Comparative Analysis of Language Dependent Serial, Parallel, and GPU Implementations of the N-Body Problem

By

Jessica Cunningham

Florida Tech

CIS 5080 – Projects in Computer Information Systems

for

Mr. Steve Johnson

July 31, 2011

**Table of Contents**

**Introduction … 3**

**Background … 3**

**Algorithms … 5**

**Project Solutions … 6**

**Implementation … 8**

**Results … 9**

**Conclusions … 17**

**Lessons Learned … 19**

**Future Enhancements … 19**

**Gantt Chart … 22**

**References … 23**

**Appendices …**

* **A. Progress Reports … 25**
* **B. Hardware Requirements … 35**
* **C. Software Configuration … 35**
* **D. User’s Manual … 38**

**Introduction:**

*The performance of different implementations of the same parallelizable algorithm will be quantified and the advantages and disadvantages of each will be determined.*

The n-body problem will be used to benchmark the performance of multiple computer processing architectures and languages. The n-body problem is an introductory molecular dynamics simulation and is known as a massively parallel problem. These types of parallel problems can take full advantage of multithreading and multiprocessing techniques and are appropriate to measure performance metrics between different parallelizable architectures.

Many levels of comparison will be measured. The first will be to compare the performance between the serial Java and serial C implementations. This will show the differences between a compiled language like C and an interpreted language such as Java. The second is to compare the performance of the multithreaded code of C and Java. This is useful because it is important to understand the differences between the performance of heavyweight threads used in C and lightweight threads used in Java. The third comparison will be to compare both C and Java multithreading performance to that of the CUDA GPU architecture.

**The results of the performance analysis will allow a program manager to make an informed decision on what hardware and software implementation of the n-body problem, or other related parallel algorithm, will work best under the given conditions. Depending on factors such as the amount of parallelization in the problem, the amount of work required per thread, and how much time is allotted for design and implementation, this study will give insight into which architecture and implementation is appropriate.**

**Background:**

The need for fast and efficient computing power has been the long time goal of the computer industry since the day it began. Parallel processing, or parallel computing, has been a solution for this problem in the past and in the present. Supercomputers first addressed this solution with vector processors and now after 30 years, the same technique is being harnessed using video cards. ATI and NVIDIA are rapidly introducing General Purpose Computing on Graphics Processing Units, or GPGPU, into mainstream computing. This architecture has already shown promise in a wide range of application to real world problems.

Parallel computing allows a program to carry out many calculations simultaneously. Traditionally, algorithms were performed by a single central processing unit, or CPU, as a serial stream of instructions. Each instruction is executed on the single CPU and only once the instruction is finished can the next one begin. These single CPU are known as single instruction, single data stream processors. With parallel computing, an algorithm is split up into independent parts and then executed with multiple processors at once.1

From the 1980’s all the way up until 2004 the need for fast and efficient processing was improved by frequency ramping, or frequency scaling, which is a technique that increases the frequency of instruction execution. The higher the frequency, the more cycles per second the processor can handle. This process has diminishing returns because power consumption and therefore heat production is directly proportionate to the frequency.2 In May of 2004, Intel ultimately cancelled the release of the Tejas and Jayhawk processors because the power consumption was too great.3

Since 2004, the main focus for fast and efficient processing has gone toward multi-core processors. It is incredibly common to find dual and quad core processors in homes nowadays. These processors are known as multiple instruction, multiple data stream (MIMD) architectures.4 This means that different instructions and data can be sent to each individual core and can complete necessary tasks individually. This architecture helps with the execution of parallel processing by adding extra processing cores. The extent of the efficiency gain from two or four additional processors is not necessarily significant when dealing with massively parallel problems.

Today, there are well known techniques to perform parallel processing. It is possible to achieve parallel processing using multiple single-core computers in a network, which is known as cloud or cluster computing. This technique unfortunately requires distribution computing software that must take care of all of the routing information within the network. A new and upcoming method for parallel processing is known at General Purpose Computing on Graphics Processing Units or GPGPU.5

During the evolution of CPU’s and multi-core CPU’s, the evolution of graphics cards was also in full swing. The first video card was developed by IBM in 1981. With the progress of gaming and visualization the need for better graphics was imperative. Graphics processing units were pushed by the development of gaming systems such as PlayStation and Nintendo 64. GPU’s eventually become highly sophisticated units in themselves, some of the more current ones drawing more power than the general CPU in the same computer. Each pixel in an image or video can be handled by a single unit within the GPU. Current GPU have in excess of 1000 available processing units.

GPGPU is the realization that graphics processing units can be used for more than just images and movies. GPGPU uses the multitude of processing units in a GPU to perform stream processing on other types of data. Stream processing allows the execution of a limited form of parallel processing. Stream processing performs multiple tasks concurrently without managing the allocation, synchronization, or communication between the multiple processors involved. The true definition of parallel processing requires these management skills but there are many data intensive problems that don’t necessarily need the stringent adherent to these parallel processing rules.6

GPGPU aids in these data intensive processes by running a single instruction on many different data points in a stream at once. In the GPGPU scenario, all of the data needs to have the same computation performed upon each individual piece of data. To do this, each data element is sent to an individual processing unit in the GPU. The instruction is performed by each of the processing units and then reported back. The processing units in a GPU can only read in input, perform an operation, and then output the result. There is very little memory available in the GPU itself. This defines that the best kind of problem for the GPGPU scheme is highly parallelizable with data elements that are independent.

GPGPU is known as single instruction, multiple data streams, or SIMD processing. This is similar to vector processing. It is interesting to note that through the development of CPU in the 1980’s, the era of supercomputers arose. Supercomputers used the technique of vector processing to perform the same instruction on multiple data elements quickly. This was an advancing technology until the personal computer rush occurred. The basic CPU technology was cheaper and therefore was included into the personal computers and pushed into the mainstream. The vector computing of supercomputers was lost in the personal computing era but now it is returning, again with the catchphrase of supercomputing, with the rise of GPGPU.

The two companies that are leading the marking with GPU’s and therefore GPGPU techniques are ATI and NVIDIA.7 Both of these companies are now providing application programming interfaces, or API’s, for GPGPU. The ATI API is called ATI Stream and the NVIDIA API is called CUDA. These API’s provide a significant base to programmers who want to use the GPU for parallel processing. CUDA can be used with high level computing languages such as C/C++, Fortran, Java, Python, and .Net Framework. ATI and NVIDIA are competing to create the best GPGPU architecture and the results are increasingly impressive.

GPU’s are now rivaling the processing power of the most current CPU’s. Processing power is most commonly discusses in ‘floating point operations per second,’ or FLOPS. The Intel I7 can perform about 69 billion FLOPS. The ATI Radeon HD 5770 GPU, which is only a mid-grade GPU performs 1.36 trillion FLOPS. That is easily a factor of 1000 more FLOPS. A point to keep in mind is the clock rate of these processors as well. The Intel I7 has a clock rate of 3.4GHz while the ATI Radeon has a clock rate of 850MHz. The GPU is performing more FLOPS in fewer cycles, which reduces heat and power consumption. The top grade ATI Radeon HD 5970 is capable of 4.64 trillion FLOPS with a 725Hz clock rate. The ATI Radeon HD 5770 GPU costs around $260.00 and the Intel I7 costs only around $200.00. Even the top grade ATI Radeon HD 5970 is available for $620.00. It is easy to see how GPGPU is becoming a widely used processing technique.8

With multiple processing possibilities such as multithreading, multiprocessing, and now GPGPU available for execution of parallel algorithms, it is helpful to compare the options at hand. The n-body problem is used here to benchmark the performance of five available processing architectures, mixing and matching both language and processing techniques.

