Benjamin Steenkamer
CPEG 652-010
5/17/2019

Parallelizing an N-Body Simulation with Pthreads and OpenMP

Introduction:

In this project, I took on the task of parallelizing a sequential, planetary gravitation (n-body) simulation with Pthreads and OpenMP. The original, n-body simulation comes from a previous class I took as an undergraduate. The sequential C program was distributed by the professor of that class with a GNU license, and I found that the algorithm had great potential for parallelization speedups. The n-body simulation works by taking in an input file that contains the number of planets (bodies), the number of time steps to be done in the simulation, and the masses and starting positions of each body in a 2D field. The program then does the simulation by updating the effects of each planet's gravity on the other planets for a given step. The following step then uses the updated planet positions and velocities to calculate the next position of each planet and so on. After a certain number of steps are calculated, the program will write out an image of the current body field using the GD library (by default, this writing occurs every other frame). At the end of the simulation, all the images together create an animated .gif file of the simulation, which shows the planets speeding up and slowing down as they are attracted and flung away from other planets. The simulation allows for an arbitrary amount of bodies and calculation steps, which for large values is what makes this simulation slow from a single threaded program standpoint.

I wanted to mention that this project setup differs in some ways from the one which I proposed at the beginning of the semester. As we had discussed previously, I had initially chosen to write an n-body water simulation from scratch and attempt to parallelize that. At your suggestion, I decided to not do this and instead use this planetary n-body simulation instead, as the sequential version already existed. I also decided to drop my initial plans of incorporating an OpenGL output to display the animation, as I thought that was not important to the core goals of this project. I also decided to drop the MPI parallelization I initially planned from this project, as I wanted to focus on understanding the programing models for Pthreads and OpenMP instead.

Work Distribution:

In total, I created four different parallelized version of the n-body simulation. They are nbody_pthread_v1, nbody_pthread_v2, nbody_omp_v1, nbody_omp_v2. All these versions take the base version of the simulation (nbody_seq) and use Pthreads or OMP to create a parallelized implementation of the sequential version. All these version are functionally equivalent, as they produce the same output, but with the parallel versions doing the same work in a shorter amount of time. Also, in the submission files is a program (created by the previous class's professor and modified by me) called test_maker that is used to generate a large, random field of bodies to test the performance of the main n-body program. I included some test files generated with this program in the "Tests" folder, so you can just use those files as input and not have to generate your own.

Since the original sequential version was already created, I didn't need to spend much time on it. I did spend about 3 hours going over the code, cleaning up functions, and removing unneeded

sections to make the program easier to read. To create all the parallel versions and optimize them, it took me about 30 hours. The time spent on this was divided among reading about how to use Pthreads and OMP, trying different combinations of the features offered by the parallel models, testing for correctness, and looking for places and trying to use parallel enabling or optimizing techniques. In the end, I wrote or edited about 300 lines total among the four parallel versions. The parallel versions range in size form ~430 lines to ~470 lines while the sequential version is 379 lines. This final line count does not include the code I wrote and then deleted after finding out it was incorrect or made performance worse. I would say I wrote or edited about 450 lines total, including the code I have now plus the code I removed. The time spent to write the lines was high because I had to do a good amount of review for the parallelization techniques and reading how to use Pthreads and OMP.

I spent about 8 hours doing the time and performance analysis of the sequential and parallel programs. This included finding parameters that would best demonstrate the scaling benefits of my parallel versions and recording the actual times of each program with a different number of threads, compiler flags, and inputs files. Also included in this time is the creation of the graphs and analyzing the speed ups and efficiencies of each parallel version to see if they made sense. In total, I spent about 40 hours on this project, which is about 3.5 hours per week for 12 weeks. The notes I have from each week are in the Google document I already shared with you. They are too long to paste into this report, so here is the link:
https://docs.google.com/document/d/1sQ6EWT_ID4hM-7GZqADnvUApZHkGaBKI41ks6zsJ12E
In the same Google drive, I have also shared a folder with the code. The same code is also included in this .zip file with the report.

Parallelizing the N-body Simulation:

To start the parallelization process of the sequential program, I chose to use OpenMP and Pthreads. I made this decision because the sequential program is written in C, which is compatible with version of Pthreads and OMP. I have also used both parallel libraries in the past, so I wanted to review my knowledge of them. Since Pthreads and OMP are based on a shared memory model, I thought it would be easy enough to maintain variants of the parallel program and see how the performance between them compares. To get these programming models up and running, I had to install the Pthreads and OMP libraries. This was not difficult, and I included the commands to do this in my Makefile. With these installed, I could compile my programs using gcc with the appropriate library flags.

