A Novel Special Relativistic N-body Simulation

James Atwood

April 5, 2007

Contents

Chapter 1

Introduction

1.1 Dark Matter

The problem of resolving the rotation curves of galaxies with accepted theories of gravitation has been central to the study of astrophysics (see review by Bergh, 1999). Dark matter theories contend that when weakly interacting massive particles, which are nearly impossible to detect, are introduced to the composition of galaxies, observed galactic rotations are consistent with Newtonian gravitation theory. This theory has become widely accepted within the astrophysics community (Bergh, 1999).

There has recently, however, been some discussion regarding the validity of the dark matter theory (Cooperstock & Tieu, 2005; Menzies & Mathews, 2006; Fuchs & Phleps, 2006; Garfinkle, 2006). Cooperstock and Tieu have recently proposed that general relativity alone can account for observed galactic rotations (2005). This claim is based upon the success their general relativistic galactic model in the absence of dark matter. This has caused some uproar in the astrophysics community, and many papers attempting to refute the findings of Cooperstock and Tieu have been published. Some publications question specific aspects of Cooperstock and Tieu's model (Menzies & Mathews, 2006), while others point out the model's failure to resolve defining aspects of a galaxy other than rotation curves (Fuchs & Phleps, 2006), and still others argue that a model such as Cooperstock and Tieu's can not possibly work (Garfinkle, 2006). This discussion has motivated us to undertake our own investigation of large-scale

gravitational dynamics.

1.2 Our Simulation

We have constructed a parallel N-body simulation in special relativity. This simulation is conceptually more transparent than the model of Cooperstock and Tieu, as we are modeling stars as discrete bodies, whereas they model the galaxy as a fluid, and we are using the simpler special relativistic theory.

We model a system as a collection of bodies that interact under special relativistic physics. We have implemented an information lag, as gravitational interactions between bodies propagate at the speed of light in special relativity. To the author's knowledge, the information lag implemented in our simulation is novel to the study of N-body simulations.

This simulation is currently still in its infancy, and cannot yet be used to simulate real systems. Our qualitative analyses of simple systems, however, are promising, and future work with the simulation could shed some light on the dark matter issue.

1.3 This Thesis

In the following chapter we will provide a brief summary of the history of dark matter and note that it is essentially a classical theory. We will then explain our motivation for writing this simulation. Finally, we will give a brief analysis of the physics that we have implemented.

In Chapter 3 we will provide an in-depth analysis of the code of our simulation. We give an overview of parallel programming with MPI, and use this information to build up to an understanding of our simulation. We then present the current issues with the simulation.

In Chapter 4 we investigate some simple systems and provide qualitative analyses of our results.

Chapter 2

Background

The central issue of this thesis is whether observed galactic dynamics are consistent with "traditional" physical models, or if exotic non-luminous non-interacting particles must be introduced to make our current theories consistent with the available data. An investigation of the origin and subsequent explication of the dark matter theory is in order.

2.1 A Brief History of Dark Matter

Fritz Zwicky published the paper "Die Rotverschiebung von extragalaktischen Nebeln" (1933), and a similar paper "On the Masses of Nebulae and of Clusters of Nebulae" (1937). In each paper he notes that there is a large discrepancy between the mass of the Coma cluster as implied by the Newtonian virial theorem and the mass of the Coma cluster as implied by its luminosity. In the 1933 paper he posits the existence of some sort of "dunkle Materie" (dark matter) that would account for the missing mass. (It is interesting to note that in the 1937 paper Zwicky does not posit dark matter but rather notes that further investigation is needed).

In 1936, Sinclair Smith published a study of the mass of the Virgo cluster in which he analyzed the spatial velocities of the constituent nebulae. He assumed that these nebulae were executing circular orbits and applied the Newtonian relation $m = \frac{v^2 r}{G}$ to find that the individual masses of the constituent nebulae were about 200 times greater than the mass Hubble had previously predicted for

a single nebula. Smith concludes that "the difference between [Smith's analysis] and Hubble's value for the average mass of a nebula apparently must remain unexplained until further information becomes available.", but speculates that "It is... possible that both [Hubble's analysis and Smith's analysis] are essentially correct, with the difference representing internebular material, either uniformly distributed or in the form of great clouds of low luminosity surrounding the nebulae." Note that both Zwicky and Smith adopted an incorrect value for the Hubble constant, which caused them to over-estimate the mass of the missing material. Still, similar analyses with the modern value for the constant still imply a considerable mass discrepancy.

The work of Ostriker & Peebles (1973) is considered to be instrumental in convincing astrophysicists that dark matter exists (Bergh, 1999). Ostriker and Peebles constructed a Newtonian N-body simulation and determined that, within the context of their simulation, galaxies such as the Milky Way are grossly and irreversibly unstable. They determined that introducing a halo of matter that surrounds the galaxy was the most plausible method for solving the dilemma.

For a more detailed exposition of the history of dark matter, the reader is referred to Sidney Van Den Bergh's 1999 review of the subject.

The last few years have also seen considerable work in dark matter theories. The Millennium Simulation, which is the largest N-body simulation to date with 10^9 particles, has enjoyed enormous success in predicting large-scale universal formation by including dark matter (Springel et al., 2005). Their simulation is based upon N-body integration approaches that do not seem to implement the finite speed of gravitational interactions.

2.2 Motivation

It is extremely important to note that all of the previous papers, which are considered to be the foundation of dark matter theory (Bergh, 1999), are based upon *classical analyses* of what is essentially a *relativistic problem*. It would seem that applying the finite speed of gravitational interaction to a such a spatially immense problem would be not only appropriate, but essential to physi-

cally correct results, and it is thus a bit mystifying that no one has undertaken such a study. This is most likely due to the difficulty of the task, as creating a special relativistic N-body simulation is extremely arduous, and running such a simulation requires enormous computational resources. With increased resources at our disposal, we argue that now is the time to integrate special relativistic effects into N-body simulations.

The purpose of our simulation is to investigate whether adding relativistic effects to a conceptually simple N-body simulation significantly improves the correspondence of that simulation to observed data. Positive findings would hopefully provide some impetus for further work in the area.

2.3 The Physics of the Simulation

Our simulation implements discrete special relativistic physics.
Consider a body with position $\vec{x}$, velocity $\vec{v}$, and mass m. The position and velocity vectors are measured relative to a fixed origin $O = (0, 0, 0)$. This body will have a relativistic momentum p_r (read as "p relativistic"), where

$$\vec{p_r} = \gamma m \vec{v} \tag{2.1}$$

where

$$\gamma = \frac{1}{\sqrt{1 - (\frac{v}{c})^2}} \tag{2.2}$$

where v is the magnitude of $\vec{v}$ and c is the speed of light.
We define forces via Newton's second law:

$$\vec{F} = \frac{d\vec{p}}{dt} \tag{2.3}$$

By substituting p_r for p and taking the derivative, we find that:

$$\frac{d\vec{p_r}}{dt} = \gamma m \frac{d\vec{v}}{dt} + \frac{d\gamma}{dt} m\vec{v} \tag{2.4}$$

The velocities in our simulation are on the order of $.001c$ — $.01c$. In this semi-relativistic domain, variations in γ will be negligible, so we can ignore the second

term on the right-hand side of the equation, leaving us with with a momentum p_{sr} (read as "p semi-relativistic"):

$$\frac{d\vec{p}_{sr}}{dt} = \gamma m \frac{d\vec{v}}{dt} \tag{2.5}$$

Combining equations 2.3 and 2.5, we find:

$$\vec{F} = \gamma m \frac{d\vec{v}}{dt} \tag{2.6}$$

or

$$\vec{F} = \frac{d\vec{p}_{sr}}{dt} \tag{2.7}$$

Position and momentum are related by:

$$\frac{\vec{p}_{sr}}{m} = \frac{d\vec{x}}{dt} \tag{2.8}$$

or, rearranging:

$$d\vec{x} = \frac{\vec{p}_{sr}}{m} dt \tag{2.9}$$

Our simulation is discrete; instead of modeling the behavior of our system with continuous time derivatives, we iterate over discrete time steps (Δt). Replacing the differential equations 2.7 and 2.9 with difference equations, we find that momentum is given by:

$$\Delta\vec{p}_{sr} = \vec{F}\Delta t \tag{2.10}$$

and position is given by:

$$\Delta\vec{x} = \frac{\vec{p}_{sr}}{m} \Delta t \tag{2.11}$$

The only force in our simulation is gravity, so:

$$\vec{F} = \frac{G m_i m_j}{r^2} \hat{r} \tag{2.12}$$

where m_i and m_j are the masses of star i and star j, and r is the displacement of the two stars. It can be shown that, if $F = (F_x, F_y, F_z)$, then

$$F_k = \frac{Gm_i m_j}{r^2}\frac{x_k^j - x_k^i}{r} \tag{2.13}$$

Consider a simulation with m stars. Star i has initial position $\overrightarrow{x_0^i} = (x_{0_1}^i, x_{0_2}^i, x_{0_3}^i)$ and momentum $\overrightarrow{p_0^i} = (p_{0_1}^i, p_{0_2}^i, p_{0_3}^i)$ (we now refer to $\vec{p}_{sr}$ as $\vec{p}$ to reduce clutter). We can combine equations 2.10 and 2.11 with equation 2.13 to find $\overrightarrow{p_1^i}$ and $\overrightarrow{x_1^i}$, the momentum and position of star i after one iteration:

$$p_{1_k}^i = p_{0_k}^i + \sum_{\substack{j=1 \\ i\neq j}}^{m} \frac{Gm_i m_j}{r^2}\frac{x_{0_k}^j - x_{0_k}^i}{r}\Delta t \tag{2.14}$$

$$x_{1_k}^i = \frac{p_{1_k}^i}{m}\Delta t \tag{2.15}$$

We can easily generalize equations 2.14 and 2.15 to find $\overrightarrow{p_n^i}$ and $\overrightarrow{x_n^i}$, the momentum and position of star i after n iterations:

$$p_{n_k}^i = p_{n-1_k}^i + \sum_{\substack{j=1 \\ i\neq j}}^{m} \frac{Gm_i m_j}{r^2}\frac{x_{n-1_k}^j - x_{n-1_k}^i}{r}\Delta t \tag{2.16}$$

$$x_{n_k}^i = \frac{p_{n_k}^i}{m}\Delta t \tag{2.17}$$

Note that gravitational interactions propagate at a finite speed, which implies that equations 2.16 and 2.17 are not complete in their current form. We have implemented the information delay using techniques from computer science, which will be discussed in §3.1.2.3.

Chapter 3

Code

3.1 Overview

Our highly parallel simulation is written in C and implements the standard Message Passing Interface (MPI). We will present an extremely brief and qualitative overview of the mechanics of MPI; for a more thorough explanation, please see Peter Pacheco's book on the subject (1997). We will then informally explain the structure of our simulation. Finally, we will give an in-depth analysis of the role of each component in the program.

