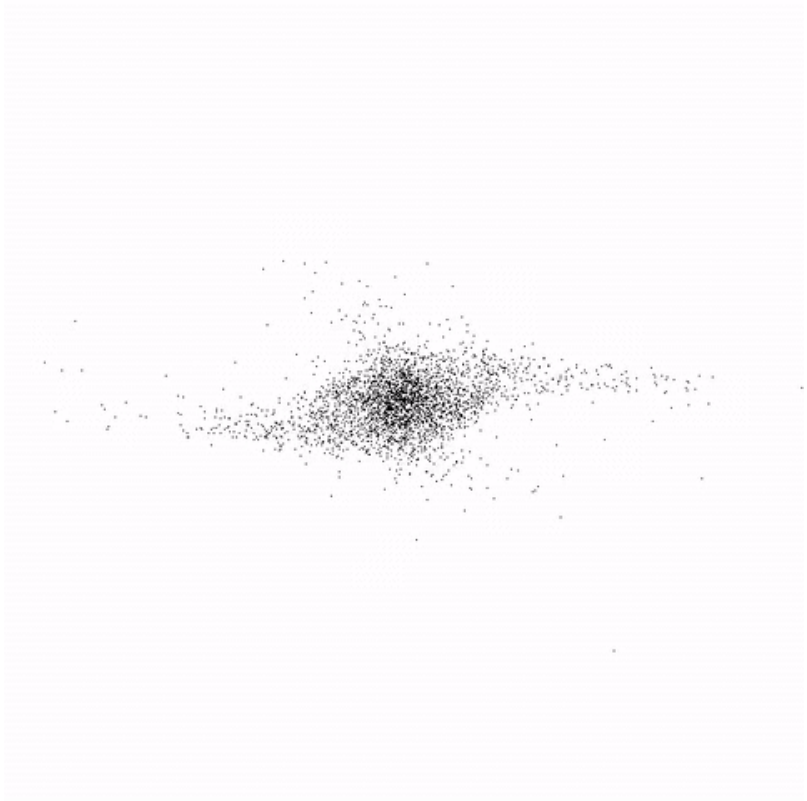
Final Project Report

Parallel Particle-Mesh Methods Applied to the N-body Problem

Michael Whittaker (mjw297)

Sheroze Sheriffdeen (mss385)

December 15, 2015



Contents

Chapter 1

Introduction

This final project explores parallel particle-mesh methods applied to the collision-less N-body problem in a shared memory programming model.

The N-body system is a correlated dynamical system where each element can have a mass, radius, velocity, position and acceleration and each element is under the influence of a physical force, in our case, Newtonian gravity.

For the general N-body problem with $N > 2$, barring selected configurations, the system cannot be solved analytically. To solve a system with $N > 2$, we create numerical models to approximate the evolution of the system with advancing time. Chapter 2 introduces the physical and numerical models that enable us to simulate the behavior of bodies under the influence of gravity and discusses the performance and accuracy tradeoffs of the models.

Chapter 3 documents the implementation process of the simulation. The chapter presents a description and usage of the numerical libraries employed and the data marshaling and animation process.

Chapter 4 evaluates the performance of the system. The chapter analyzes the time profiles of the serial implementation and the parallel implementations, the strong and weak scaling properties of the parallel system and performance scaling with the number of particles and grid points.

Chapter 5 discusses a specific application of the N-body simulator and the results of advancing a system of a large number of particles with time. This chapter introduces the correspondence between the simulation scales and the physical system and contains plots of the system evolution.

In conclusion, Chapter 6 mentions possible future work on the simulation and advanced numerical methods and parallelization schemes to further improve performance.

Chapter 2

Model Theory

To simulate the N-body problem, we have to employ models that enable us to track the attributes of the bodies of the system as we step forward in time. This section describes two such models and comments on their accuracy and performance tradeoffs.

2.1 Particle-Particle Model

The first model of the N-body problem is a particle-particle model. At each time step of the system, the velocity, position and acceleration attributes of an element need to be updated according to the following set of equations.

$$\frac{\vec{v_i}}{dt} = \frac{\vec{F_i}}{m_i} = G\sum_{j\neq i} m_j \frac{\vec{x}_j - \vec{x}_i}{|\vec{x}_j - \vec{x}_i|^3} \tag{2.1}$$

$$\frac{\vec{x}_i}{dt} = \vec{v}_i \tag{2.2}$$

where m_i is the mass of the i^{th} particle and G is the gravitational constant. Given the acceleration vector acting on a particle, we can then use a time integration scheme to solve the above coupled set of equations. For example, we could use the automatic step Runge-Kutta method to update the position and velocity of each particle.

Although, this model provides good accuracy, the attributes modeled for a single element depend on every other elements in the system, leading to an algorithmic complexity of $O(N_p^2)$ where N_p is the number of particles modeled by the system.

Therefore, for large number of particles we have to adopt a different model. Therefore, this project does not use the automatic stepsize Runge-Kutta method but further discussion of the method can be found in Chapter 17.2 of Numerical Recipes [1].

2.2 Particle-Mesh Model

In the particle-mesh approach, we exploit the force-at-a-point formulation and the field equation for the gravitational potential to compute a faster force calculation for the particles at the cost of accuracy. The space of interest is divided into a mesh and each particle is assigned to a mesh point depending on its location. The mesh sampling point defines the center of a cell.

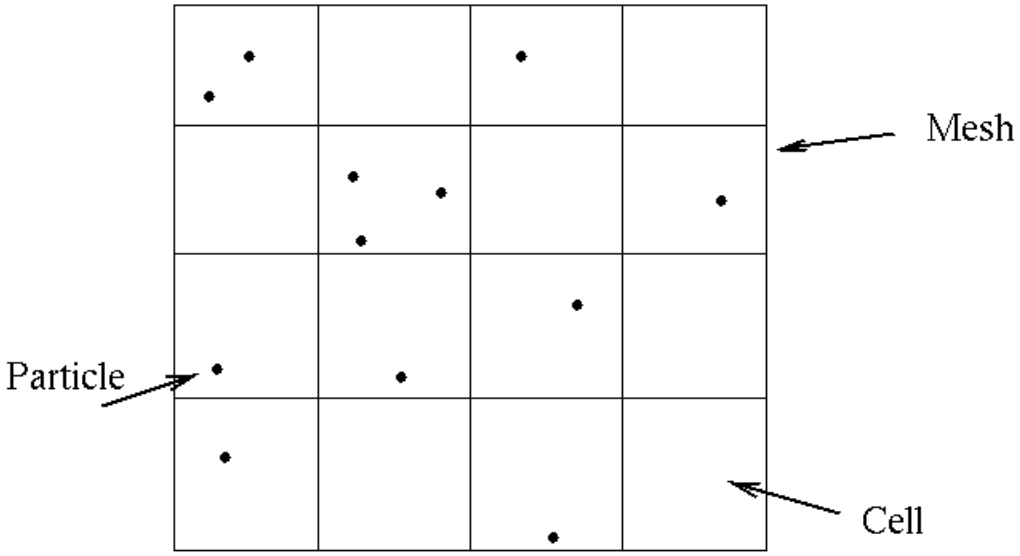


Figure 2.1: Particle-Mesh Model [4]

2.2.1 Physical Interpretation

The gravitational field is defined at the center of a cell. Using the gravitational field at a mesh point, we can interpolate the force acting on a particle. To compute the gravitational field on a particle at a time step, we use Newton's law of universal gravitation:

$$\frac{\vec{F}}{m} = \vec{g} \tag{2.3}$$

The gravitational field $\vec{g}$ on a particle can be related to the gravitational potential ϕ by,

$$\vec{g} = -\nabla\phi \tag{2.4}$$

and the potential $\phi(r)$ is related to the mass distribution by,

$$\phi(r) = \frac{-Gm}{r} \tag{2.5}$$

and obtaining the Poisson's equation for gravity:

$$\nabla^2\phi = 4\pi G\rho \tag{2.6}$$

We can use Fourier analysis to solve the Poisson's equation for gravity. In Fourier space, the equation becomes,

$$k^2\phi(\vec{k}) = 4\pi G\rho(\vec{k}) \tag{2.7}$$

where $\vec{k}$ is the spatial frequency vector.

2.2.2 Mass Density and the NGP scheme

The mass density of the system is sampled at the cell centers of the mesh. To relate the masses of particles to the cell centers, a density assignment scheme is needed. The simplest method is the Nearest Grid Point (NGP) scheme.

$$\rho_{i,j} = \frac{1}{(\Delta d)^2}\sum_{k=1}^{N_p} m_k\,\delta\left(\frac{x_k}{\Delta d} - i\right)\delta\left(\frac{y_k}{\Delta d} - j\right) \tag{2.8}$$

where $\rho_{i,j}$ is the mass density at the cell (i, j), k iterates over the particles and Δd is the sampling interval in space.

Algorithm 1 describes the 2 dimensional NGP scheme on a square mesh where cell_ρ holds the mass density at the cell centers.

Algorithm 1 NGP scheme on square mesh

1: **for all** particles i **do**
2: find (m, n): the index of the cell center in the mesh
3: $\text{cell}_\rho(m, n) \leftarrow \text{cell}_\rho(m, n) + m_i$
4: **end for**
5: **for all** cell centers (m, n) **do**
6: $\text{cell}_\rho(m, n) \leftarrow \text{cell}_\rho(m, n)/(\Delta d)^2$
7: **end for**

The NGP scheme provides a fast method to compute the mass distribution of the system since the core operation is a summation that runs in linear time proportional to the number of particles. The disadvantage of the NGP scheme is that the mass densities are discontinuous.

2.2.3 Gravitational Potential and Acceleration

To solve for the gravitational potential ϕ, we obtained equation 2.7. Using the NGP scheme, we obtain the spatial distribution of density. Converting this density distribution to Fourier space, we obtain the relationship,

$$\phi(\vec{k}) = \frac{4\pi G \cdot \text{FFT}\left[\rho(x,y)\right]}{k^2} \tag{2.9}$$

$$\phi(x,y) = \text{FFT}^{-1}\left[\frac{4\pi G \cdot \text{FFT}\left[\rho(x,y)\right]}{k^2}\right] \tag{2.10}$$

where FFT is the Fast Fourier Transform [5]. The implementation details of FFT is described in Section 3.1.1.

Then, using equation 2.4, we can solve for the gravitational acceleration at a cell center using the central difference method:

$$g_x(i,j) = -\frac{\phi(i+1,j) - \phi(i-1,j)}{2(\Delta d)} \tag{2.11}$$

$$g_y(i,j) = -\frac{\phi(i,j+1) - \phi(i,j-1)}{2(\Delta d)} \tag{2.12}$$

The gravitational acceleration of the cell centers can now be associated with the particles in the simulation using the NGP scheme.

2.2.4 Particle-Mesh Algorithm

To summarize the particle-mesh method, Algorithm 2 describes a single time step [2].

