Planetary Impact Simulation

Alex Kashi[1], Henry Jin[1], Florian Juengermann[1], and Johannes Portik[1]

[1]Institute of Applied Computational Science (IACS), Harvard University

May 14, 2022

Abstract

In this study, we aim to simulate impacts of gaseous planets using Smoothed Particle Hydrodynamics (SPH). We began the simulation using brute force approaches that scale in $O(n^2)$ in time complexity. To optimize performance, a variety of improvements were implemented. Firstly, the Barnes Hut algorithm was used to accelerate the algorithm of calculating gravitational forces between particles. The mesh-free smoothed particle hydrodynamics technique was also used to calculate local pressure forces on each particle. The gravitational and pressure forces experienced by each particle were used to integrate the state of the system. Further improvements we implemented include OpenMP for thread-level parallelization of several for loops, ranging from force computation, to velocity integration. The Eigen package was also used to vectorize operations for three dimensional computations. We demonstrate a strong and weak scaling analysis of our code, and we show that to scale our simulation, more compute hours are necessary.

1 Background and Significance

Humans have been looking to the skies for as long as we've existed. Ancient civilisations from the Egyptians to the Roman Empire, from the Aztecs to the early Chinese dynasties, all demonstrated an inspiration by the night sky, and the possibilities of the great beyond. Through the ages, as human technology improved, our understanding of outer space grew more sophisticated and thorough. We are now at a level of unprecedented technological competence and scientific knowledge, enabled primarily by key innovations in computing.

With computing limits growing in adherence to Moore's Law, we are at a stage of computing that permits the simulation of massive, highly complex, and interactive systems. Using such compute-intense modeling techniques allows us to study hypothetical scenarios by constructing systems with conditions of our choosing, and observing how the system evolves.

The field of astronomy today thus greatly benefits from leveraging computational methods. Thanks to high precision computation, we are able to calculate the trajectory of planetary bodies centuries into the future, model the formation of planets from gaseous nebulae, and even predict the weather on other planets[2][6].

In this study, we set out to simulate planetary impact using smoothed particle hydrodynamics and the Barnes Hut algorithm. Understanding planetary collisions is an import sub-field of astronomy that can reveal profound insights about the origins of planets, stars and galaxies. For instance, the Milky Way galaxy is predicted to collide with the Andromeda galaxy in 4 billion years time. By treating bodies as particles, we can simulate the trajectory to understand how this collision will evolve, and how this impacts planet Earth.

There are two forces that must be considered in order to simulate planetary collisions. One is the gravitational force. When modelling gaseous planets as clusters of particles, we must include the gravitational force that every particle exerts on every other particle in the system. With brute force calculation, these forces can be calculated for the system in $\mathcal{O}(n^2)$ time. For large simulations with millions of particles, this can be excruciatingly slow, even with parallelisation techniques. The Barnes Hut Algorithm approximates this force, rather than calculating it precisely as in the brute force approach, and is able to calculate the forces exerted on all the particles in $\mathcal{O}(n \log n)$ time.

The other type of force that needs to be computed is local pressure forces between gas particles. This is the force that accounts for collision. When particles are close together, a collision is modelled via local pressure forces. These forces are computed using smoothed particle hydrodynamics, a technique that applies a kernel around every particle to compute a force relative to its nearest particles in the range of said kernel[1][5].

There is a similar simulation performed by Dr Jacob Kegerreis from Durham University, where direct and glancing collisions of two gaseous planets were simulated on COSMA supercomputer. There was also a lightweight python implementation of smoothed particle hydrodynamics by Philip Mocz, which we used as inspiration and guidance.

2 Scientific Goals and Objectives

Our overarching aim is to simulate the collision of two gaseous planetary bodies. We want to achieve a simulations 1000 time steps with 10,000,000 particles in the system. In order to achieve this, we will need to parallelise our code, since for each timestep, using multiple threads can greatly accelerate both the gravitational force calculations as well as the local gas pressure calculations. The more threads we are able to execute on different cores, the greater the achievable factor of acceleration becomes. We are limited in the number of cores for running this simulation locally, and by using compute hours on an HPC cluster, we can use up to 32 or 64 cores.