3.1.1 MPI

Our simulation is designed to run on *distributed memory clusters* (Figure 3.1). These clusters can be envisioned as many separate computers (henceforth nodes), each with their own processor and memory, that are able to communicate over some sort of network. Each node runs an instance of the simulation; all of these instances are exactly the same. Every node performs operations (via its processor) on the data stored in its own local memory.

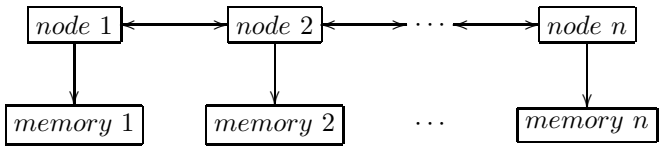

Figure 3.1: A distributed memory cluster.

If a node requires data that another node controls, the data must be communicated. The simplest form of communication is a *send-receive operation*, in which the node that controls the data sends it to the node that needs the data. There also exists another form of communication, called *collective communication*, which comes in many different flavors. A node can send data to every other node (a *broadcast operation*), or data can be gathered from every node to a single node (a *gather operation*), or every processor can collect the same data from every other processor (an *all-gather operation*). There are many more varieties of collective communication, but these are the only three utilized by our simulation. The MPI library provides C-callable functions that are designed to accomplish these various communication operations. A broadcast operation, for example, can be executed via a call to `MPI_Bcast`.

The largest complication in designing and implementing a piece of parallel software is ensuring that data is communicated properly. This is a very important task, because, while all of the nodes are running the same program, they are each managing different sets of data. Extreme care needs to be taken to prevent a node from trying to access data that it does not control — this will cause a run-time segmentation fault and crash the execution of the program. Furthermore, sending anything other than standard C datatypes via MPI requires meticulous datatype definitions, so that nodes sending a chunk of data can properly package it and nodes receiving the data can properly parse it.

3.1.2 Code Structure

The structure of our program is fairly complicated. We'll work up to an understanding of the structure of our parallel relativistic simulation by informally analyzing some simpler cases: a serial classical simulation, a parallel classical simulation, and a serial relativistic simulation

3.1.2.1 Simulation 1: Serial and Classical

Let's suppose that we wish to write a classical N-body simulator in C. We would probably want to make some sort of structure that represents a star. It would contain information such as position, momentum, mass, and whatever other

physical parameters we may want to keep track of. Let's call this structure `body`, and typedefine it for convenience:

```
typedef struct body {
  float x[3]; --- The position of the body.
  float p[3]; --- The momentum of the body.
  float mass; --- The mass of the body.
}
```

We could keep track of all the bodies with an array of type `body` and length `nstars`, where `nstars` is the number of stars in the simulation. Instead (for reasons that will become clear later), let's create a new structure that has only one field of type `body*` (a pointer of type `body`). We'll call this new structure `owners` and type-define it as well:

```
typedef struct owners {
  body* star; --- The star that is being tracked
}
```

We can now declare a global array `owner` of type `owners` and length `nstars` (`owners owner[nstars]`). We can then malloc[1] `nstars` stars, and have each entry in the `owner` array point its `body* star` pointer to each star. In order to access star `i`, we simply refer to `*owner[i].star`. This provides an easy mechanism for representing physical interactions. For instance, if we want to apply the function `fun(body* star1, body* star2)` to every unique pair of stars, we would simply execute:

```
for (i=0 ; i<nstars ; i++)
  for (j=(i+1) ; j<nstars ; j++)
    fun(owner[i].star, owner[j].star);
```

If we replace the function `fun` with something useful, such as function that simultaneously moves all of the stars around according to Newtonian force laws, we have a simulation.

[1] malloc is a C function used to initialize data

3.1.2.2 Simulation 2: Parallel and Classical

Now that we have an understanding of the control structure of the simplest serial N-body simulator, let's dive into some parallel programming (we'll keep the physics the same for simplicity's sake). Our program is now operating on several computers (nodes) that have distinct memory. Each of these nodes controls a certain fraction of the total bodies of the simulation. Because the memory of each node is distinct, the nodes must communicate in order to accomplish some unified task.

The structure that represents a body remains the same in our new parallel simulation (reproduced below) — introducing more processors does not change the physical parameters of a body. The `owners` structure, however, must be modified. As it is shown below, the new owners structure contains the additional field `int rank`. Why? MPI, the library we have utilized for communication, assigns a rank to each node. Adding the rank field to the owner control structure allows us to associate a *rank* with a *body*, so that we know where to look for it.

```
typedef struct {
  float x[3]; --- The position of the body.
  float p[3]; --- The momentum of the body.
  float mass; --- The mass of the body.
} body;


typedef struct {
  int rank;    ---  The rank of the processor that body* star exists on
  body* star;  ---  The star that is being tracked
} owners;
```

How is this actually accomplished? Well, let's assume that we want all of the nodes start with the same amount of stars, `my_nstars`. Each node creates two arrays of type `owners`, `owners my_owner[my_nstars]` and `owners owner[nstars]`. Each node mallocs `my_nstars` and tracks these bodies with the `my_owner` array. Each node then assigns the `int rank` field of each entry in the `my_owner` array to the node's rank. Finally, this information is gathered

from every node to every node into the `owner` array via an *allgather* call.

Now suppose we want apply the function `fun(body* 1, body* 2)` to every unique pair of stars, regardless of what node they reside on. We simply have every processor execute something similar to the following:

```
body* other;

for (i=0 ; i<nstars ; i++) {
  for (j=(i+1) ; j<nstars ; j++) {
    if(owner[i].rank == myrank) {

      if(owner[j].rank == myrank)
        other = owner[j].star;
      else
        communicate(myrank, j, other);

      fun(owner[i].star, j, other);
    }
  }
}
```

Where `communicate(int rank, int index, body* buffer)` communicates `*owner[j].star` from node `owner[j].rank` to node *rank* at memory address `body* buffer`. This control structure prevents the segmentation faults that occur when a node tries to access a body that it does not control. Note that the actual C call is more complicated then the code above due to the complicated nature of MPI syntax. Note also that `fun(body* star1, body* star2)` is a *serial* function; our control structure very nearly allows us to sweep the communication under the rug.

3.1.2.3 Simulation 3: Serial and Relativistic

Let's now step back into the world of serial programming and construct a special relativistic simulation. Several changes need to be made to the control structures of the program in order to implement the finite speed of the gravitational

interaction.

We will again represent a body with the type-defined structure `body`. This structure will now have two roles: it will track the physical parameters of the body (as it did in the classical case), and it will store information that is needed to implement the information lag. Let us first analyze the portion of the type that tracks physical data:

```
typedef struct bodies {
  float x[3];           --- The position of the body.
  float p[3];           --- The momentum of the body.
  float mass;           --- The mass of the body.
  float gamma;          --- The relativistic correction factor.
  ...
  (various other fields)
} body;
```

We track the physical parameters of the star, such as position, momentum, and mass, as we did in our classical approach. Notice that we must now keep track of the relativistic correction factor γ.

In order to understand the other fields, we must first turn our attention to the relativistic `owners` structure:

```
typedef struct {
  body* star;            --- The star that is being tracked.
  body* others[nstars];  --- The time-delayed linked list of every other star in the s
} owners;
```

Note that `owners` still has a `body* star` field, which is still used to keep track of a star. It also has a mysterious `body* others[nstars]`, which is an array of `body` pointers of length `nstars`. This seems to allows us to have each entry in the owner array point to a star, as well as to every other star in the simulation. This may seem like spurious control: If we are already keeping track of every star via the owner array, then why would we need to have *each* entry in the owner array keep track of *every other star*? Well, in order to implement

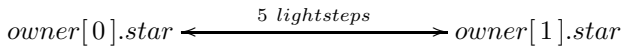

Figure 3.2: Two bodies, 5 light-steps apart

a time delay for gravitational interactions, each star needs to know where every other star *once was* rather than where every other star *is now*. We solved this problem by creating linked lists of type `body`, which are tracked by the `body* others` field.

Just how these linked lists solve the problem is best investigated with an example. Consider a simulation with two bodies: `owner[0].star` and `owner[1].star` (Figure 3.2). Let's say that they are five light-steps apart, where a light-step is the distance that light would travel in one timestep of the simulation. `owner[0].star` and `owner[1].star` should thus interact after five iterations, as their mutual gravitational interaction propagates at the speed of light. In order to accomplish this, the `owner[0]` entry establishes a linked list of type `body` that is five entries long, and `owner[1]` does the same (Figure 3.3). `owner[0]` and `owner[1]` store these linked lists in `owner[0].others[1]` and `owner[1].others[0]`, respectively. The first entry in the `owner[0].others[1]` linked list corresponds to the position of `owner[1].star` that `owner[0].star` will see after 1 iteration, and vice versa. The second entry will correspond to the data seen after two iterations, the third entry to the data seen after three iterations, and so on. In our example, then, the first four entries in the `owner[0].others[1]` and `owner[1].others[0]` linked list will be *placeholders*, and the fifth entry will be the physical data of `owner[1].star` and `owner[0].star`, respectively.

Why? Well, whatever propagator we were using in our serial classical example would simply have each star `owner[i].star` look at every other star `owner[j].star`, then apply some Newtonian physics. This implies an infinite speed of gravitational interaction: every star knows where every other star is during a given timestep. In special relativity, however, the gravitational interaction has a finite speed. In order to implement this, we have our propagator have each star owner[i].star look at every other star *via owner[i].others[j]*. If

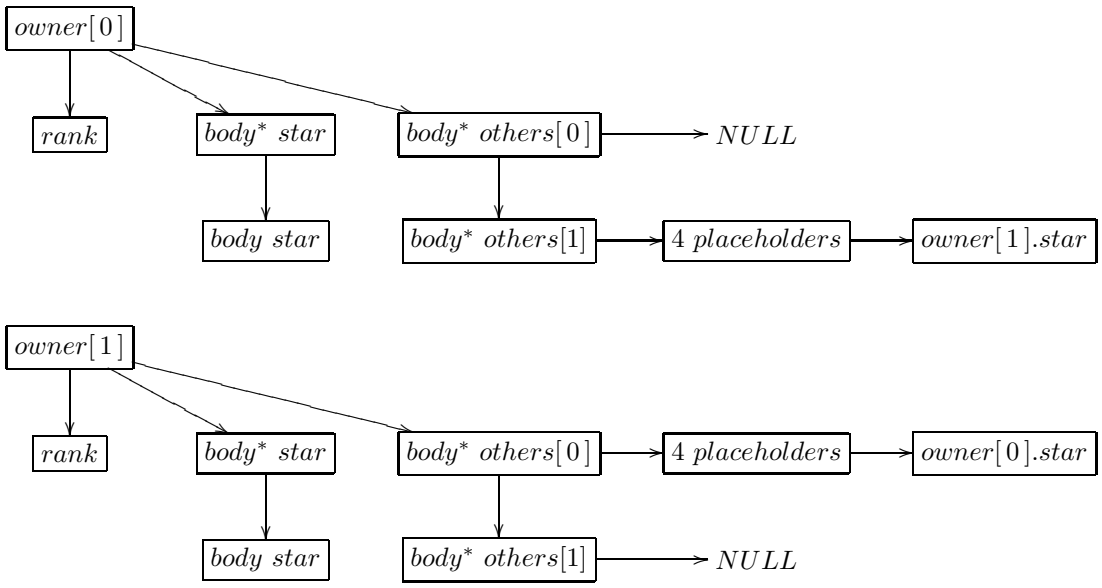

Figure 3.3: The layout of the two owner entries.

the first entry in the linked list is physical, the stars interact; if not, they don't. After each iteration, the list is moved up by one entry, and the physical data `owner[j].star` is once again tacked on to the end of the list. The information lag is implemented by the length of the linked list: `owner[0].star` will see `owner[1].star`, which is 5 light-steps away, after 5 timesteps.

