Far From the Tree: Newtonian Gravitational N-Body Simulation

Jonathan Robey, Dov Shlachter, Ryan Marcus

April 1, 2009

Team 55

Los Alamos High School

Teacher:

Lee Goodwin

Mentor:

Robert Robey

Contents

Executive Summary

The purpose of this project is to create a three dimensional model that can can accurately predict the behavior of a Newtonian N-Body system, subject to movement exclusively through the force of gravity. This was achieved using a first order symplectic method. After having tested the simulation against various test problems, it has been concluded that the model calculates with a high degree of accuracy. However, certain test conditions can lead to a degeneration of the model, often to such a degree that the test run must be abandoned. This could be dealt with by using a higher order method, but these were deemed too complex to implement. The model could be improved with the addition of a run-time display, as the potential applications for the model extend to the world of realistic video games, or a workaround scheme to allow the Newtonian scheme to be used in situations that would be more accurate with the use of Relativity, done for the sake of computational efficiency.

Two versions of the model were developed: one written in C, the other in Java. This was mostly done to see how the various languages, each coupled with a different method of parallelization, scaled, although absolute performance differences are also important.

1 Introduction

1.1 Problem Statement

Every object in the universe, and certainly every object in the solar system, causes a slight change in the hypothetically perfect elliptical path of the earth around the sun. As most of these objects are comparatively tiny or far away, they are of little concern. However, if a star was to pass through the solar system, our path would be driven erratically out of control. A slight difference in position or velocity could mean the difference between life and death. If this sort of situation were to occur, the exact fate of the earth would need to be predicted, either so that that fate could be prevented or so that the problem could be prudently ignored. Isaac Asimov raises this problem in his book *Nemesis*[2], the name by which this problem is now known. The simulation of this, however, is mostly a side benefit of the more interesting (to us) project: a study on data intensive computational models.

In order to continue the trend of increasing performance on their CPUs, manufacturers have started adding cores instead of simply ratcheting up clock speed. This has led to a problem for code developers: how does one take advantage of the extra cores? A minor revolution in code design has occurred due to the above question, coupled with a new emphasis on how well a given task scales over multiple cores.

One would think that the above-mentioned problem would scale easily: each processor is allowed to calculate new values for as many bodies as there are, divided by the number of available processors. However, that division, coupled with the needed communication between processors, often leads to decreased performance for a small number of bodies. This decrease is not strictly predictable for complex reasons that will be explained more in depth later on.

1.2 Objective

We propose to create an accurate, scalable, fully three dimensional program, capable of being tested and verified, that models the behavior of a system driven entirely by the influence of gravity. This model will be based on Newton's equation of gravitational attraction. The code will be written for high efficiency on multiple processors, in order to maintain accuracy and speed, and will output data to be examined after runtime.

1.3 Background

1.3.1 N-Body Codes

The applied problem was first made widely known by Isaac Asimov in one of his later books, *Nemesis*. The underlying premise is that small changes in a gravitational system will lead to vastly different results in the long term, with the assumption that some of the potential outcomes are benign to life on earth and that other outcomes will kill all life.

N-Body codes are the most extreme version of a Lagrangian model. In all Lagrangian methods, the model moves with the objects, instead of objects through the model. This is slightly obtuse, but makes more sense with specific examples. With a Lagrangian method, the values in each time-step refer to a moving object, instead of an immobile cell.

At the most extreme version of an N-Body code, no particles are abstracted, meaning that every particle of dust and sand and water would be its own object in the simulation. This is obviously impractical in this specific case, as it would require the enormously complex task of giving all objects a volume and compressibility. In the case of a gravitational N-body code, the normal abstraction level is at that of planets, moons, and asteroids.

1.3.2 Parallelization

The benefits and disadvantages of parallelization have been debated for years. In 1967, Gene Amdahl argued that the portion of any algorithm that could not be parallelized would severely limit the extent to which parallelization could occur.[1] This argument was based on the assumption that the problem size was constant. In 1988, that assumption was challenged by John Gustafson. Gustafson argued that effective scaling could be achieved if the problem size was not considered constant. He asserted that the purpose of having a large supercomputer was not to run a small simulation faster, but to run a much larger simulation as fast as the small simulation.[5]

2 Description

2.1 Mathematical Model

The model is based around the standard Newtonian gravitational model:[3]

$$F_g = G\frac{m_1 m_2}{r^2}$$

This model is inaccurate in situations where the side effects of Einsteinian Relativity become more apparant. This could occur when the velocity of an object approaches the speed of light, or when two objects whose gravitational attraction is non-negligible are sufficiently distant from each other that the time light would take to travel between them is a significant fraction of the time step.

Newton's equation does not directly solve for the change in velocity of two objects. Instead, it is used to calculate the force between two objects. Force is mass times acceleration, according to Newton's second law of motion[3]. Therefore, the first object's acceleration due to gravity is

$$F_g = m_1 a_g$$

Substituting for F_g:

$$m_1 a_g = F_g = G\frac{m_1 m_2}{r^2}$$

Solving for a_g:

$$a_g = G\frac{m_2}{r^2}$$

2.2 Computational Model

The computational model is only first-order. Because of time limitations and the difficulty of finding any integrators suitable for this model, it was decided the first order method would be sufficient. With the current computational model, every processor that calculates the new position for a planet, even if that processor does not calculate the new position for every planet, requires all the data. In other words, subsets of the problem require all of the data of the previous iteration of the problem. This could be avoided with a different method, but would require substantially more inter-processor communication and would provide negligible net benefits.

The method is a simple first order example of a symplectic method.[6]

$$v_{n+1} = v_n + a_n t$$

$$a_{n+1} = a_n + v_{n+1} t$$

The implementation of a higher order method is currently a work in progress.

