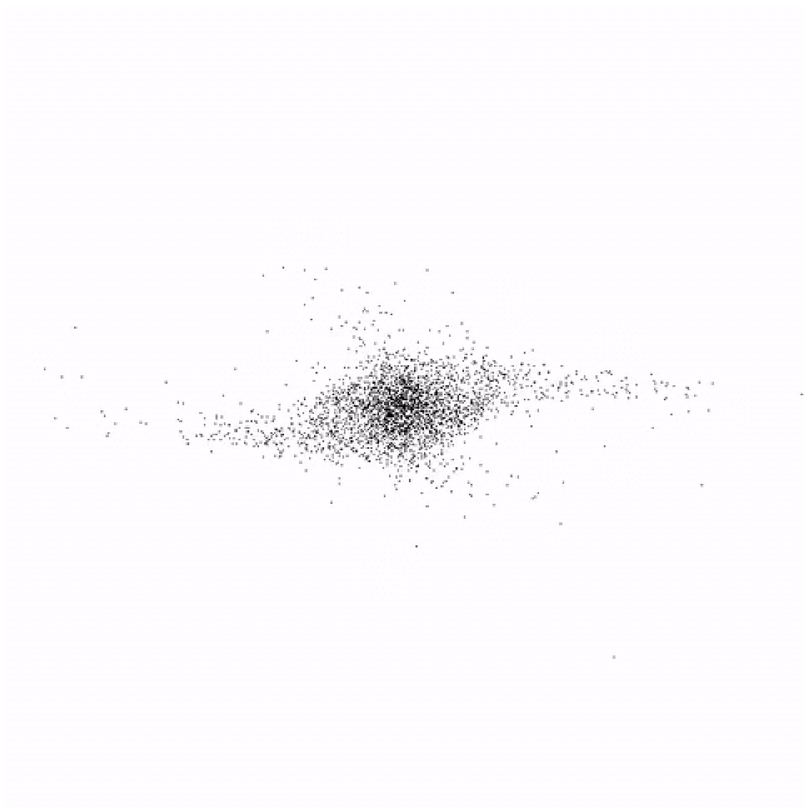
Final Project Report

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

Michael Whittaker (mjw297)

Sheroze Sheriffdeen (mss385)

December 14, 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.

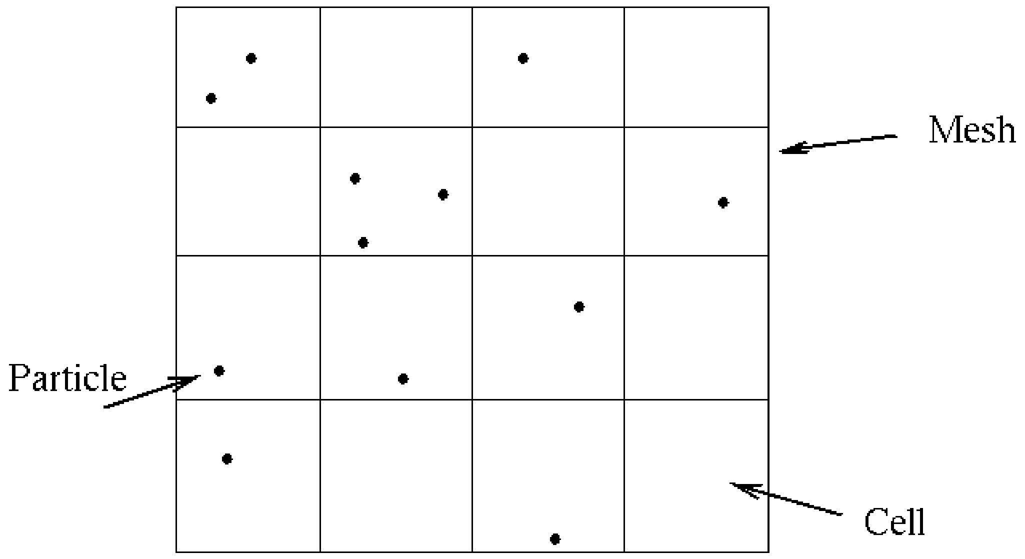


Figure 2.1: Particle-Mesh Model [4]

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.

The following algorithm describes the 2 dimensional NGP scheme on a square mesh.

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

where cell_ρ holds the mass density at the cell centers.

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.

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 Summary

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

1. Assign mass to cell centers using the NGP scheme.
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 Details

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

3.1 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,

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

3.1.1 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);
```

3.1.2 FFTW Usage

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

```
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.

3.2 Marshalling and Animations

Our simulator 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

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 report is shown in Figure 4.1.

TODO(mjw): amplxe stuff

4.2 Tuning

4.3 Design Decisions

// Section to talk about our design decisions

```
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
---------------------------  ------------  --------
main                         serial_opt      6.907s
n1bv_64                      serial_opt      3.855s
n2fv_64                      serial_opt      2.387s
n1fv_32                      serial_opt      1.808s
hc2cbdftv_16                 serial_opt      1.120s
__intel_memset               serial_opt      1.100s
hc2cfdftv_16                 serial_opt      0.960s
fftw_cpy1d                   serial_opt      0.940s
t2fv_32                      serial_opt      0.900s
fftw_cpy2d                   serial_opt      0.898s
t2bv_16                      serial_opt      0.749s
```

Figure 4.2: A snippet of an `amplxe` profile of our serial code

// Vectorization report analysis, vectorized loops etc.

4.3.1 OpenMP

// I left the OpenMP intro here. You can leave it and just talk about the design decisions or we can get rid of it and go straight to design.

OpenMP (Open Multi-Processing) is an Application Programming Interface (API) that supports shared memory programming. The N-body simulation using the particle-mesh method decouples the particle attribute updates so that each particle's velocity and position can be independently updated using the shared resource, the array that holds the gravitational potential ϕ.

The following example that updates the acceleration values in the x direction highlights some of the useful directives in OpenMP.

```
Function                              Module             CPU Time
------------------------------------  -----------------  --------
__kmp_fork_barrier                    libiomp5.so        283.260s
n2fv_64                               ppm_omp             13.749s
n1bv_128                              ppm_omp             12.812s
n1bv_64                               ppm_omp              9.395s
main                                  ppm_omp              7.493s
hc2cbdftv_8                           ppm_omp              5.470s
n1fv_32                               ppm_omp              4.710s
fftw_cpy1d                            ppm_omp              4.427s
__kmp_launch_thread                   libiomp5.so          3.570s
hc2cfdftv_16                          ppm_omp              2.587s
__kmp_join_call                       libiomp5.so          2.354s
```

Figure 4.3: A snippet of an `amplxe` profile of our parallel code

```
#pragma omp parallel for
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;
  }
}
```

- `#pragma omp` signifies that the current line is an OpenMP directive.
- `parallel` clause creates a thread pool that enables multiple threads to perform operations on the shared memory `phi`.
- `for` communicates to OpenMP that the following operation is a `for` loop to be executed in parallel.