We have not yet established how we create these linked lists. The trick is in the extra fields of the `body` type:

```
typedef struct bodies {
  int physical;         --- If the body type is being used in its
                            linked list capacity, this flag
                            indicates whether the linked
                            list entry is a physical star
                            or a placeholder.
  float x[3];           --- The position of the body.
  float p[3];           --- The momentum of the body.
  float mass;           --- The mass of the body.
```

```
  float gamma;          --- The relativistic correction factor.
  struct bodies* prev;  --- Used for establishing a doubly-linked list of bodies.
  struct bodies* next;  --- Used for establishing a doubly-linked list of bodies.
} body;
```

The `struct bodies* prev` and `struct bodies* next` fields are used to establish the linked lists. The `int physical` field determines whether an entry in the linked list contains physical data or is a placeholder, with a value of `1` signifying physical data and a value of `0` signifying a placeholder. Note that the `body` type is used to represent *physical bodies* and *linked lists of data*. Physical stars always have the `int physical` field set to `1` and ignore the `struct bodies* prev` and `struct bodies* next` fields.

We can now easily understand the control structure of a special relativistic simulation with `nstars` stars in an arbitrary configuration. A single entry in the owner array, `owner[i]`, is shown in Figure 3.4. Each entry in the owner array fills its `body* others` array with the physical data of every other star in the simulation, properly time-delayed. Notice that this requires far more memory than the serial-classical case - the linked lists can grow very, very large as the stars move apart in space, and there are $nstars \cdot (nstars - 1)$ lists to keep track of. This raises some issues with modeling real systems, which we will explore in §3.3.

3.1.2.4 Simulation 4: Parallel and Relativistic

Now we'll try to put it all together with a parallel, relativistic simulation. Our `body` type is the same as that for a serial relativistic simulation:

```
typedef struct bodies {
  int physical;         --- If the body type is being used in its
                            linked list capacity, this flag
                            inidicates whether the linked
                            list entry is a physical star
                            or a placeholder.
  float x[3];           --- The position of the body.
  float p[3];           --- The momentum of the body.
```

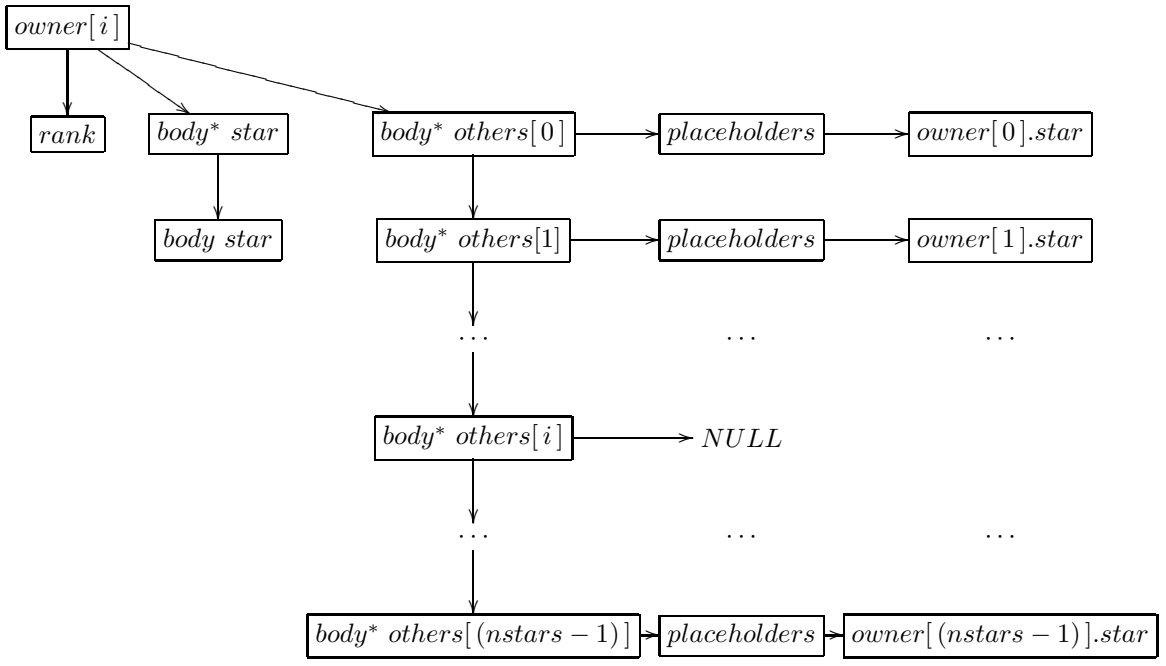

Figure 3.4: An entry in the owner array.

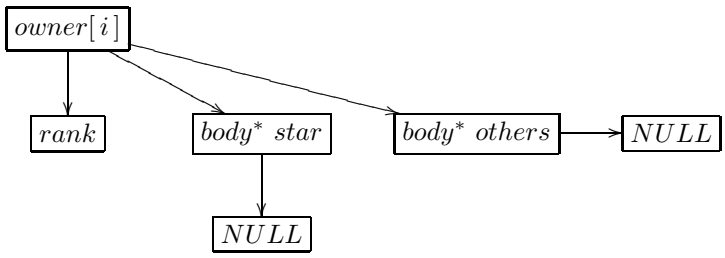

Figure 3.5: An entry in the owner array for which the owned star does not reside on the current processor.

```
  float mass;           --- The mass of the body.
  float gamma;          --- The relativistic correction factor.
  struct bodies* prev;  --- Used for establishing a doubly-linked list of bodies.
  struct bodies* next;  --- Used for establishing a doubly-linked list of bodies.
} body;
```

Our owners type now contains both the int rank field and the body* others[nstars] field:

```
typedef struct {
  int rank;              ---  The rank of the processor that body* star exists on
  body* star;            ---  The star that is being tracked
  body* others[nstars];  --- The time-delayed linked list of every other star in the
                               simulation.
} owners;
```

The behavior of the body* others[nstars] field is more complex in the parallel case. If the star that an entry in the owner array is tracking resides on the current node, then the owner entry will have the same character as it would in a serial simulation (Figure 3.4). If the star being tracked by the owner array entry does not reside on the current node, then there is no need to track the star (because it does not exist locally), and there is subsequently no need to track its relationship to other stars (Figure 3.5).

All of this control may seem cumbersome, but it reduces the difficulty of performing operations on the stars enormously. Consider the following code

sample:

```
1:  init();
2:
3:  for(i=0 ; i<nstars ; i++) {
4:    if(owner[i].rank == myrank) {
5:      for(j=0 ; j<nstars ; j++) {
6:        if(i != j) {
7:          if(owner[i].others[j]->physical == 1)
8:            move(owner[i].star, owner[i].others[j])
9:        }
10:     }
11:   }
12: }
```

Analyzing this simplified code sample line-by-line will lend some insight into how the program really operates:

1. In line 1, the simulation is initialized. This function instructs each node to malloc its portion of the total bodies. The bodies are then allgathered onto every processor. This allows each node to fill in the `body* others[nstars]` field for bodies that it controls. After this function has terminated, then, every node controls an even number of bodies, and each of these bodies has access to properly time-delayed linked lists of the physical data of every other body in the simulation.

2. In line 3 each node begins iterating over every star in the simulation.

3. In line 4 each node checks whether or not it owns `owner[i].star`.

4. In line 5 the node that owns `owner[i].star` begins a nested iteration over every star in the simulation.

5. In line 8 the node that owns `owner[i].star` moves `owner[i].star` according to the information stored in `owner[i].others[j]`, provided that `owner[i].others[j]` is not a placeholder (checked by line 7) and that the star is not trying to move itself (checked by line 6).

Note that this is an oversimplification. Executing the code in this fashion would cause body 0 to be moved by body 1, followed by body 2, and so forth, rather than calculating the net force on body 0 and moving it accordingly. We must now move on to an in-depth analysis of the functions of the program and their interactions with the control structures so that such problems can be avoided.

3.1.3 User-Callable Functions

In this section we define and analyze each of the user-callable functions of the simulation:

1. `void init()`
2. `void look()`
3. `void iterate()`
4. `void write_out()`

There are many more functions within the program; these four are effectively "wrappers" that perform the four major tasks of the simulation:

1. `init()` initializes the simulation.
2. `look()` locally updates the `body* others` field for every entry in the owner array.
3. `iterate()` has each node move its the stars around.
4. `write_out()` has each node write two dimensions of position and velocity data to a file.

3.1.4 void init()

`void init()` must be called at the beginning of any simulation. It accomplishes four tasks: MPI variable initialization, physical data initialization, information lag linked list initialization, and the communication of the initialized data throughout the cluster.

