parallel	2	35 s	11 s
	4	25 s	7 s
	6	22 s	7 s
parallel-div	2	$-$ s	$-$ s
	4	$-$ s	$-$ s
	6	$-$ s	$-$ s
No. of Particles		651	340

Table 2: Computational time, in seconds, of serial and parallel implementations with number of processes listed. Simulations are carried out with $T = 5.0$, $\Delta t = 0.0001$, and $h = 0.01$, and $-O3$ complier optimisation applied. No I/O is done during time stepping. The simulation was ran on a system with an Intel Core i7-10750H with 16 GB of RAM.

5 *parallel-div* Implementation

Again, we start off by branching off *parallel*, as it has all the necessary communication functionality (*MPI_Manager* communication member functions) and attributes (particle global IDs, distinction between global and local arrays) to start our implementation. Considering time constraints, the scope of this implementation is limited to the $2D$ simulation box, and hence a sub-domain division in only the x-direction. However, we would implement a Cartesian grid topology, creating a nice interface for identifying process rank of neighbouring sub-domains for point-to-point communications.

5.1 Point-to-Point Communication

Take k as the number of processes in the simulation. After any arbitrary state update, to split up the domains, each process require a global view of particle's $\mathbf{x}$ state. With it, we tabulate and sort the x-position of particles and split them into intervals of n/k in a load balanced manner. The particle position at the interval point would be used to update all sub-domain boundaries. Then each processes would to work out the communications required to hop particles across neighbouring sub-domains, and facilitate the communication of only its $\mathbf{v}$ states. (The dynamics of a state update is uniquely determined by only $\mathbf{x}$ and $\mathbf{v}$ of the particle, since each process have the global $\mathbf{x}$ states required for the global sub-domain evaluation, we only require message passing for the $\mathbf{v}$ of each particle crossing a

sub-domain.) Then, for each sub-domain, we figure out the associated ghost particles, and perform another set of communication. To avoid deadlock, for every point-to-point communication phase, we have the even-numbered grid perform a send in the $+ve$ direction, while the odd-numbered grid a *Recv* in the $-ve$ grid direction. The opposite is done next, which completes a communication phase.

5.2 Sets of State Arrays

Since particles are now free to move across processes, *vector* would be a natural container for the state arrays in each process. The maximum number of particles per process is uncertain. It is non-obvious what kind of theoretical guarantees we are working with. As a crude arbitrary safety factor, we allocate vector capacity based on a maximum of $1.3\,n/k$ number of particles per process. For each local process, we have a set of state arrays corresponding the sub-domain particles, another set holding the $\mathbf{x}$, $\mathbf{v}$, ρ and p states of the ghost particles.

On the global level, at every time step, all processes will have an *Allgatherv* communication update of $\mathbf{x}$. It will be used for the sub-domain boundary update. Only the root process holds a set of global state arrays, $\mathbf{x}$, $\mathbf{v}$, etc, representing all particles, sorted by their IDs. Communication for this global dataset is only updated as required by I/O, reducing message passing latency.