3 Algorithms and Code Parallelization

For each iteration step, we do the following steps:

1. Initialize the fixed sized grid and sort particles.
2. Compute the density and pressure values for all particles.
3. Construct the Barnes Hut tree.
4. Compute gravity forces with the Barnes Hut tree.
5. Compute SPH forces.
6. Integrate forces and update position.

We will now outline what the individual steps do.

Barnes Hut Algorithm

The Barnes Hut Algorithm approximates the n-body gravitational force computation by aggregating many bodies together when such aggregations result in sufficiently small approximation error. The algorithm has two phases.

The first phase is the recursive construction of an octree that represents the position of bodies in the space. The tree consists of nodes, where each node represents an octant of its parent node. The root node represents the full space. We start with a root node, and sequentially add bodies starting from the root node. The tree is constructed recursively in the following way. If the node is empty, then place the body in that node. Otherwise, if the node is an internal node, update the nodes center of mass and total mass and place the body in the appropriate child octant. Otherwise, if the node is an external node and has an existing body, then subdivide this node into 8 octants and place the two bodies into the appropriate child nodes. This algorithm takes $\mathcal{O}(n \log n)$ time, since for each body that needs to be placed in the tree, there is on average $\mathcal{O}(\log n)$ steps. Here we assume that particles are reasonable well separated. Even a tree with only 2 particles that are ϵ apart can have $\Omega(\log \frac{1}{\epsilon})$ nodes.

The second phase is the traversal of this tree to calculate the force on a particular body. This occurs in the following way. Suppose we are calculating the force on a particle A. Starting with the root node, if the current node is an external node (and is not particle A), then calculate the force exerted by the current node on particle A and add it to its force experienced. Otherwise, calculate the ratio s/d where s is the width of the region represented by the current node, and d is the distance from particle A to the current nodes center of mass. If this quantity is below a threshold θ, then calculate the force of this node on the particle A and increment its force experienced. Otherwise, call this calculate force function on each of the current nodes children. The $s/d < \theta$ is the approximation criteria that ensures approximations are appropriate. Specifically, this approximation is triggered when the distance from the node to the particle is significantly greater than the width of the region. This step also takes $\mathcal{O}(n \log n)$ time. Together with the tree construction phase, Barnes Hut algorithm has a time complexity of $\mathcal{O}(n \log n)$.

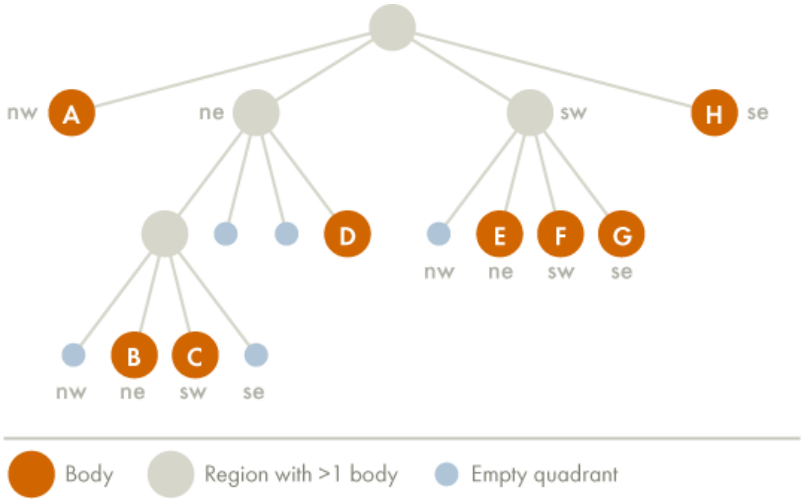


Figure 1: This figure shows what the Barnes Hut tree looks like for a two dimensional system, where each node can be split into four quadrants. In our three dimensional case, every node has eight children, one for each octant subspace. Nodes can be external (holds one body or is a leaf node), or internal (has children nodes).

Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics is a mesh-free, particle-based algorithm that we used for calculating pressure forces between local particles, and is responsible for the fluid dynamics of the system. Our implementation involves two stages. The first stage iterates through all particles, and establishes local

densities around every particle. This is then used in the following stage, which takes the density and viscosity of the fluid around each particle to calculate the force experienced. The strength of the particle interaction impact strongly reduces with distance. The exact relationship is defined by a kernel function as shown in Figure 2.