Algorithm 2 Particle-Mesh

1: Assign mass to cell centers using the NGP scheme (Algorithm 1).
2: Solve the Poisson's equation on the mesh in Fourier space by taking the Fourier transform of the mass distribution to obtain $\phi(\vec{k})$.
3: Compute the inverse Fourier transform to obtain $\phi(x,y)$.
4: Use central difference to obtain the gravitational acceleration at cell centers from $\phi(x,y)$.
5: Use the NGP scheme to assign gravitational acceleration values to each particle.
6: Use time integration to update the particle mass and velocity using the gravitational acceleration.

2.2.5 Performance and Accuracy tradeoffs

The Particle-Particle model described in Section 2.1 although accurate requires a high computational cost, taking $O(N_p^2)$ time.

The advantage of the Particle-Mesh method is the performance. Updates to the particle attributes occur in $O(N_p)$ time. The slowest step in the method is the computation of the two dimensional Fourier transform which takes $O(N^2 \log(N^2))$ time.

The increase in performance comes with an accuracy cost. Using the NGP scheme to compute acceleration on a particle decreases the resolution of the solver and to obtain fairly accurate time integration, the spatial sampling interval should be smaller than the wavelengths of importance in the physical system. Finer mass interpolation schemes such as CIC and mixed schemes are discussed in Hockney and Eastwood's Computer simulation using particles [2].

Chapter 3

Implementation

We have implemented a set of N-body simulators using the Particle-Mesh model and algorithm in C++ and a pair of simulation visualizers in Python. In this chapter, we discuss the implementation of both the simulators and visualizers.

3.1 Simulators

In this section, we first discuss FFTW: the fast Fourier transform library we use in our simulators. We then discuss the periodic boundary conditions we use in our algorithm. Finally, we discuss the implementation details of our simulators.

3.1.1 FFTW

To compute the Fourier transform required in the Particle-Mesh method, we use the Fast Fourier Transform, obtained from the FFTW package. (www.fftw.org, the "Fastest Fourier Transform in the West" [6]). In FFTW, the Discrete Fourier Transform of a complex one dimensional array X computes Y where,

$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi jki/n} \tag{3.1}$$

In the solution implementation, to compute the Fourier space representation of the density distribution ρ, we perform a forward transform.

```
1 fftw_plan rho_plan =  fftw_plan_dft_r2c_2d(N, N, rho, rho_k, FFTW_MEASURE);
  fftw_execute(rho_plan);
```

which takes `rho`, that holds $N \times N$ 2D samplings of $\rho(x, y)$ and stores the Fourier space representation in `rho_k`.

The function `fftw_plan_dft_r2c_2d` creates a plan that binds two arrays to perform an FFT operation. `r2c` variants of this function denotes a transform that goes from real numbers to complex numbers.

To compute ϕ, we perform the computation in equation 2.7 and use an inverse transform to obtain $\phi(x, y)$.

```
fftw_plan phi_plan =  fftw_plan_dft_c2r_2d(N, N, rho_k, phi, FFTW_MEASURE);
fftw_execute(phi_plan);
```

Since we perform this FFT operation for multiple time steps, FFTW offers an option to analyze the fastest style of computation for the compute by passing in the `FFTW_MEASURE` flag to the function.

3.1.2 Periodic Boundary Conditions

The Discrete Fourier Transform assumes periodic boundary conditions. Therefore, our 2D gravitational potential is implemented with a toroidal geometry:

$$\phi(x + L, y) = \phi(x, y) \tag{3.2}$$
$$\phi(x, y + L) = \phi(x, y) \tag{3.3}$$

where L is the side length of the simulation in physical units. Therefore, any particle that goes out of the grid is 'wrapped around' and introduced from the opposite end.

```
if (particle_pos[2*i] < 0.0) particle_pos[2*i]     = fmod(particle_pos[2*i], L) + L;
if (particle_pos[2*i] > L) particle_pos[2*i]       = fmod(particle_pos[2*i], L);
if (particle_pos[2*i+1] < 0.0) particle_pos[2*i+1] = fmod(particle_pos[2*i+1], L) + L;
if (particle_pos[2*i+1] > L) particle_pos[2*i+1]   = fmod(particle_pos[2*i+1],L);
```

3.1.3 Serial Simulator

Our serial simulator, found in `serial_opt.cpp`, is a direct implementation of Algorithm 2 which uses FFTW for the fast Fourier transforms. The serial simulator, as well as the other simulators, accepts a set of command line arguments that alter the execution of the simulation.

- The `-l` <L> flag instructs our simulator to simulate on an underlying $L \times L$ space. By default particles are distributed uniformly at random with x and y coordinates between $\frac{2}{5}L$ and $\frac{3}{5}L$.
- The `-n` <N> flag instructs our simulator to use an $N \times N$ mesh grid. The $N \times N$ grid is overlaid uniformly over the $L \times L$ space.
- The `-p` <N_p> flag instructs our simulator to simulate N_p particles.

- The `-s <`s`>` flag instructs our simulator to perform s time steps.
- The `-t <`Δt`>` flag instructs our simulator to use time step Δt.

3.1.4 `fftw_omp` Simulator

The `fftw_omp` simulator is the first of three parallel simulators and is found in `ppm_omp.cpp`. The `fftw_omp` simulator is identical to the serial simulator except for one key difference: the fast Fourier transforms are done in parallel. FFTW provides parallel implementations of its fast Fourier transform functions that use OMP for parallelization, so this process is relatively easy. We simply add the following boilerplate to our serial simulator:

```
fftw_init_threads();                    // initialize FFTW for parallelization
fftw_plan_with_nthreads(num_threads); // set the number of OMP threads to use
// ...                                  // plan and execute FFT's as usual
fftw_cleanup_threads();                 // destroy threads
```

The `fftw_omp` simulator and the other two parallel simulators also accept a `-o` o flag which instructs the simulators to use o OMP threads.

3.1.5 `partial_omp` Simulator

The `partial_omp` simulator, found in `ppm_omp_partial.cpp`, adds additional parallelization on top of the `fftw_omp` simulator. Specifically, it uses OMP to parallelize all loops that range over the particles. For example, consider the following loop found in the `fftw_omp` simulator which implements part of Algorithm 1:

```
int ind_x, ind_y, index, mass;
for (int i=0; i<N_p; i++) {
    ind_x =  floor(particle_pos[2*i]/delta_d);
    ind_y =  floor(particle_pos[2*i+1]/delta_d);
    index = ind_y*N + ind_x;
    mass = particle_mass[i];
    rho[index] += mass;
}
```

The `partial_omp` simulator parallelizes this loop with the `#pragma omp parallel for` construct. It also protects shared memory accesses with appropriate mutual exclusion or atomic updates.

```
int ind_x, ind_y, index, mass;
#pragma omp parallel for                        // <-- new
for (int i=0; i<N_p; i++) {
    ind_x =  floor(particle_pos[2*i]/delta_d);
    ind_y =  floor(particle_pos[2*i+1]/delta_d);
    index = ind_y*N + ind_x;
    mass = particle_mass[i];
    #pragma omp critical                        // <-- new
    rho[index] += mass;
}
```

Note that the `partial_omp` simulator *does not* parallelize all loops, only those that range over particles.

3.1.6 `full_omp` Simulator

The `full_omp` simulator, found in `ppm_omp_full.cpp`, builds off of the `partial_omp` simulator and very aggressively parallelizes *all* loops. For example, consider the following loop which ranges over the $N \times N$ mesh. Since this loop does not range over the particles, the `partial_omp` simulator would not parallelize it, but the `full_omp` simulator does.

```
#pragma omp for collapse(2) nowait
for (int j=0; j<N; j++) {
    for (int i=1; i<N-1; i++) {
        a_x[j*N + i] = (-phi[j*N + i-1] +  phi[j*N + i+1]) / scaling_factor;
    }
}
```

3.2 Visualizers

Our simulators iteratively updates the positions of the particles being simulated. After each iteration, it writes the particles' positions to a text file: a process known as marshalling. In addition to the positions, our simulator also marshals the size of the particle mesh (N), the number of particles (N_p), the number of iterations, and the masses of each particle. The marshalling logic is implemented in `marshaller.cpp`. Also note that our simulator only marshals data when debugging and is elided via conditional compilation otherwise.

The marshalled files are then parsed and animated by one of two visualizers. The first, `scripts/mp4.py`, uses the MoviePy library [7] to construct an MP4 visualization of the simulation. The second, `scripts/gui.py`, uses the TkInter GUI library [8] to animate the simulation in a GUI, without producing any output media files. Both visualizers also accept a set of command line options to modify the behavior of the simulation. For example, the `-f` flag sets the frames per second of the animation, and the `-p` flag sets the resolution of the animation.

Simulations with a large number of time steps or a large number of particles can produce very large marshalled files. To alleviate the burden of storing and transferring large files, the GUI visualizer can read and animate marshalled data from the network rather than from a file. This allows us to animate simulations without ever marshalling data to a file.

Chapter 4

Performance and Evaluation

In this chapter, we discuss how we used profiling to guide our optimization decisions, we discuss the tuning and optimizations we performed, and finally we evaluate the performance of our simulator with a large number of scaling studies.

4.1 Profiling

The first implementation of our particle simulator was naively implemented and single-threaded. By quickly writing a reference implementation of the simulator, we were able to quickly develop and debug our visualizers and check the correctness of our later optimized implementations. We also profiled the compilation and execution of our first implementation to guide our optimization efforts.

First, we examined the vectorization reports produced by `icpc`. The reports showed that almost every one of our loops was being vectorized, not too surprising given the parallel nature of the Particle-Mesh algorithm. Moreover, the vectorization reports indicated that some of our arrays were unaligned. We vectorized the remaining of our loops and replaced invocations of `malloc` and `free` with invocations of `_mm_malloc` and `_mm_free` to align our arrays. After this, our vectorization reports showed fully vectorized loops with minor memory misalignment. A snippet of one of our vectorization reports is shown in Figure 4.1.