Prologue functions:

Going over the general steps I took to parallelize the program, I started in the main function and worked my way through the flow of the program to see with parts could be parallelized. Overall, I found the only place that benefits greatly from parallel performance are the nested calculation loops inside the simulation step loop of the program. The prologue and epilogue code for this step loop wasn't parallelized because it either had poor parallel scaling (did not beat the overhead) or because the operations were inherently sequential (e.g. file IO).

Starting off, the first function called is the init() function, which creates the body arrays and reads in the header of the input file which contains the starting positions for each planet. This first part must be sequential since it involves memory allocation and the opening of a file. The function then goes through the input file and initializes the main bodies array with this input data. It then does another loop after where it copies this data to the secondary bodies array used in the simulation step. I did not parallelize these loops in this input procedure because the amount of work and time spent in these loops was not enough to justify the overhead of using multiple threads and coordinating where in the file each thread reads. During my tests, I found that even for what I consider a large number of planets in the input file (n=10,000), a single thread would still beat two or more threads in execution time. The single thread beat the parallel threads because the work of reading from the file and writing to the arrays has so few operations that it only took a few microseconds. Even combining or leaving separate the two copy loops did not make a significant difference for the single thread performance since it was so fast.

For these initialization loops, adding more threads made the parallel versions even slower since while the number of tasks given to each thread would get smaller, the overhead from creating more threads increased. For my performance tests, the number of planets in the input file would range from 2000 to 3000, so the slowdown from the parallelization overhead would be even worse compared to 10,000. For this initialization to start benefiting from parallelization, I estimate that there would have be around 100,000 planets in the input file. However, 100,000 bodies for just 1000 steps of simulation would take a very long time to calculate in the simulation loop (in the range of hours for a single thread), so I did not consider this parameter size reasonable for my purposes. Because of these reasons, I left the init() loop sequential. After all these loops, the init() function then finished with a call to prepgif(), which also has been sequential, since it involves opening a file and allocating memory.

Following the init() function is the write_frame() function to write out the first frame of the output .gif file. This function is also called in the main step function of the simulation when a certain number of frames have been processed (when “step % period == 0”, the frame is written). The first part of this function involves creating the usable colors or copying the background palette for the image. Since palette copying only needs to be done once per frame, it has to be sequential (done by one thread). The work done inside the color loop is also too small to benefit from parallelization (only 254 iterations).

Following this, there is then a loop to write out the bodies to the frame. This requires function calls to the gd library, which are thread safe if there is only one thread writing to a given image. Since there is only one image being written out for this program, it would not be safe for multiple threads to call these functions on the same file. Therefore, a critical section would be needed on the gdImageFilledEllipse() function to be thread safe. However, in a similar circumstance to the init() function, the iteration space of this loop was too small to benefit from parallelization. The addition of the critical section also added overhead and increased the runtime of the loop compared to a single thread without critical sections. For this parallel loop to beat the overhead of the synchronization, the number of bodies in the simulation had to be very large. Again, since this massive number of bodies is far larger than the parameters I would be using to test with, I decide to leave this loop purely sequential, with no parallelization or critical sections. Afterwards, there is again a section that calls gd functions that are only needed once per frame, so

they have to be handled by one thread. Overall, the relatively small amount of iteration space made it not feasible to parallelize this function given my intended input data.

Main update loop:

Up to this point the program has been sequential for various reasons. However, now that everything is setup, the main "step" loop of the program can be started. For each step, the update function is called, which calculates the gravitational effects each body has on the other. Once this update is done, the write_frame() function is called if the specified number of frames have been processed (for my tests I left the period value at a default of two, so a frame is written every two steps). As mentioned before, this write_frame() function is sequential. However this update() function can be parallelized, and it is where the main performance improvement from parallelized can be had for the overall program. On thing to note is that that this outer step loop can't be parallelized, as each future step relies on the information of the previous step. Therefore, the parallel threads will have to execute within the current step, wait until every thread is finished with the step, and then all move on to the next step.

With this call to the update() function, I found that there were two main ways to do parallelization. The first is to start the threads at the start, let them do their portion of the update function in parallel, and then join them after the update is done. Then a pointer operation on the global arrays (has to be sequential) and write_frame() function can be called by the master thread. The next step can then start with the master thread creating the threads and so on. This method was implemented in the nbody_pthread_v1 and nobody_omp_v1. In terms of parallel functionality, these two versions can be considered equivalent since they both spawn and join the threads at the start and end of each step. The only real difference between the Pthreads and OMP versions is the syntax and language requirements. Pthreads needs a prologue and epilogue loop to call the pthread_create() and pthread_join() functions for every thread. For OMP, just a #pragma omp parallel num_threads(num_threads) is needed at the start.