We calculate the density ρ_p of particle p at position $\mathbf{r}_p$ as:

$$\rho_p = \sum_{q \in \mathrm{NH}(p)} m c_k \exp\left(-\frac{\|\mathbf{r}_q - \mathbf{r}_p\|^2}{H^2}\right) \tag{1}$$

where H is a constant determining the influence radius, $\mathrm{NH}(p) = \{q \in P | \quad \|\mathbf{r}_q - \mathbf{r}_p\| \leq R\}$ are all particles within radius $R = \alpha H$ of p, m is the mass of a particle with pressure 1 and c_k is a constant based on the Gaussian kernel we chose. For large enough α, this is a valid approximation as particles with distance $> R$ only have an impact of $\mathcal{O}(\exp(-\alpha))$. We also compute each particle's pressure $p_p = R^{gas}/\rho_p^2$ where R^{gas} is a gas constant.

The forces $\mathbf{f}_p$ acting on particle p are computed by

$$\mathbf{f}_p = \sum_{q \in \mathrm{NH}(p)} \frac{\mathbf{r}_q - \mathbf{r}_p}{\|\mathbf{r}_q - \mathbf{r}_p\|} m c_k' \frac{p_p + q_p}{2\rho_q} \exp\left(-\frac{\|\mathbf{r}_q - \mathbf{r}_p\|^2}{H^2}\right) \tag{2}$$

where c_k' is another scaling constant.

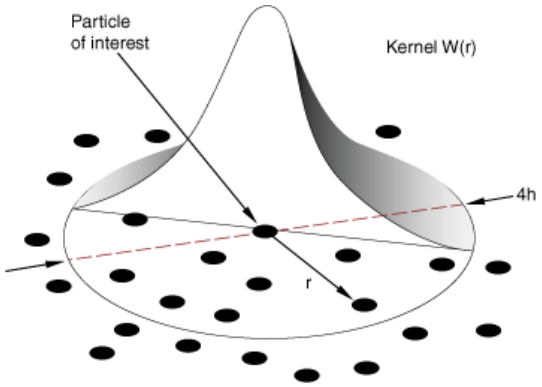


Figure 2: Smoothed Particle Hydrodynamics calculates a particle's local force experienced by adding force contributions from neighbouring particles under a kernel (we use a Gaussian kernel).

Finding Neighboring Particles

To efficiently implement the function $\mathrm{NH}(p)$, one option is to use the existing octree data structure provided by the Barnes Hut tree. However, we found traversing the tree to be unnecessarily slow and implemented a fixed-sized grid instead. Figure 3 shows how to find neighboring particles with a grid with resolution $R/2$. We first find the grid cell that the particle is contained in, then iterate over all neighbor cell that are within R distance of that cell. In this example, in 3d, this leads to $C = 5^3$ cell neighbors we need to iterate over for each particle. However, As we can see in the figure, not all particles in the neighboring cells are within R of the original particle. In fact, by calculating the volume of the sphere of radius R and comparing it to the volume of the cube with side length $5R/2$, we find that this approach only has a 27% volume efficiency. By increasing the resolution, we can increase this efficiency. For example for a grid resolution of $R/3$ we get 33% and with $R/4$ we get 37% efficiency. As a further

optimization, we can directly skip cell that are too far away such as the ones in the diagonal corner. This way we can reach an efficiency of 44% with the $R/4$ resolution. However, this resolution comes at a cost. Even when there are no particles around, each particle must visit all neighboring cells. For $R/4$ resolution, this is $C = 9^3 = 729$ cells. In practice we found the resolution $R/2$ to work the best, slightly better than the naive implementation with resolution R and only 16% efficiency.

Memory Layout

For the SPH force calculation (Equation 2), we need to access the particle's position, pressure and density, so we decided to use an array of structs data structure to maximize locality. The way the fixed sized grid is implemented is as follows. The particles are sorted based on their cell index. Then to retrieve all element of a cell, we store a pointer to cell's the start and end position in the global particle array. To further optimize spacial locality, we use a space-filling z-curve for indexing. This way, most particle's neighbors are stored in close-by storage locations.