Next, we profiled the bottlenecks in the execution of our serial and parallel code using `amplxe-cl`, a snippet of which can be found in Figure 4.2. The profiles show that the majority of the execution of our serial code is spent inside of the FFTW library. This is consistent with the Particle-Mesh algorithm which spends a considerable amount of time performing fast Fourier transforms. Moreover, this is desirable; the bottleneck of our code is in a library that has been *very* thoroughly optimized. This means that we have successfully offloaded the bulk of the computation to a well-tuned library. The profiles report a similar pattern for our parallel code. The majority of the execution time is spent within the FFTW library. One notable difference is that the parallel code spends most of its time performing barrier operations to synchronize code. This suggested to us that increasing the

number of threads would only improve the performance of the code given a large enough grid size. This hypothesis was confirmed by our scaling studies, as discussed later.

```
LOOP BEGIN at common.cpp(57,5) inlined into serial_opt.cpp(78,9)
   remark #15388: vectorization support: reference rho_18337 has aligned access   [ common.cpp(58,9) ]
   remark #15388: vectorization support: reference rho_18337 has aligned access   [ common.cpp(58,9) ]
   remark #15399: vectorization support: unroll factor set to 4
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 35
   remark #15477: vector loop cost: 1.250
   remark #15478: estimated potential speedup: 13.440
   remark #15479: lightweight vector operations: 5
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Figure 4.1: A snippet of a vectorization report

Function	Module	CPU Time	Function	Module	CPU Time
main	serial_opt	6.907s	__kmp_fork_barrier	libiomp5.so	283.260s
n1bv_64	serial_opt	3.855s	n2fv_64	ppm_omp	13.749s
n2fv_64	serial_opt	2.387s	n1bv_128	ppm_omp	12.812s
n1fv_32	serial_opt	1.808s	n1bv_64	ppm_omp	9.395s
hc2cbdftv_16	serial_opt	1.120s	main	ppm_omp	7.493s
__intel_memset	serial_opt	1.100s	hc2cbdftv_8	ppm_omp	5.470s
hc2cfdftv_16	serial_opt	0.960s	n1fv_32	ppm_omp	4.710s
fftw_cpy1d	serial_opt	0.940s	fftw_cpy1d	ppm_omp	4.427s
t2fv_32	serial_opt	0.900s	__kmp_launch_thread	libiomp5.so	3.570s
fftw_cpy2d	serial_opt	0.898s	hc2cfdftv_16	ppm_omp	2.587s
t2bv_16	serial_opt	0.749s	__kmp_join_call	libiomp5.so	2.354s

Figure 4.2: A snippet of an `amplxe-cl` profile for our serial code (left) and parallel code (right)

4.2 Optimizations

One benefit of the Particle-Mesh algorithm is that majority of the complexity and runtime is taken by the fast Fourier transforms. By leveraging the FFTW library to perform these transforms, our algorithm gets a heavily optimized kernel for free! Still, we performed a variety of optimizations to further improve the performance of our simulators.

- We compiled our simulators with the `-O3 -no-prec-div`, `-ipo`, `-restrict`, and `-xCORE-AVX2` `icpc` compiler flags. This provided a quick and easy speedup for all our simulators.
- We download, configure, build, and install the FFTW libraries from scratch on the totient cluster. By default, FFTW is built with `gcc` and does not include some optimizations. We

configured FFTW to build with `icc` and we enabled vectorization during its compilation. This sped up all our simulators significantly.