1. **Serial Java implementation**
2. **Serial C implementation**
3. **Java multithreading implementation**
4. **C multiprocessing implementation**
5. **GPGPU implementation using CUDA**

**The results of the performance analysis will show the advantages and disadvantages of each architecture. Many components will be considered such as processing time, ease of setup, ease of use, etc…**

**Algorithms:**

The n-body problem is an introductory molecular dynamics simulation in which given only the present positions and velocities of a group of bodies, all future motion can be predicted.9 Though there are many implementations of the n-body problem, the most simple and direct method will be used. This method, known as the “particle-particle” method directly computes all body pair interactions at every time interval.10

The total force acting upon each individual body in an n-body system is calculated each iteration. The forces in our simulation are defined by the law of gravity.11

Where  *and*  are the masses of two interacting particles (kg)

and are the position vectors of the two interacting particles

Once the total force on a particle is known, Newton’s second law can be used to translate this force into acceleration, velocity, and ultimately the body’s new position. These new positions will be used to calculate the concurrent positions, and so on.

Where  *is the acceleration calculated by the n-body algorithm on body i at time t1*

and *is the velocity of body i at time t0 and t1*

and *is the position of body i at time t0 and t1*

*is the time interval between each discrete simulation iteration*

A smaller will provide a higher time resolution model, but requires many more iterations of the model and therefore more time and computing power. A should be chosen to maximize the time resolution while keeping within the constraints of available computing power.12

**Project Solution:**

A sequential implementation of the n-body problem will be written in Java and C. In a sequential implementation, each individual body will update its position in a sequential manner, one at a time. This is because in a sequential algorithm, all instructions are executed on one CPU, restricting only one instruction to execute at a time. The pseudo code for this implementation is as follows:

*Initialize n particle positions and velocities for m iterations*

*for iteration = 1:1:m*

*for i = 1:n*

*for j = 1:n*

*if i≠j*

*total F = total F + force(body i, body j)*

*end*

*acc = total F / mass*

*vel = vel + acc \* dt*

*pos = pos + vel \* dt*

*end*

*end*

C is a compiled language where the high level user code is completely pre-compiled into machine code and distributed as a native executable all before runtime.13 This allows the CPU to run the machine code with no interruptions as a batch process. The downside to a compiled language is that it is non-transferable to any other machine architecture other than the one that it was specifically compiled under.

Java on the other hand is an interpreted language. The high level user code is compiled into an intermediate representation called Java bytecode. This Java bytecode is then interpreted by the Java virtual machine, or JVM, in order to be run on the CPU.14 This allows the Java bytecode to be non-platform-specific, only requiring an instance of the Java Runtime environment to be running on the platform that is required. This intermediate step has been thought to result in slower run time performance than native executables such as C. However, in recent years, the JVM has been augmented with many performance optimizations that challenge that conventional wisdom. In comparing the n-body performance under both situations, these underlying differences can be quantified.

Implementing the n-body problem using a multithreaded algorithm will exploit the parallelism of the problem. Because each individual body can update its position independently of all other bodies in the system, these calculations could be parallelized, allowing for a speed-up in computation time. The pseudo code for the multithreaded implementation is as follows.

*Initialize n particle positions and velocities for m iterations*

*Set up thread pool/equivalent*

*for iteration = 1:1:m*

*threadExecute(updateParticles) -------> for j = 1:n*

*if i≠j*

*total F = total F + force(body i, body j)*

*end*

*acc = total F / mass*

*vel = vel + acc \* dt*

*pos = pos + vel \* dt*

*synchronize threads*

*end*

The implementation of a multithreaded algorithm in C and Java will likely produce different results due to the different scheduling schemes used with a compiled language and with an interpreted language. In a compiled language, kernel-level threads will be implemented and will be scheduled directly by the operating system. Depending on the operating system installed, the performance will differ. With an interpreted language such as Java, the threads are user-level, thus the scheduling of the threads will be done by the JVM. The only thing the operating system scheduler will see is the JVM.15 Comparing the n-body performance under both of these scheduling schemes will also quantify the underlying system differences.

The implementation of the n-body problem on the GPU will follow the same algorithm implementation as the multithreading in both Java and C but will be executed on the graphics processing unit using CUDA C. The inspiration of the project and design of the CUDA code is given in *GPU Gems 3.*16 CUDA, or Compute Unified Device Architecture, allows threads to be run on the GPU processing cores instead of the computer processing cores. This will change the entire dynamics of the management of the threads. A comparison of the GPU performance against both the Java and C multithreaded implementations will give insight into how a GPU enabled computer might further increase the capabilities of a parallel implementation.

**Implementation:**

All implementations are run on 32-bit Ubuntu 10.10 Linux. Ubuntu is one of the CUDA supported operating systems. This is important to ensure that communication between the operating system and the device protocols are compatible.

Multiple compilers are required for the various implementations of the n-body algorithm. The gcc compiler is required for both the C and CUDA C implementations and a Java environment with JVM is required for all Java implementations. CUDA C requires a separate compiler for the graphics card device specific code. The compiler is the NVIDIA CUDA compiler, or nvcc. This is downloaded and installed with the CUDA toolkit. (See Appendix C – Software Configuration)

**CUDA also requires some additional software. The NVIDIA driver is required to support the communications between the operating system Ubuntu 10.10 and the graphics card. The CUDA toolkit, mentioned before, includes the CUDA compiler,** development tools, libraries, and documentation needed to utilize the CUDA architecture. The CUDA SDK is **not explicitly required to utilize the graphics card but it provides a number of examples and white papers that can speed up the learning and operating curve.** (See Appendix C – Software Configuration)

The computer hardware used here is an Intel® Core™ i7 CPU 720QM with a clock rate of 1.60GHz. There is 6.00 GB of DDR3 RAM and 6 MB of L3 cache.

The graphics card used is an NVIDIA GeForce GT 230M. The specifications are as follows.

|  |  |
| --- | --- |
| CUDA Cores | 48 |
| Core clock | 500 MHz |
| Memory clock | 790 MHz (1580 MHz data rate) |
| Memory interface | 128-bit |
| Total available graphics memory | 3834 MB |
| Shared system memory | 2810 MB |
| Bus | PCI Express x16 Gen2 |

**Results:**

**The n-body algorithm was implemented successfully in each of the proposed architectures to update every particle 30 times. These 30 iterations were timed for different numbers of particles to be updated. The algorithm was run 30 times for every number of particles under each implementation to provide confidence in the true performance metrics.**

**To begin, consider the serial implementations of the n-body algorithm. The figure below shows the average performance times for the serial n-body algorithm implementations for 128 particles up to 16384 particles.**

Figure - Implementation performance comparison of serial n-body algorithm implementations for varying number of particles.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Particle # | **128** | **256** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| Java Serial (ms) | 6.75 | 21.17 | 70.97 | 282.87 | 1130.67 | 4500.67 | 18126.33 | 73583.87 |
| C Serial (ms) | 13.37 | 51.33 | 212.33 | 811.00 | 3139.60 | 12382.97 | 51881.70 | 210323.73 |

Table – Average performance times in milliseconds of serial n-body algorithm implementations for varying number of particles.