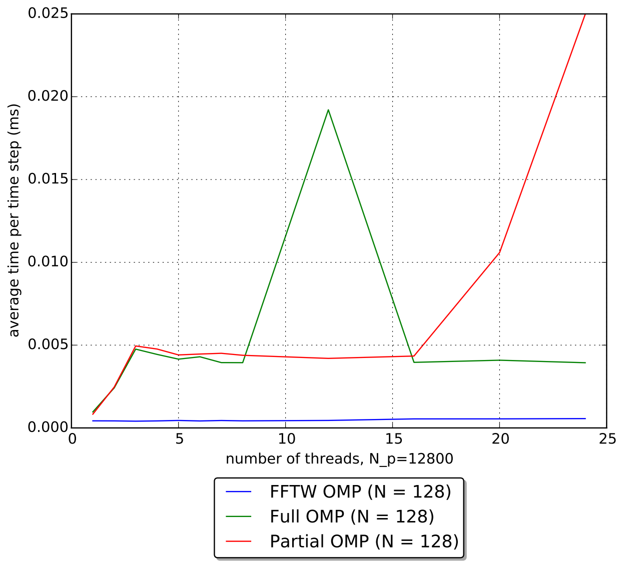
4.4 Scaling Studies

4.4.1 Strong Scaling Study

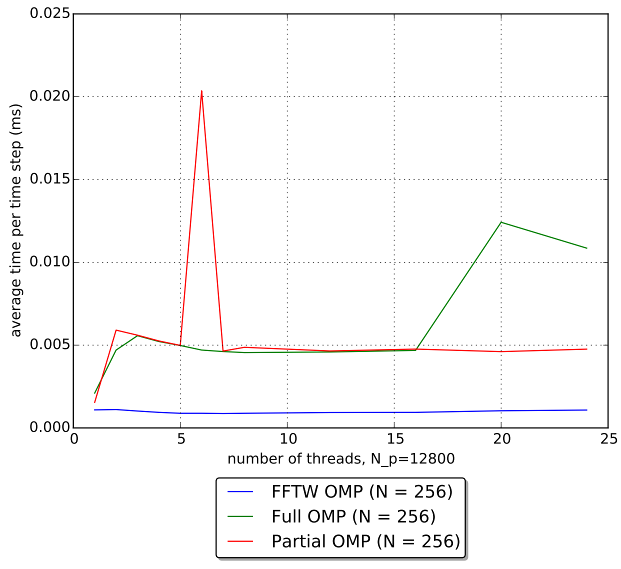
4.4.2 Weak Scaling Study

4.4.3 Particle Scaling

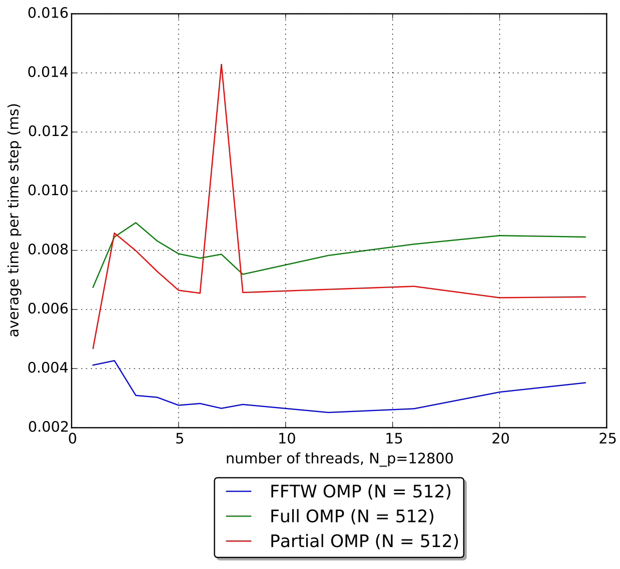
4.4.4 Grid Size Scaling



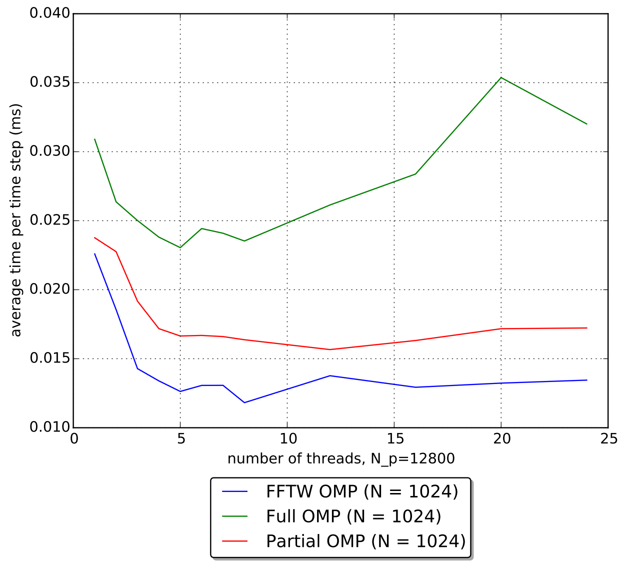
(a)



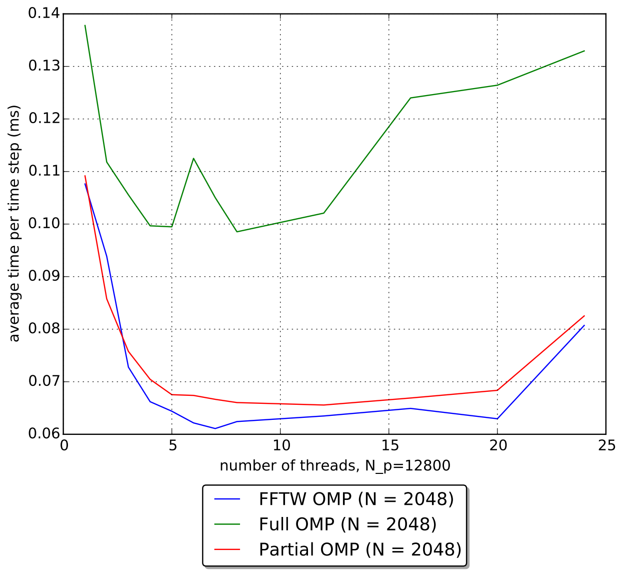
(b)



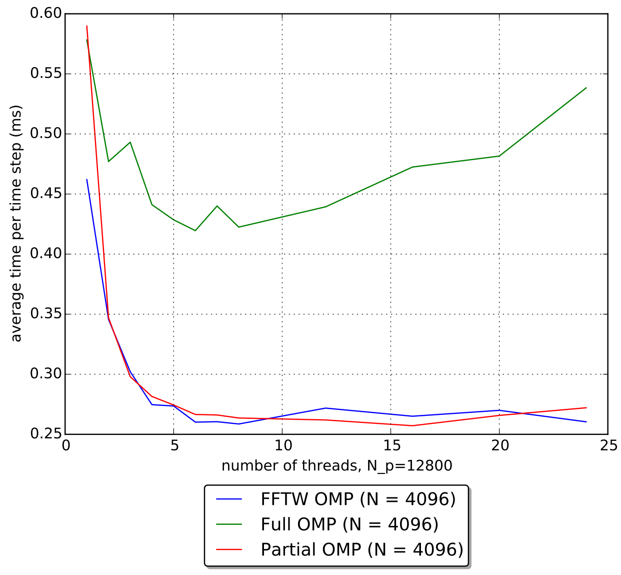
(c)



(d)

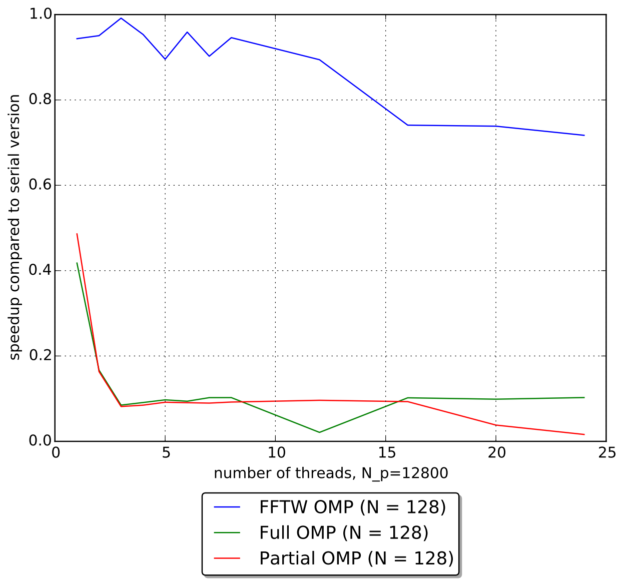


(e)

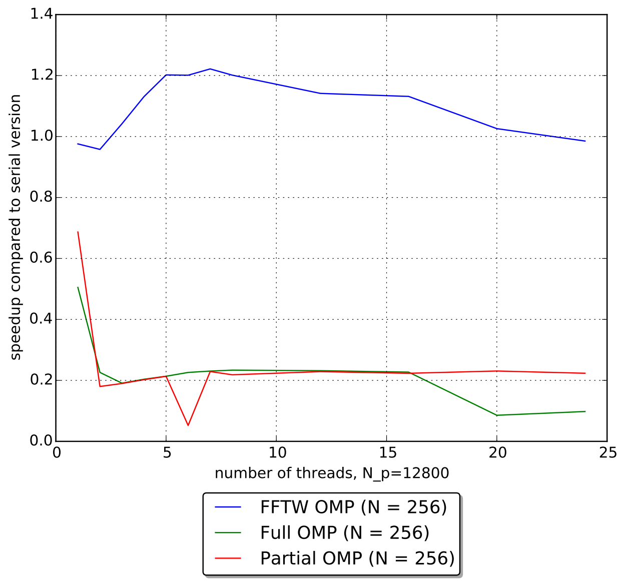


(f)