- As discussed in above, we vectorized every loop in our simulators and aligned memory with calls to `_mm_malloc`.
- The FFTW library plan generation function accepts a flag that is either `FFTW_ESTIMATE` or `FFTW_MEASURE`. When `FFTW_ESTIMATE` is specified, FFTW estimates a reasonable execution plan. If `FFTW_MEASURE` is specified, then FFTW performs a set of experiments to determine which execution plan is likely to be optimal. Since we generate an execution plan once and re-use it for every iteration of the simulation, we make sure to initialize plans with the `FFTW_MEASURE` flag.
- We made sure to avoid certain pitfalls when using OMP in our `partial_omp` and `full_omp` simulators. For example, we use a single `#pragma omp parallel` construct and multiple `#pragma omp for` constructs in our `full_omp` simulator rather than using multiple `#pragma omp parallel for`. We also use `#pragma omp atomic` rather than `#pragma omp critical` when possible. Finally, we use the `nowait` specifier whenever possible to avoid barrier overheads.

4.3 Scaling Studies

In this section, we perform a large number of scaling studies to thoroughly evaluate the performance and scalability of our particle simulator. Figure 4.3, Figure 4.4, Figure 4.5, Figure 4.6, Figure 4.7, Figure 4.8, and Figure 4.9 present the data gathered during our experiments.

4.3.1 Strong Scaling Study

We conducted a strong scaling study to measure the scalability of our implementations. We fixed the number of particles at $N_p = 12800$. Then, for each grid size (N) in $\{128, 256, 512, 1024, 2048, 4096\}$, we executed our particle simulators for 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, and 24 threads. Figure 4.3 charts the average iteration execution time; Figure 4.4 charts the average speedup compared to our serial implementation.

The strong scaling study provides a number of interesting insights. First, for small N, increasing the number of threads never decreases and sometimes increases the execution time of the simulators. This is consistent with our intuition and consistent with the profiles presented in Figure 4.2. When the grid is too small, there is simply not enough work for a large number of threads. As a result, the majority of the execution time is spent synchronizing threads. On the other hand, for large N, increasing the number of threads does significantly improve the performance of the code. Notably, for $N = 2048$ and $N = 4096$, the `fftw_omp` and `partial_omp` simulators execute almost twice as fast as our serial implementation. We hypothesize that for even larger grids, and for fewer particles, the speedup will continue to increase. Of course, these results are not surprising. The FFTW documentation even reports that "if the problem size is too small, you may want to use fewer threads than you have processors" [18].