2.3 C Code Development

The model has been designed to dump the state variables to a binary file at both the end of the program and during the run of the program. In order to lose as little data as possible, the data is written to the files directly. In order to manipulate and view these files, a set of data utilities have been written. To show the data in a visual manner, the Open GL libraries were used for simplicity of the 3D display.

Parallelization was achieved using the MPI library[4], which spawns multiple processes to deal with a task on the assumption that every process will be identical in design. The use of processes makes the code easier to implement over a networked supercomputer, but quite memory intensive on each node.

2.4 Java Code Development

After data is loaded in RAM, the code creates a static number of threads (defined at runtime) to begin processing the data. Each thread requests a single object within the model, runs all the required calculations on that object, then returns the new data. All threads run until all data is calculated. Minimal effort was required to make the program "thread-safe" thanks to Java's 'synchronized' keyword.

Because of limitations with Java's virtual machine, it was decided to run the code using the Java "server" as opposed to the Java "client." This can be done easily by adding the switch "-server" to a "java" command. The Java server takes slightly longer to start up then the client. Once running, the Java server does run significantly faster (4 to 8%) then the client.

Networking was not implemented using a pre-made library like MPI. In the Java version, a socket is created that listens for a network connection. Once a connection is recieved, the socket sends the connected machine a number of objects (generally more than 400) and waits for the connected machine to return its calculations. This method is incredibly efficient

because it offloads a large amount of data onto an potentially unlimited number of clients.

Most Java is incredibly verbose: this code is no expception. Short cuts were studiously avoided when possible (and when short cuts were taken, they were clearly labeled and explained) for the sake of creating the code "the right way." It is firmly and widely believed that standardization is important, as no one likes three ternary operators embedded in a dereferenced inline-class.

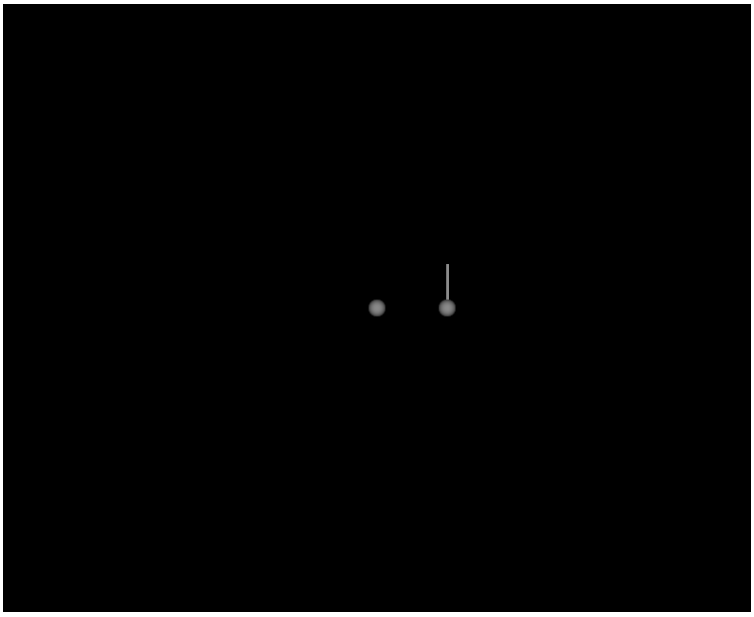
3 Results

3.1 Test Problems

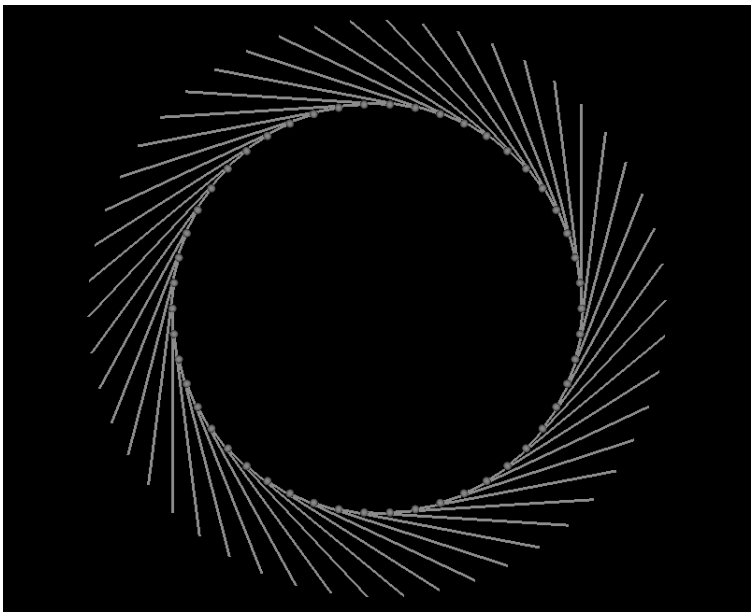
Two test problems were chosen: a ring of identical bodies, and a small body orbiting a massive body. The first test scenario was chosen because it is easy to test with a wide number of parameters. It is easy to conduct a scaling study with this setup.

3.1.1 C Model

The ring problem tests very well, but has a few eccentricities due to an inherent sensitivity. This may be exacerbated by the use of a first order method, but there is no current way to verify this. If two objects get too close together, the dynamic time step decreases in size in order to prevent instability. This slows down the entire model significantly, and eventually grinds the simulation to a halt. If, for whatever reason, the relative position, velocity, or acceleration is even slightly different for any of the bodies, the end results are wildly different. This verifies the underlying thesis of the Nemesis problem: a slight change in the initial conditions of a system of this sort can lead to massively different results in the long term.