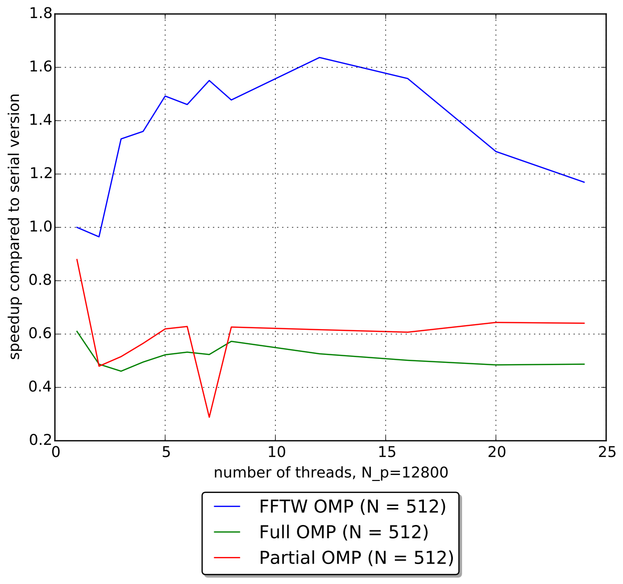
Figure 4.4: Average iteration time vs number of threads



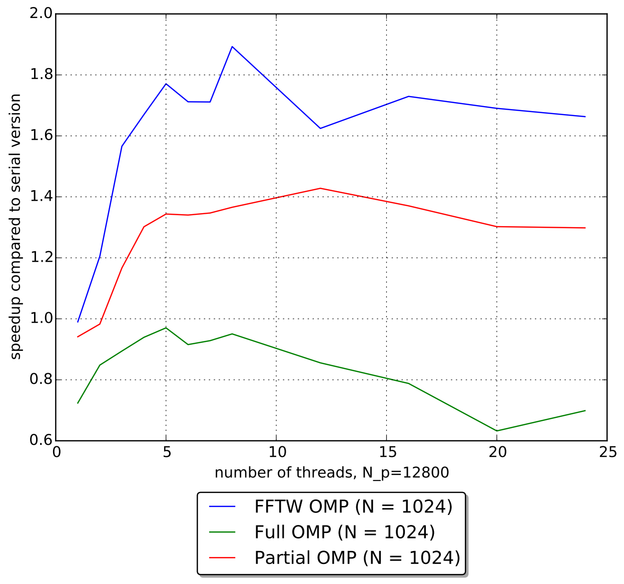
(a)



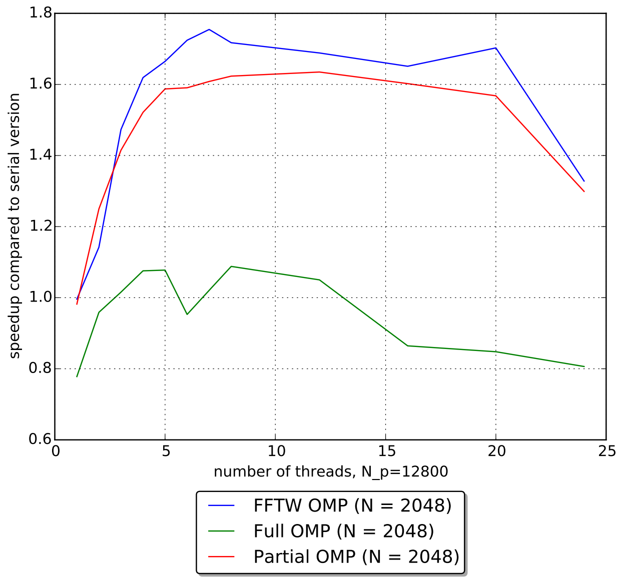
(b)



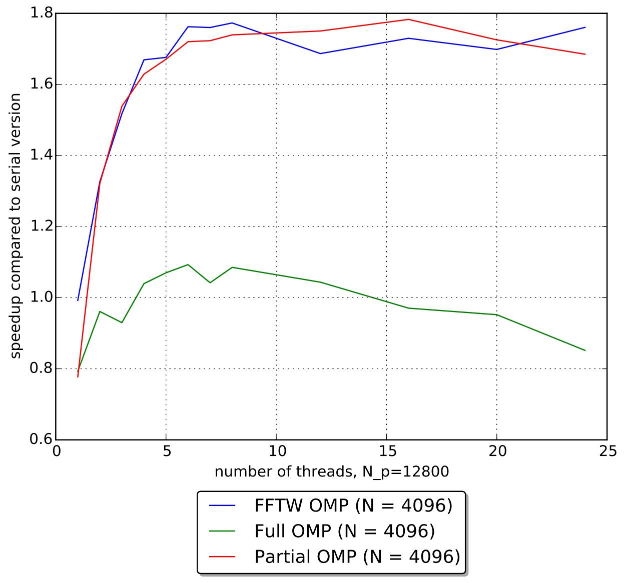
(c)



(d)

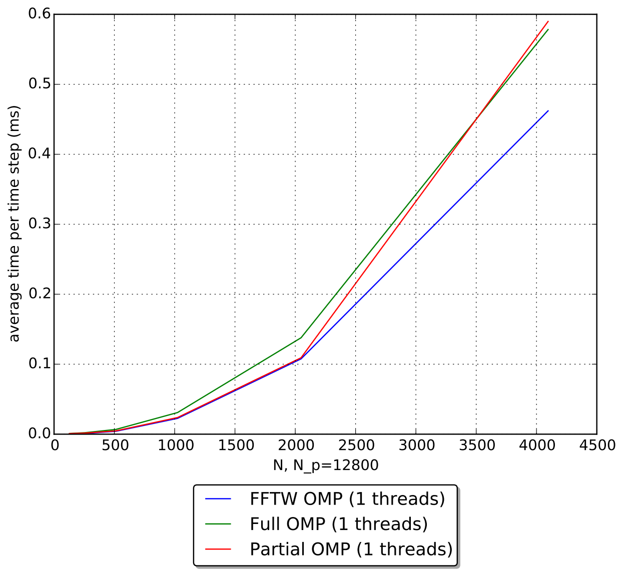


(e)

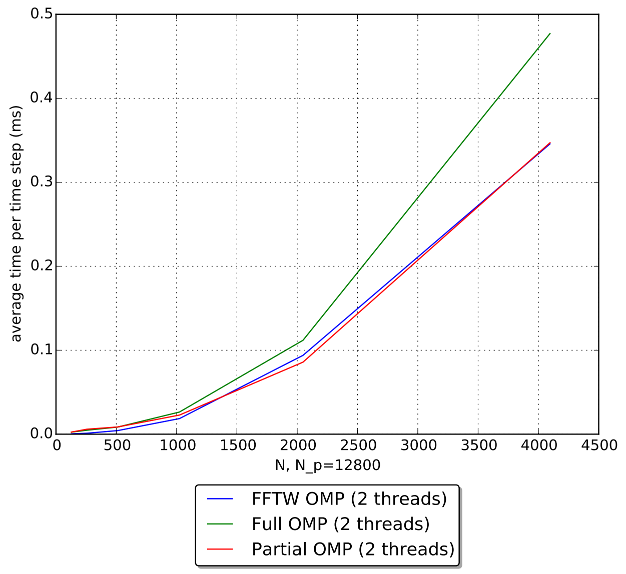


(f)

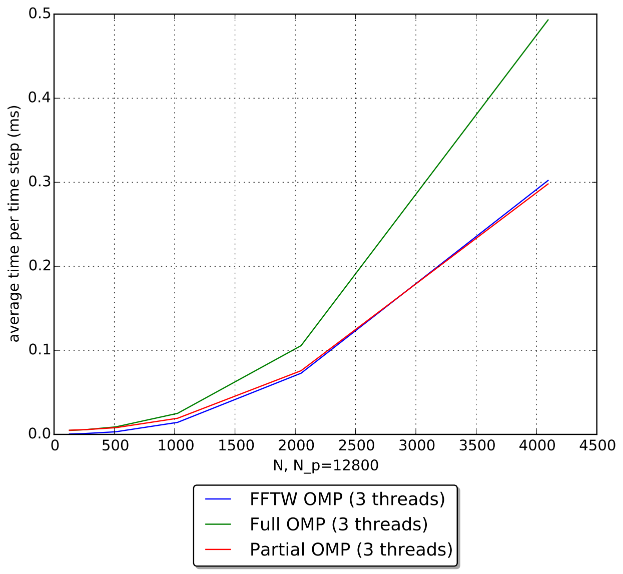
Figure 4.5: Speedup vs number of threads