3.1.4.1 MPI variable initialization:

`int myrank`, the global variable that stores the rank of the node, is initialized via the following call:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

`int nprocs`, the global variable that stores the total number of nodes in the simulation, is initialized via the following call:

```
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

3.1.4.2 Physical Data Initialization:

Each processor initializes the local array `owners* my_owner[nchunks]`, where `nchunks` = `nstars/nprocs`. Each entry in the array points, via `my_owner[i].star`, to a mallocced body, and has the `int rank` field assigned to `myrank`. The star's physical parameters are initialized. The linked list aspects of the body type are ignored for these stars, as they are physical.

3.1.4.3 Communication

The local `owners my_owner[nchunks]` arrays need to be gathered onto every node so that each node knows the rank information of every star. This is accomplished via an allgather call. After the call, `owner[i]` will be equivalent on all processors. If a node wishes to operate on a star `owner[i].star`, it simply checks whether `owner[i].rank == myrank`, and performs the operation if the equality is true.

3.1.4.4 Information Lag Linked List Initialization

Every entry in the `owner` array initializes its `body* other` array with properly time-delayed linked lists of the physical data of every other star in the simulation. Note that `owner[i].others[i]` is set to `NULL`, as there is no need for a body to track itself.

3.1.5 void look()

A call to `look()` adds elements to the information lag linked list stored in the `body* owner[i].others` array. `look()` regathers every star onto each node via an allgather call. This allows each node to update the `body* others` array for stars it controls. The update itself is complicated, and the mechanics of it need not be discussed.

3.1.6 void iterate()

`iterate()` moves the stars in the simulation around according to special relativistic physics. This is accomplished via nested iterations over the `owner` array and the `body* others` field of the owner array:

```
float xs[nstars][3];
float ps[nstars][3];
float r;

for(i=0 ; i<nstars ; i++) {
   for(j=0 ; j<NDIMS ; j++) {
     xs[i][j] = owner[i].star->x[j];
     ps[i][j] = owner[i].star->p[j];
   }
}

for(i=0 ; i<nstars ; i++) {
  if(owner[i].rank == myrank) {
    for(j=0 ; j<nstars ; j++) {
      if (i != j) {
if (owner[i].others[j]->physical == 1) {
  r = Calc_displacement(owner[i].star->x, owner[i].others[j]->x);
  for (k=0 ; k<(NDIMS-1) ; k++) {
    dS[k] = owner[i].others[j]->x[k] - owner[i].star->x[k];
            ps[i][k] +=
      owner[i].star->gamma *
```

```
      ((G*owner[i].star->mass*owner[i].others[j]->mass)/(r*r))
               * (dS[k]/r) * timestep;

    xs[i][k] +=
       (owner[i].star->p[k] * timestep)/owner[i].star->mass;
  }
}
else {
  for ( k=0 ; k<(NDIMS-1) ; k++) {
    xs[i][k] += (owner[i].star->p[k] * timestep) /
                owner[i].star->mass;
  }
}
      }
    }
  }
}
for(i=0 ; i<nstars ; i++) {
  if(owner[i].rank == myrank) {
    for(k=0 ; k<NDIMS ; k++) {
      owner[i].star->x[k] = xs[i][k];
      owner[i].star->p[k] = ps[i][k];
    }
  }
}
```

`iterate()` begins by iterating over the owner array. If a node owns the star of the current owner entry, `iterate()` instructs that node to begin a nested iteration over the `body* others` array. If the current entry in the `body* others` array is physical, `iterate()` has the two stars interact, and the resulting change in momentum is added to the current entry of the `ps` array. The position is then changed by this new momentum, with the result stored in the `xs` array. If the current entry in the `body* others` array is not physical, the momentum is stored

as is in the `ps` array, and the position is updated with the same momentum and stored in the `xs` array.

Note that the position and momentum of `owner[i].star` are not immediately updated. The updated position and momentum are rather stored in a separate array, and the physical parameters of `owner[i].star` are assigned to this array only after all of the iterations have taken place. This prevents the problem we saw in §3.1.2.4 where stars would be see every other star "piecewise" rather than as a collective whole exerting a net force.

3.1.7 void write_out()

`void write_out` writes two dimensions of position and velocity data to a file. The first time `write_out` is called, each node creates a file named `figure_#`, where `#` is the rank of the node. Each node then writes the position data to `figure_#` in the following format:

```
star 0 position 1, star 0 position 2, star 0 velocity 1, star 0 velocity 2
star 1 position 1, star 1 position 2, star 1 velocity 1, star 1 velocity 2
...
star n position 1, star n position 2, star n velocity 1, star n velocity 2
```

If `write_out()` is called again, each node appends its position data to `figure_#` in the above format. Users can write a graphing program to plot this data.

3.2 A Sample Simulation

We can use the four functions `init()`, `look()`, `iterate()`, and `write_out()` to build a simulation:

```
int main(int* argc, char* argv[]) {
  int i;


  MPI_Init(argc, &argv);
```

```
  initrand();

  init();

  MPI_Barrier(MPI_COMM_WORLD);

  for(i=0 ; i<NITER ; i++) {
      look();
      MPI_Barrier(MPI_COMM_WORLD);
      iterate();
      MPI_Barrier(MPI_COMM_WORLD);
      write_out();
      MPI_Barrier(MPI_COMM_WORLD);
  }

  cleanup();

  MPI_Barrier(MPI_COMM_WORLD);

  MPI_Finalize();