**This figure shows that the Java serial implementation completes 30 iterations of the n-body algorithm faster than C serial. This may not immediately make sense when considering the fact that in general, C is faster because it is a fully pre-compiled language. This is largely true when considering algorithms that may require many classes and objects where pre-compiling can drastically reduce computation time. For an algorithm such as the n-body problem, a relatively simple algorithm where the implementation requires a very small number of classes, the performance will be not be as affected by the pre-compiling vs. just in time compiling differences. In this way, performance is based on other language specific techniques. It is suspected that Java performed faster than C because of the optimizations that are included in the Java language, such as garbage collection and the ability to dynamically choose whether to pre-compile or interpret sections of the code while processing.**

**The quadratic nature of the performance curve with respect to the number of particles is seen in the above figure and will be seen in all subsequent discussions of the results of the n-body problem. The n-body algorithm, because of its full “particle-particle” interaction is what is known in big O notations as an O(n2) algorithm. This means that the algorithms performance is proportional to the square of the input data size. This is because the algorithm must process each particle and that processing requires another pass through each particle. In this way, doubling the amount of particles actually squares the amount of total operations and therefore also squares the performance time.**

**Next, consider the parallel implementations of the n-body algorithm. The figure below shows the average performance times for the parallel n-body algorithm implementations for 128 particles up to 16384 particles.**

Figure - Implementation performance comparison of parallel n-body algorithm implementations for varying number of particles.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Particle # | **128** | | **256** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| Java Parallel (ms) | | 188.17 | 385.43 | 814.97 | 1772.40 | 3699.97 | 8194.33 | 28089.87 | 51753.57 |
| C Parallel (ms) | 147.30 | | 253.13 | 505.43 | 1250.10 | 2212.00 | 6471.60 | 22001.80 | 80764.60 |
| CUDA (ms) | 115.07 | | 116.60 | 123.87 | 139.03 | 184.97 | 311.53 | 1156.53 | 3198.37 |

Table – Average performance times in milliseconds of parallel n-body algorithm implementations for varying number of particles.

**Immediately it is observed that the CUDA implementation is drastically faster than both the Java parallel and C parallel implementations. This is expected due to the nature of the CUDA parallel processing architecture. CUDA is able to utilize the equivalent of 48 computing cores to complete the execution of threads. CUDA is also in full control over scheduling the threads over all 48 cores because the GPU is not busy performing other operating system tasks. Java and C only have the ability to access the four computing cores on the host system, and even then there are limitations on the actual usage of the cores.**

**The C parallel implementation is able to utilize all four processors of the host system because the threads are executed as kernel level threads, allowing the operating system to schedule them over all four available cores. Java on the other hand, executes threads as user level threads. The operating system is only able to schedule the JVM as one process and therefor limits it to one processor. In both cases, Java and C, the processors that the threads are to be executed on are also being utilized for other operating system tasks, such as handling interrupts and I/O. It is now easy to see why the utilization of a GPU for execution of parallel algorithms has many advantages over other commonly used multithreading schemes.**

**It is interesting to note that when there are a very large number of particles, 16,384, the Java parallel implementation actually becomes faster than the C parallel implementation. It is expected that C parallel will be faster because of the kernel level execution utilizing all four cores, but at some point there seems to be diminishing returns. It is not completely clear why the Java parallel overtakes the C parallel in performance once there are a very large number of particles. A hypothesis is that the longer the C parallel algorithm is requiring the operating system to schedule it alongside all other operating system tasks, the more we see the other tasks affecting the performance time. Another hypothesis is that there are diminishing returns on opening a very large number of C threads. The memory management and lack of garbage collection could cause the system as a whole to slow down as the number of threads increases. Java will not experience these problems to the same extent because all of the threads are kept out of view of the operating system within the JVM.**

**Next, consider just the Java implementations of the n-body algorithm. The figure below shows the average performance times for the Java n-body algorithm implementations for 128 particles up to 16384 particles.**

Figure - Implementation performance comparison of Java n-body algorithm implementations for varying number of particles.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Particle # | **128** | **256** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| Java Serial (ms) | 6.75 | 21.17 | 70.97 | 282.87 | 1130.67 | 4500.67 | 18126.33 | 73583.87 |
| Java Parallel (ms) | 188.17 | 385.43 | 814.97 | 1772.40 | 3699.97 | 8194.33 | 28089.87 | 51753.57 |

Table – Average performance times in milliseconds of Java n-body algorithm implementations for varying number of particles.

**This figure shows that the Java serial implementation is faster than the parallel implementation until there are a very large number of particles. The thread management of the parallel implementation requires a lot of overhead and ultimately reduces the performance. The JVM is spending a large fraction of time and resources organizing and scheduling threads instead of actually executing them. This is a similar concept to memory thrashing where the management of tasks takes more time than the actual execution of tasks. It seems as if once there are enough threads to be executed that the time it takes to execute the threads overcomes the thread management costs. This is the case when there are 16,384 particles where the Java parallel is faster than the Java serial. More algorithmic tests and research would give more insight into the performance switchover of Java serial and parallel implementations.**

**Now consider just the C implementations of the n-body algorithm. The figure below shows the average performance times for the C n-body algorithm implementations for 128 particles up to 16384 particles.**

Figure - Implementation performance comparison of C n-body algorithm implementations for varying number of particles.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Particle # | **128** | **256** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| C Serial (ms) | 13.37 | 51.33 | 212.33 | 811.00 | 3139.60 | 12382.97 | 51881.70 | 210323.73 |
| C Parallel (ms) | 147.30 | 253.13 | 505.43 | 1250.10 | 2212.00 | 6471.60 | 22001.80 | 80764.60 |

Table – Average performance times in milliseconds of C n-body algorithm implementations for varying number of particles.

**This figure shows that the parallel execution of the n-body algorithm gives a substantial performance increase over the serial implementation. C executes threads as kernel level threads that allow the operating system to distribute them over all four processors in the system. The C serial implementation is bound to only one processor. These results are completely expected and give a beautiful example of performance increase due to multithreading.**

**Next, it is important to discuss overhead costs such as thread management and communication between the host and the device when comparing the differences between serial and parallel implementations. Performance metrics are greatly affected by overhead costs, and are especially apparent when the ratio of overhead time to actual algorithm execution time is great. When there are only a small number of particles, the cost of the overhead of parallel execution generally outweighs the benefits. In previous figures, the resolution in the small number of particles was not very clear. The figure below shows the average performance times for all n-body algorithm implementations for up to only 1024 particles.**

Figure - Implementation performance comparison of all n-body algorithm implementations for small number of particles.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Particle # | **128** | | **256** | **512** | **1024** |
| Java Serial (ms) | 6.75 | | 21.17 | 70.97 | 282.87 |
| Java Parallel (ms) | | 188.17 | 385.43 | 814.97 | 1772.40 |
| C Serial (ms) | 13.37 | | 51.33 | 212.33 | 811.00 |
| C Parallel (ms) | 147.30 | | 253.13 | 505.43 | 1250.10 |
| CUDA (ms) | 115.07 | | 116.60 | 123.87 | 139.03 |

Table – Average performance times in milliseconds of all n-body algorithm implementations for small number of particles.

**This figure, only showing small numbers of particles, better illustrates the tradeoff between the benefits of costs of serial and parallel implementations. Java parallel is consistently the slowest implementation. This is because the benefits of parallel execution within the JVM are small because it is not actually utilizing multiple processing cores while the costs of thread management are quite high. The C parallel implementation receives more benefit from multithreading, making it faster than the Java parallel, though the costs of thread management cause it to still be slower than serial implementations. This concludes that unless the desired algorithm requires a large number of particles, it is not beneficial to utilize a parallel approach because the costs outweigh the benefits.**

**The serial implementations do not suffer from overhead costs and are therefore the better approach for algorithms addressing small numbers of particles. Java serial is the best choice because of the optimizations that make it moderately faster than C.**

**CUDA shows the most interesting results when comparing small numbers of particles. CUDA does not seem to share the same O(n2) attribute as the other implementations, but rather seems to be quite flat. What this suggests is that the performance of CUDA with small numbers of particles is characterized solely by pure overhead time of communication between the host and the device. The copies of data back and forth from the system to the device are costly. The more time spent contiguously on the device looping through iterations, the more performance gain will be seen. Not until the amount of time contiguously spent on the device executing threads overcomes the time spent transferring data between the host and the device do we see any marked increase in execution time. This does not occur until more than 1024 particles are processed (shown below in figure 6). Therefore, using the CUDA architecture is overkill for algorithms that only require small numbers of particles or small numbers of iterations.**

Figure - Implementation performance of CUDA n-body algorithm for varying number of particles.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Particle # | **128** | **256** | **512** | **1024** |
| CUDA (ms) | 115.07 | 116.60 | 123.87 | 139.03 |

Table – Average performance times in milliseconds of CUDA n-body algorithm implementations for varying number of particles.