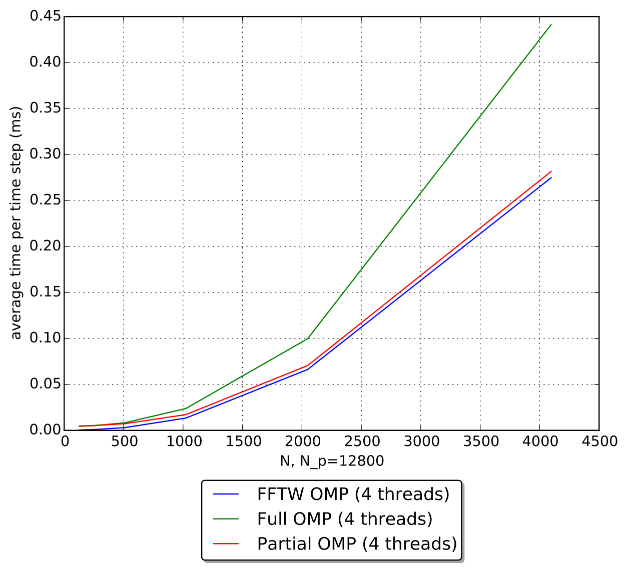
(a)



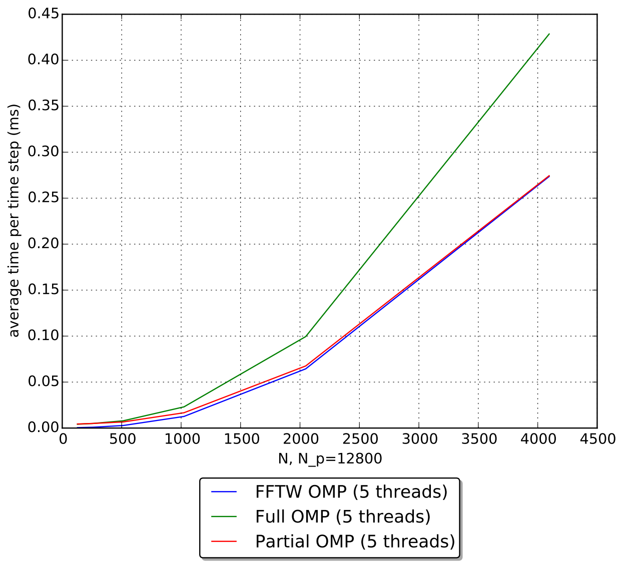
(b)



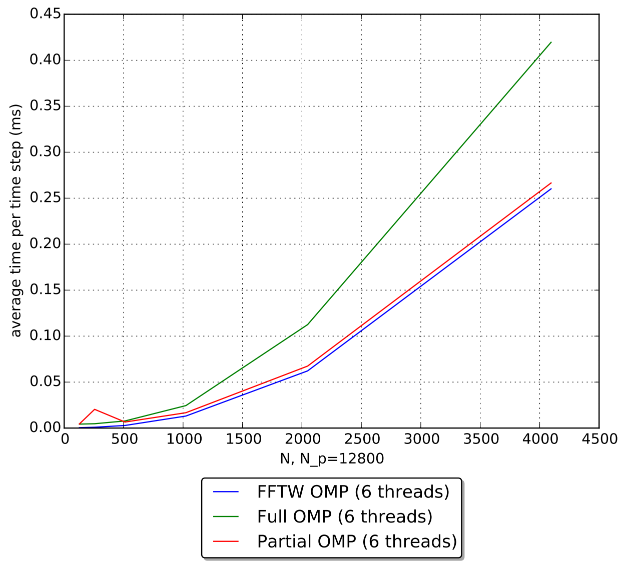
(c)



(d)

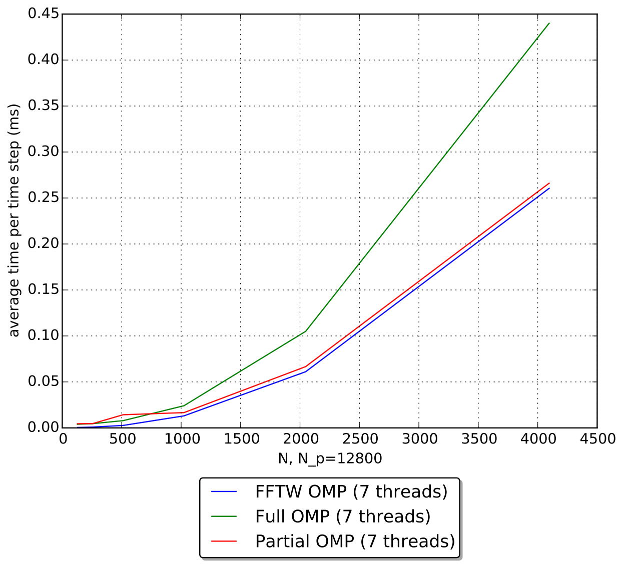


(e)

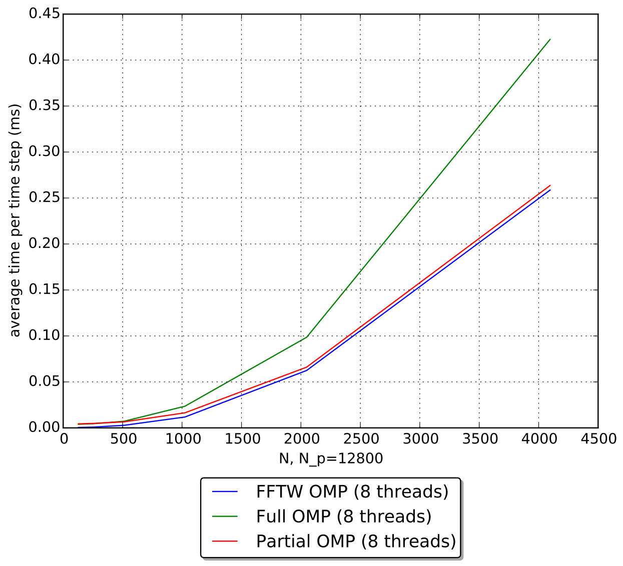


(f)

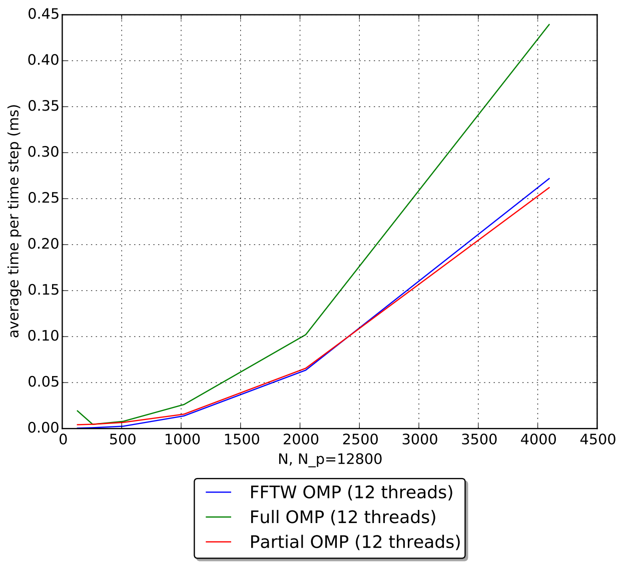
Figure 4.6: Average iteration time vs grid size (N)



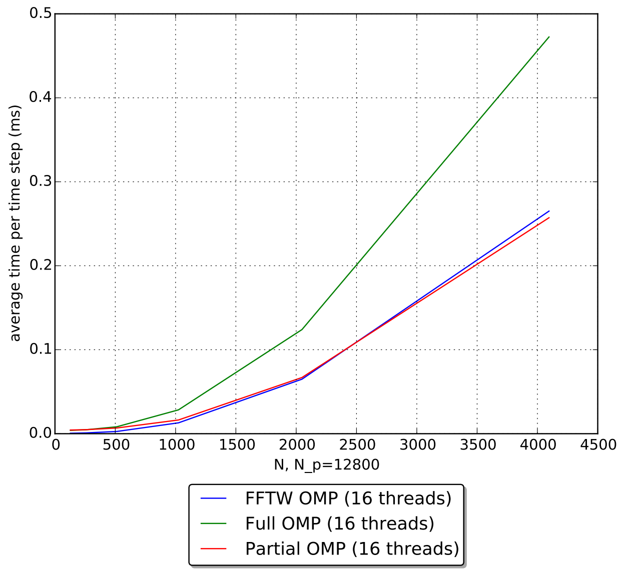
(a)