  return 0;
}
```

We begin by calling `MPI_Init()`, which allows use to call MPI functions, and then calling `initrand()`, which allows us to use C's random number generator. We then initialize the simulation with `init()`, and set up an `MPI_Barrier`, which instructs all nodes to wait at the barrier until every node has reached it – this prevents synchronization bugs. Next we call `look()`, `iterate()`, and `write_out()`, with `MPI_Barriers` between calls, `NITER` times, where `NITER` is a previously defined parameter that determines the number of iterations we will execute. We then call `cleanup`, which makes sure that any memory we have allocated is properly freed. Finally, we call `MPI_Finalize`, signifying that we

no longer need to make MPI calls, and return 0, indicating that the program executed successfully.

If this code were executed, the user would find *nprocs* `figure_#` files in their working directory. The data in these files is the final result of the simulation — plotting and analyzing the data is a separate issue.

3.3 Further Work / Issues

Our simulation is far from finished. While the prototype we currently have is adequate for a qualitative investigations, the errors introduced by our integration techniques bar us from rigorous quantitative analysis. There are also some severe memory issues that need to be addressed before large systems can be analyzed. The parallel capabilities of the simulation could also be improved.

3.3.1 Computational Error

Discrete approaches to the N-body problem are subject to computational errors. Energy, for instance, is not conserved in our simulation. This could be rectified by replacing our current iteration procedure with a symplectic procedure. Symplectic integrators produce systems whose energies vacillate about a constant value (rather than varying unpredictably), which is about as good as a discrete approach to a continuous problem can accomplish. Please see Kinoshita et al.'s 1991 article on the applications of symplectic integrators to dynamical astronomy for more information.

The discrete nature of our simulation adds constraints to the sizes and timescales we can work with. Discrete approaches to continuous problems can not handle strong interactions. If two bodies are very close and interact strongly, their momenta will skyrocket, and over one timestep they will fly apart. They will then be "lost," because subsequent interactions will be too weak to counteract the initial interaction. We must integrate over small timesteps and keep stars adequately spaced to avoid this issue. This is extremely difficult in practice, Nb many stars are thus erroneously ejected from the simulation.

3.3.2 Memory Issues

As was previously mentioned in §3.1.2.3, the `body* others[nstars]` arrays tracked by the owner array can become gargantuan. This consumes a lot of memory, limiting the number of stars that can be reasonably included in a simulation.

More troublesome is the fact that the memory used by the simulation is directly proportional to the spatial dimension of the simulation. The length of an entry in the `others` linked list is determined by the spatial separation of the two stars, as each each lightstep of separation requires its own unique entry. This caps the size of the simulation at the outset - very large simulations require memory that is simply unavailable. This makes simulating real systems (which have dimensions on the many thousands of lightyears scale) very difficult.