The two body problem is much easier to test in depth, as it is not unreasonable to calculate by hand the positions of objects, even for several time steps. According to Kepler's laws of planetary motion, one planet orbiting another generates an elliptical path. This can be verified using the output positions of the two bodies, or less accurately with the display function.



The above graphic is taken from the run of the ring-body problem run with fifty bodies. The white lines tangent to the circle are the current velocity vectors of each body. The ring is almost perfectly uniform, which implies that the model is being calculated accurately.

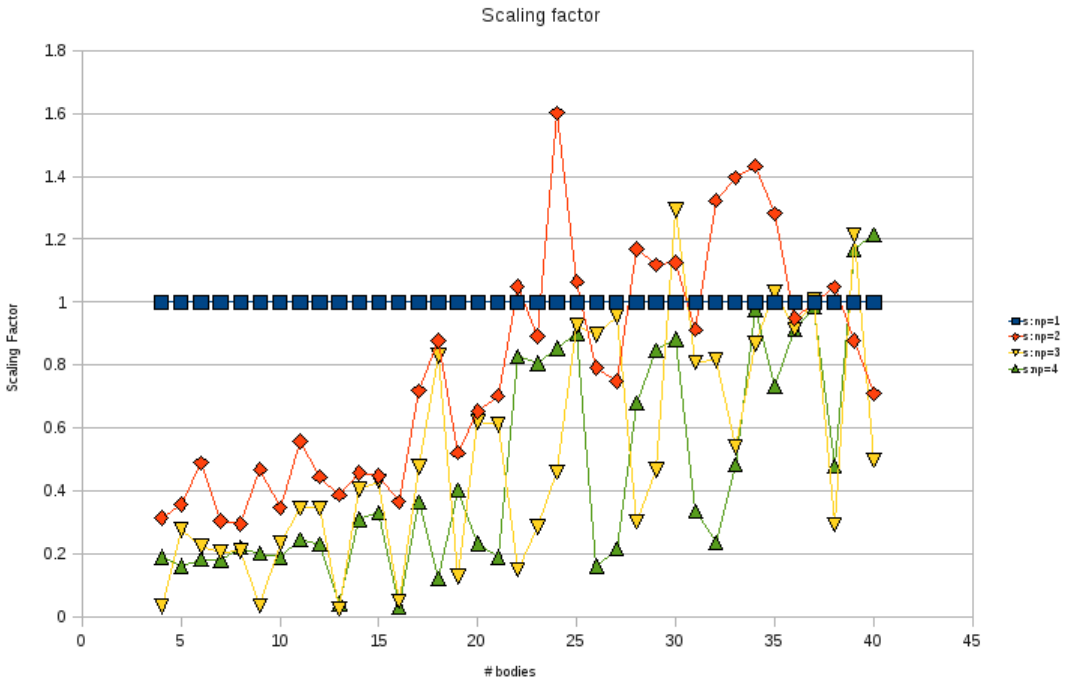
3.1.2 Java Model

No test problems were run with the Java model as the locations and velocities of the objects could not be set at or before runtime.

3.2 Scaling

3.2.1 C Model

In order to test whether or not the computational power of a second processor is worth the communication backlog at certain problem sizes, a scaling study is normally done. There have been debates on how this should be done. Problems of various sizes were tested in order to show not only how well the model scales in specific instances, but how well it is likely to scale on a problem of a certain size with a certain number of available processors. The chart below shows the scaling factor for the ring of bodies problem with an increasing numbers of bodies.



Due to the relatively low amount of data and communications, whether the number of bodies is divisible by the number of processors is an important factor in how well the program scales. The lowest break even point for 2 processors is 22 bodies.

3.2.2 Java Model

No scaling study was run with the Java model because of bugs which prevent the program from running for more than one iteration.

4 Conclusions

4.1 Model

4.1.1 C Model

The model performs with a high degree of accuracy as can be verified with the two test problems. The ring of bodies verifies the model as being accurate because the bodies maintain almost perfect rotational symmetry, and the two body problem verifies the model with the existence of the elliptical path. The model does include some inherent limitations, but this is only a natural side effect of implementing the more simple Newtonian gravity as opposed the that defined by Relativity and using only a first order method. There are some work-around solutions to this, but most of them apply only to specific cases, and any of them would take some time to design and implement.

As of yet a run-time display has not been written, so the 'smoothness' of the model cannot be tested. The creation of such a display is a reasonably high priority, however, as it may provide further insight into the entire model.

4.2 Scaling

4.2.1 C Model

From our tests of the model, it appears that the method scales in a roughly linear way. As the complexity of the model increases in a more than exponential manner with regard to the number of bodies being accounted for, the additional overhead communication increases very quickly. This may account for the somewhat erratic scaling. Other possible sources include the sizes of various on-die caches, communication latency, and access to necessary memory.

The multi-processor library used was MPI, which was chosen for its ability to work on networked clusters as well as individual, non-connected nodes. Each processor calculates the new position and velocity for a fraction of total bodies per time-step.

Every communication between processors takes time and system resources that could have been used in calculating the new positions and velocities of planets. With a first order method, the number of calculations required per planet per time-step is relatively small. This means that one communication between processors replaces a certain amount of time that could have been spent in computation, the specific amount that is consumed varies depending on the machine and the design of the code. If a higher order method were used, one or both of two things would happen that would most likely reduce the relative amount

of communication overhead. Larger time-steps could be used to achieve the same accuracy, which would mean that fewer inter-processor communications would be required per run of the model. Conversely, far more calculations would be required for a time-step of the same size that would result in higher accuracy, which would mean that each communication replaces a smaller fraction of the model's computation.