Analysis of the variability in execution times of the individual implementations also gives insight into how other tasks the system is performing can affect performance. **The percent error is used to quantify this variability and is defined as follows.17**

**The standard error for each implementation for each number of particles is given by**

**Where, *s* = sample standard deviation**

***n* = sample size, which is 30 for all calculations**

**The percent error is then calculated using this standard error.**

**Where, = sample standard error**

**= sample mean**

**The figure below shows the percent error of execution times over 30 repetitions for each implementation for 128 particles up to 16384 particles.**

Figure - Percent error for all n-body algorithms for varying number of particles.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Particle # | **128** | | **256** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| Java Serial (%) | 3.09 | | 2.98 | 0.23 | 0.48 | 0.13 | 0.07 | 0.38 | 0.74 |
| Java Parallel (%) | | 2.16 | 1.59 | 0.54 | 0.97 | 0.51 | 1.44 | 0.11 | 0.17 |
| C Serial (%) | 0.91 | | 0.51 | 0.72 | 0.26 | 0.04 | 0.34 | 0.47 | 0.22 |
| C Parallel (%) | 0.99 | | 3.81 | 3.19 | 0.75 | 2.00 | 0.08 | 0.19 | 0.08 |
| CUDA (%) | 0.53 | | 0.19 | 0.40 | 0.24 | 0.19 | 0.17 | 0.19 | 0.05 |

Table – Percent error for all n-body algorithms for varying number of particles.

**The overall variability of all of the implementations for any number of particles is quite low. The percent error less than 5% statistically indicates that all implementations are repeatable. Nevertheless, the most variability is shown in the small number of particles, where a higher percentage of the overall performance measure is more susceptible to other tasks that the operating system is performing. This explains why the C parallel implementation is the most variable because the threads are kernel level and are directly scheduled by the operating system. The serial implementations have consistently very low variability because one processor is devoted to the execution of the algorithm generally without being interrupted.**

**The CUDA implementation, although a parallel implementation, has a very small level of variability. This is expected due to the fact that the thread scheduling is not performed by the operating system, but is instead scheduled locally on the graphics card device. In this way the thread execution is not affected by other operating system tasks in any way.**

**Conclusions**

*The performance of different implementations of the same parallelizable algorithm will be quantified and the advantages and disadvantages of each will be determined.*

**The performance of the n-body problem implemented in Java serial, Java parallel, C serial, C parallel and CUDA C was quantified using the total run time of updating varying number of particles 30 iterations. This allowed a discussion of the advantages and disadvantages of each type of implementation under varying circumstances. Depending on the specific type of problem at hand, it is possible to choose the correct implementation using the information gathered**

**The figure below shows the average performance times for all n-body algorithm implementations for 128 particles up to 16384 particles.**

Figure - Implementation performance of all n-body algorithms for varying number of particles.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Particle # | **128** | | **256** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| Java Serial (ms) | 6.75 | | 21.17 | 70.97 | 282.87 | 1130.67 | 4500.67 | 18126.33 | 73583.87 |
| Java Parallel (ms) | | 188.17 | 385.43 | 814.97 | 1772.40 | 3699.97 | 8194.33 | 28089.87 | 51753.57 |
| C Serial (ms) | 13.37 | | 51.33 | 212.33 | 811.00 | 3139.60 | 12382.97 | 51881.70 | 210323.73 |
| C Parallel (ms) | 147.30 | | 253.13 | 505.43 | 1250.10 | 2212.00 | 6471.60 | 22001.80 | 80764.60 |
| CUDA (ms) | 115.07 | | 116.60 | 123.87 | 139.03 | 184.97 | 311.53 | 1156.53 | 3198.37 |

Table – Average performance times in milliseconds of all n-body algorithm implementations for varying number of particles.

**When considering a simulation where only a small number of particles are needed, the parallelization of an algorithm is not the best choice. The overhead cost of thread management outweighs the possible benefits of parallelization, making the overall run time longer than implementing in a serial manner. The Java parallel implementation is the slowest because there is the overhead cost of thread management *and* multithreading in Java does not utilize multiple cores. C parallel is moderately better than Java parallel because the operating system schedules the threads individually as kernel level threads, utilizing multiple cores. However, C parallel still suffers from thread management costs. CUDA is also not a great choice for small numbers of particles because of the overhead of communication between the host and the device.**

**The best choice for a small number of particles is a serial implementation. From the results generated, it is better to use a Java serial implementation than a C serial implementation because the optimizations included in the Java language increase performance speed in general. This may not be true for a more complicated algorithm where the pre-compiled nature of C will make a difference, but for the n-body algorithm, which is relatively simple, the just in time compilation does not cause any performance issues.**

**When considering the situation where increasing numbers of particles are necessary it is better to implement the algorithm in a parallel manner. The best possible implementation is the CUDA architecture. While the setup requirements and learning curve of utilizing the CUDA architecture may be a little more than with Java or C, the performance increase is dramatic. The exploitation of the large number of floating point computing cores available on a graphics card, without the cost of involving a busy operating system, causes the observed drastic performance increase.**

**Here are the performance increases for 16,384 particles. The speedup will only increase as the number of particles increases.**

* 16.18 times **faster than Java parallel**
* 23.01 times faster than Java serial
* 25.25 times faster than C parallel
* 65.75 times faster than C serial

**If the CUDA architecture cannot be utilized because one does not have a compatible graphics card, the next best implementation is Java parallel. The optimizations in Java as well as the ability to control very large numbers of open threads with garbage collection and memory management, makes Java parallel a better choice than C parallel. The data show that C parallel and Java serial actually converge in terms of performance, where the optimizations of Java and the utilization of multiple cores in C begin to behave the same. The C serial implementation is the slowest, which is expected, because it is not utilizing multithreading and is not optimized.**

**Lessons Learned:**

Installation of all required software to utilize the CUDA architecture proved to be difficult. Following the directions given explicitly in the “Getting Started with CUDA Linux” is recommended. Although, even after following these directions, it is beneficial to keep in mind that additional libraries may need to be installed. Once the installation and setup of CUDA is completed, programming and executing CUDA C programs is straightforward.

Algorithm design is very important when programming for multiple architectures, especially CUDA. Simple data structures tend to be the best when considering that performance can always be related back to total operations. Using complicated structures adds computation time for packaging and can add a lot of time during copies. A simple vector that held all needed data was appropriate for the n-body problem, but may not be appropriate for all parallel algorithm implementations.

Memory management needs to be seriously addressed when programming for any multithreaded application. In the n-body algorithm there is a shared memory vector though the use of semaphores or mutexes was not needed due to the nature of the threads. Each thread accessed a unique set of indices in the data vector which eliminated the need for shared memory techniques. Again, this may not be appropriate for many parallel algorithms, as it is seen in literature, so careful design of memory access techniques needs to be addressed.

When writing multithreaded algorithms in C that require a very large number of threads to be created at one time, it is inevitable that the maximum number of threads allowed by the operating system will be reached. The error that is generally thrown states only that a segmentation fault has occurred. To avoid this, the total number of threads opened at one time should be specified in the algorithm implementation. When deciding the value of the maximum number of threads, take into account the number of processors that the system has. In theory, the maximum number of threads should be the number of processors, as this is the maximum number of threads that could be executing at the same time. In practice, it is best to give each processor more than one thread, to make sure that a processor is never idle.