I realized that this method would have some overhead with the creating and joining for each loop of the step, so I then created nbody_pthread_v2 and nbody_omp_v2. In these versions, the threads are only created once at the very start of the step function and are joined only once at the end. To prevent write_frame() and the global pointer operations from being done by multiple threads, I used a barrier and thread IDs to only allow one thread to do this work. The barrier prevents the write_out() from occurring until all threads have finished the current step. For Pthreads, I had to use explicit barriers and single out thread 0 using an if statement to have it do the writing. However, with OMP I was able to use just the implicit barriers at the end of the OMP work distribution sections and the #pragma omp single to let any single thread do the write_out() and pointer work. To make it easier to view this new layout, I did some slight code arrangement by placing the step loop inside the update function. This doesn't change the functionality of the program as the threads still all work in parallel within the same step of the simulation. This rearrangement just allowed me to have all the parallelization starting/end points and loop distribution code in the same function.

Iteration Space Distribution:

For a given step of the update() function, the program needs to execute a doubly nested loop. The outer loop iterates over all the bodies and the inner loop is for calculating the effects of all other bodies on the current body from the outer loop. For several reasons, I was only able to distribute the outer loops iteration space among the threads. I will explain these reasons in the next section. For both Pthreads versions, the outer loops iteration space is divided into equal blocks. I chose block distribution because it would allow each thread to have good cache locality when reading from the bodies array and when it writes to the bodies_new array. The arrays are global variables, and bodies_new is written to by each thread with the new body data from the current step. With block distribution, each thread will get its own, equal sized chunk to write to in a stride-one manner, which provides good spatial locality and effectively removes most false sharing issues. There is still potentially a false sharing issue on cache pages where one thread's block starts and the previous one ends.

For partitioning the outer loop with the OMP v1 and v2 versions, I tried several different distribution systems, since OMP makes it easy to do. I tried the static, dynamic, guided, and auto arguments for scheduling and compared the timing results of each. For the static distribution size with no chunk size specified, the iteration space is divided into about equal size chunks, and only one chunk at most is given to each thread. This is the same way the Pthread versions distribute their iteration spaces. I initially chose static scheduling for the same reasons I chose it for the Pthreads versions. For dynamic and guided, I didn't see any significant performance improvements for the chunk sizes I specified. In some cases, the chunk sizes I chose made the performance worse for a given number of threads. For auto, "the decision regarding scheduling is delegated to the compiler and/or runtime system" and it can "choose any possible mapping of iterations to threads in the team" (quotes from the OMP documentation). When using auto, I found that this scheduling type ran better for most thread counts compared to the other types. With 4 threads, 1000 bodies, and 1500 steps, the OMP auto version executed over 1 second faster than the static version. I found that the scheduling type mattered less as the number of threads increased, but auto seemed to perform consistently better, so I chose that scheme. I believe auto works better than the others because it allows the runtime to potentially choose a dynamic or guided distribution with consideration to how many threads are available and the size of the iteration space. It can find a more effective chunk size than I was able to and make the overall performance better for a variety of thread counts. The benefit of a dynamic or guided scheduling is that is creates a producer/consumer style distribution that would allow faster threads to pick up the slack of slower threads. This is opposed to static distribution, where each thread always gets the same amount of work each time. If a thread finishes faster than all the other, it will just have to wait, which is waste of resources.

Trying to Parallelize/Optimize the Inner Loop:

Going back to the inner loop issue mentioned earlier, I was having trouble finding any real chance for using the various parallel enabling or optimizing techniques we discussed in class (strip-mining, induction variables, forward substitution, loop tiling, loop interchange, etc.) as the dependencies of this inner loop would not be resolved or prevented the use of these techniques. The issue is that the inner loop needs several initial values from the outer loop. The following part of the outer loop after this then depends on results of the inner loop. I tried relocating the inner loop before the outer loop to see if there was any benefit from precomputing the inner loop values, but this just shifted the same amount of work earlier in the program, as it still required two nested

loops. This also added an overhead from extra arrays needed to store the data for use in the following loop. I tried to collapse this now relocated loop, but it was not in a perfectly nested form. To make it perfectly nested, I would have to add another loop before that to initialize the arrays where values are written to. This made the program run even slower, as there were now three, for-loops total (two single, one double nested, but collapsed) instead of just being one double nested for-loop originally. Overall, the need to add these extra loops introduced too much overhead compared to the original version. Because of these issues, I left these two nested loops are they were in the original sequential program and just parallelized the outer for-loop. This gave the best performance.