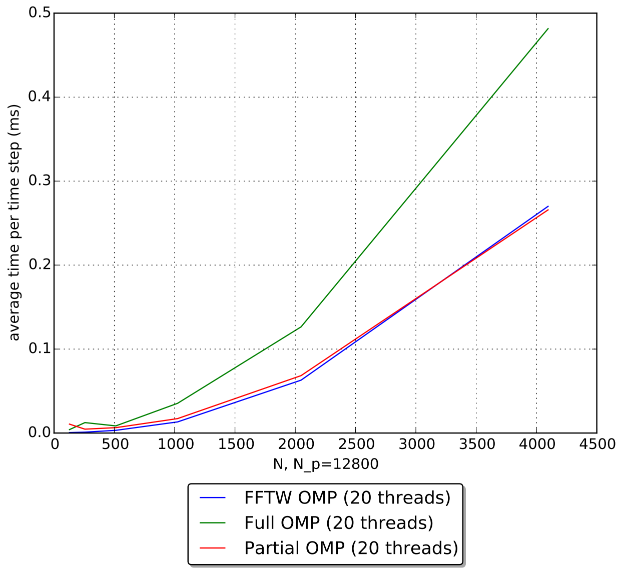
(b)



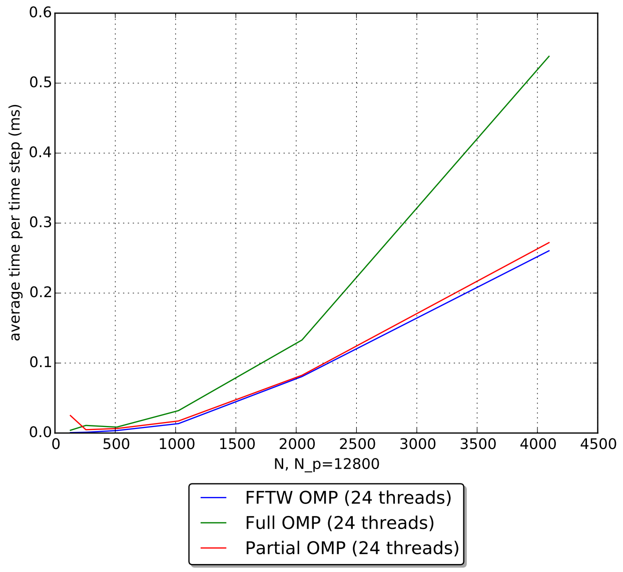
(c)



(d)

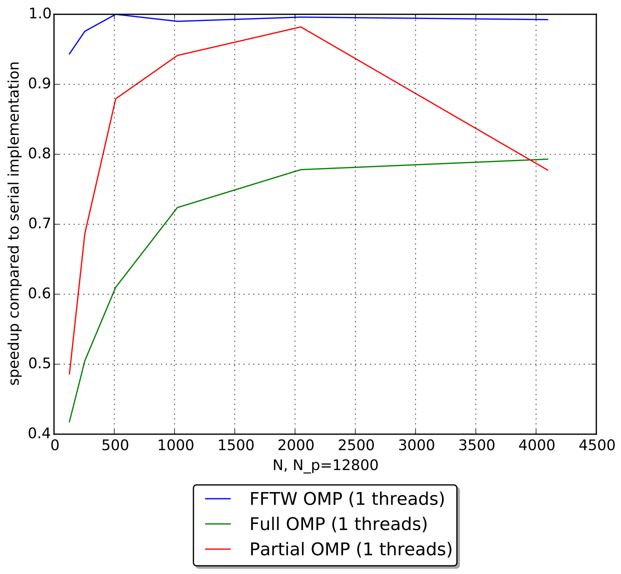


(e)

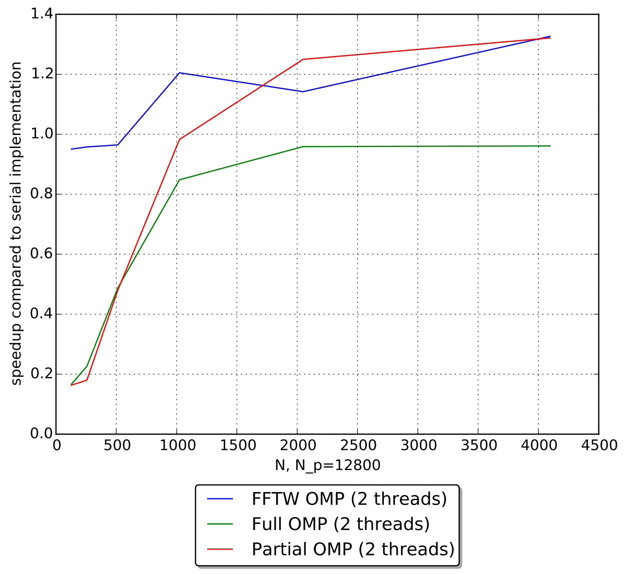


(f)

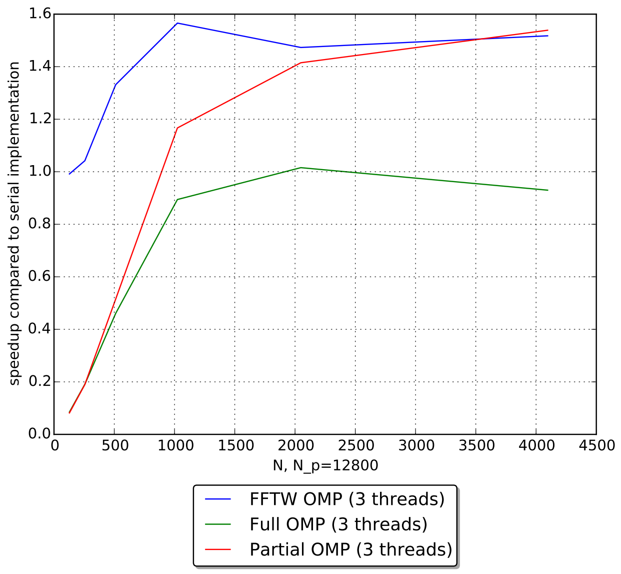
Figure 4.7: Average iteration time vs grid size (N)



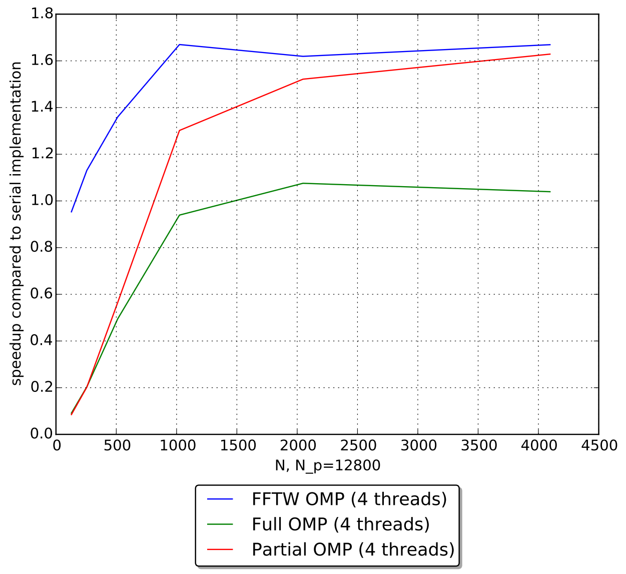
(a)



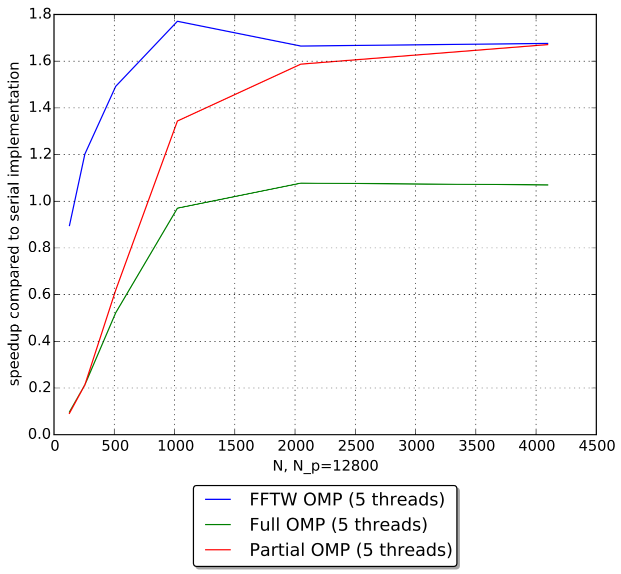
(b)