**Future Enhancements:**

# **I have proposed to utilize the CUDA architecture for a project that I am involved in at my current job at the Moffitt Cancer Center. An excerpt from a paper published in the PLOS ONE journal, “**Coulomb Interactions between Cytoplasmic Electric Fields and Phosphorylated Messenger Proteins Optimize Information Flow in Cells”18, is shown below.

*“Normal cell function requires timely and accurate transmission of information from receptors on the cell membrane (CM) to the nucleus. Movement of messenger proteins in the cytoplasm is thought to be dependent on random walk. However, Brownian motion will disperse messenger proteins throughout the cytosol resulting in slow and highly variable transit times. We propose that a critical component of information transfer is an intracellular electric field generated by distribution of charge on the nuclear membrane (NM). While the latter has been demonstrated experimentally for decades, the role of the consequent electric field has been assumed to be minimal due to a Debye length of about 1 nanometer that results from screening by intracellular Cl− and K+. We propose inclusion of these inorganic ions in the Debye-Huckel equation is incorrect because nuclear pores allow transit through the membrane at a rate far faster than the time to thermodynamic equilibrium. In our model, only the charged, mobile messenger proteins contribute to the Debye length.*

*Using this revised model and published data, we estimate the NM possesses a Debye-Huckel length of a few microns and find this is consistent with recent measurement using intracellular nano-voltmeters. We demonstrate the field will accelerate isolated messenger proteins toward the nucleus through Coulomb interactions with negative charges added by phosphorylation. We calculate transit times as short as 0.01 sec. When large numbers of phosphorylated messenger proteins are generated by increasing concentrations of extracellular ligands, we demonstrate they generate a self-screening environment that regionally attenuates the cytoplasmic field, slowing movement but permitting greater cross talk among pathways. Preliminary experimental results with phosphorylated RAF are consistent with model predictions.”*

**The paper theoretically calculates particle movement under pure diffusion and then under the influence of an electric field. This is a perfect application for the use of a highly parallelizable algorithm such as the n-body problem, simulating the particles individually. While it is a form of the n-body problem, a number of changes would need to be made to the very simple implementation of the n-body problem shown in this project.**

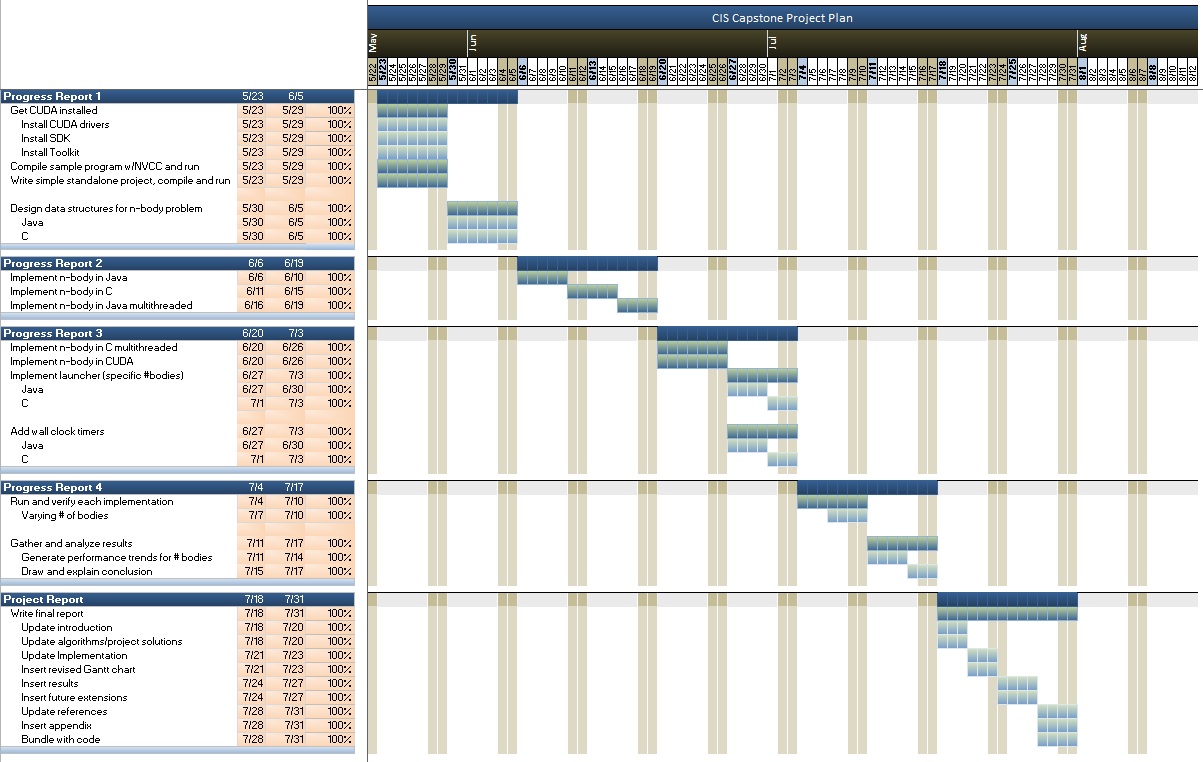
**A new algorithm would have to be written that would simulate Brownian motion. This would be a special case of the n-body problem where the particles would not interact with each other at all, barring physical interaction such as collisions. This implementation would be, in big O notation, a O(n) problem because each particle is completely independent. In this way, the CUDA implementation could allow tens of thousands of particles to be modeled in one simulation.**

**The simulation modeling the particles under the influence of an electric field would be more similar to the n-body problem discussed in the project. Instead of undergoing gravitational interactions between particles, the particles would undergo Coulomb interactions. Coulomb interactions calculate the force exerted on each particle due to an electric field and also the force exerted due to the charge of every other particle in the system. This would be an O(n2) problem, as every particle must update itself in relation to every other particle.**

**If I am able to work on this project I would attempt both of these algorithms using the CUDA architecture. If the implementation was successful, the number of particles to be run in a simulation would need to be much greater than 16,384 as there are hundreds of thousands of particles in a single cell. This would most likely require a rewrite of the n-body problem to one of the more advanced techniques that minimizes the number of computations needed for each particle. While the particle-particle method is the most representative of true nature, algorithms exist that result in very good approximations. The Barnes-Hut algorithm, for example, segments the simulation space of particles into subsections. Within each subsection the particles at short distances are updated using full particle-particle interactions, but the long range interactions are done using the subsections as a whole.22**

**To enhance the implementation in terms of hardware, it would be very beneficial to expand the GPU code to utilize multiple GPU’s simultaneously. The performance benefit gained by using multiple GPU’s would increase the possible amount of particles in a single simulation drastically. The utilization of multiple GPU’s would also allow the algorithm to become more complicated and include other real world phenomena such as particle collisions. Increasing the complexity of the particle interactions and increasing the number of particles will more closely simulate the real world problem, and therefore be more accurate is predicting future events.**