Worse still is the implication for small systems. We already stated that stars interacting in close quarters will most likely acquire large momenta and go flying off to infinity. Their spatial separation grows larger and larger, eating more and more memory until it is all consumed, causing the program to hang or crash. We will henceforth refer to this as the *spatial memory bug*.

This spatial memory issue is not limited to small systems. If for any reason any star in any simulation is ejected from the rest of the stars, it will eventually crash the simulation. Now, real systems are very immense, and thus will not be interacting particularly strongly. This means that, in order to see results, we must either increase the length of a single timestep, or increase the number of iterations. Either of these changes greatly increases the probability that a star will be ejected: increasing the length of the timestep leads to stronger interactions, which can eject stars, and adding iterations provides more chances for stars to be ejected. The end result is that *our simulation can not simulate real systems at this time*. This is a solvable problem, but not on the timescale of this project.

3.3.3 MPI-related issues

According to our current control structure, each node is initialized with the same amount of stars, and controls the stars throughout the simulation. This

could be improved upon with an *adaptive mesh*. If nodes were able to exchange control of stars, a more efficient distribution could be reached. This would result in much faster execution times for simulations utilizing a large number of nodes. Furthermore, if nodes were able to release ownership of stars, they could stop tracking stars that have been ejected, potentially solving the spatial memory issue.

Chapter 4

Results

We will now simulate a few simple systems to demonstrate the capabilities and limitations of the simulation. The first two simulations are designed to give a clear picture of how the information lag works. The third simulation demonstrates the potential of the simulator.

4.1 System 1: 2 bodies, 2 light-steps apart

We will start with the simplest possible case: two stationary bodies of equal mass which are 2 light-steps[1] apart from each other. This simulation will clearly show the information lag in action. The masses and timestep were chosen so that the interaction would be visible over short timescales.

We can see the bodies at initialization in Figure 4.1. After one iteration (Figure 4.2) the bodies still have not interacted, as each star's gravitational information has not yet reached the other star, but has only made it to the origin. After another iteration, the gravitational information reaches each star, and they rapidly move towards each other (Figure 4.3). They continue to move toward each other (Figure 4.4), move past each other (Figure 4.5), and finally fly off towards infinity.

This is qualitatively what we should expect. The stars would not, in reality, fly off to infinity, but would rather collide at the origin. If they did not collide, and the simulation were continuous, they would oscillate about the origin. The

[1] Recall that a light-step is the distance that light would travel in one iteration of the simulation

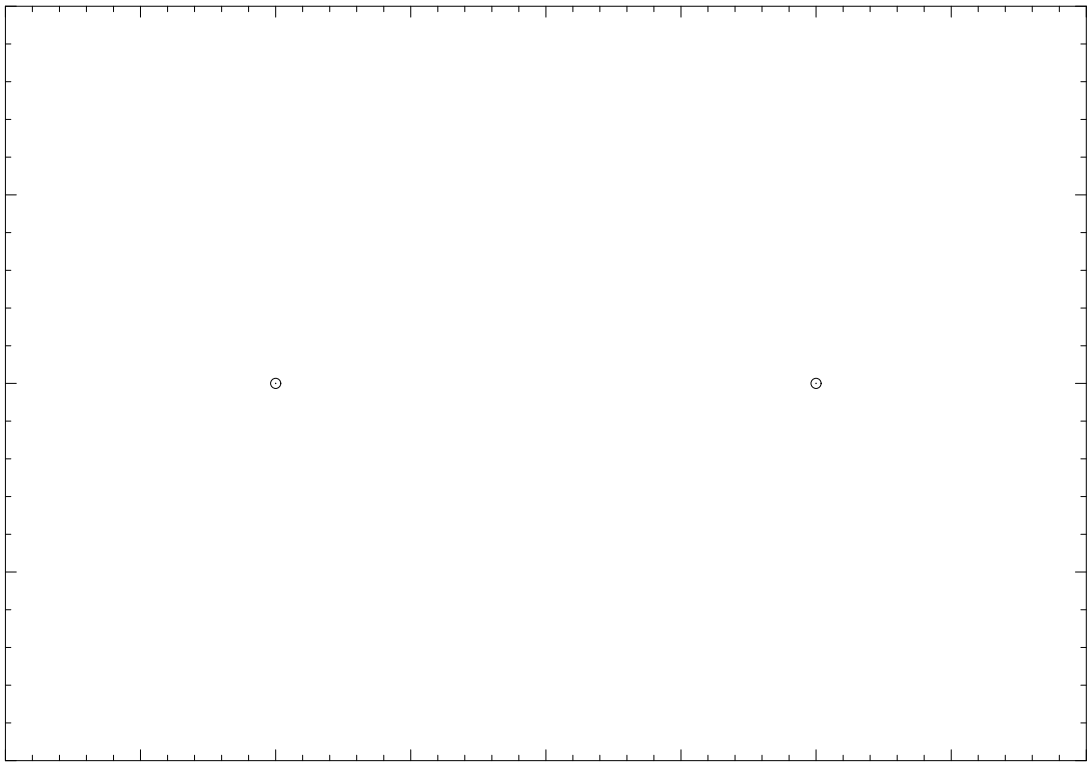

Figure 4.1: The two bodies at initialization. They have a displacement of two light-steps and no initial momentum.

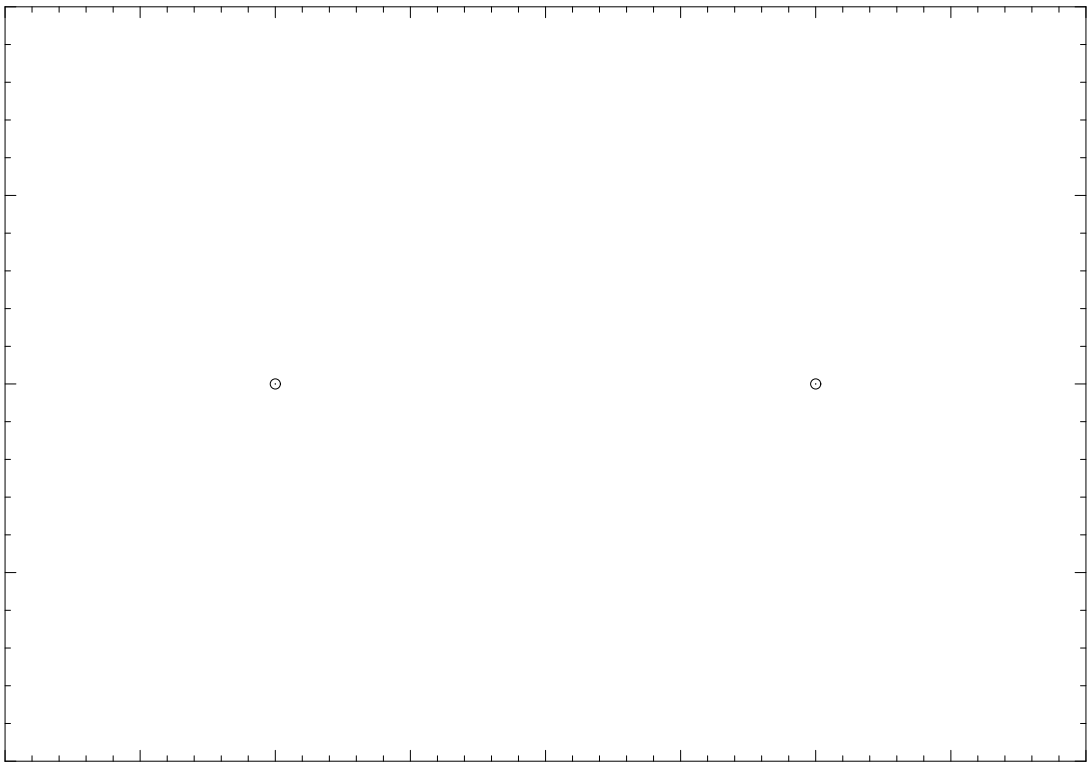

Figure 4.2: The two bodies after one iteration. Note that they have not yet interacted.

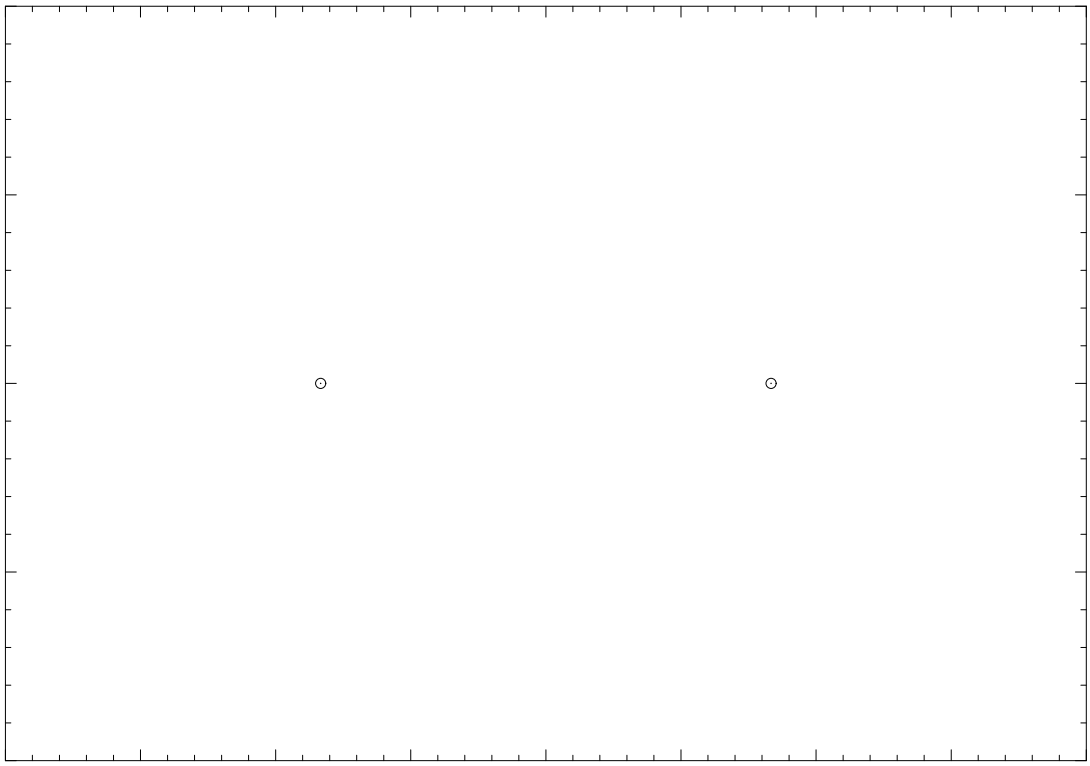