Furthermore, it requires almost the same amount of communication overhead in MPI to pass one variable between processors as it does to pass an entire array. This leads to the conclusion that variables might be grouped together in arrays to diminish overhead, although this would lead to more arcane and less readable code.

4.3 Teamwork

Two of the team members are programmers of superior skill, with the third competent but overshadowed in skill and experience. This third person is a competent and fluent technical writer, one who can more than make up for his team mates lack of disposition to put anything into words. The C version of the model was written entirely by one programmer, with the Java version created solely by the other. The technical writer was able to get most of the work done while the C programmer and the Java programmer were arguing as to the superiority of their preferred languages.

All members of the team have experience with presentation. This is most obvious in the technical writer, who is closely followed in skill by the Java programmer in presentation and speaking. The C programmer is competent at speaking, but prefers not to do so.

Acknowledgements

- Robert W. Robey
- Lee Goodwin
- Cleve Moler

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[2] I. Asimov. Nemesis^ Doubleday, Bantam Division, 1989.

[3] S.R. Diamond. *Fundamental Concepts of Modern Physics.* AMSCO School Pub., 1970.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8:159–416, 1994.

[5] J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[6] Cleve Moler. Chapter 17. draft of book intended to demonstrate MATLAB, presented at New Mexico Supercomputing Challenge Kickoff, 2009.