**Gantt Chart:**



**References:**

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2004). *Operating System Concepts with Java.* John Wiley and Sons, Inc.
2. Hennessy, J., & Patterson, D. (2003). *Computer Architecture: A Quantitative Approach.* Elsevier Science.
3. Krazit, T. (2004, May 7). *Intel Changes Plans for Pentium 4*. Retrieved 2011, from PC World: http://www.pcworld.com/article/116053/intel\_changes\_plans\_for\_pentium\_4.html
4. Hennessy, J., & Patterson, D. (1998). *Computer Organization and Design: The Hardware/Software Interface.* San Francisco: Morgan Kaufmann Publishers.
5. Sanders, J., & Kandrot, E. (2011). *CUDA by Example: An Introduction to General Purpose GPU Programming.* Addison Wesley.
6. NVIDIA. (2011). *Developer Zone*. Retrieved 2011, from NVIDIA: http://developer.nvidia.com/category/zone/cuda-zone
7. Advanced Micro Devices. (2011). *ATI Stream Technology*. Retrieved 2011, from AMD: http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx
8. AMD. (2011). *ATI Radeon and ATI FirePro Graphics Cards from AMD*. Retrieved 2011, from http://www.amd.com/us/products/Pages/graphics.aspx
9. Gould, H. (2006). *An Introduction to Computer Simulation Methods: Applications to Physical Systems.* Addison Wesley.
10. Troyer, M. (2006). *Computational Physics*. Retrieved 2011, from http://www.ifb.ethz.ch/education/IntroductionComPhys/ScriptTroyer.pdf
11. Halliday, D., Resnick, R., & Walker, J. (2005). *Fundamentals of Physics.* John Wiley & Sons, Inc.
12. Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2001). *Introduction to Algorithms.* McGraw Hill Book Company.
13. Deitel, P., & Deitel, H. (2005). *C++: How to Program.* Upper Saddle River, NJ: Pearson Education.
14. Deitel, P., & Deitel, H. (2010). *Java: How to Program.* Upper Saddle River, NJ: Pearson Education Inc.
15. Englander, I. (2009). *The Architecture of Computer Hardware, Systems Software, and Networking: An Information System Approach.* John Wiley and Sons, Inc.
16. Nyland, L., Harris, M., & Prins, J. (2007). Chapter 31: Fast N-Body Simulation with CUDA. In *GPU Gems 3.* Addison-Wesley Professional
17. Walpole, R. E. (2011). *Probability and Statistics for Engineers and Scientists (9th Edition).* Prentice Hall.
18. Gatenby RA, Frieden BR (2010) Coulomb Interactions between Cytoplasmic Electric Fields and Phosphorylated Messenger Proteins Optimize Information Flow in Cells. PLoS ONE 5(8): e12084. doi:10.1371/journal.pone.0012084
19. Barnes, J, Hut, P. (1986). A hierarchical O(N log N) force-calculation algorithm. *Nature*, 446-449.
20. C. W. (2010, Aug). *NVIDIA CUDA C Getting Started for Linux.* Retrieved 2011, from http://developer.download.nvidia.com/compute/cuda/3\_2\_prod/docs/Getting\_Started\_Linux.pdf
21. Ubuntu. (2011). *Download Ubuntu.* Retrieved 2011, from http://www.ubuntu.com/download/ubuntu/download
22. The Java Foundation (2011). *Eclipse Downloads.* Retrieved 2011, from http://www.eclipse.org/downloads/

**Appendix A – Status Reports**

**A.1 - Capstone Project Week 5 Status Report for “Comparative Analysis of Language Dependent Sequential, Multithreaded, and GPU Implementations of the N-Body Problem”**

**Jessica Cunningham**

**Date:** 6/2/11

**Accomplishments**

Progress Report 1: 5/23/11 – 6/5/11

1. Install CUDA drivers – completed
2. Install CUDA SDK – completed
3. Install CUDA Toolkit – completed
4. Compile sample program w/NVCC and run – completed
5. Write simple standalone project, compile and run – completed
6. Design data structures for n-body problem
   1. Java – completed
   2. C – completed

**Specific Requests**

The performance metrics of each of the implementations of the n-body problem will be measured in wall clock time in milliseconds. The resolution of these measurements is capped at milliseconds because that is the highest resolution that Java provides. The measurements of performance will only measure the run time within the n-body algorithm. I will not be measuring the allocation of arrays, any other preprocessing, or any other post processing.

Java

The method *System.currentTimeMillis()* will be used in the Java implementations. This method returns the current time in milliseconds from January 1, 1970 UTC. This allows a relative measurement to be made from two times reported before and after the n-body code. The code in Java will be implemented as follows:

fStart = System.*currentTimeMillis*();

MyJavaNBodyCode();

fStop = System.*currentTimeMillis*();

runTimeInMilliseconds = fStop – fStart;

C

There is no direct analog in C to the method *System.currentTimeMillis()*. The approach that most closely matches the Java method is the UNIX *gettimeofday()* function. This in included in the header ‘sys/time.h’.

The function *gettimeofday()* returns the current time, expressed as seconds and microseconds since the January 1, 1970 UTC. This also will allow a relative measurement to be made between the times before and after the n-body code. The code in C, as well as CUDA, will be implemented as follows:

gettimeofday(&startTime, NULL);

MyCNBodyCode();

gettimeofday(&stopTime, NULL);

runTimeInMilliseconds = ((stopTime.tv\_sec - startTime.tv\_sec)\*1000 +

(stopTime.tv\_usec - startTime.tv\_usec)/1000.0)

+ 0.5;

The equation used here is a conversion from the returned time as a combination of seconds and microseconds to milliseconds. The first line takes the different in seconds and coverts to milliseconds by multiplying by 1000. The second line takes the different in microseconds and converts to milliseconds by dividing by 1000. The +0.5 at is needed to round the result to the nearest millisecond.

To resolve any issues with background processing changing the wall clock processing time of the algorithm, like servicing interrupts and other operating system functions, many runs of each implementation will be averaged.

Confidence intervals of the run times will be computed to provide an accurate expectation of performance.

Using these measurements the following table and subsequent visual graphs will be able to be created.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *# of Particles*  *Implementation* | **128** | **1024** | **2048** | **4096** | **Etc... 🡪** |
| **Java** | - Average run time (milliseconds)  - Confidence Intervals | … | … | … | … |
| **Java Multithreaded** | - Average run time (milliseconds)  - Confidence Intervals | … | … | … | … |
| **C** | - Average run time (milliseconds)  - Confidence Intervals | … | … | … | … |
| **C Multithreaded** | - Average run time (milliseconds)  - Confidence Intervals | … | … | … | … |
| **CUDA** | - Average run time (milliseconds)  - Confidence Intervals | … | … | … | … |

**Current Activities**

Progress Report 2: 6/6/11 – 6/19/11

1. Implement n-body algorithm in Java – *In progress*
2. Implement n-body algorithm in multithreaded Java - *TBD*
3. Implement n-body algorithm in C - *TBD*

**Challenges**

The challenges encountered and may be encountered are:

1. Choosing the correct driver for the CUDA video card is important. I passed up on the driver that was given on the CUDA developer site for a newer version. Unfortunately, this newer version was missing libcuda.so and probably many other key pieces. Install the driver provided on the CUDA developer page.
2. Most everything worked but there were still several libraries that were missing from the install that needed to me installed separately. These are the extra commands needed.
   1. sudo apt-get install g++
   2. sudo apt-get install libx11-dev (might not be necessary)
   3. sudo apt-get install mesa-common-dev
   4. sudo apt-get install xorg-dev
   5. sudo apt-get install freeglut3-dev
3. The data structures designed may need to be changed when very large numbers of particles are run in the n-body algorithms. For now it is a simple vector containing all particle information sequentially.   
     