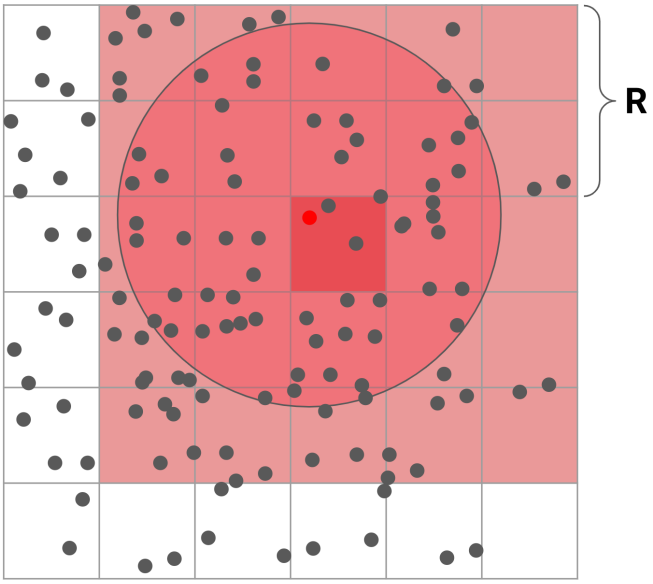


Figure 3: Finding neighbors using a fixed size grid with resolution $R/2$.

Code Parallelization

Our code uses OpenMP for multi-threaded parallel processing, which accelerates our simulation by executing `for` loop iterations with multiple threads in parallel. Our implementation of Barnes Hut and Smoothed Particle Hydrodynamics is readily parallelisable with OpenMP. In many cases, we iterate over all particles and perform an update that does not influence the other iterations. In those cases, the parallelization is easily implementable with the `pragma omp parallel for` directive. This applies to the computation of density and pressure values, computing the gravity and SPH forces, and integrating the forces to update the positions. The initialization of the grid can partly parallelized but as this method runs magnitudes faster than the other methods, this is not our main concern.

The only function left is the construction of the Barnes Hut tree. Here, we use a similar approach as before. Every OMP thread insert a batch of particles into the tree. However, now we have to control write accesses that might cause race conditions. We use a unique lock for each node in the octree and

only allow a single thread to modify the node at a time. Note that it is still possible for other threads to traverse the node as we only update the weights but never the children pointers. The idea is that for large systems, most work will be done in the leafs of the tree where there are few collisions. Additionally, sorting the particles according to the z-curve order helps that different threads work on spatially separated sets of particles. Upon closer inspection, this method has one specific bottleneck many threads need to modify nodes near to top of the tree. The locks then partly serialize the access and slow down the speedup. While this function is not the main bottleneck, future work could use the fixed sized grid to build subtrees independently and parallel and then merge the sequentially at the end.

We also use the Eigen library in C++. Eigen is a high-level C++ library for linear algebra algorithms, from calculating norms to performing matrix multiplication. Since our planetary impact simulation involves working with three-dimensional data of position, velocity and forces, it is easily data-parallelised. The Eigen library has a data class called Vector3f, which holds three elements of floats designed to represent quantities like position, velocity, acceleration and force. Representing these quantities with the Vector3f data type enabled us to perform standard operations such as addition and subtraction on all three dimensions at once. This way, we can repetitive calculations that occur in different directions by simply parallelising them vai data-parallelisation. The relevant functions for us were squaredNorm() and the norm() functions, which calculates the sum of squares of the vector's elements and the square root of the sum of squares respectively.

MPI

We experimented with MPI to determine the possible speedup by using multiple machines. We identified that our for larger initializations, $N > 10^6$ SPH became the limiting factor as the number of neighbors for each particle grows to 1000. In this case, our runtime is significantly longer than linear so a linear copy overhead becomes almost negligible. We implemented a simple MPI scheme that distributes a copy of all the data to each node, where each node performs an update on their chunk of data. After the update is complete, we gather all the particle updates back to the root node ass seen in Figure 4. With this approach we do not need need to worry about communicating between ranks, after the data is copied to each node. We saw near perfect scaling for updating forces, however our new bottle neck became the actual construction of the Barnes-hut tree, which needs to work on the full problem size.