Figure 4.3: The two bodies after two iterations. The stars have each received the gravitational information from the other star, causing them to interact.

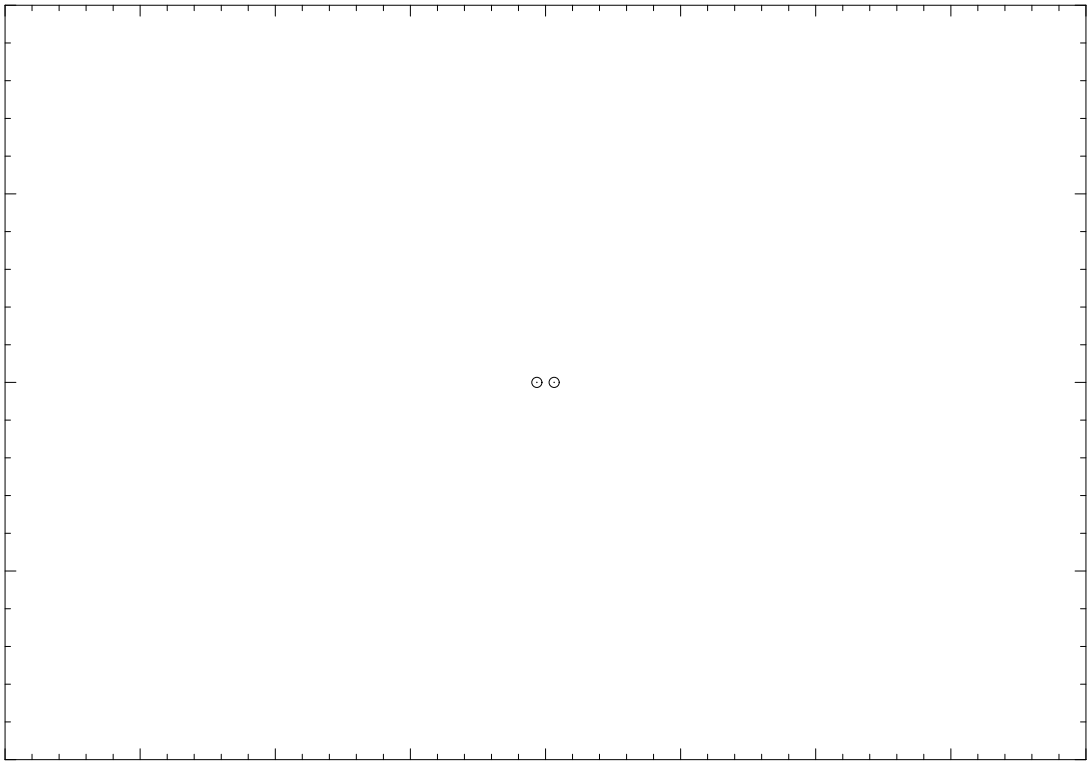

Figure 4.4: The two bodies after 5 iterations.

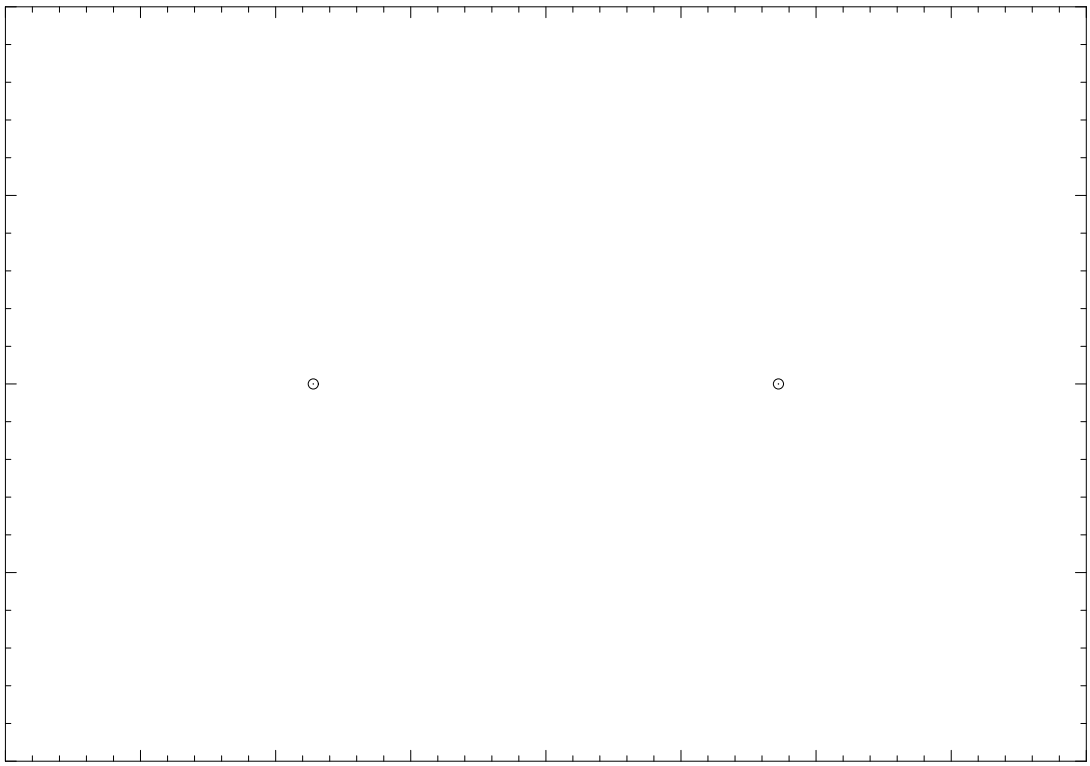

Figure 4.5: The two bodies after six iterations. Note that they have passed each other.

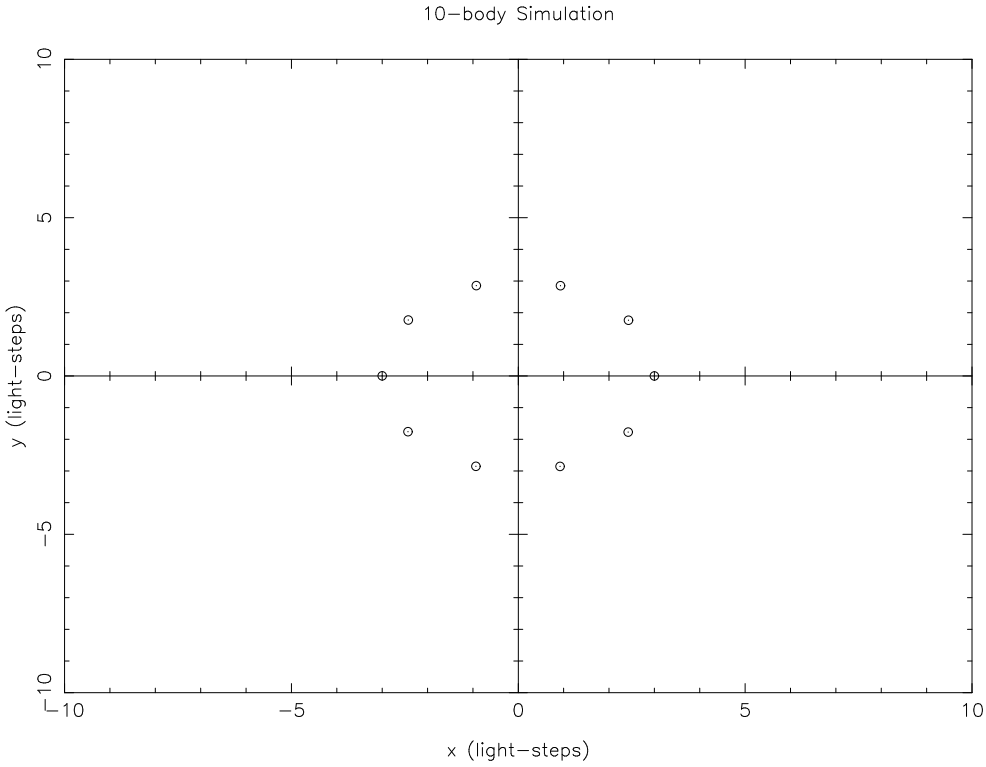

Figure 4.6: Simulation 2: The ten bodies.

stars acquire such a large velocity due to discrete calculation error. Reducing the masses or decreasing the timestep would result in more continuous behavior, and potentially prevent the stars from acquiring such large momenta.

If we were to continue to run the simulation indefinitely, the stars would continue towards infinity, eventually crashing the simulation, due to the spatial memory bug.

4.2 System 2: 10 bodies

Next we have a simulation with 10 bodies arranged in a circle of radius 3 light-steps about the origin (Figure 4.6). These bodies have no initial momentum. Note that iteration 1 and 2 (Figure 4.7) have the same character as the initialization figure. Stars adjacent to each other will see each other right away (they are less than a lightstep apart), which would lead us to expect them to

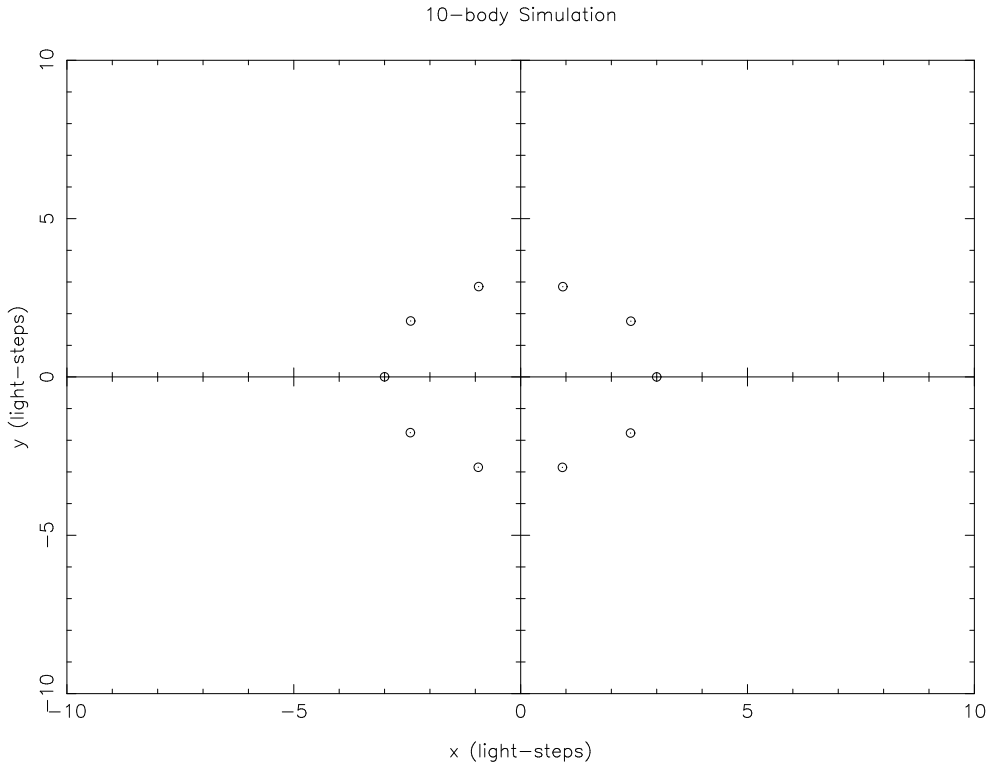

Figure 4.7: Simulation 2: The ten bodies after two iterations. Note that the stars have not moved.

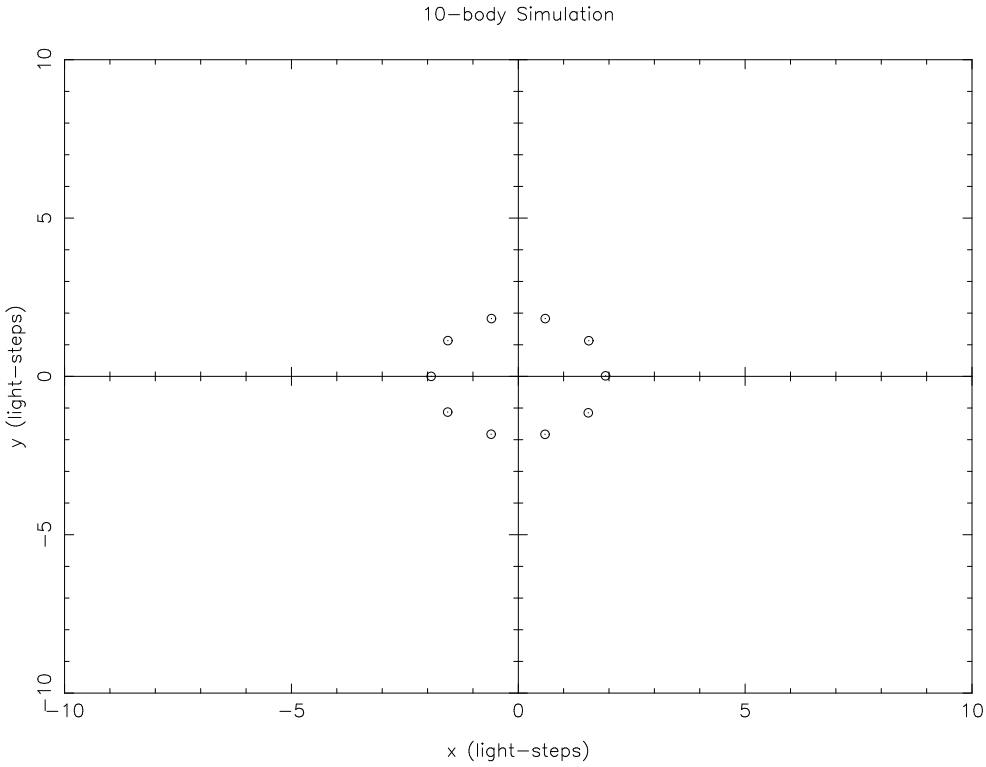

Figure 4.8: Simulation 2: The ten bodies after three iterations. Note that the stars are now collapsing towards the origin.

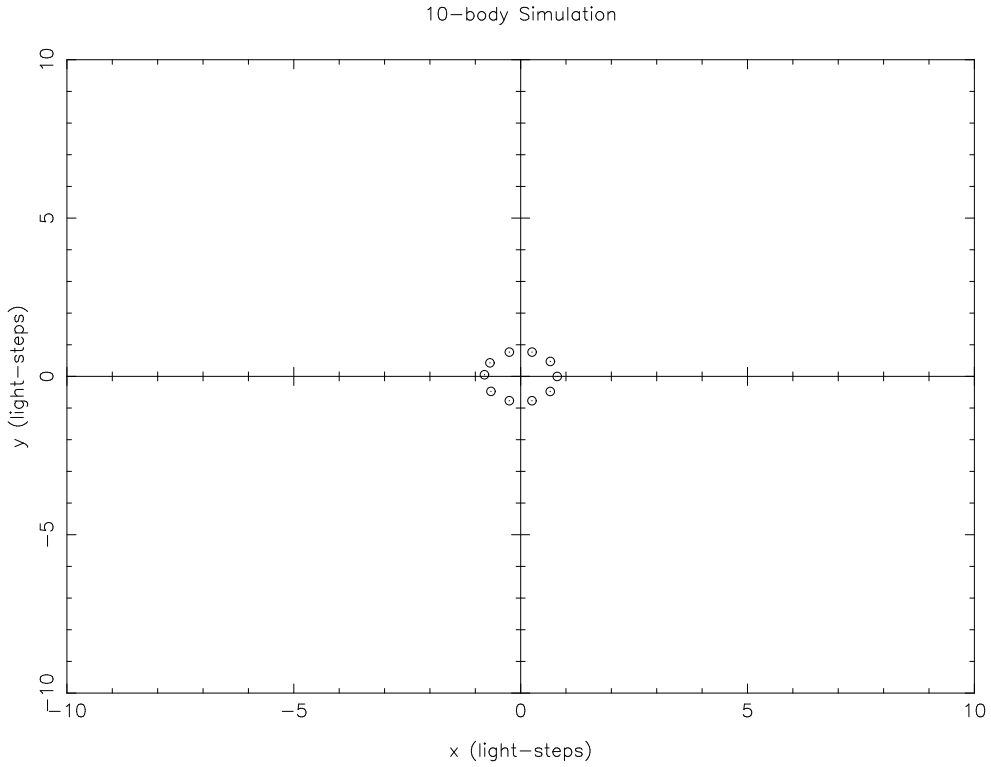

Figure 4.9: Simulation 2: The ten bodies after four iterations. Note that the stars have continued to collapse towards the origin.

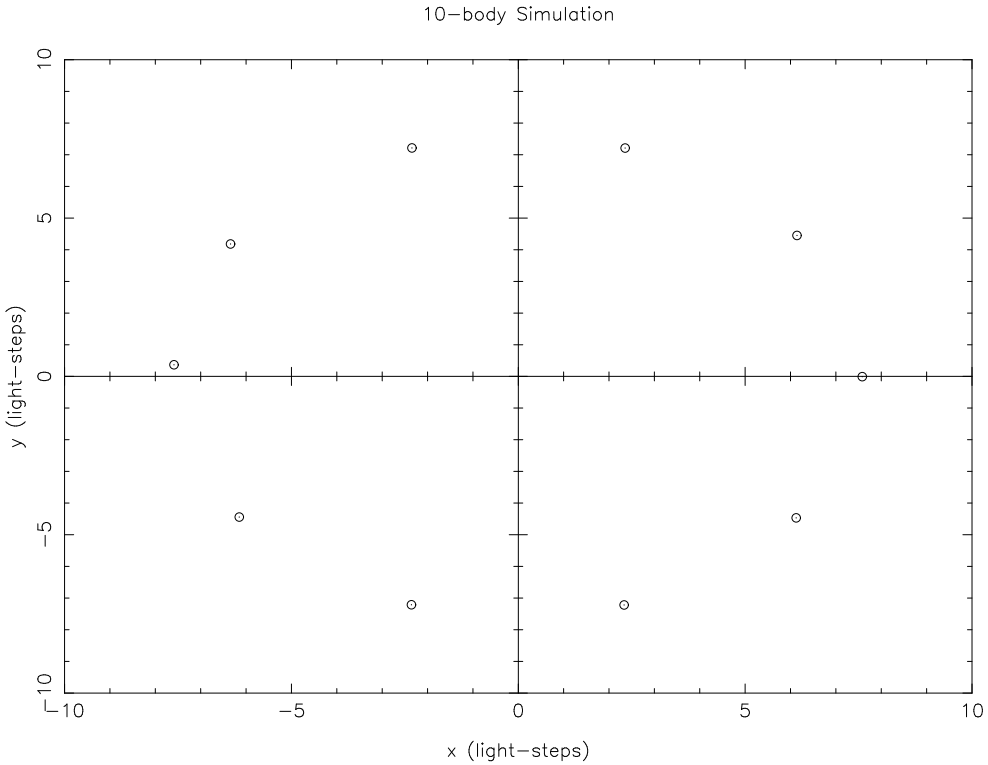

Figure 4.10: Simulation 2: The ten bodies after six iterations. Note that the stars are moving away from the origin.

interact. They actually are interacting, but the net force of all adjacent stars is fairly small. The force on the star will become significant when it is interacting with all of the other stars, which should be seen on the third iteration. Indeed, we see in Figure 4.8 that this is the case; the stars collapse towards the center. This collapse continues (Figure 4.9) until the stars reach the origin, at which time they explode out to infinity (Figure 4.10) due to discrete calculation error.

Again, if we allowed the simulation to run indefinitely, it would crash due to the spatial memory bug. This crash would occur after many fewer iterations than it would in Simulation 1, as we are now tracking 10 stars, and thus have 90 `body* others` linked lists to keep track of, rather than 2.

4.3 System 3: 101 Bodies

We will now demonstrate a simulation of 101 bodies. 100 of these bodies are initialized in ten concentric rings of radii 10 light-steps, 20 lightsteps, ..., 100 lightsteps around a body located at the origin, which is 3000 times as massive as the other bodies (Figure 4.11). Bodies are now represented as arrows which point in the direction of the velocity of the body. After a few iterations, many of the bodies fly off towards infinity due to their large initial velocities (Figure 4.12), but the inner three rings seem to be somewhat stable. After many more iterations, we begin to see pulsation behavior, in which the center-most stable ring collapses to a small radius about the central body, rotates rapidly about it, and then expands outwards into what may be another semi-stable orbit - unfortunately, the timescale of the simulation is too short to tell. This process repeats a few times before the simulation ends. This is, unfortunately, very difficult to show with two-dimensional spatial plots; Figures 4.13, 4.14, and 4.15 provide the best picture possible.