   (position-X, position-Y, position-Z, mass, velocity-X, velocity-Y, velocity-Z)   
     
   This can cause memory problems when this vector will be 7 \* (# particles) in length. Other possible data structures may be used as development continues. The main restriction is that it must be implementable in all three languages used, Java, C, and CUDA for comparability. That is why this most simple structure is used for now. The other possible ideas are as follows.
   1. Original sequential as shown here. *Most simple and can be implemented in all languages.*
   2. Two separate vectors. One for position-XYZ and mass. Other for velocity-XYZ. *Could help with contiguous memory problems, if any.*
   3. Use other constructs such as float4. – *Need comparable Java equivalent*. *Test TBD.*

**Work to be completed by 6/19/11**

1. Implement n-body algorithm in Java – *In progress.*
2. Implement n-body algorithm in multithreaded Java - *TBD*
3. Implement n-body algorithm in C - *TBD*

**A.2 Capstone Project Week 7 Status Report for “Comparative Analysis of Language Dependent Sequential, Multithreaded, and GPU Implementations of the N-Body Problem”**

**Jessica Cunningham**

**Date:** 6/19/11

**Accomplishments**

Progress Report 2: 6/6/11 – 6/19/11

1. Implement n-body algorithm in Java - completed
2. Implement n-body algorithm in C - completed
3. Implement n-body algorithm in Java multithreaded – 80%

**Current Activities**

Progress Report 3: 6/20/11 – 7/3/11

1. Finishing touches on implementation of n-body algorithm in Java multithreaded – *In progress*
2. Implement n-body algorithm C multithreaded – *In progress*

**Challenges**

1. When programming the Java multithreaded implementation I ran across the decision of whether to use existing objects for multhreading or to do it manually. In the past I had used the Executor interface in Java. An Executor is normally used instead of explicitly creating threads. This example is taken from the JAVA doc.

For example, rather than invoking new Thread(new(RunnableTask())).start() for each of a set of tasks, you might use:

Executor executor = *anExecutor*;

executor.execute(new RunnableTask1());

executor.execute(new RunnableTask2());

...

Though this approach is more robust and ultimately more simple, I decided not to use it. I needed to program the multithreading explicitly so that I can recreate the same multithreading scheme in the C implementation as C does not have these existing objects available. The multiple different implementations need to be programmed as similarly as possible to one another so that when the execution time comparisons are done, I am comparing “apples to apples.” This is why the Java multithreading is only at 80% instead of completed.

1. Keeping with the “apples to apples” mentality I know that in the implementation of the launchers coming up that I will have to be very careful about where I place the wall clock timer starts and stops. It is important to compare the overhead of the actual algorithm but not the overhead of the initiation of each of the n-body implementations. Should not be difficult, but needs to be paid close attention to.

**Work to be completed by 7/3/11**

1. Finish implementation of n-body algorithm in Java multithreaded – 80%
2. Implement n-body algorithm C multithreaded – *In progress*
3. Implement n-body algorithm CUDA - *TBD*
4. Implement launcher - TBD
   1. C
   2. Java
5. Add wall clock timers - TBD
   1. C
   2. Java

**A.3 - Capstone Project Week 9 Status Report for “Comparative Analysis of Language Dependent Sequential, Multithreaded, and GPU Implementations of the N-Body Problem”**

**Jessica Cunningham**

**Date:** 7/3/11

**Accomplishments**

Progress Report 3: 6/20/11 – 7/3/11

1. Implement n-body algorithm in Java multithreaded – completed
2. Implement n-body algorithm in C multithreaded – 75%
3. Implement n-body algorithm in CUDA – completed
4. Implement launchers
   1. Java – completed
   2. C – *TBD*
5. Implement wall clock
   1. Java – completed
   2. C – *TBD*

**Current Activities**

Progress Report 4: 7/4/11 – 7/17/11

1. Finishing touches on implementation of n-body algorithm in C multithreaded – *In progress*
2. Implement launcher in C – *In progress*
3. Implement wall clock in C – *In progress*
4. Run and verify each implementation – *In progress*

**Challenges**

1. The writing of the critical section containing the n-body algorithm of each thread was easy, but the allocation of shared memory for all threads caused some problems. I unfortunately spent a lot of time writing a semaphore and trying to make sure that the memory was not being written to by more than one thread at a time. This is not needed in the current implementation of the n-body algorithm. Each thread will only ever be writing to its designated space in the allocated memory. In this way I do not need to implement any locking mechanism such as a mutex or a semaphore. Instead I just need to send a handle to each of the threads to the allocated memory space. I was making the problem much more difficult than it needed to be.
2. Because of the time spent on multhreaded C I was not able to fully complete the C launcher and wall clock. This is only listed here because it has fully slipped past the deadline set on the Gantt chart. This will not cause problems as I have made sure to leave room in the schedule for such types of slips.

**Work to be completed by 7/17/11**

1. Run and verify each implementation
   1. Varying number of time slices – *In progress*
   2. Varying number of bodies – *TBD*
2. Gather and analyze results
   1. Generate performance trends for number of bodies – *TBD*
   2. Generate performace treads for number of time slices – *TBD*
   3. Draw and explain conclusion – *TBD*

**A.4 - Capstone Project Week 11 Status Report for “Comparative Analysis of Language Dependent Sequential, Multithreaded, and GPU Implementations of the N-Body Problem”**

**Jessica Cunningham**

**Date:** 7/17/11

**Accomplishments**

Progress Report 4: 7/4/11 – 7/17/11

1. Finishing touches on implementation of n-body algorithm in C multithreaded – *completed*
2. Implement launcher in C – *completed*
3. Implement wall clock in C – *completed*
4. Run and verify each implementation – *completed*
5. Gather and analyze results – *In progress*

**Current Activities**

Progress Report 5: 7/18/11 – 7/31/11

1. Analyze results – *In progress*
2. Generate performance trends for # bodies – *In progress*
3. Draw and explain conclusions – *In progress*
4. Write final report – *In progress*

**Challenges**

1. When finishing the C multithreading, yet another major problem arose. During the execution of runs with higher number of particles a segmentation fault error kept getting thrown. After some research I found that there are a maximum number of allowed concurrent threads. It seems to be an operating system limit, not a language limit. To fix this I ended up having to cap the number of threads open at one time. The system I am running on has 4 processors so really any number of threads above 4 is theoretically not going to add performance. I decided to cap at 32 threads to give each processor some extra room to play to make sure that the processors were always being utilized to the maximum. 32 was also chosen because it is a multiple of all number of particles being run. This will minimize the times that less than the cap would be executed at one time, which could cause a performance drop.
2. In running these implementations it became clear that it is not needed to vary the number of iterations of the molecular dynamics model to gain insight into the speed of each algorithm. Just using one set number of iterations through each implementation gives the data that is necessary. 30 iterations within the molecular dynamics model is used.
3. The main challenge this progress report was verifying that each of the 5 implementations of the n-body algorithm were in fact giving the same results, making sure that performance comparisons were “apples to apples” and no mistakes were giving false results. This took quite a bit of debugging though this was not surprising. In the end all 5 algorithms result in the same data output. Here are the outputs of the first particle of the runs with 128 particles from each implementation. The full output files of each of the runs for all number of particles will be included in the files turned in with the final report.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Java Serial** | **Java Multithread** | **C Serial** | **C Multithread** | **CUDA** |
| 1.52588E-5 | 1.52588E-5 | 1.52588e-05 | 1.52588e-05 | 1.52588e-05 |
| -0.5579325 | -0.5579325 | -0.557932 | -0.557932 | -0.557932 |
| -0.06652169 | -0.06652169 | -0.0665217 | -0.0665217 | -0.0665217 |
| -0.64009863 | -0.64009863 | -0.640099 | -0.640099 | -0.640099 |
| -0.085472465 | -0.085472465 | -0.0854725 | -0.0854725 | -0.0854725 |
| -0.0987731 | -0.0987731 | -0.0987731 | -0.0987731 | -0.0987731 |
| 0.33613238 | 0.33613238 | 0.336132 | 0.336132 | 0.336132 |

**Results**

This is the first draft image of all data collected for all runs through all implementations. The C serial implementation ended up being overall the slowest. CUDA was amazingly fast. More graphs and in depth analysis to be added in the final report.