Validation and Verification

We can verify the Barnes Hut algorithm implementation by comparing it with the brute force calculation which takes $\mathcal{O}(n^2)$. While this verification is infeasible for large simulations (due to the quadratic time complexity), we can use this to verify small systems to ensure our Barnes Hut algorithm is reliable. Indeed, this is what we did for small simulations on the order of 10000 particles. Since the Barnes Hut algorithm is an approximation of full pairwise relationships, the Barnes Hut and brute force simulations were not identical, but were similar.

Similarly, we have three different implementation of the SPH updates: a naive $\mathcal{O}(n^2)$ implementation, an implementation using the Barnes Hut tree, and a linear time implementation using the fixed size grid. Across the three implementations, we see consistent behavior, giving us confidence in the correctness of the implementation[1][3][4][5].

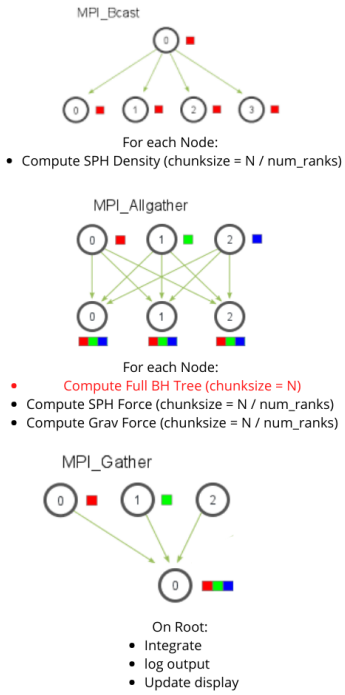


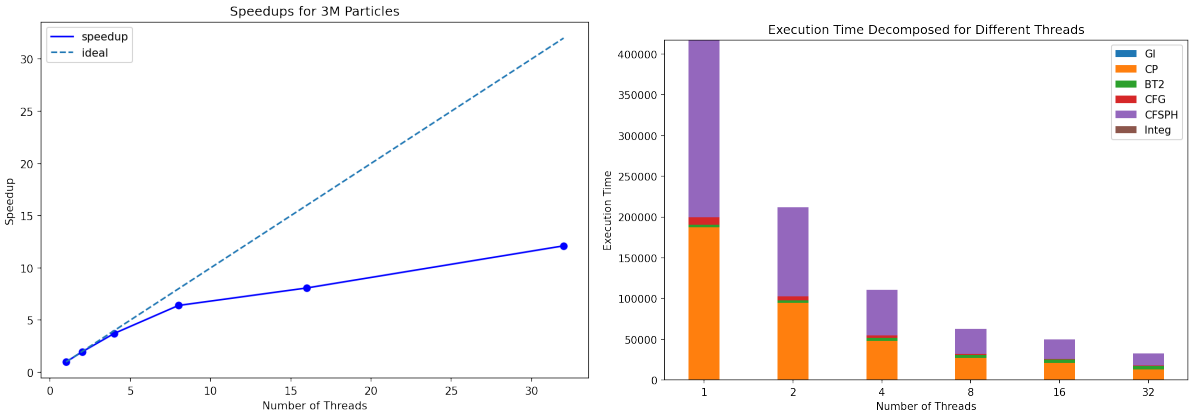
Figure 4: Main computational loop with MPI enabled

4 Performance Benchmarks and Scaling Analysis

Strong Scaling Analysis

Here, we perform a strong scaling analysis. In the strong scaling analysis approach, we assume a constant work load, and a varying number of processors. For the simulation time measurements, we measured the wall clock time for 1, 2, 4, 8, 16, 32 and 64 threads. All benchmarks were conducted on the Harvard FAS cluster running Intel Broadwell 32-core CPUs. We choose a problem size of 3,000,000 particles. In this setting, we approximated the gravity forces with an average of 234 terms per particle. For the SPH calculation, we used 1850 terms on average, however, we visited 6850 neighbor particles for each update due to our over-approximation described in section 3. In runtime with a single processor is 419s which we could reduce to 35s with 32 cores yielding a speedup of 12x. In Figure 5a we show the speedup for different numbers of threads.