A Java Code

```
File: Station.java
package grandCentral;
public class Station implements ThreadCommunicator {
protected int localThreadCount;
protected int networkThreadCount;
protected double timeToRun;
protected double timestep;
protected ArrayList<AbstractThread> threadList;
protected ArrayList<SpaceObject> currentObjects;
protected int currentIndex;
protected int doneCount;
protected Album theAlbum;
protected Socketeer theSock;
public Station(int numLocalThreads, int numNetworkThreads, double times, doub
localThreadCount = numLocalThreads;
networkThreadCount = numNetworkThreads;
timeToRun = times;
timestep = theTimeStep;
threadList = new ArrayList<AbstractThread>();
currentObjects = new ArrayList<SpaceObject>();
currentIndex = 0;
doneCount = 0;
theAlbum = new Album();
theSock = new Socketeer(7737);
}
public void addObject(SpaceObject o) {
currentObjects.add(o);
}
public void start() {
int i = 0;
// store the inital conditions
theAlbum.addRecord(timeToRun + 1.0, currentObjects);
// create the local threads
while (i != localThreadCount) {
```

```
LocalSpaceObjectThread myThread = new LocalSpaceObjectThread(this,
timestep);
threadList.add(myThread);
// run the thread
threadList.get(threadList.size() - 1).start();
i++;
}
// create the network threads
i = 0;
while (i != networkThreadCount) {
NetworkSpaceObjectThread myThread = new NetworkSpaceObjectThread(this, timest
threadList.add(myThread);
threadList.get(threadList.size() - 1).start();
i++;
}
}
public SpaceObjectLooper getLooper() {
return (new SpaceObjectLooper(theAlbum.getLast()));
}
public synchronized ArrayList<SpaceObject> getObject(int numObjects) {
if (!isReady()) {
return null;
}
ArrayList<SpaceObject> toReturn = new ArrayList<SpaceObject>();
while (isReady() && toReturn.size() != numObjects) {
SpaceObject myObject = currentObjects.get(currentIndex);
toReturn.add(myObject);
currentIndex++;
}
return toReturn;
}
public boolean isFinished() {
return (timeToRun == 0);
}
protected boolean isReady() {
return ((currentIndex != (currentObjects.size())));
```

```
}
public synchronized void saveObject(SpaceObject ourObject) {
doneCount++;
// the id of the object should also be its index.
// but in the crazy world of threads, we can't be so sure.
// but it would still be nice to have the preformance increase...
// so let's just check!
if (currentObjects.get(ourObject.id).id == ourObject.id) {
// yay! It is working!
currentObjects.set(ourObject.id, ourObject);
} else {
// ah... we got threaded!
int i = 0;
while (i != currentObjects.size()) {
if (currentObjects.get(i).id == ourObject.id) {
currentObjects.set(i, ourObject);
break;
}
i++;
}
}
if (isReady() || isFinished() || doneCount != currentObjects.size()) {
if (doneCount % 1000 == 0) {
System.out.println("Got object: " + doneCount);
}
return;
}
// there are no more objects
// time to copy...
theAlbum.addRecord(timeToRun, currentObjects);
timeToRun--;
currentIndex = 0;
}
public Stillframe getCurrentFrame() {
return theAlbum.getLastFrame();
}
```

```
}
File: ThreadCommunicator.java
package grandCentral;
public interface ThreadCommunicator {
public SpaceObjectLooper getLooper();
public Stillframe getCurrentFrame();
public void saveObject(SpaceObject ourObject);
public ArrayList<SpaceObject> getObject(int numObjects);
public boolean isFinished();
}
File: NBodySocket.java
package networkStuff;
public class NBodySocket {
protected int numObjects;
protected SocketTalker theTalker;
protected Socket theSocket;
public NBodySocket(Socket theSock) {
theSocket = theSock;
}
public void setTalker(SocketTalker ST) {
theTalker = ST;
}
// precondition: We've already set the SocketTalker
public void checkBuffer() {
try {
BufferedReader ourReader = new BufferedReader(
new InputStreamReader(theSocket.getInputStream()));
String theLine = ourReader.readLine();
// parse theLine
// the first character of any line will contain the lines
// "command"
// possible commands are:
// 1: set optimal number of objects, and send me my first set
// 2: save a space object burst and send the next set
switch (Integer.valueOf(theLine.substring(0, 1))) {
case 1:
```

```
numObjects = Integer.valueOf(theLine.substring(1));
theTalker.sendNextSet();
break;
case 2:
Stillframe myFrame = new Stillframe(0);
myFrame.setData(theLine);
theTalker.gotObjectsToSave(myFrame);
theTalker.sendNextSet();
break;
}
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
public void sendDataset(ArrayList<SpaceObject> objectIDs,
Stillframe theFrame) {
// formatted as such:
// object.ids,for,this,machine||stillframe
String toSend = "";
for (SpaceObject o : objectIDs) {
toSend += String.valueOf(o.id) + ((char) 1);
}
// remove the last comma and add the seperator and add the frame
toSend = toSend.substring(0, toSend.length() - 1) + ((char) 0)
+ theFrame.getData();
try {
DataOutputStream myStream = new DataOutputStream(theSocket
.getOutputStream());
myStream.writeBytes(toSend + ((char) 10)); // char 10 is the end of
// line character that we
// look for with
// readLine.
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
```

```
}
}
public int getOptimalObjects() {
return numObjects;
}
public boolean isConnected() {
return theSocket.isConnected();
}
public void close() throws IOException {
theSocket.close();
}
}
File: Socketeer.java
package networkStuff;
public class Socketeer {
protected ServerSocket theSocket;
public Socketeer(int portNumber) {
try {
theSocket = new ServerSocket(portNumber);
} catch (IOException e) {
e.printStackTrace();
}
}
public synchronized NBodySocket getSocket() {
NBodySocket mySocket = null;
try {
mySocket = new NBodySocket(theSocket.accept());
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
return mySocket;
}
}
File: SocketTalker.java;
package networkStuff;
```

```
public interface SocketTalker {
public void gotObjectsToSave(Stillframe sf);
public void sendNextSet();
}
File: NBody.java
package runner;
public class NBody {
public static void main(String[] args) {
if (args.length != 4) {
System.out.println("NBody, SCC 2008-2009");
System.out.println("Dov, Ryan, and Jon");
System.out.println("----------------------------------------");
System.out.println("Usage: nbody.jar <num local threads> <num network threads
System.out.println("");
System.out.println("If there is no objects.xml file, random objects will be g
System.out.println("Upon the end of the run, the entire album will be saved t