The scaling study also shows the relative performance of the `fftw_omp`, `full_omp`, and `partial_omp` simulators. For small values of N, the `partial_omp` and `full_omp` simulators perform similarly and both perform worse than the `fftw_omp` simulator. When N increases, the performance of the `partial_omp` simulator approaches that of the `fftw_omp` simulator, and the performance of the `full_omp` simulator decreases. This can again be attributed to thread synchronization overhead. The `full_omp` simulator aggressively parallelizes loops over the grid, and some of these loops have critical sections within them. As the grid size increases, the overhead of synchronizing threads within the loop outweighs the benefits of parallelizing the loop to start with. Moreover, the fact that the `fftw_omp` and `partial_omp` simulator converge in performance suggests that for large grid sizes and a modest number of particles, the majority of the time is spent performing fast Fourier transforms and not on particle processing. This is again consistent with the profiles presented in Figure 4.2 and is confirmed by our particle scaling experiments discussed below.

4.3.2 Weak Scaling Study

4.3.3 Grid Size Scaling

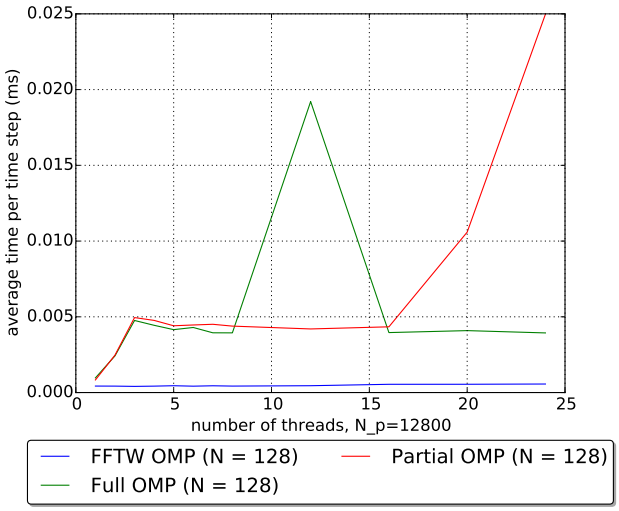
In addition to strong and weak scaling studies, we also measured the performance of our simulators with respect to grid size for a varying number of threads. The data—presented in Figure 4.5, Figure 4.6, Figure 4.7, and Figure 4.8— is similar to the data found in the strong scaling study. Figure 4.5 and Figure 4.6 confirm the $N^2 \log(N^2)$ runtime of the Particle-Mesh algorithm. They also confirm that the `full_omp` simulator performs worse than the other two simulators, no matter the number of threads. The data also makes it clear that the `fftw_omp` and `partial_omp` simulators have very similar performance. Figure 4.7 and Figure 4.8 show an alternate view of the same data. Most importantly, they highlight the fact that the `fftw_omp` and `partial_omp` simulators perform almost twice as well as the serial implementation for large N.

4.3.4 Particle Scaling

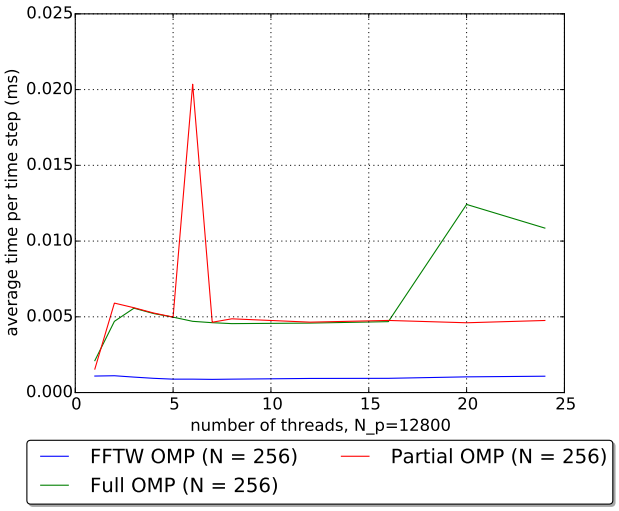
As discussed in Chapter 3, `fftw_omp` invokes a parallelized implementation of FFTW's fast Fourier transform function; the `partial_omp` simulator introduces additional parallelization by parallelizing loops ranging over particles; and the `full_omp` simulator additionally parallelizes all loops. Our previous scaling studies show that `full_omp` is too aggressive and has poor performance because of its heavy synchronization overhead. Our `partial_omp` simulator has less overhead because it only parallelizes loops over the particles. If the number of particles is large, it's possible that the parallelization benefits outweigh the synchronization overhead. We performed a scaling study with respect to the number of particles to explore this hypothesis. We fixed a small grid size of $N = 128$ and benchmarked the performance of our simulators for 1, 2, 3, and 4 threads. The data is presented in Figure 4.9.

The data shows that for all three simulators, increasing the number of threads decreases the performance of the simulators for all number of particles. These results are again confirmed by the profile in Figure 4.2. The bottleneck of our simulators is in FFTW, not in particle processing. This means that if we fix a small grid size of $N = 128$ so that additional threads do not increase the performance of the fast Fourier transform, then adding more threads does not significantly help process

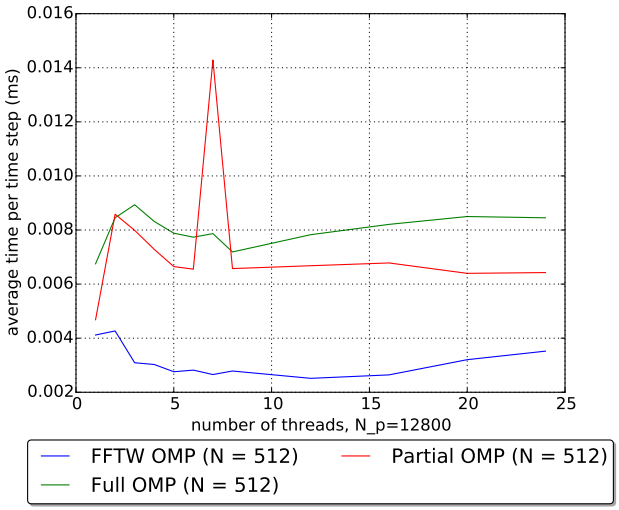