In order to better understand the time that each segment of our code takes up, we timed specific sections of our code. Figure 5b shows this breakdown. We can see that, when using a single thread, the largest code segments are the SPH force calculation (CFSPH), the SPH pressure calculation (CP), and the gravity force calculation (CFG). The integration and grid initialisation (GI) steps are negligible across all threads. Measuring the times for these parallelisable segments separately, we find that the SPH speedups are approximately 15x from 1 thread to 32 threads. There is the tree building segment however, does not see much improvement with more threads. This is because of the lock bottleneck discussed in section 3. As the method only takes negligible amount of time, this is however not a big problem.

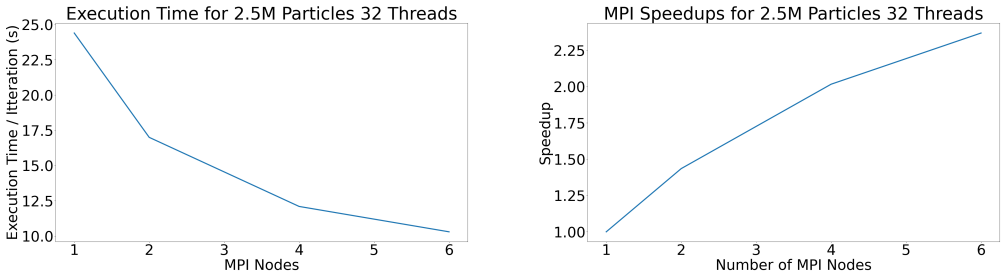


(a) (b)

Figure 5: Strong scaling for 3,000,000 particles. (a) Speedup, (b) Breakdown of the execution time in ms. GI is Grid Initialization. CP is Compute Pressure. BT2 is Building Tree. CFG is Compute Force Gravity. CFSPH is Compute Force SPH. Integ is the Integration step.

MPI Strong Scaling analysis

To perform a strong scaling analysis of our MPI algorithm we allocated a six Broadwell nodes with 32 cores each, and calculated the time per iteration of for 2.5M particles. We observe that even though we are sharing all of the data between nodes, we achieve a 50% scaling efficiency at four MPI nodes, and near 50% scaling efficiency at six nodes 6a. and 6b. At this point build tree replaces SPH as the new bottleneck, future work would be dedicated to constructing the Barnes-hut tree hierarchically for one or two levels using MPI.

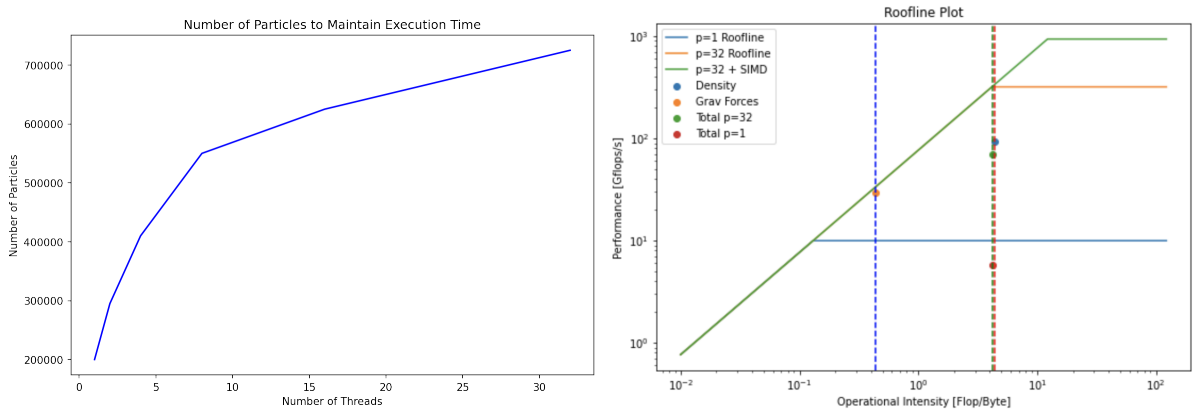


(a) (b)

Figure 6: (a) Total execution time per iteration of a simulation with 10 iterations and 2,500,000 particles for various numbers of MPI nodes. (b) Speedup achieved for a simulation with 10 iterations and 2,500,000 particles for various numbers of MPI nodes.