System.out.println("");
System.out.println("Happy computing!");
System.exit(0);
}
int numThreads = Integer.valueOf(args[0]);
int numNetworkThreads = Integer.valueOf(args[1]);
int numObjects = Integer.valueOf(args[2]);
int numSteps = Integer.valueOf(args[3]);
Station myStation = new Station(numThreads, numNetworkThreads, numSteps, 1);
SpaceObject myObject;
int i = 0;
while (i != numObjects) {
myObject = new SpaceObject();
myObject.makeRandom(i);
myStation.addObject(myObject);
i++;
}
// let 'er rip!
myStation.start();
}
```

```
}
File: Album.java
package scribe;
public class Album {
protected ArrayList<Stillframe> theFrames;
public Album() {
theFrames = new ArrayList<Stillframe>();
}
public void addRecord(double timestep, ArrayList<SpaceObject> objects) {
// stillframe makes the deep copy for us
Stillframe myFrame = new Stillframe(timestep, objects);
theFrames.add(myFrame);
}
public ArrayList<SpaceObject> getStillframe(double timestep) {
for (Stillframe sf : theFrames) {
if (sf.getTimeStep() == timestep) {
return sf.getRecord();
}
}
return null;
}
public ArrayList<SpaceObject> getLast() {
return theFrames.get(theFrames.size() - 1).getRecord();
}
public Stillframe getLastFrame() {
return theFrames.get(theFrames.size() - 1);
}
}
File: Stillframe.java
package scribe;
public class Stillframe {
protected double timeStep;
protected ArrayList<SpaceObject> record;
public Stillframe(double timeStep, ArrayList<SpaceObject> theWorld) {
this.timeStep = timeStep;
record = new ArrayList<SpaceObject>();
```

```
// we need to make a deep copy
for (SpaceObject o : theWorld) {
record.add(o);
}
}
public Stillframe(double timeStep) {
this.timeStep = timeStep;
record = new ArrayList<SpaceObject>();
}
public double getTimeStep() {
// int is a primitive type so we will not need any kind of deep copy. We
// can just return it.
return timeStep;
}
public ArrayList<SpaceObject> getRecord() {
// this may seem dumb, but we can't pass a reference to the protected
// property. We need to create a deep copy.
ArrayList<SpaceObject> toReturn = new ArrayList<SpaceObject>();
for (SpaceObject o : record) {
toReturn.add(o);
}
return toReturn;
}
public String getData() {
StringBuilder toReturn = new StringBuilder();
for (SpaceObject o : record) {
toReturn.append(o.getData() + ((char) 2));
}
// remove the last comma
return toReturn.toString().substring(0, toReturn.length() - 1);
}
public void setData(String data) {
String[] temp = data.split("" + ((char) 2));
for (String s : temp) {
SpaceObject o = new SpaceObject();
o.fillData(s);
```

```
record.add(o);
}
}
public int getNumberOfObjects() {
// this won't be a pointer
// int is primitive
return record.size();
}
}
File: Dimension.java
package spaceObjectStuff;
public class Dimension {
public double x;
public double y;
public double z;
}
File: SpaceObject.java
package spaceObjectStuff;
public class SpaceObject {
public int id;
public Dimension location;
public Dimension velocity;
public double mass;
public double radius;
public SpaceObject() {
location = new Dimension();
velocity = new Dimension();
}
public void makeRandom(int theId) {
id = theId;
location.x = (Math.random() * 10000.0);
location.y = (Math.random() * 10000.0);
location.z = (Math.random() * 10000.0);
velocity.x = (Math.random() * 10000.0);
velocity.y = (Math.random() * 10000.0);
velocity.z = (Math.random() * 10000.0);
```

```
mass = (Math.random() * 10000000.0);
radius = (Math.random() * 1000000.0);
}
public void fillData(String theData) {
String[] temp = theData.split(( "" + (char) 1));
int i = 0;
while (i < temp.length) {
switch (i) {
case 0:
id = Integer.valueOf(temp[0]);
case 1:
location.x = Double.valueOf(temp[1]);
case 2:
location.y = Double.valueOf(temp[2]);
case 3:
location.z = Double.valueOf(temp[3]);
case 4:
velocity.x = Double.valueOf(temp[4]);
case 5:
velocity.y = Double.valueOf(temp[5]);
case 6:
velocity.z = Double.valueOf(temp[6]);
case 7:
mass = Double.valueOf(temp[7]);
case 8:
radius = Double.valueOf(temp[8]);
}
i++;
}
}
public String getData() {
return "" + id + ((char) 1) + location.x + ((char) 1) + location.y + ((char)
}
}
File: AbstractThread.java
package threadStuff;
```

```
public interface AbstractThread {
public void start();
}
File: LocalSpaceObjectThread.java
package threadStuff;
public class LocalSpaceObjectThread extends Thread implements AbstractThread
public ThreadCommunicator theMain;
public SpaceObject theObject;
public double timestep;
public final double gravity = 0.00000000006674;
public LocalSpaceObjectThread(ThreadCommunicator theTalker, double theStep) {
theMain = theTalker;
timestep = theStep;
}
public void run() {
while (!theMain.isFinished()) {
theObject = getNextObject();
if (theObject != null) {
calculateVelocity();
calculatePosition();
theMain.saveObject(theObject);
}
}
}
protected SpaceObject getNextObject() {
ArrayList<SpaceObject> theList = theMain.getObject(1);
if (theList == null) { return null; }
return theList.get(0);
}
protected void calculatePosition() {
theObject.location.x = theObject.location.x
+ (theObject.velocity.x * timestep);
theObject.location.y = theObject.location.y
+ (theObject.velocity.y * timestep);
theObject.location.z = theObject.location.z
+ (theObject.velocity.z * timestep);
```

```
}
protected void calculateVelocity() {
SpaceObjectLooper ourLoop = theMain.getLooper();
SpaceObject currentObject;
Dimension dist = new Dimension();
Dimension accel = new Dimension();
double distance = 0.0;
double sqDistance = 0.0;
double magAccel = 0.0;
while (!ourLoop.isFinished()) {
currentObject = new SpaceObject();
currentObject = ourLoop.nextObject();
if (currentObject.id != theObject.id) {
// it isn't us. We can use it.
dist.x = currentObject.location.x - theObject.location.x;
dist.y = currentObject.location.y - theObject.location.y;
dist.z = currentObject.location.z - theObject.location.z;
sqDistance = Math.pow(dist.x, 2) + Math.pow(dist.y, 2)
+ Math.pow(dist.z, 2);
distance = Math.sqrt(sqDistance);
magAccel = (currentObject.mass / (sqDistance)) * gravity;
accel.x = (magAccel * dist.x) / distance;
accel.y = (magAccel * dist.y) / distance;
accel.z = (magAccel * dist.z) / distance;
theObject.velocity.x = theObject.velocity.x
+ (accel.x * timestep);
theObject.velocity.y = theObject.velocity.y
+ (accel.y * timestep);
theObject.velocity.z = theObject.velocity.z
+ (accel.z * timestep);
}
}
}
}
```