the particles. In other words, adding more threads doesn't help particle processing. It's possible that if we increase the number of particles by one or two orders of magnitude, that multiple threads could begin to be beneficial, but given an enormous amount of particles, it becomes practical to increase the grid size to maintain a decent level of accuracy. When we increase the grid size though, the overhead of FFTW increasingly dominates the cost of particle processing. Thus, it is unclear that parallelization will ever benefit particle processing, no matter the number of particles.



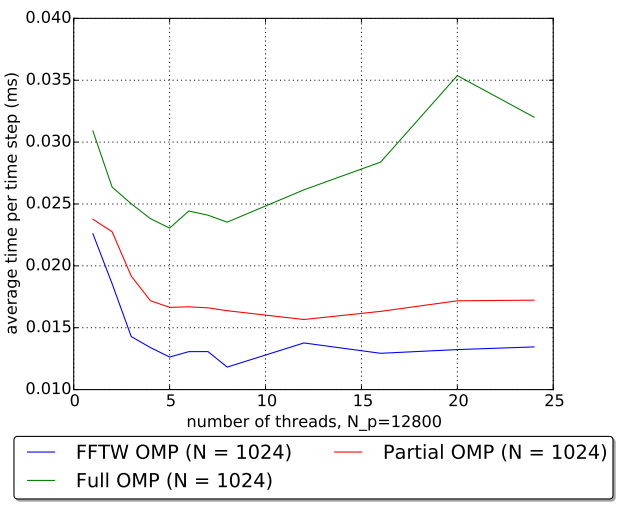
(a)



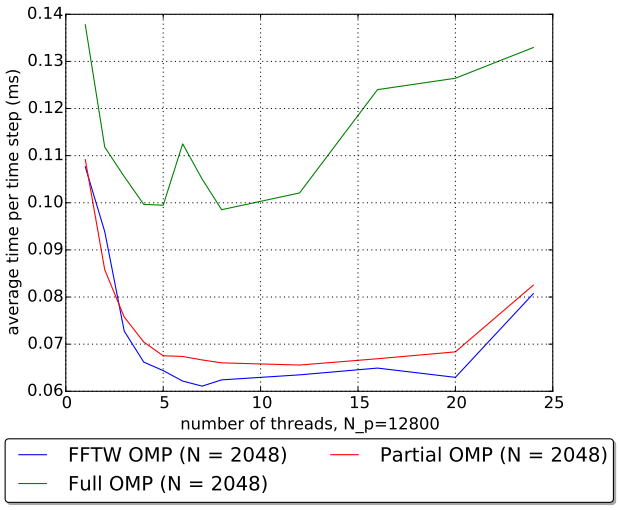
(b)



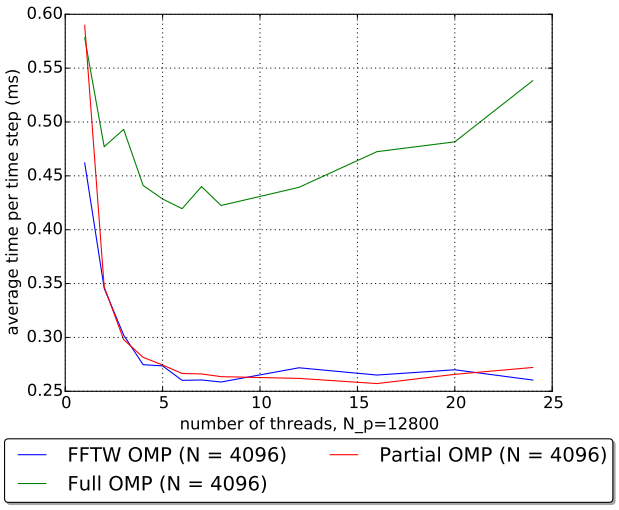
(c)



(d)

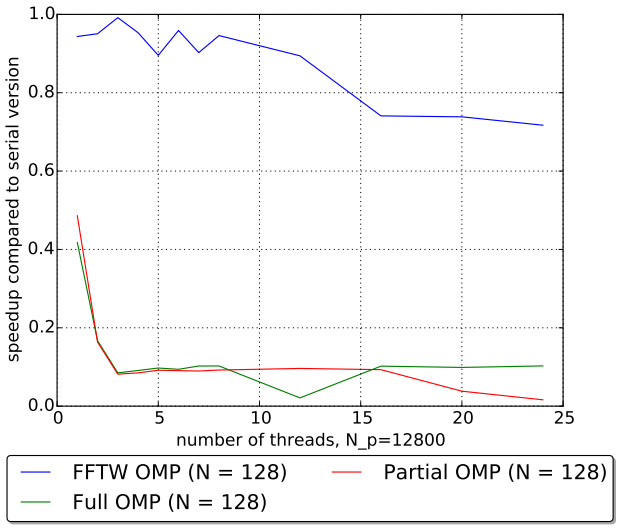


(e)

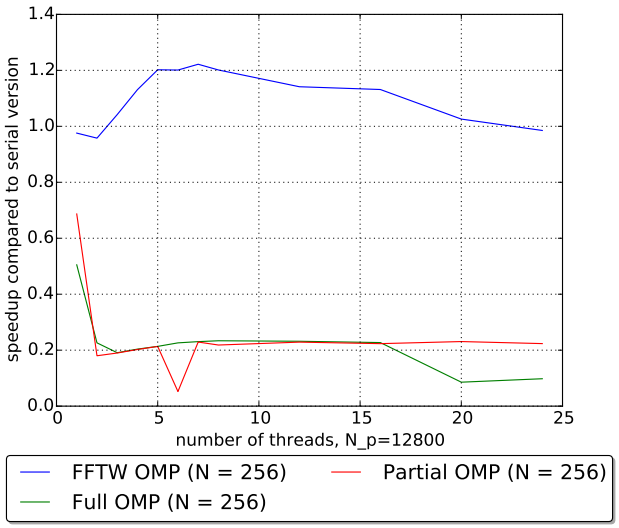


(f)

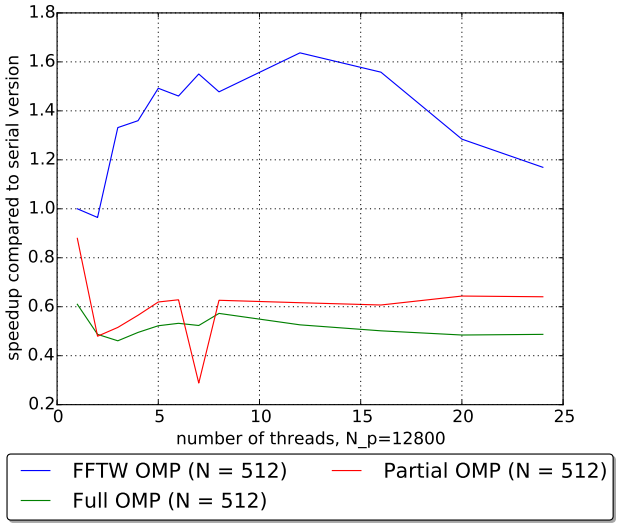
Figure 4.3: Average iteration time vs number of threads



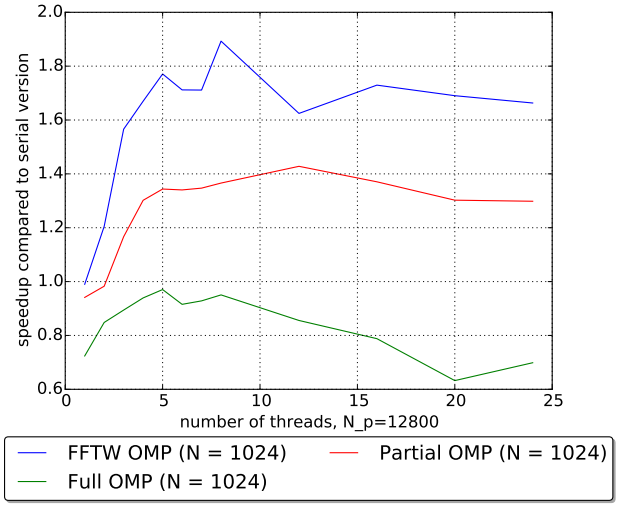
(a)



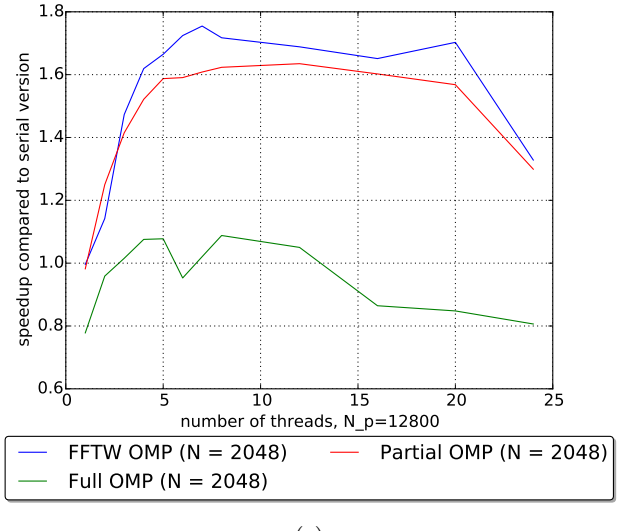
(b)



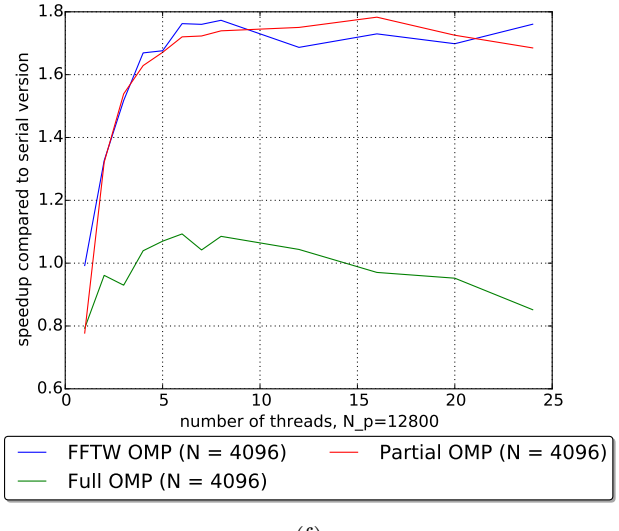
(c)



(d)

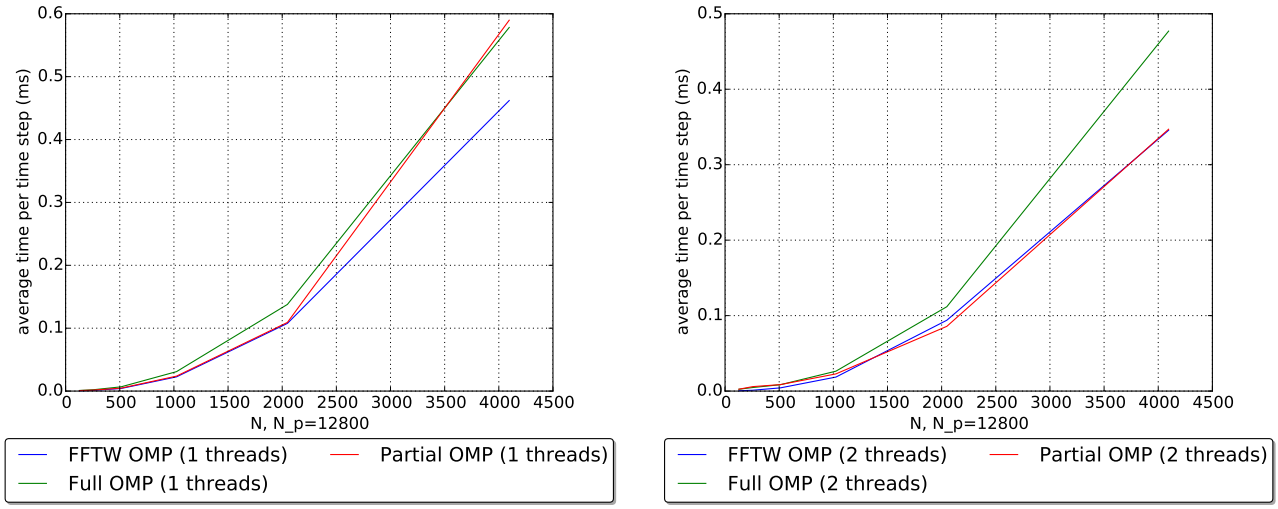


(e)



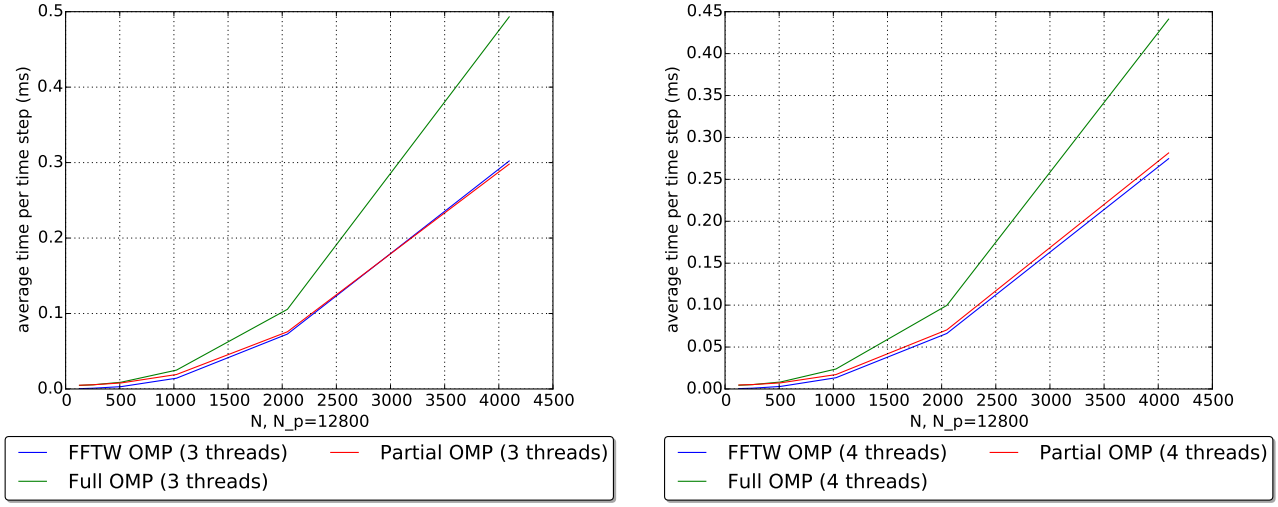
(f)

Figure 4.4: Speedup vs number of threads



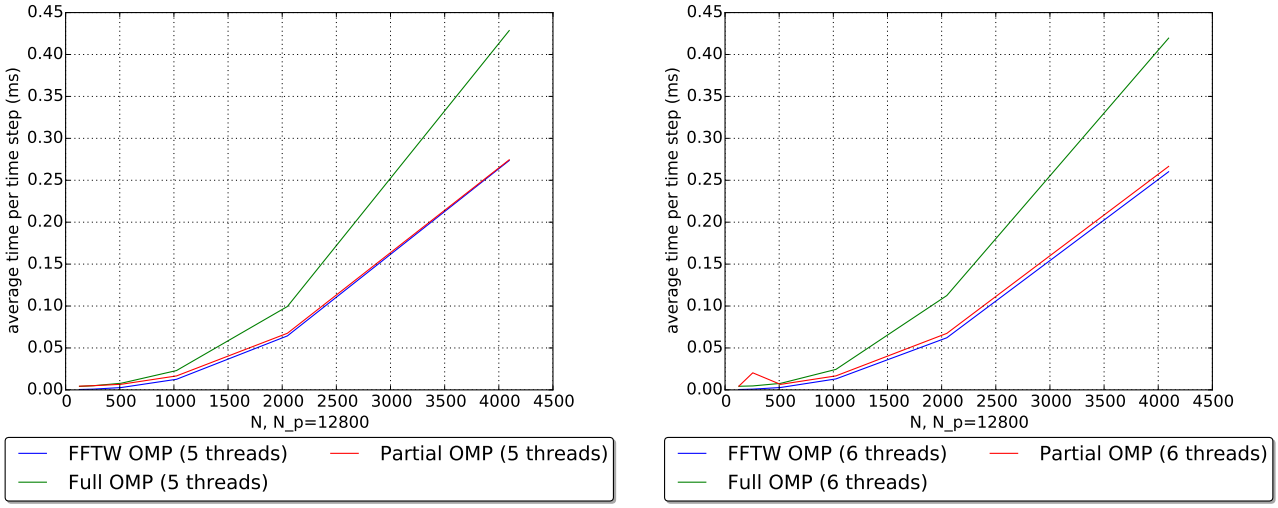
(a)

(b)



(c)

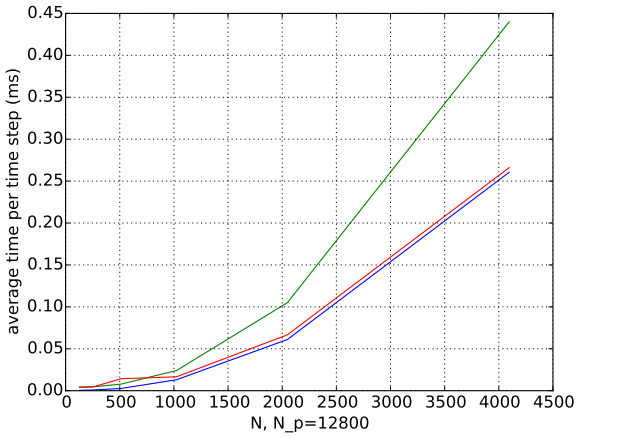
(d)



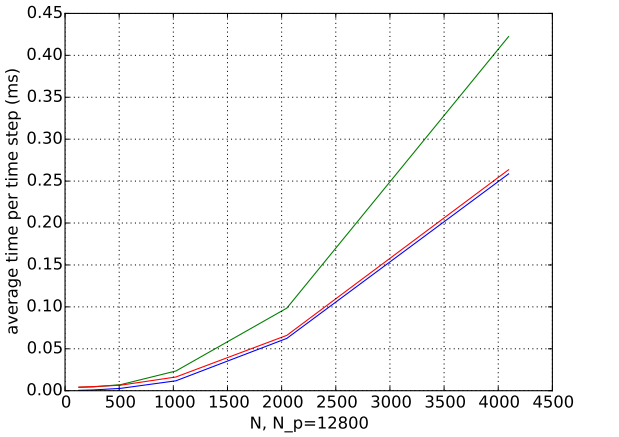
(e)

(f)

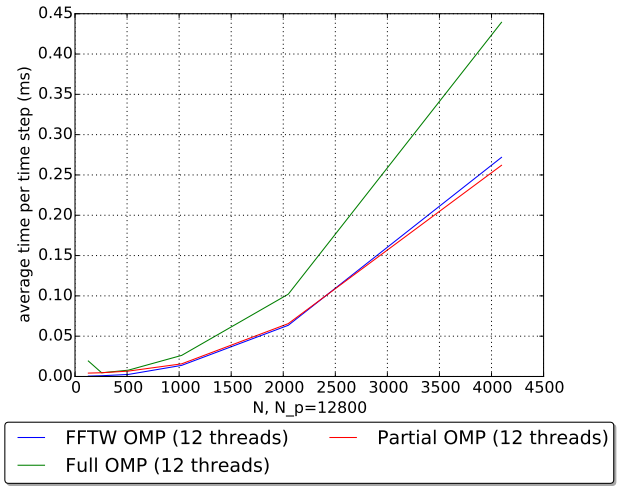
Figure 4.5: Average iteration time vs grid size (N)



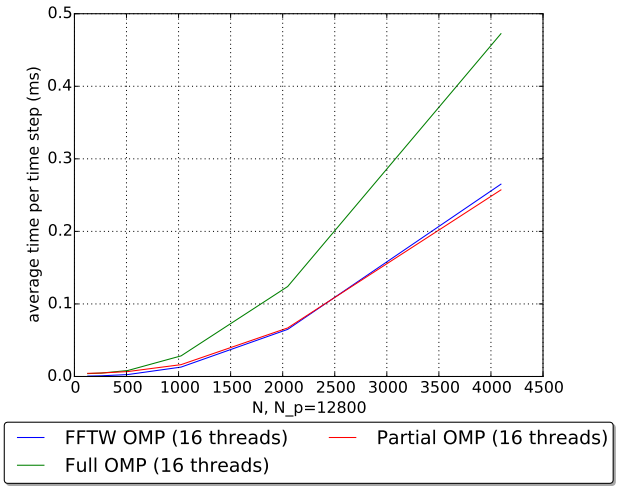
(a)



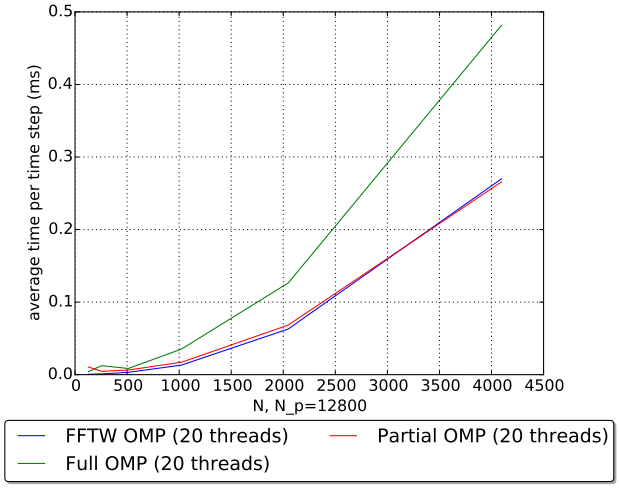
(b)



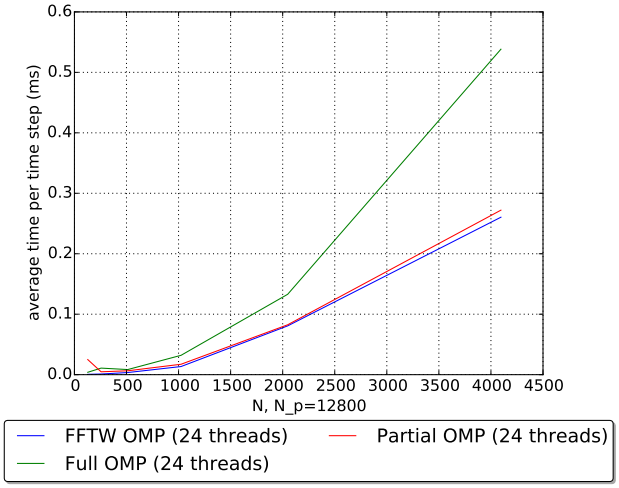
(c)



(d)

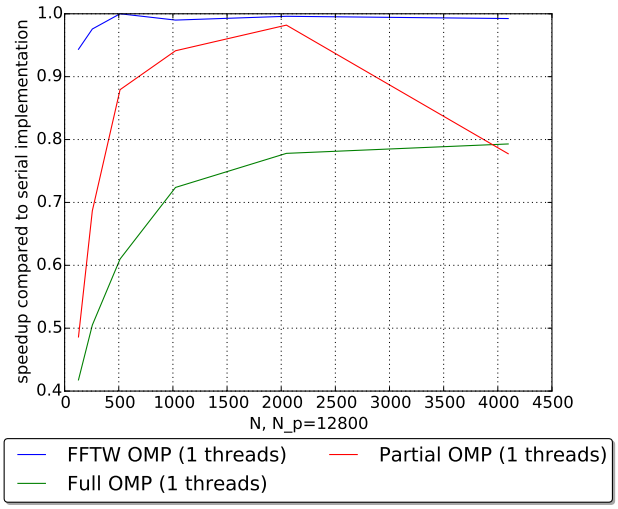


(e)

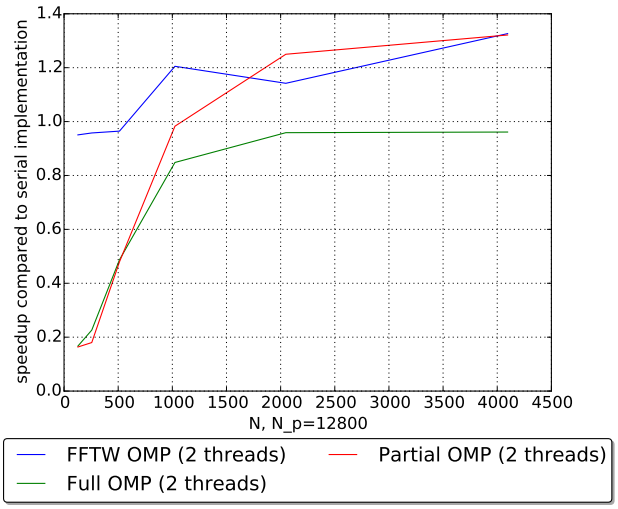


(f)

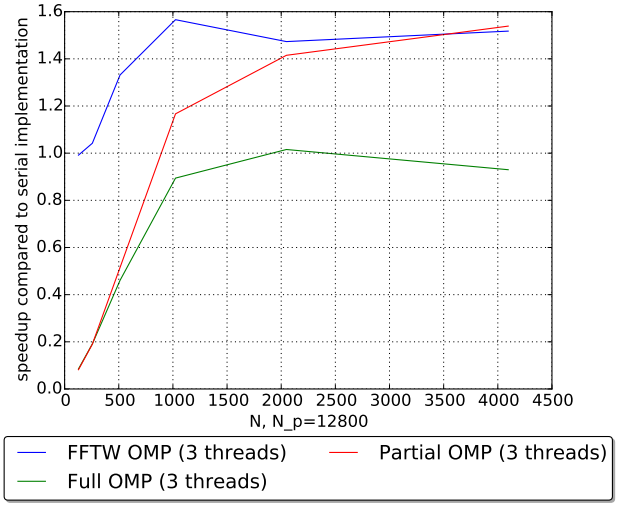
Figure 4.6: Average iteration time vs grid size (N)



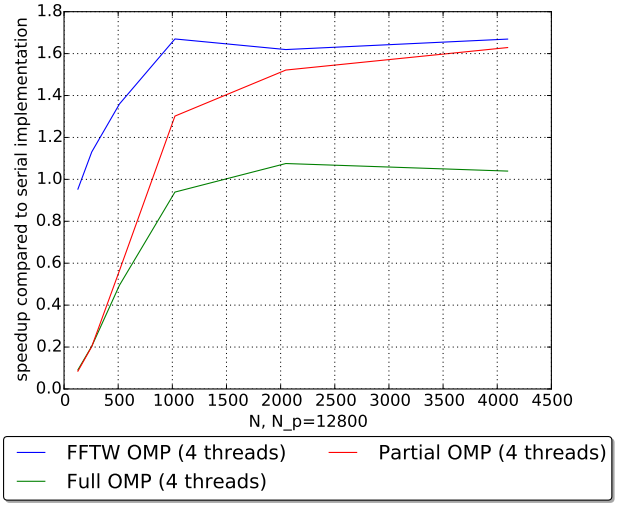
(a)



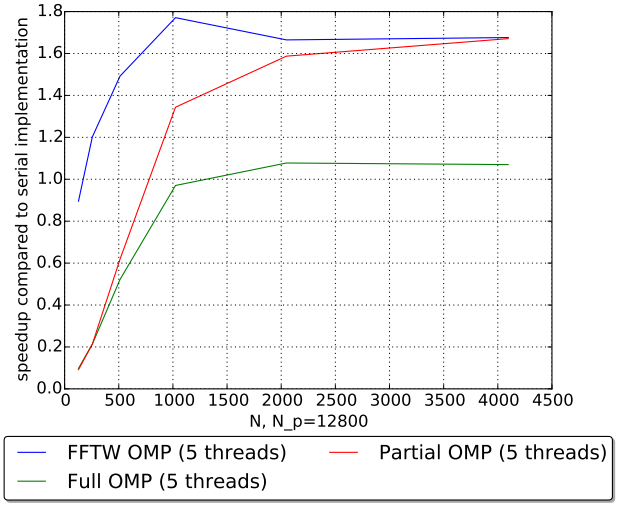
(b)



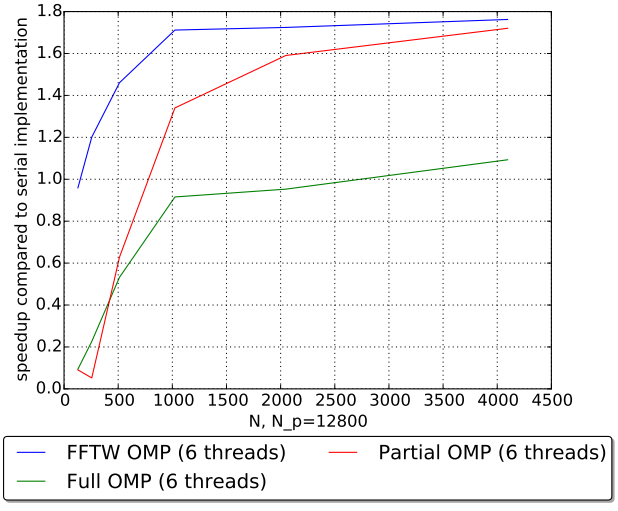
(c)



(d)

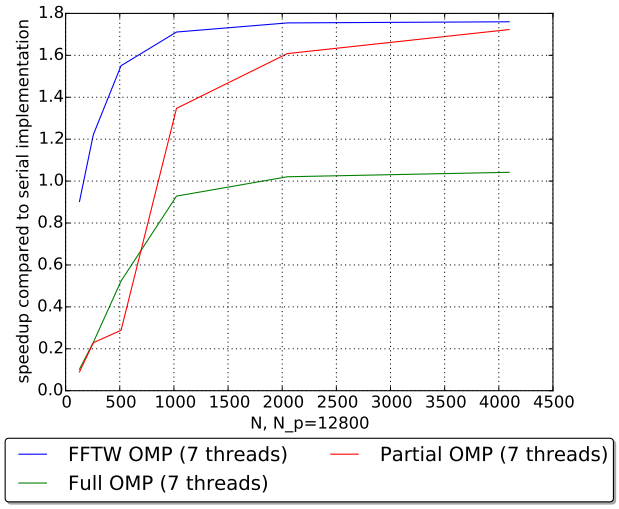


(e)

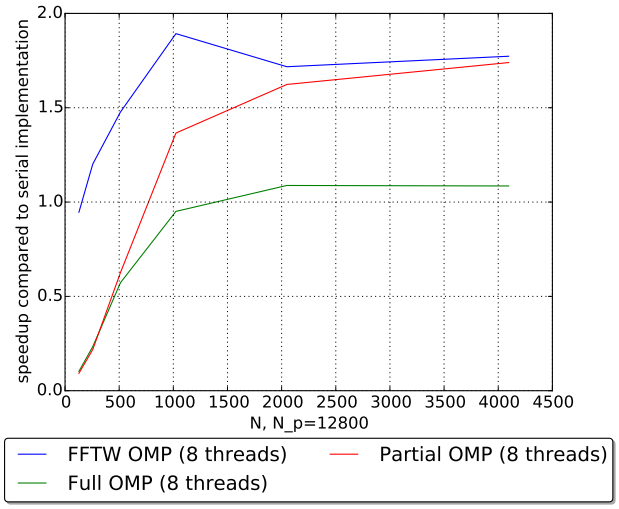


(f)

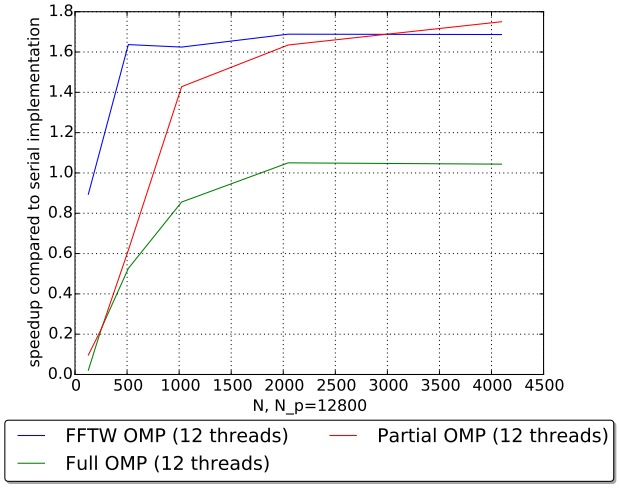
Figure 4.7: Speedup vs grid size (N)



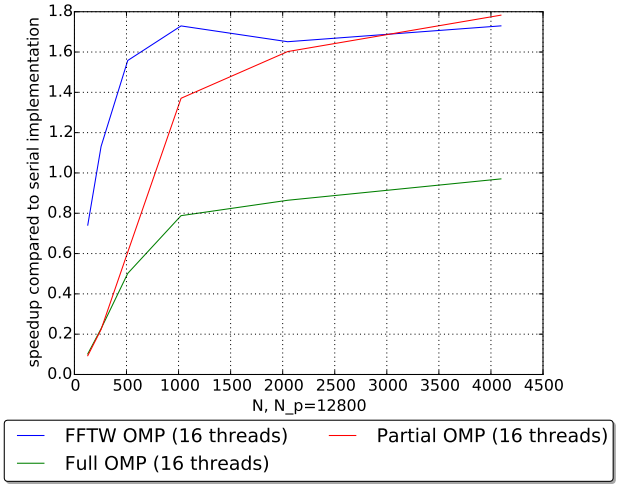
(a)



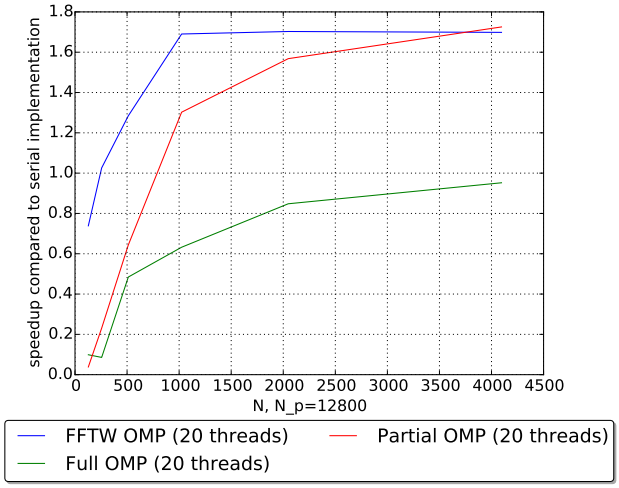
(b)



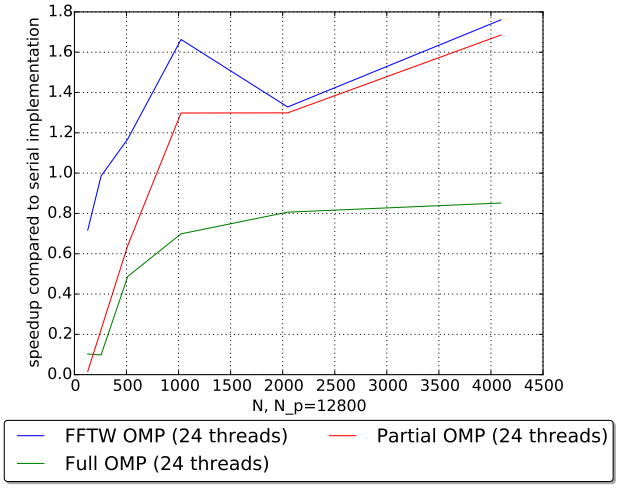
(c)



(d)



(e)



(f)

Figure 4.8: Speedup vs grid size (N)

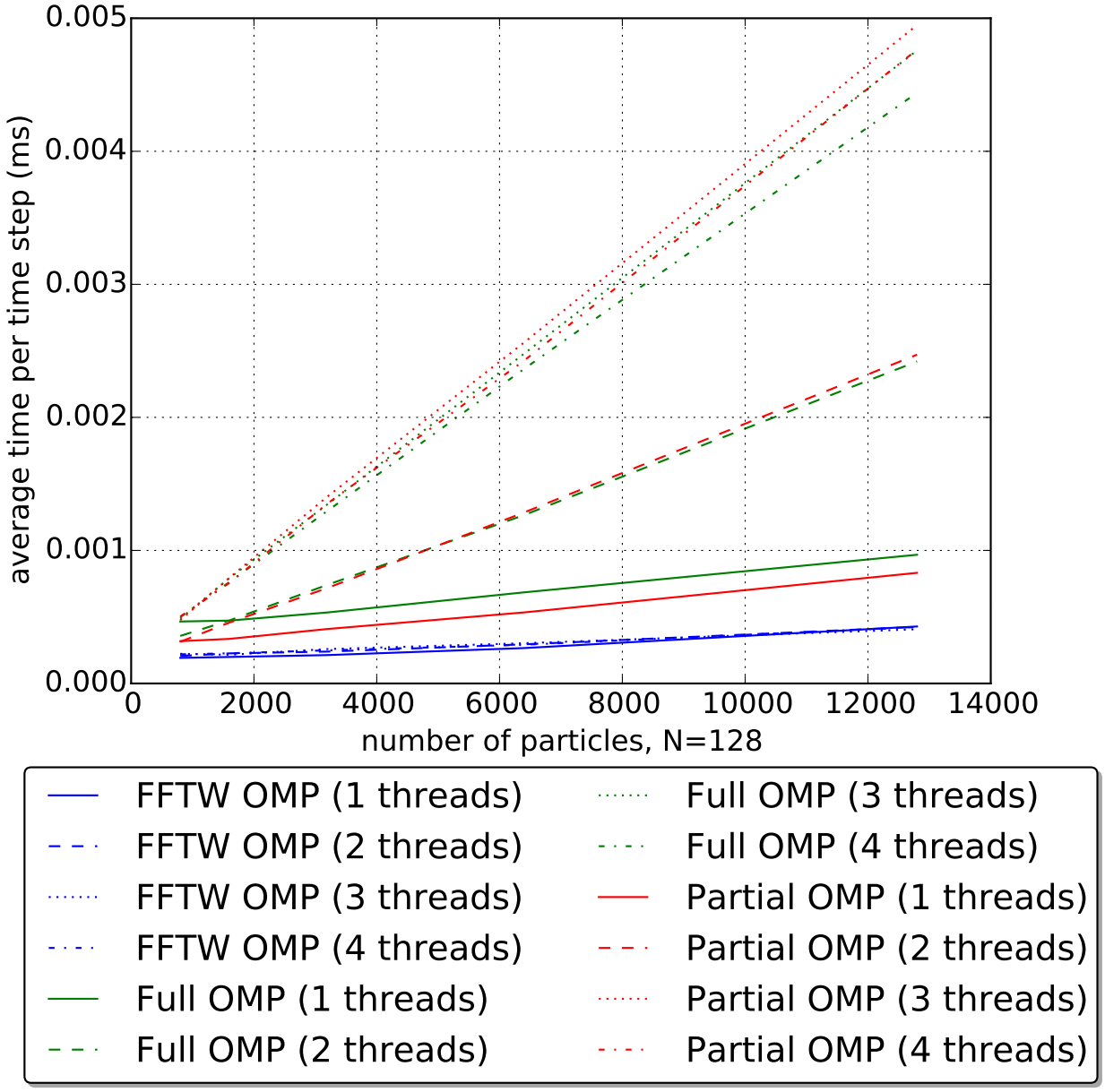


Figure 4.9: Average iteration time vs number of particles (N_p)

Chapter 5

Simulation Results

To see the N-body simulation in action, we model a distribution of bodies with a mean mass of one solar mass around a massive object at the center.

5.1 Conversion Units

To scale the floating point numbers of the simulation, we adopt the following conversion units [9].

Simulation Units	**Physical Units**
Time unit ($\Delta t = 1$)	1 year (3.15×10^7 seconds)
Space unit ($\Delta d = 1$)	1 AU (1.4960×10^{11} meters)
Mass unit ($\Delta m = 1$)	1 solar mass (1.9891×10^{30} kilograms)

5.2 Initial Conditions

5.2.1 Space distribution

The simulation space is an $L \times L$ torus. The positions of the particles are randomly distributed according to a bivariate Gaussian distribution. The positions are sampled from the distributions,

$$p_x = \frac{1}{\sqrt{2\pi}\sigma} \exp - \frac{(x-\mu_x)^2}{2\sigma^2} \tag{5.1}$$

$$p_y = \frac{1}{\sqrt{2\pi}\sigma} \exp - \frac{(y-\mu_y)^2}{2\sigma^2} \tag{5.2}$$

where $\mu_x = \mu_y = \frac{L}{2}$. This sampling produces particles that are centered around $\left(\frac{L}{2}, \frac{L}{2}\right)$ where a massive particle with mass $1000M_\odot$ lies. The bodies form a "bulge" around the center of the configuration.