After all steps have been calculated in the step loop, all that remains is the warpup() function, which finalizes the .gif file and frees the allocated memory. This must be sequential, so all spawned threads will be joined at the end of the final step loop to allow just the main thread to do this. At this point, all sections of the program that I could parallelize have been parallelized and, it is time to test the performance.

Testing and Results:

Since I chose to use Pthreads and OpenMP as my parallel languages, I needed to choose a parallel, shared memory machine to run my tests on. I decided to use a lab server I have access to, which has two Intel Xeon E5-2690 V3 processors, with clock speeds from 2.60 GHz to 3.50 GHz and 128 GB of RAM. Each processor has 12 cores with 24 threads (2 threads/core). This allows for a total of 48 threads. The large number of parallel threads and the fact that it is a shared memory system makes this a good choice to see how well my programs scale with an increasing thread count. To test the timing of each program, I added several timing functions to record the total execution time of the program and the total time each thread spent executing code in the main step loop. I found that inserting the timing functions adds ~0.001 seconds to the execution of 1000 loop iterations. This has a negligible impact on the timing of the overall program if I only measure the time of the outer loop in the update function. I did not place timing functions inside the inner loop of the update function because then this overhead would become significant with how many times the inner loop is run.

For testing, I had to select a large number of bodies and steps so that the program speedup would be clearly visible as the thread count was increased. After some tuning, I selected 3000 bodies with 2000 steps for the main tests of the sequential, 2, 4, 8, 16, 32, and 48 threaded parallel programs. When run with a single thread, all parallel versions ran at essentially the same speed or marginally slower than the sequential version, so I excluded these one-threaded results and just compared each parallel version's multithreaded runtimes to the original sequential version to provide a consistent baseline. I did not test any versions with higher than 48 threads, as I believe this would cause threads to starve each other of processor resources on the parallel system I selected, and it would result in very poor performance. The system only supports 48 threads, so I did not want to go higher than this (please also see the final note at the end of this report). I also added the option to every program to turn off the creation of the .gif file (-NO_OUT). I did this because I wanted to see how much of the runtime was spent inside the sequential part of the otherwise parallel update() function. I also decided to test if gcc compiler optimizations (-O0 vs -O3) would make a significant difference in performance. Most tests were run with just the -O0 optimization as I believed this would best demonstrate the benefits of the parallel performance

optimizations I previously described. A few more tests were done with the -O3 and -NO_OUT to see how much difference they'd make. Finally, I did one last test where I ran the -O0 version again, but with switched body and step values (3000 steps and 2000 bodies).

Since I generated many graphs from these timing test, I will not paste them into this report. They are in the separate Timing.xlsx file included in this submission. The first sheet in the document contains the raw results for each test. The second sheet contains the graphs of the most interesting or relevant results. The first column of graphs shows the speedup (left y-axis) and efficiency (right y-axis) of each parallel program version for 2 up to 48 threads compared to the original sequential version. The sequential version's times can be viewed at the top of the data sheet. The second column shows the graph comparing the runtime of each parallel version. The third column shows how the -O3 and -NO_OUT flags affected each program's speedup and efficiency compared to the -O0 program for 48 threads. The final column shows the switched parameter speedups (3000 steps and 2000 bodies) versus the regular parameter tests (2000 steps and 3000 bodies) for each program version.

As explained previously, each version of the parallelized program has sequential prologue code to set up the arrays and image file before the main step loop and sequential epilogue code that frees memory and finishes up the .gif at the end. This sequential code, in theory, acts as a bottleneck for the maximum parallel speedup of each program. However, as the results I've collected show, the time it takes to do these specific sequential sections on a single thread ended up being very small compared to the time taken inside the main step loop. Looking ahead, this can be seen with the OMP v2 speedup graph where it has effectively linear speedup for up to 16 threads. The drop-off after 16 threads has to do with something else I will explain in a future section, but this linear region tells me the sequential prologue and epilogue code around the step loop is effectively inconsequential to the runtime of the program. This makes sense because the prologue has only three for-loops that iterate over numBodies and are only run once in the entire program. In my tests, numBodies is 3000, so this totals at 9000 iterations. A single thread in the main step function will do significantly more iterations than this for just one step. For example, say there are two threads, then for any one step, a thread will do approximately 1500*3000 = 4,500,000 loop iterations (or for 48 threads, about 187,500 iterations). This number of iterations is then done for nsteps=2000 loops. Because of this, the step loop's execution time far outweighs the sequential prologue and epilogue code of this program, meaning they don't pose a significant performance impact to any of the parallel versions for my current testing setup. As I will explain next, the loss of efficiency in the parallel versions has to do with either the method of parallelization, the work distribution scheme, or the sequential write out that occurs within the step loop.