Weak Scaling Analysis

In this section, we discuss the weak scaling analysis approach. Weak scaling assumes we hold the processing time constant (3165ms in our case), while scaling the job load. Our approach for this task was to start with a 200,000 particle simulation on a single thread, and then adjust the simulation size to maintain the same execution time. The result is displayed in Figure 7a.



(a) (b)

Figure 7: (a)Weak scaling analysis. This diagram shows the number of particles required in the simulation to maintain the same execution time as a simulation with 200,000 particles run with a single thread, which took 3165 milliseconds.(b) Roofline analysis plot.

Roofline Analysis

Here in Figure 7b we have our Roofline analysis. We have plotted three performance peak boundaries in the compute region as depicted by three horizontal lines, all for the Broadwell architecture. The first horizontal line is at 10 Gflops/s and it represents the compute limits of a single thread execution. The next plateau is orange, and is at about 320 Gflops/s. This performance peak is for the case of 32 threads. Lastly, at 940 Gflops/s, there is the peak performance for 32 threads with SIMD vectorization as well.

For the case of a single thread, we can see that we are well in the compute bound region, and are 63% of the peak performance attainable. Once increasing to 32 threads with vectorised computations, we are able to reach a much higher of performance on the order of 90 Gflops/s at an operational intensity of 4.2 flops/byte.

Notably, we almost reach the memory bound for the gravity force computation. This is however, because we assume that all particle particle data accesses are non-cached. However, because of our space-filling curve exploits spacial locality, we probably do significantly less memory accesses. This would increase operational intensity and move our point to the right.

Key Figures

A single-threaded simulation of 3 million particles takes approximately 7 minutes per iteration. So for 50 iterations, this takes approximately 5.8 hours. With 32 threads however, this 3 million particle simulation with 50 iterations takes approximately 0.5 hours. In the multi-threaded scheme, we used 1 node, and 32 cores with 32 threads. We require 0 input files, because the state is self-generated, and no external file is necessary to start or run the simulation. For the output, however, we output 1 file. This file stores the trajectory of particles, to allow us to render the simulation. The file size is the number of particles times the number of iterations times four times four bytes (3 space coordinates and density as single precision floats). So the size of the output file for 50 iterations is 2.4 GB. Lastly, the memory per node is given by the size of the simulation. Each particle has 7 single precision floating point attributes. This gives 28 bytes per particle. For 3 million particles, then, this becomes 0.084 Gigabytes.

5 Resource Justification

To reach tens or hundreds of millions of particles, or to generate extended simulation times, we will need multiple cores to achieve as high a speedup as possible. From the analysis in the previous section, with only 3 million particles, simulating 1000 iterations requires approximately 10 hours, even with multi-threaded parallelisation with 32 cores. The generated data for this simulation is 48GB.

Ideally, we would want to simulate 30 million particles for 1000 timesteps resulting in a 480GB output and an estimated runtime of 130 hours.

References

[1] Robert A Gingold and Joseph J Monaghan. *Smoothed particle hydrodynamics: theory and application to non-spherical stars*. Monthly notices of the royal astronomical society, 181(3):375–389, 1977.

[2] J. A. Kegerreis, V. R. Eke, R. J. Massey, and L. F. A. Teodoro. *Atmospheric erosion by giant impacts onto terrestrial planets*. The Astrophysical Journal, 897(2):161, Jul 2020.

[3] L. B. Lucy. *A numerical approach to the testing of the fission hypothesis*. 82:1013–1024, December 1977.

[4] Matthias Mu ller, David Charypar, and Markus Gross. *Particle-based fluid simulation for interactive applications*. In Proceedings of the 2003 ACM SIG- GRAPH/Eurographics symposium on Computer animation, pages 154–159. Citeseer, 2003.

[5] Joe J Monaghan. Smoothed particle hydrodynamics. *Smoothed Particle Hydrodynamics*. Annual review of astron- omy and astrophysics, 30(1):543–574, 1992.

[6] S Ruiz-Bonilla, V R Eke, J A Kegerreis, R J Massey, and L F A Teodoro. *The effect of pre-impact spin on the moon-forming collision*. Monthly Notices of the Royal Astronomical Society, 500(3):2861–2870, Dec 2020.