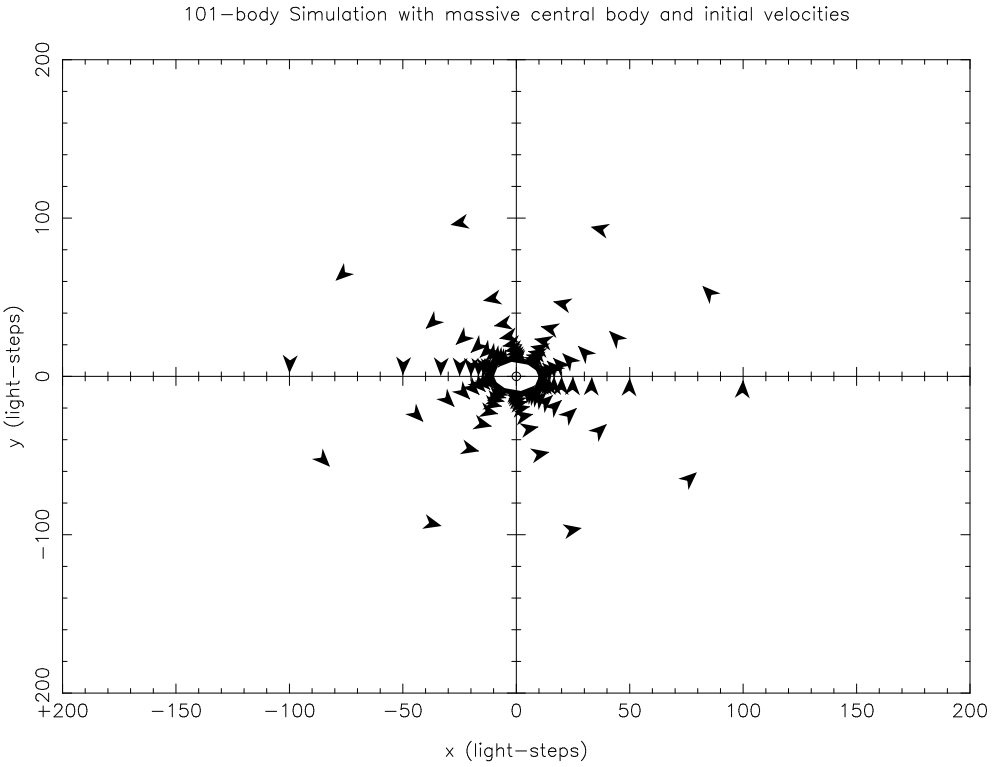

Figure 4.11: Simulation 3: The 101 bodies at initialization. Note that the arrows point in the direction of a body's velocity.

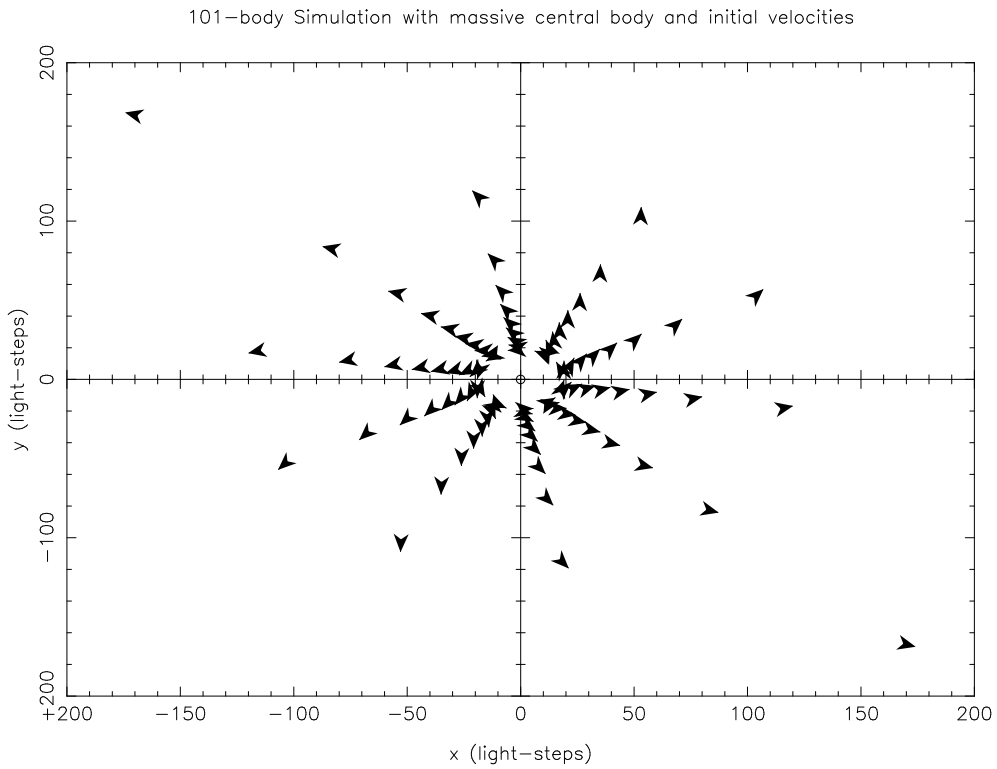

Figure 4.12: Simulation 3: The 101 bodies after 23 iterations. Note that many bodies in the outer rings are moving out towards infinity, whereas the bodies in the three central rings seem somewhat stable.

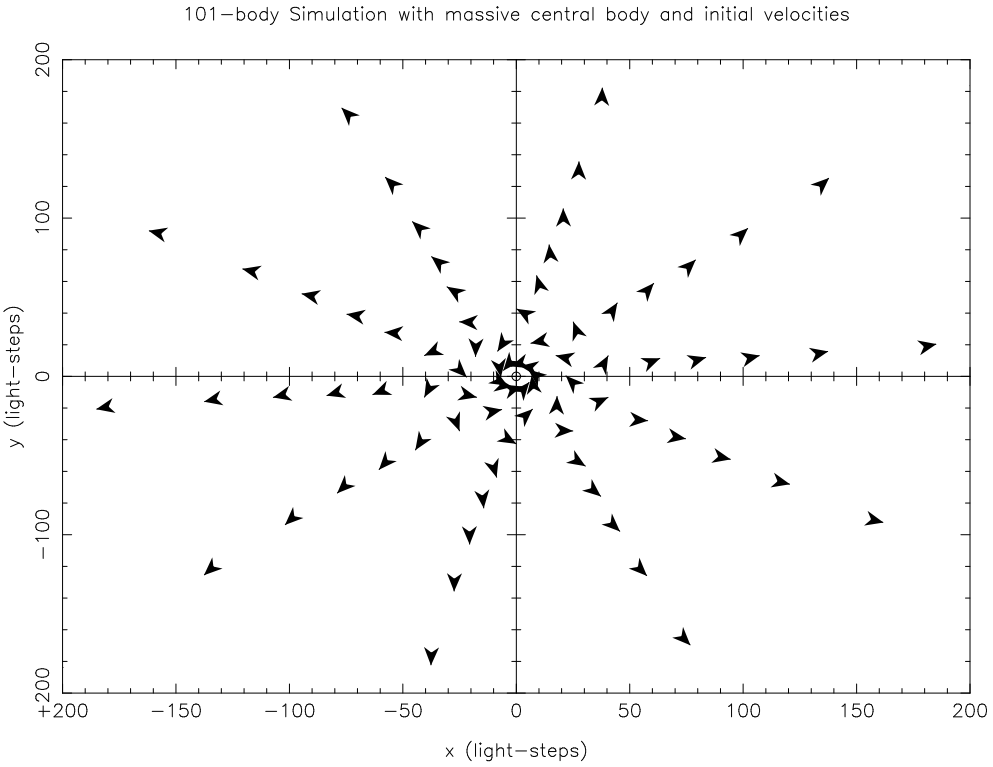

Figure 4.13: Simulation 3: The 101 bodies after 56 iterations. Note the three stable central rings and the rapidly rotating innermost ring.

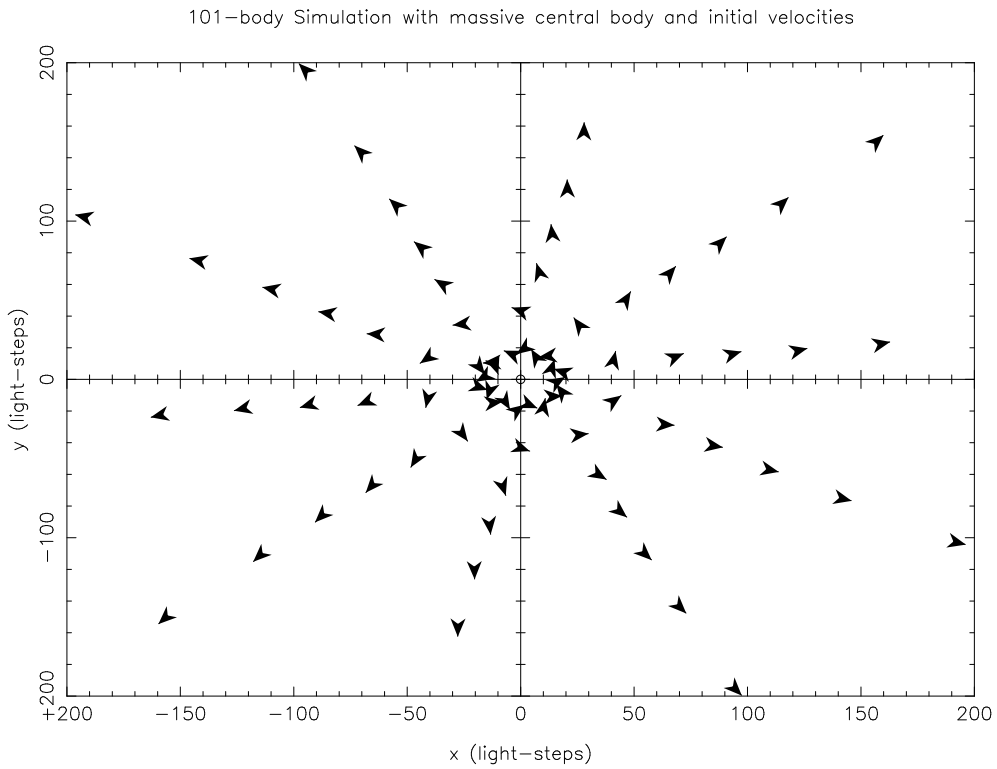

Figure 4.14: Simulation 3: The 101 bodies after 67 iterations. Note the innermost stable ring is beginning to move outwards, while the interstitial stable ring is beginning to move inwards.

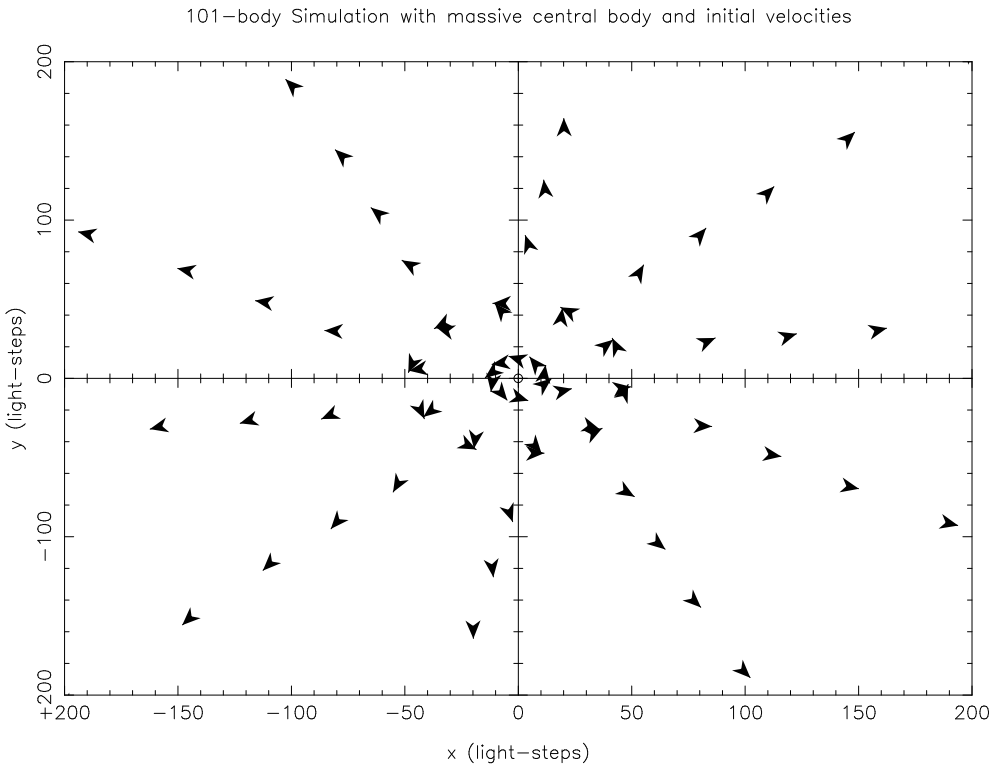

Figure 4.15: Simulation 3: The 101 bodies after 88 iterations. Note the innermost stable ring from Figure 4.13 is now the outermost stable ring. The interstitial ring from Figure 4.13 is now the rapidly rotating innermost ring. Note that one body seems to have been ejected from the innermost ring, most likely due to computation error.

Chapter 5

Conclusion

We have suggested that the investigations of galactic and nebular dynamics that led to the theory of dark matter were classical. This, coupled with the work of Cooperstock and Tieu (2005) in general relativistic galactic dynamics, has motivated us to produce an easily understandable special relativistic N-body simulation. The simulation is not yet capable of simulating a realistic system. We have, however, shown that we obtain qualitatively accurate results for a few simple systems.

5.1 Future Work

During the next few months we will address the various problems with the simulation, such as the spatial memory bug, so that we may simulate more realistic systems. This will allow us to produce quantitative rotation curves for galactic systems, which will in turn allow us to determine whether including special relativistic effects in N-body simulations significantly improves their correspondence to the physical world.

5.2 Acknowledgments

I would like to thank Chris Martin, my thesis adviser, for all of the hours of help he happily provided, for his patience with the project, and for generally being a fantastic guy.

Bibliography

Bergh, V. D. 1999, Publications of the Astronomical Society of the Pacific, 111, 657

Cooperstock, F. I. & Tieu, S. 2005, Astrophysical Journal

Fuchs, B. & Phleps, S. 2006, New Astronomy, 11, 608

Garfinkle, D. 2006, Classical and Quantum Gravity, 23, 1391

Kinoshita, H., Yoshida, H., & Nakai, H. 1991, Celestial Mechanics and Dynamical Astronomy, 50, 59

Menzies, D. & Mathews, G. J. 2006, Astrophysical Journal

Ostriker, J. & Peebles, P. J. E. 1973, The Astrophysical Journal, 186, 467

Pacheco, P. S. 1997, Parallel Programming with MPI (Morgan Kaufmann Publishers, Inc.)

Smith, S. 1936, Astrophysical Journal, 83, 23

Springel, V., White, S. D. M., Jenkins, A., Frenk, C. S., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J. A., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., & Pearce, F. 2005, Nature, 435, 629

Zwicky, F. 1933, Helvetica Physica Acta, 6, 110

—. 1937, Astrophysical Journal, 86, 217