File: NetworkSpaceObjectThread.java

```
package threadStuff;
```

```
public class NetworkSpaceObjectThread extends Thread implements AbstractThrea
protected double timestep;
protected ThreadCommunicator theMain;
protected NBodySocket theSocket;
protected Socketeer theSock;
public NetworkSpaceObjectThread(ThreadCommunicator theTalker, double ts, Sock
timestep = ts;
theMain = theTalker;
theSock = socks;
}
public void run() {
// first, get a socket.
theSocket = theSock.getSocket();
theSocket.setTalker(this);
while (theSocket.isConnected() && (!theMain.isFinished())) {
theSocket.checkBuffer();
}
}
public void sendNextSet() {
ArrayList<SpaceObject> toGive = theMain.getObject(theSocket.getOptimalObjects
while (toGive == null) {
toGive = theMain.getObject(theSocket.getOptimalObjects());
}
theSocket.sendDataset(toGive, theMain.getCurrentFrame());
}
public void gotObjectsToSave(Stillframe sf) {
ArrayList<SpaceObject> theList = sf.getRecord();
for (SpaceObject o : theList) {
theMain.saveObject(o);
}
}
}
File: SpaceObjectLooper.java
package threadStuff;
public class SpaceObjectLooper {
private ArrayList<SpaceObject> ourList;
```

```
public SpaceObjectLooper(ArrayList<SpaceObject> theList) {
ourList = new ArrayList<SpaceObject>();
for (SpaceObject o : theList) {
ourList.add(o);
}
}
public boolean isFinished() {
return ourList.isEmpty();
}
public SpaceObject nextObject() {
SpaceObject toReturn = ourList.get(0);
ourList.remove(0);
return toReturn;
}
}
```