5.2.2 Mass distribution

Masses of the particles, excepting the central mass, are Poisson distributed with an expected value λ of 1 solar mass.

$$p_m = \frac{\lambda^k}{k!} e^{-\lambda} \qquad \text{where } k \in \{0, 1, 2, 3, \ldots\} \tag{5.3}$$

5.2.3 Velocity distribution

The initial velocities of the particles are assigned assuming a stable circular orbit around the central mass neglecting the surrounding particles.

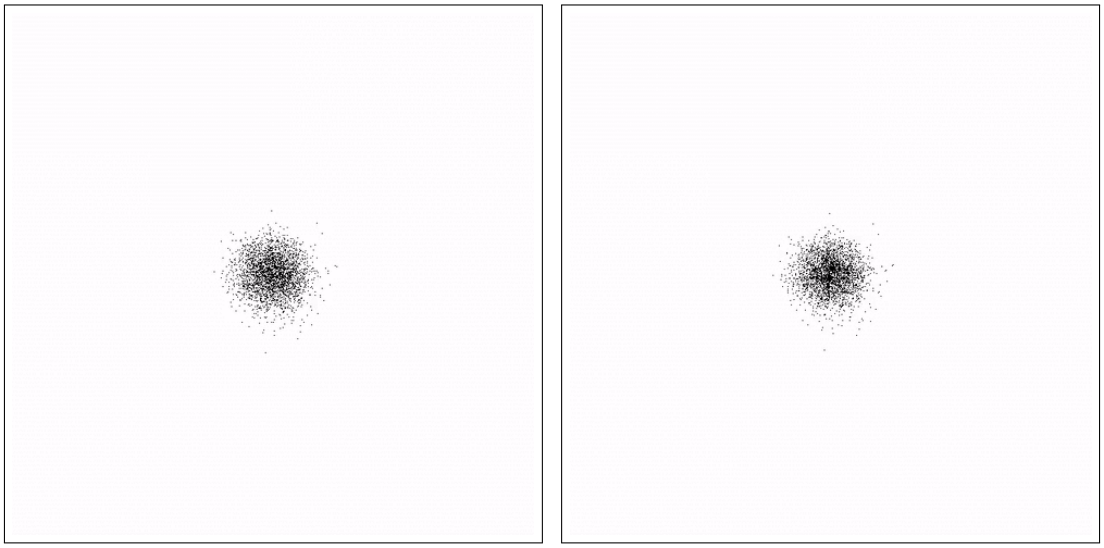
$$v_x = -\sin(\theta)\sqrt{\frac{GM}{r}} \tag{5.4}$$

$$v_y = \cos(\theta)\sqrt{\frac{GM}{r}} \tag{5.5}$$

where M is the mass of the central body, G is the gravitational constant, r is the separation between the central mass and the particle, and θ is the azimuthal angle between the central body and the particle body where the central body is at the origin. This provides the initial rotational component for the galaxy but does not create a stable configuration.

5.2.4 Time Evolution Plots

Given the initial conditions as described above, with simulation side length of 60AU, 3000 particles, and a time step of 0.86 hours, we advance the simulation and plot the resulting particle configuration at specific time intervals. The complete simulation can be found in `particles.gif`.



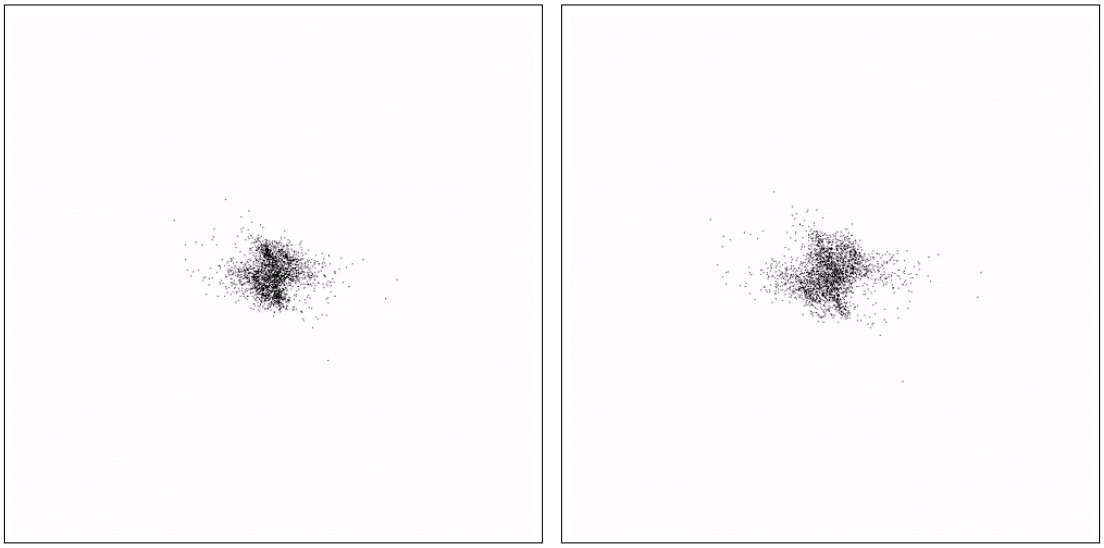
(a) Bodies at $t = 0.0$ days, $L = 60$AU

(b) Bodies at $t = 2.43$ days, $L = 60$AU

(c) Bodies at $t = 4.86$ days, $L = 60$AU

(d) Bodies at $t = 7.3$ days, $L = 60$AU

Figure 5.1: Configuration evolution with $\Delta t = 0.86$ hours, 3000 particles, $\mu_m = 1 \times 10^{30}$ kg



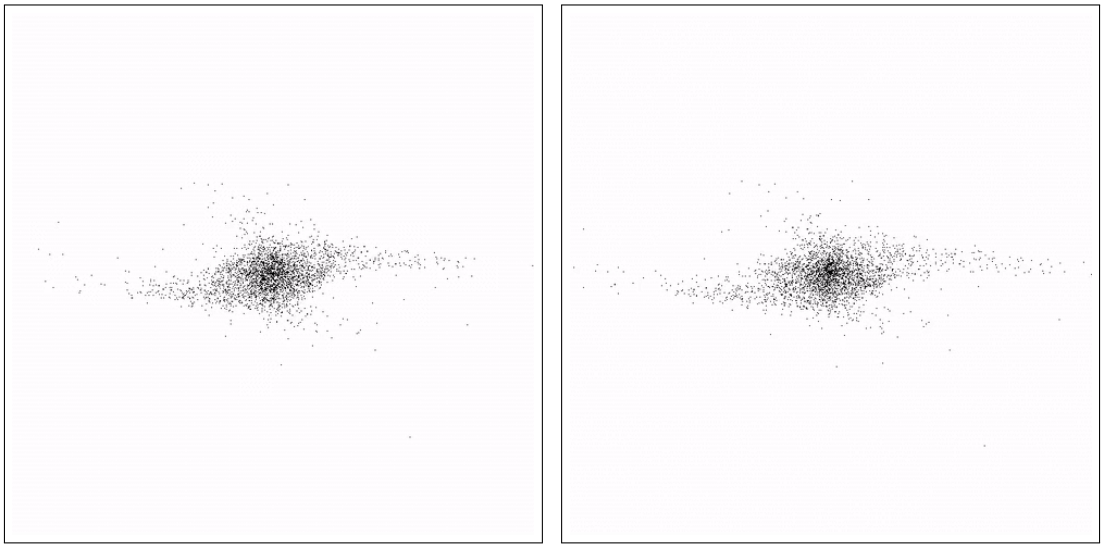
(a) Bodies at $t = 9.73$ days, $L = 60$AU

(b) Bodies at $t = 12.16$ days, $L = 60$AU

(c) Bodies at $t = 14.6$ days, $L = 60$AU

(d) Bodies at $t = 17.03$ days, $L = 60$AU

Figure 5.2: Configuration evolution with $\Delta t = 0.86$ hours, 3000 particles, $\mu_m = 1 \times 10^{30}$ kg



(a) Bodies at $t = 19.46$ days, $L = 60$AU

(b) Bodies at $t = 21.9$ days, $L = 60$AU

(c) Bodies at $t = 24.33$ days, $L = 60$AU

(d) Bodies at $t = 26.76$ days, $L = 60$AU

Figure 5.3: Configuration evolution with $\Delta t = 0.86$ hours, 3000 particles, $\mu_m = 1 \times 10^{30}$ kg

Chapter 6

Conclusion

To improve the time integration process, future work can be undertaken to automatically scale the time step size to prevent instabilities. A maximum velocity can be enforced on the mesh, proportional to the cell widths, to limit the information propagation rate across the cells and the next time step can be dilated accordingly.

A better approximation for mass assignments to cells than the NGP scheme discussed here is the CIC method or the "cloud-in-cell" method described in Chapter 5-2-3 of Hockney and Eastwood's Computer simulation using particles [2]. It is costlier in terms of floating point operations per mass assignment run, but provides a more accurate estimate, giving forces continuous in volume. This method considers the closest 2^N neighbors where N is the dimensionality of the problem, to assign masses.