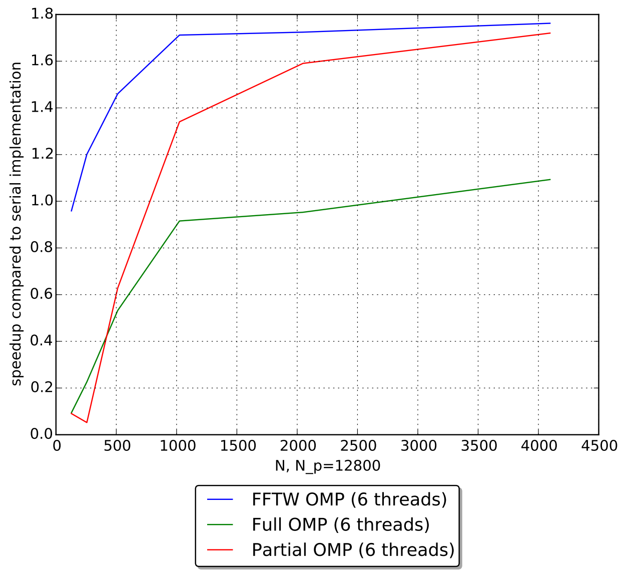
(c)



(d)

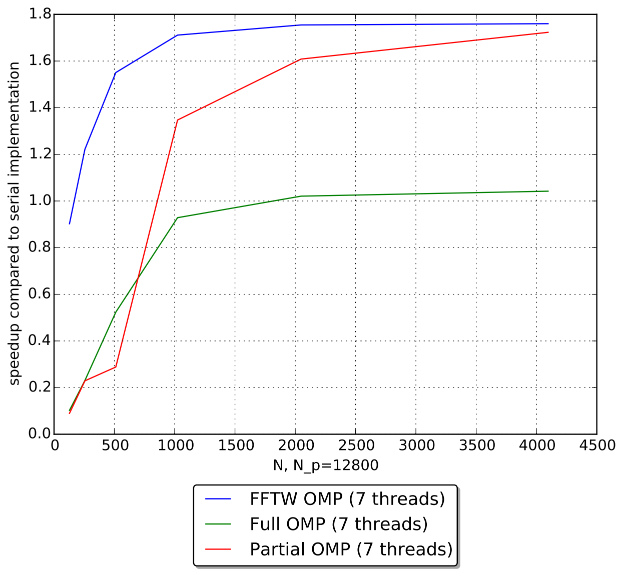


(e)

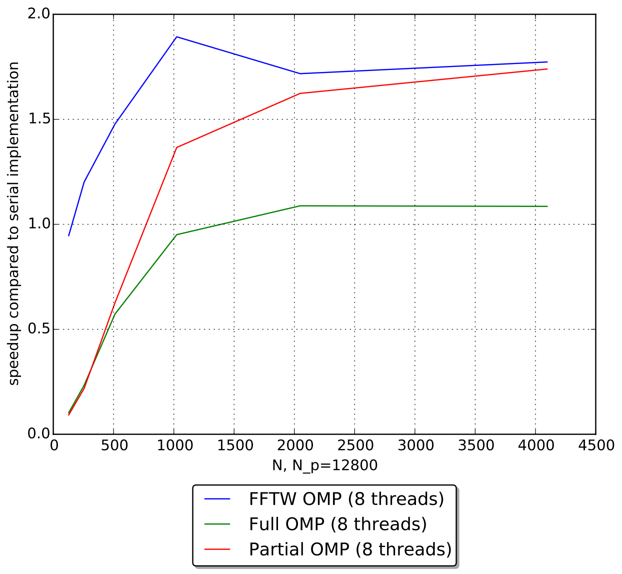


(f)

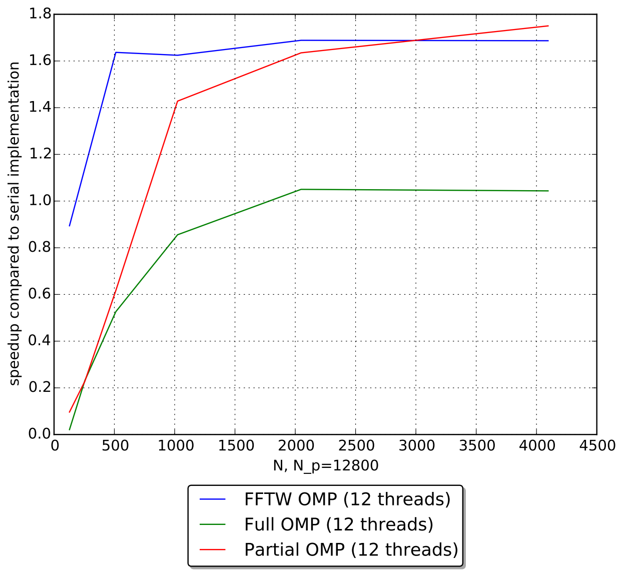
Figure 4.8: Speedup vs grid size (N)



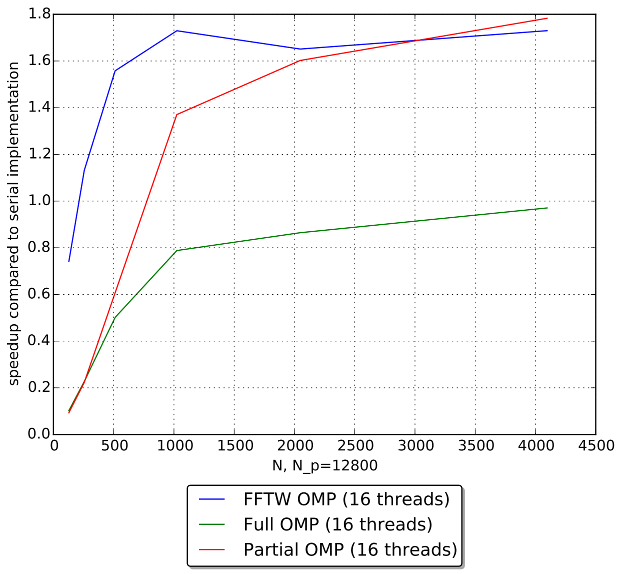
(a)



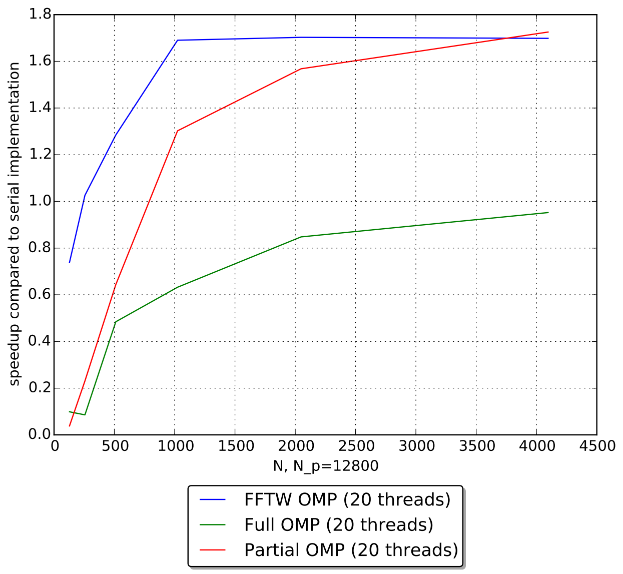
(b)



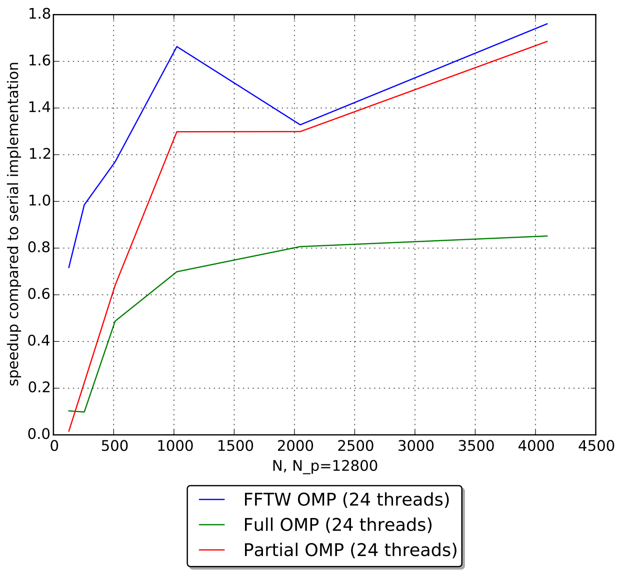
(c)