**Work to be completed by 7/31/11**

1. Analyze results – *In progress*
2. Draw and explain conclusion – *In progress*
3. Write final report. - *TBD*

**Appendix B - Hardware Requirements**

**Either a 32-bit or 64-bit operating system can be used for execution of this project but it is imperative to install all software respective to the operating system being used.**

**An NVIDIA CUDA enabled GPU device is required to run the CUDA software of this project. A complete list of CUDA-enabled GPUs can be found on the NVIDIA CUDA web site at http://www.nvidia.com/object/cuda\_gpus.html.**

**Multiple processing cores are technically required to show the true power of the multithreading portion of the project but can be run without multiple cores.**

***Note: Results for all implementations will vary greatly depending on the specific hardware available.***

**Appendix C - Software Configuration**

**Be sure to keep in mind whether the system used is a 32-bit or a 64-bit operating system. All software installation instructions below are for a 64-bit operating system. Information is available on provider websites for slight changes needed to accommodate a 32-bit machine.**

**A CUDA supported version of Linux is required.19 The supported versions for the most current NVIDIA tools are as follows. All are available online and Ubuntu is downloadable free of charge.**20

**Supported Operating Systems (32-bit and 64-bit)**

**- Red Hat Enterprise Linux 5.5\***

**- OpenSUSE 11.2**

**- SUSE Linux Enterprise 11 SP1\***

**- Fedora 13\***

**- Ubuntu 10.04\***

**\* New support with CUDA v3.2**

**The gcc compiler is required for both the C and CUDA implementations. This should be pre-installed with the supported version of Linux. If it is not installed, it is available online at http://gcc.gnu.org/releases.html.**

**A Java environment is required to execute the Java implementations. Many are available online. Eclipse is downloadable free of charge.**21

**It is assumed that these previous baseline installations can be completed by the user. If not, ample installation instructions are available from the providers.**

**Three additional software tools must be installed for all CUDA execution:**

1. **NVIDIA driver - The NVIDIA driver required for all supported versions of Linux can be found at** http://www.nvidia.com/object/cuda\_get.html. Run the driver installation package as *superuser*. More detailed information on installing the driver if the typical installation does not work is available at http://us.download.nvidia.com/XFree86/Linux-x86/256.35/README/index.html.
2. **CUDA toolkit – The CUDA toolkit can be downloaded from** http://developer.nvidia.com/cuda-toolkit-32-downloads#Linux. **Run the .run file provided in the download as *superuser* and install into the default directory provided. Update the environment variables within the ~/.bash\_profile using the following commands:**

export PATH=/usr/local/cuda/bin:$PATH

export LD\_LIBRARY\_PATH=/usr/local/cuda/lib64:$LD\_LIBRARY\_PATH

1. **CUDA SDK. The CUDA SDK can be downloaded from** http://developer.nvidia.com/cuda-toolkit-32-downloads#Linux. Run the .run file provided in the download as a *regular user* and install into the default directory provided.

Verify CUDA installation

Using a terminal window, go to the default installation directory of the SDK ~/NVIDIA\_GPU\_Computing\_SDK/C and type make. This will install the subsequent binary files in ~/NVIDIA\_GPU\_Computing\_SDK/C/bin/linux/release.

Go to the new release directory and run deviceQuery. If the CUDA configuration was successful the output of the device query should be as follows.

-----------------------------------------------------------------------~/NVIDIA\_GPU\_Computing\_SDK/C/bin/linux/release$  
./deviceQuery

./deviceQuery Starting...  
  
 CUDA Device Query (Runtime API) version (CUDART static linking)  
  
There is 1 device supporting CUDA  
  
Device 0: "GeForce GT 230M"  
 CUDA Driver Version:                           3.20  
 CUDA Runtime Version:                          3.20  
 CUDA Capability Major/Minor version number:    1.2  
 Total amount of global memory:                 1073020928 bytes  
 Multiprocessors x Cores/MP = Cores:            6 (MP) x 8 (Cores/MP)  
= 48 (Cores)  
 Total amount of constant memory:               65536 bytes  
 Total amount of shared memory per block:       16384 bytes  
 Total number of registers available per block: 16384  
 Warp size:                                     32  
 Maximum number of threads per block:           512  
 Maximum sizes of each dimension of a block:    512 x 512 x 64  
 Maximum sizes of each dimension of a grid:     65535 x 65535 x 1  
 Maximum memory pitch:                          [2147483647](tel:2147483647) bytes  
 Texture alignment:                             256 bytes  
 Clock rate:                                    1.10 GHz  
 Concurrent copy and execution:                 Yes  
 Run time limit on kernels:                     Yes  
 Integrated:                                    No  
 Support host page-locked memory mapping:       Yes  
 Compute mode:                                  Default (multiple  
host threads can use this device simultaneously)  
 Concurrent kernel execution:                   No  
 Device has ECC support enabled:                No  
 Device is using TCC driver mode:               No  
  
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 3.20, CUDA  
Runtime Version = 3.20, NumDevs = 1, Device = GeForce GT 230M  
  
  
PASSED  
  
Press <Enter> to Quit...  
-----------------------------------------------------------------------

To ensure that the CUDA device identified in the device query is able to communicate correctly with the system, run the bandwidth test.

-----------------------------------------------------------------------~/NVIDIA\_GPU\_Computing\_SDK/C/bin/linux/release$  
./bandwidthTest  
[bandwidthTest]  
./bandwidthTest Starting...  
  
Running on...  
  
 Device 0: GeForce GT 230M  
 Quick Mode  
  
 Host to Device Bandwidth, 1 Device(s), Paged memory  
 Transfer Size (Bytes)        Bandwidth(MB/s)  
 33554432                     4243.7  
  
 Device to Host Bandwidth, 1 Device(s), Paged memory  
 Transfer Size (Bytes)        Bandwidth(MB/s)  
 33554432                     3114.4  
  
 Device to Device Bandwidth, 1 Device(s)  
 Transfer Size (Bytes)        Bandwidth(MB/s)  
 33554432                     20098.0  
  
  
[bandwidthTest] - Test results:  
PASSED  
  
  
Press <Enter> to Quit...  
-----------------------------------------------------------------------

Installations may vary depending on the operating system and other system parameters. If the previous tests do not run properly some additional explicit installations may be required. The installation performed for this research project required the following additional libraries. Others may be required for varying installations.

sudo apt-get install g++  
sudo apt-get install libx11-dev (might not be necessary)  
sudo apt-get install mesa-common-dev  
sudo apt-get install xorg-dev  
sudo apt-get install freeglut3-dev

**Appendix D - User’s Manual**

The n-body source code for both Java and C/CUDA are provided in the .zip folder “nbody” included in the project submission. Unzip the folder to the path of your choice that will be subsequently referenced as userPath/.

For the Java implementation, import into a Java environment or directly open in an editor the source code provided in userPath/code/Java. Both the serial and parallel implementations are provided in the single project. There will be three hardcoded paths for the input and output of data that must be changed to your userPath/. To do this, find the importData, outputDataToFile, and writeTimeToFile methods near the end of the main code file. There will be a hardcoded path in each file pointing to read in the data and to write out the data and to write out the execution times. Change these respectively as follows.

“userPath/input/tab"

“userPath/output/"

“userPath/output/"

It is now possible to run the Java code directly from your Java environment or to export to a .jar file and run as an executable. The data shown this project is created from running the code as a .jar executable. The output data will be accessible in the userPath/output folder.

For the C and CUDA implementation, open the source code provided in userPath/code/C in an editor. The C serial, C parallel, and CUDA implementations are provided in the single code set. Again, the input and output paths are hardcoded and must be changed to your userPath/. To do this, find the importData, writeOutputDataToFile, and writeTimeToFile methods near the end of the main code file. Change these respectively as follows.

“userPath/input/tab%d”

“userPath/output/%s%d”

“userPath/output/%s%d”

Compile and execute the code within your userPath/ using the command line arguments

nvcc -o nbody nbody.cu

./nbody

Again, the output data is accessible in the userPath/output folder.