B C Code

```
#include <stdio.h>
#include <stdlib.h>
#include "body.h"
#ifndef NOMPI
#include <mpi.h>
#endif
#include <math.h>
//define macro to find maximum of 2 values
#define max(f,g) ((f)>(g))?(f):(g)
//define macro to find minimum of 2 values
#define min(f,g) ((f)<(g))?(f):(g)
#define PI 3.14159265358979323846
int saveFile(char *filename, modelState *state);
int readFile(char *filename, modelState *state);
int engine(int argc, char* argv[], model ins){
double *tpx, *tpy, *tpz;//temporary position vectors
double *tvx, *tvy, *tvz;//temporary velocity vectors
double *ax, *ay, *az, a;//acceleration vector
double rx, ry, rz, rSq, r;//difference in position vectors
double starttime, slavetime, totaltime;//timeing variables
double v, lMaxA, lMaxV, lMinD, deltaT;//local dynamic timestep vars
double gMaxA, gMaxV, gMinD;
int size, rank;//basic processor data
int n, rem, myStart, mysize, myEnd;//basic processor data location info
int *recvCounts, *offset;//Arrays for communication
int i,j;
double initTime;
int d;
char savefile[50];
int iter;
double m,s,c;
modelState curr;
//start MPI
#ifndef NOMPI
```

```
MPI_Init (&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
#else
rank=0;
size=1;
#endif
if(ins.sigma>=1||ins.sigma<=0){
ins.sigma=0.9;
}
//Load data from file
if(rank==0&&(!ins.scaleStudy)){
if(readFile(ins.loadFile, &curr)){
#ifndef NOMPI
MPI_Finalize();
#endif
return 1;
}
}
//Hard coded init for scaling study
if(ins.scaleStudy&&rank==0){
double rad=25.0;
curr.nBodies=ins.nBodies;
curr.time=0.0;
curr.mass=dvector(curr.nBodies);
curr.px=dvector(curr.nBodies);
curr.py=dvector(curr.nBodies);
curr.pz=dvector(curr.nBodies);
curr.vx=dvector(curr.nBodies);
curr.vy=dvector(curr.nBodies);
curr.vz=dvector(curr.nBodies);
for(i=0;i<ins.nBodies;i++){
c=cos(2.0*PI*((double)i)/((double)(ins.nBodies)));
s=sin(2.0*PI*((double)i)/((double)(ins.nBodies)));
curr.mass[i]=1.0;
curr.px[i]=c*rad;
```

```
curr.py[i]=s*rad;
curr.pz[i]=0;
curr.vx[i]=-s*rad;
curr.vy[i]=c*rad;
curr.vz[i]=0;
}
if(strcmp("", ins.dataFile)&&rank==0){
sprintf(savefile, "%s%.2f.planet", ins.dataFile, curr.time);
saveFile(savefile, &curr);
}
}
//Print initial data
if(ins.debugOutput&&rank==0){
printf("Output for time %f\n", curr.time);
for(i=0;i<curr.nBodies;i++){
printf("Body %d: mass %f pos (%f,%f,%f) vel (%f,%f,%f)\n", i, curr.mass[i], c
}
printf("\n\n");
}
#ifndef NOMPI
//Broadcast data to other processors from root process
MPI_Bcast(&(curr.nBodies), 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&(curr.time), 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
//allocate memory
if(rank!=0){
curr.mass=dvector(curr.nBodies);
curr.px=dvector(curr.nBodies);
curr.py=dvector(curr.nBodies);
curr.pz=dvector(curr.nBodies);
curr.vx=dvector(curr.nBodies);
curr.vy=dvector(curr.nBodies);
curr.vz=dvector(curr.nBodies);
}
MPI_Bcast(curr.mass, curr.nBodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(curr.px, curr.nBodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(curr.py, curr.nBodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
MPI_Bcast(curr.pz, curr.nBodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(curr.vx, curr.nBodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(curr.vy, curr.nBodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(curr.vz, curr.nBodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
//Broadcast intructions to other processors from root process
MPI_Bcast(&(ins.debugOutput), 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&(ins.timeRun), 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&(ins.endTime), 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&(ins.timeStep), 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&(ins.sigma), 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&(ins.G), 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
#endif
ax=dvector(curr.nBodies);
ay=dvector(curr.nBodies);
az=dvector(curr.nBodies);
//get data dump info
initTime=curr.time;
d=1;
//split up mesh
//Allocate communication arrays
recvCounts=ivector(size);
offset=ivector(size);
n=curr.nBodies/size;//calculate default size
rem=curr.nBodies%size;//calculate remaining bodies to split between processor
j=0;
for(i=0;i<size;i++){//calculate values for communication arrays
offset[i]=j;
recvCounts[i]=n+(rem>i?1:0);
j+=recvCounts[i];
}
myStart=offset[rank];
mysize=recvCounts[rank];
myEnd=offset[rank]+recvCounts[rank];
//allocate memory
tpx=dvector(mysize);
tpy=dvector(mysize);
```

```
tpz=dvector(mysize);
tvx=dvector(mysize);
tvy=dvector(mysize);
tvz=dvector(mysize);
//get first time
if(ins.timeRun){
#ifndef NOMPI
starttime=MPI_Wtime();
#endif
}
//begin main loop
iter=0;
if(rank==0){printf("Beginning model run with %d bodies\n", curr.nBodies);}
for(;curr.time<ins.endTime;){
//do calculations
for(i=myStart;i<myEnd;i++){
//zero acceleration
ax[i]=0;
ay[i]=0;
az[i]=0;
//zero timestep calcs
lMaxA=0;
lMaxV=0;
lMinD=-1;
for(j=0;j<curr.nBodies;j++){
if(i!=j){//do not do calculation if refering to self
rx=curr.px[j]-curr.px[i];
ry=curr.py[j]-curr.py[i];
rz=curr.pz[j]-curr.pz[i];
rSq=rx*rx+ry*ry+rz*rz;
r=sqrt(rSq);
//get magnitude of gravitational force on i due to j
a=ins.G*curr.mass[j]/(rSq);
//calculate components
ax[i]+=a*rx/r;
ay[i]+=a*ry/r;
```

```
az[i]+=a*rz/r;
if(r<lMinD||lMinD<0){
lMinD=r;
}
}
}
a=ax[i]*ax[i]+ay[i]*ay[i]+az[i]*az[i];
if(a>lMaxA){
lMaxA=a;
}
v=curr.vx[i]*curr.vx[i]+curr.vy[i]*curr.vy[i]+curr.vz[i]+curr.vz[i];
if(v>lMaxV){
lMaxV=v;
}
}
lMaxA=sqrt(lMaxA);
lMaxV=sqrt(lMaxV);
//NOTE: Some error occurs in the dynamic timestep calculation code, most obvi
#ifndef NOMPI
MPI_Allreduce(&lMaxA, &gMaxA, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(&lMaxV, &gMaxV, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(&lMinD, &gMinD, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
#else
gMaxA=lMaxA;
gMaxV=lMaxV;
gMinD=lMinD;
#endif
deltaT=(-(gMaxV/gMaxA))+(sqrt(gMaxV*gMaxV+2*gMaxA*gMinD*ins.sigma)/gMaxA);
deltaT=min(deltaT,ins.endTime-curr.time);
//deltaT=ins.timeStep;//debug: constant deltaT
//if(rank==0){printf("iter %3d dT=%15.10g, mV=%15.10g, mA=%15.10g, mD=%15.10g
for(i=myStart;i<myEnd;i++){
//update velocity in temporary variables
tvx[i-myStart]=curr.vx[i]+ax[i]*deltaT;
tvy[i-myStart]=curr.vy[i]+ay[i]*deltaT;
tvz[i-myStart]=curr.vz[i]+az[i]*deltaT;
```

```
//update position in temporary variables
tpx[i-myStart]=curr.px[i]+tvx[i-myStart]*deltaT;
tpy[i-myStart]=curr.py[i]+tvy[i-myStart]*deltaT;
tpz[i-myStart]=curr.pz[i]+tvz[i-myStart]*deltaT;
}
#ifndef NOMPI
//wait for all processes to finish before communication
MPI_Barrier(MPI_COMM_WORLD);
//do communication
MPI_Allgatherv(tpx,mysize,MPI_DOUBLE,curr.px,recvCounts,offset,MPI_DOUBLE,MPI_
MPI_Allgatherv(tpy,mysize,MPI_DOUBLE,curr.py,recvCounts,offset,MPI_DOUBLE,MPI_
MPI_Allgatherv(tpz,mysize,MPI_DOUBLE,curr.pz,recvCounts,offset,MPI_DOUBLE,MPI_
MPI_Allgatherv(tvx,mysize,MPI_DOUBLE,curr.vx,recvCounts,offset,MPI_DOUBLE,MPI_
MPI_Allgatherv(tvy,mysize,MPI_DOUBLE,curr.vy,recvCounts,offset,MPI_DOUBLE,MPI_
MPI_Allgatherv(tvz,mysize,MPI_DOUBLE,curr.vz,recvCounts,offset,MPI_DOUBLE,MPI_
#else
for(i=myStart;i<myEnd;i++){
curr.px[i]=tpx[i-myStart];
curr.py[i]=tpy[i-myStart];
curr.pz[i]=tpz[i-myStart];
curr.vx[i]=tvx[i-myStart];
curr.vy[i]=tvy[i-myStart];
curr.vz[i]=tvz[i-myStart];
}
#endif
//update time
curr.time+=deltaT;
//print iter every 100 iters
if(rank==0){iter++;}
//if(iter%1000==0&&rank==0){printf("iter %d time: %f\n", iter, curr.time);}
//dump data if necessary
if(rank==0&&ins.dataDump>0&&(curr.time-initTime)/ins.dataDump>=d){
d++;
//print to terminal if option active
if(ins.debugOutput){
printf("Output for time %f\n", curr.time);
```

```
for(i=0;i<curr.nBodies;i++){
printf("Body %d: mass %f pos (%f,%f,%f) vel (%f,%f,%f)\n", i, curr.mass[i], c
curr.vx[i], curr.vy[i], curr.vz[i]);
}
printf("\n\n");
}
//dump data
if(strcmp("", ins.dataFile)){
sprintf(savefile, "%s%.2f.planet", ins.dataFile, curr.time);
saveFile(savefile, &curr);
}
}
}
//dump data
if(strcmp("", ins.dataFile)&&rank==0){
sprintf(savefile, "%s%.2f.planet", ins.dataFile, curr.time);
saveFile(savefile, &curr);
}
//print to terminal if option active
if(rank==0&&ins.debugOutput){
printf("Output for time %f\n", curr.time);
for(i=0;i<curr.nBodies;i++){
printf("Body %d: mass %f pos (%f,%f,%f) vel (%f,%f,%f)\n", i, curr.mass[i], c
curr.vx[i], curr.vy[i], curr.vz[i]);
}
printf("\n\n");
}
if(rank==0){printf("%d iters, avg timestep=%g, Model end time=%g\n",iter, (cu
#ifndef NOMPI
//find time taken
if(ins.timeRun){
slavetime = MPI_Wtime() - starttime;
MPI_Reduce (&slavetime, &totaltime, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD
if(rank==0){
printf("Time for run: %f sec, avg time/timestep %g sec\n", (totaltime/((doubl
}
```

```
}
//end MPI
MPI_Finalize();
#endif
return(0);
}
```