(d)



(e)



(f)

Figure 4.9: Speedup vs grid size (N)

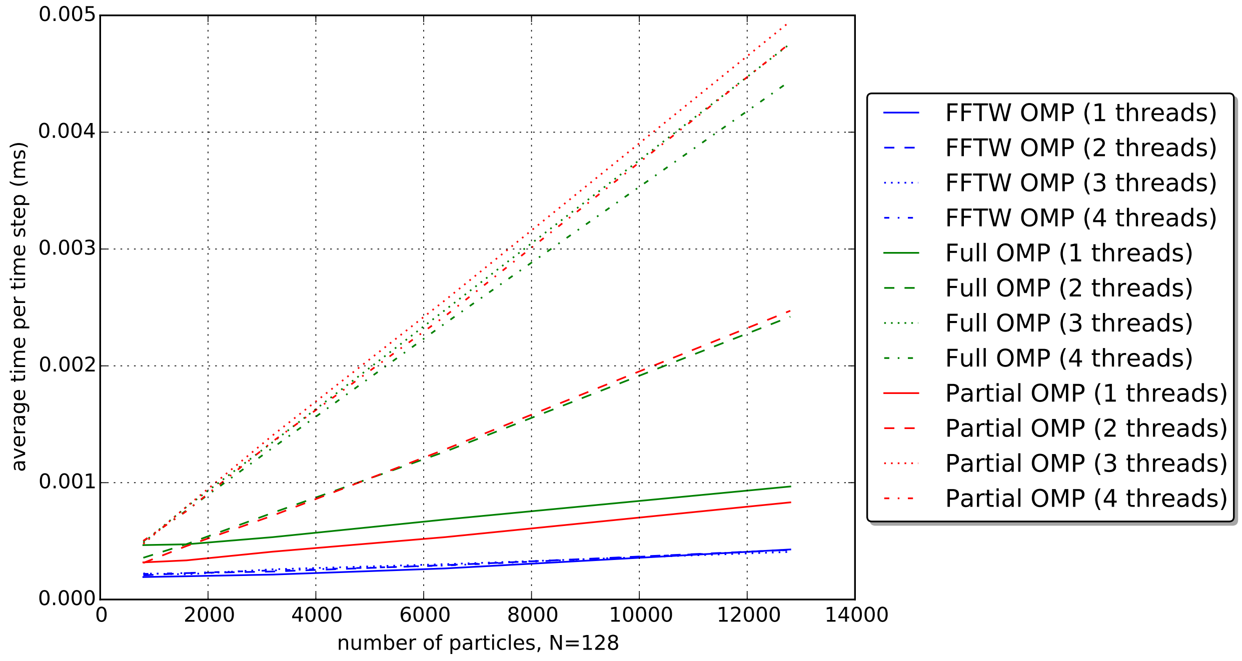


Figure 4.10: Average iteration time vs number of particles (N_p)

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 total space of the simulation is L. 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.

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

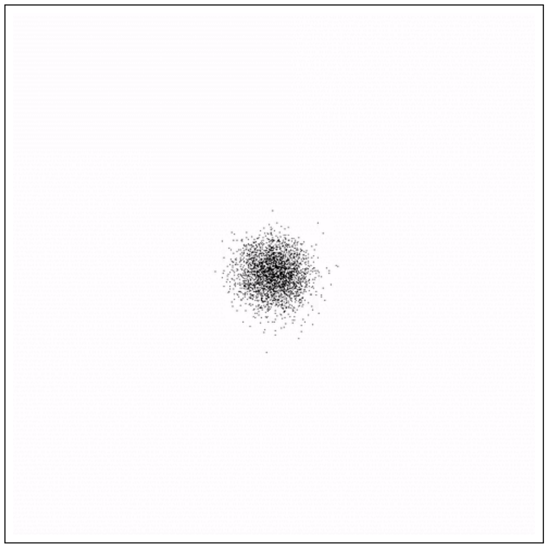
$$v_y = \cos(\theta)\sqrt{\frac{GM}{r}} \tag{5.5}$$

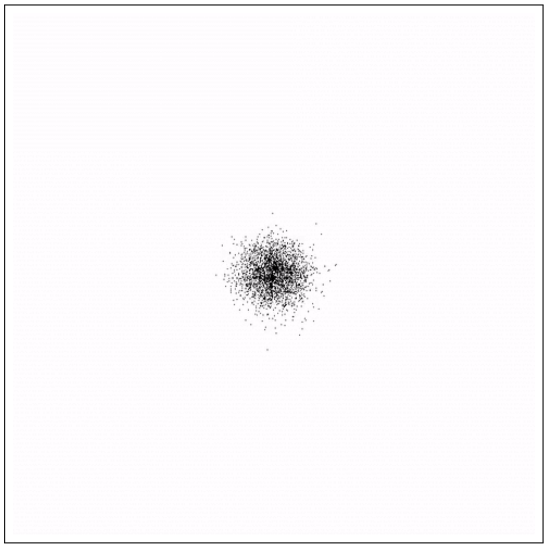
$$\tag{5.6}$$

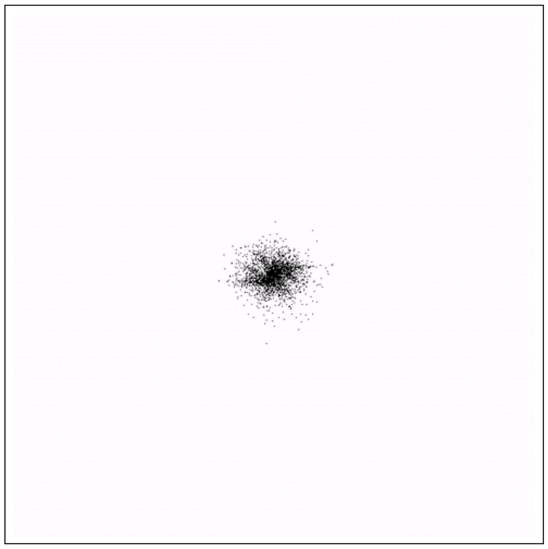
where M is the mass of the central body, G is the gravitational constant, r is the separation between the central mass and the particle, and θ is the azimuthal angle between the central body and the particle body where the central body is at the origin. This provides the initial rotational component 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.mp4` or generated using `mp4.py`. The `README` file documents the simulation generation process.


(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

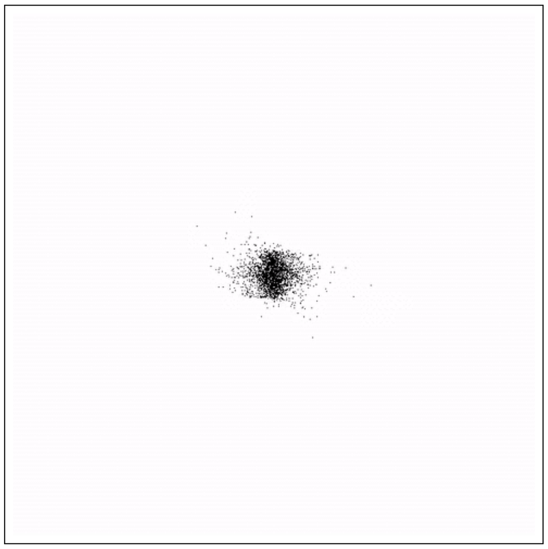
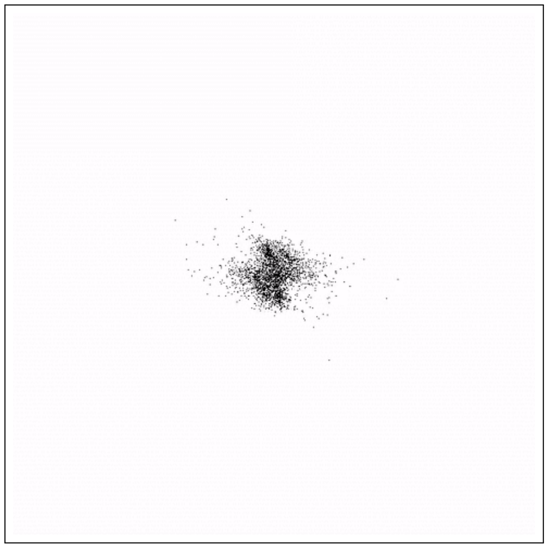
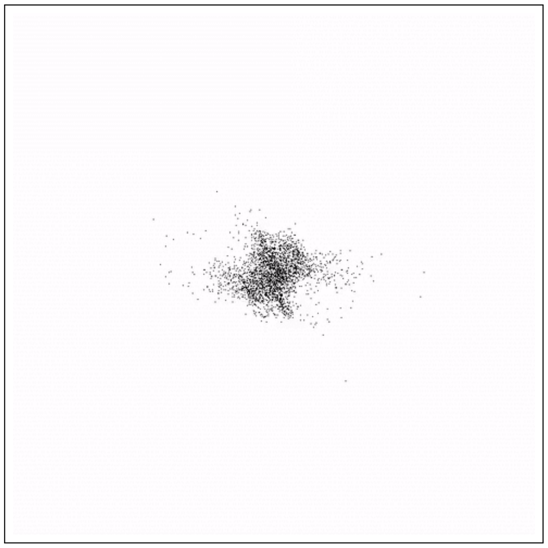