The biggest shortcoming of the Particle-Mesh method is the low resolution of forces computed when particles are near each other. This arises from the regular discretization scheme and is proportional to the cell width Δd. To overcome this shortcoming, a possible numerical model is the P3M model [5]. The Particle-Particle/Particle-Mesh model refines the mesh method by computing the pair-wise force interaction between particles if the distance between particles are below a certain threshold. P4M method is a parallel variant of the P3M and improves on the performance [11].

To scale the N-body simulation to large values, a smarter scheme is needed that is able to distribute the computation across multiple computers. OpenMP, due to its shared memory programming model, does not provide this ability to scale. MPI does enable us to perform computations in parallel on separate processors that can span multiple machines. FFTW offers an MPI variant of the FFT computation that can be used to explore the distributed computation model [12].

Bibliography

[1] W. H. Press, S. A, Teukolsky, W. T. Vetterling and B. P. Flannery *Numerical Recipes, The Art of Scientific Computing, 3rd Edition*. Camb. Univ. Press 2007.

[2] R. Hockney, J. Eastwood, *Computer simulation using particles* (Special student ed.). Bristol [England: A. Hilger.(1988)

[3] J. Dawson. *Reviews of Modern Physics* 55, p.403 (1983)

[4] M. Elmohamed (n.d.). *N-Body Simulations*. Retrieved December 13, 2015, from http://www.new-npac.org/projects/cdroms/cewes-1999-06-vol2/cps615course/nbody-materials/nbody-simulations.html

[5] F. Vesely (n.d.). *9.2 Particle-Mesh Methods (PM and P3M)*:. Retrieved December 13, 2015, from http://homepage.univie.ac.at/franz.vesely/simsp/dx/node48.html

[6] M. Frigo, S. G. Johnson, *Design and Implementation of FFTW3*, Proc. of the IEEE, 93 (2005), p. 216-231.

[7] *MoviePy User Guide*. (n.d.). Retrieved December 14, 2015, from https://zulko.github.io/moviepy/

[8] *TkInter Documentation*. (n.d.). Retrieved December 14, 2015, from https://wiki.python.org/moin/TkInter

[9] C. Mihos. (n.d.). *Spiral Galaxies*. Retrieved December 14, 2015, from http://burro.case.edu/Academics/Astr222/equations.pdf

[10] *N-Body/Particle Simulation Methods*. (n.d.). Retrieved December 14, 2015, from https://www.cs.cmu.edu/afs/cs/academic/class/15850c-s96/www/nbody.html

[11] P. Brieu, A. Evrard. (2000). *P4M: A parallel version of P3M*. In New Astronomy (3rd ed., Vol. 5, pp. 163-180). Elsevier.

[12] *Distributed-memory FFTW with MPI*. (n.d.). Retrieved December 14, 2015, from `http://fftw.org/doc/Distributed_002dmemory-FFTW-with-MPI.html`

[13] S. Sheriffdeen, M. Whittaker, (n.d.). *Parallel Particle Mesh applied to N-body simulations*. Retrieved December 14, 2015, from https://github.com/sheroze1123/ppm

[14] Intel®Xeon Phi™Coprocessor - the Architecture. (n.d.). Retrieved December 14, 2015, from https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[15] B. Barney (n.d.). *OpenMP Tutorial*. Retrieved December 14, 2015, from https://computing.llnl.gov/tutorials/openMP/

[16] *Solving the collisionless Boltzmann equation using N-body simulations*. (n.d.). Retrieved December 14, 2015, from http://www.mi.infn.it/isapp04/Moore/notes2.pdf

[17] G. Levand (n.d.). *Chapter 2Basics of SIMD Programming*. Retrieved December 14, 2015, from https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html

[18] *How Many Threads to Use?*. (n.d.). Retrieved December 14, 2015, from `http://www.fftw.org/fftw3_doc/How-Many-Threads-to-Use_003f.html#How-Many-Threads-to-Use_003f`