(d) Bodies at $t = 7.3$ days, $L = 60$AU

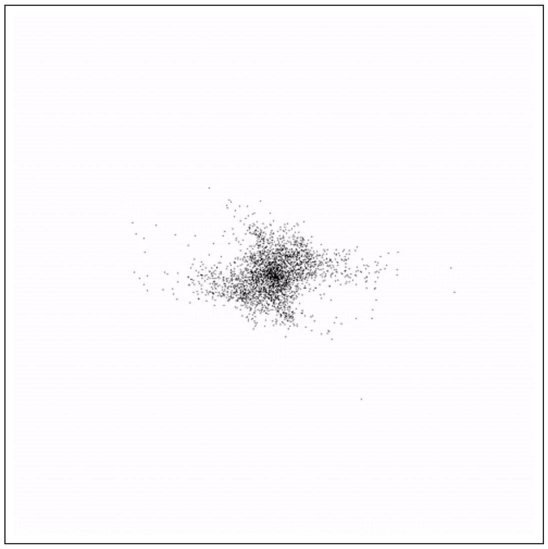
Figure 5.1: Configuration evolution with $\Delta t = 0.86$ hours, 3000 particles, $\mu_m = 1 \times 10^{30}$ kg



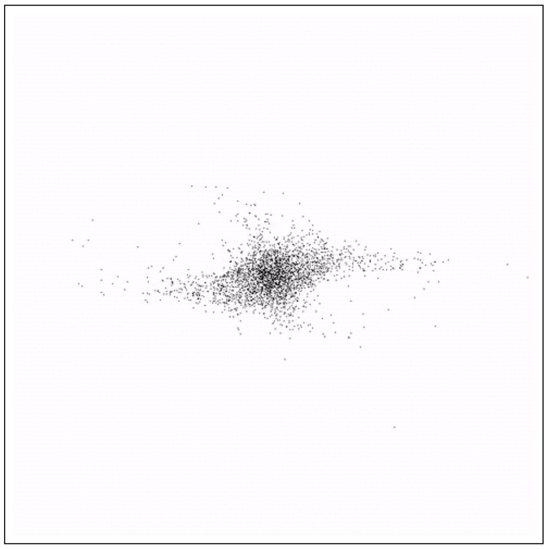
(a) Bodies at $t = 9.73$ days, $L = 60$AU



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

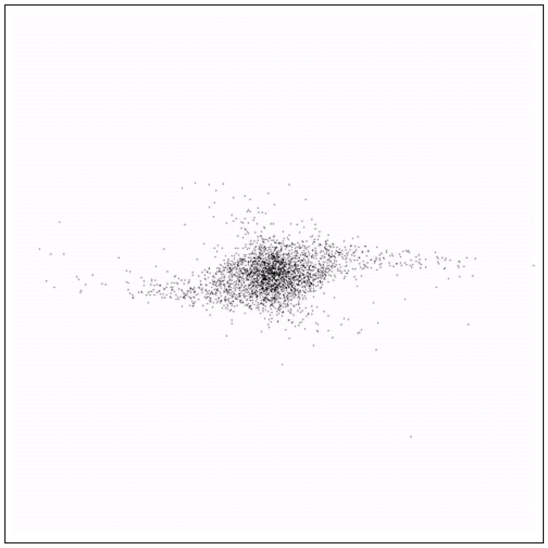


(c) Bodies at $t = 14.6$ days, $L = 60$AU

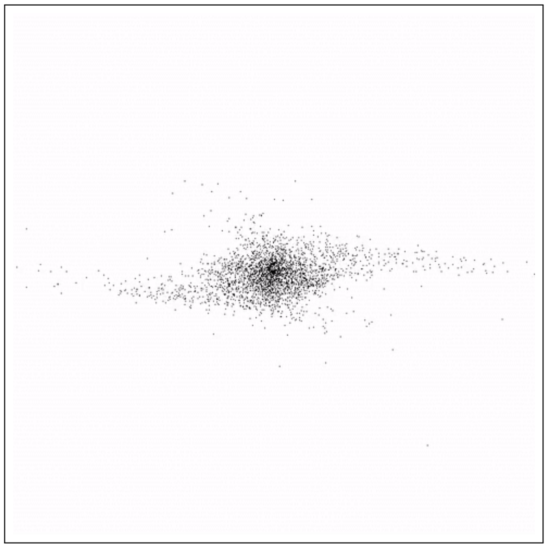


(d) Bodies at $t = 17.03$ days, $L = 60$AU

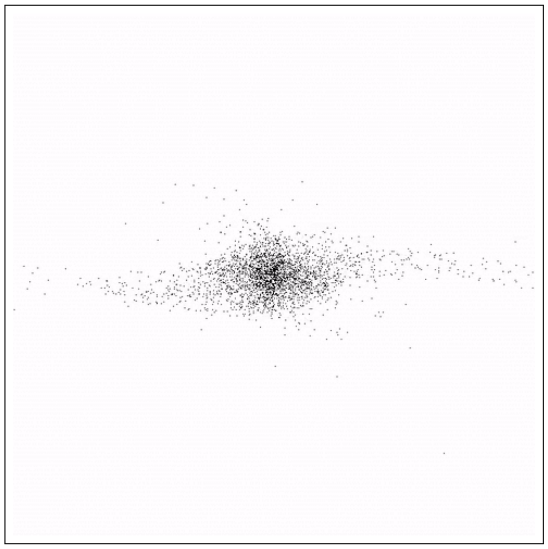
Figure 5.2: Configuration evolution with $\Delta t = 0.86$ hours, 3000 particles, $\mu_m = 1 \times 10^{30}$ kg



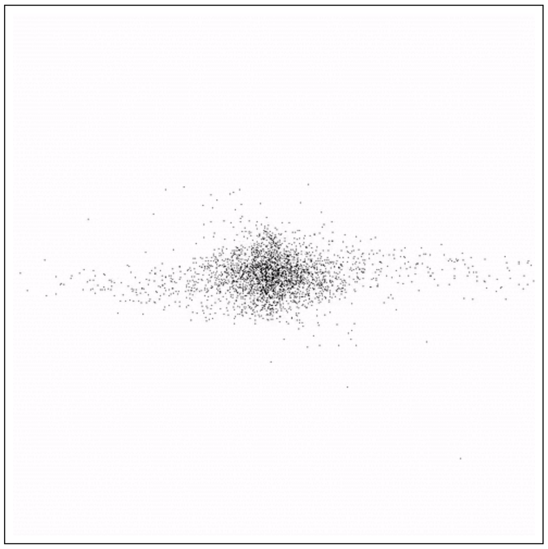
(a) Bodies at $t = 19.46$ days, $L = 60\text{AU}$



(b) Bodies at $t = 21.9$ days, $L = 60\text{AU}$



(c) Bodies at $t = 24.33$ days, $L = 60\text{AU}$



(d) Bodies at $t = 26.76$ days, $L = 60\text{AU}$

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

Chapter 6

Conclusions

// We can probably tailor this section to 5220 more

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 (Message Passing Interface), is a programming model that enables 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].

Finally, to improve the viewing experience of the simulation, we can invert the data generation process where a graphical user interface can poll simulation data in real time from a data server. The initial attempt at this server-client structure is documented in the Github page for the CS5220 final project under `scripts/fileserver.py` [13]

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