Pthread v1 Performance:

Looking at the Pthread v1 speedup and efficiency graph, it can be seen that having more threads decreases the time it takes to perform the n-body simulation compared to the sequential version. However, the graph also shows that Pthread v1 did not achieve a linear speedup even with just two threads. The speedup continues to increase but becomes less significant as the number of threads increase to 48, while the resulting efficiency drops. It then caps out at just under 11 times speedup for 48 threads compared to the sequential version. This performance gain over the

sequential version is the result of the outer loop iteration space in the step loop being distributed among multiple threads, which run in parallel. This is opposed to the sequential version, were one thread is tasked with doing all the work by itself. The benefit gained from distributing the outer loop's workload for execution in parallel is true for all the parallel versions I created, and it is the source of the general performance improvement over the sequential version. In theory, adding more threads will continue to improve this performance gain as each thread will have less work allocated to it while the total work for the step still gets done, meaning the step is completed faster overall and the total runtime is decreased.

However, the significant loss of efficiency and the non-linear speedup demonstrated by Pthread v1 shows that there is a major bottleneck, which prevents this speedup from linearly increasing as more threads are added. This bottleneck could be due in part to the sequential frame write out that happens every other step. This represents time where the parallel threads are not doing work because they need to wait for the main thread to finish this purely sequential section before spawning them again. Every moment a thread is not doing work is a waste of resources and a reduction of the parallel speedup, but it is unavoidable with this current program setup. However, what I believe to be an even bigger inefficiency for this version is the fact that for every step of the simulation, the main thread creates the threads to run inside the step loop and then joins them at the end of each step.

The creating and joining of threads takes the main thread time and resources to do, which means it is delayed by this other work before it can enter the parallel code section and get the important calculations done. This is evident from the individual thread timing results (see data sheet), which for 48 threads shows that the main thread spent 40.8 seconds total in the step loop, while the average worker thread spawned from the main thread only spent 19.8 seconds total doing work in the step loop. This means that over the course of execution, the main thread had spent 20 seconds dealing with other things like the overhead of creating and joining threads on top of doing the normal calculations for the simulation steps. This is clearly a major bottleneck to the speedup of the program. The disparity between the worker and main thread execution times is the worst at 48 threads, but it decreases as the number of threads used goes down (for two threads, the disparity is 6 seconds total). Therefore, it can be said that adding more threads to Pthread v1 incurs more sequential overhead for every step loop iteration. This explains the poor scaling and non-linear speedup of Pthreads v1 as more threads are added.

Pthread v2 Performance:

With Pthread v2, I solved the issue of creating and joining the threads for each step loop iteration. I did this by instead creating the threads once before the start of the step loop and then joining them only once at the very end of the step loop. Then within the step loop, I use Pthread barriers to ensure only the main thread does the write out of the frame. As can be seen from the graph in the separate document (column 1, graph 2), Pthread v2 achieves much closer to linear speedup for up to 8 threads compared to Pthread v1. When 48 threads are used, the speedup over the sequential version for v2 is nearly twice that of v1 (~11 for v1 and ~20 for v2). The efficiency of v2 is also almost twice as much as v1 for 48 threads. Because Pthread v2 has drastically better speedup and efficiency compared to v1, this confirms to me that the main bottle neck of v1 was

the overhead of creating and joining the threads. Overall, Pthreads v2 scales much better from the sequential version compared to Pthreads v1.

However, even with this thread creation overhead mostly removed, Pthread v2 still does not have linear speedup for 8 threads and higher. This means that a sequential bottleneck is still present in the step function that is preventing greater parallel scaling with more threads. As mentioned earlier, there is the sequential write out function that occurs every other step. This requires all threads to wait at a barrier until the main thread has switched the pointers for the old and new arrays and has started the write out process. When the main thread is writing out, the other threads are free to continue calculating the next step. The write out and the next step will only be reading from the old bodies array so there will be no interference. However, the barrier at the end of the outer calculation loop means all the other threads must wait until the main thread's execution catches back up to them before the next pointer switching and write out can occur. In this way, all threads will incur the same slowdown of the main thread doing the sequential section. This is confirmed by the results I collected (see the data tab in the spreadsheet) where it shows the total amount of time the main thread spent in the step loop is the same as the average time a worker thread spent in the loop, for all thread counts. Every time the threads are waiting at a barrier, it means they are not doing work and is a loss of efficiency. This caps the speedup of most of the program to the speed of the main thread doing the write out, which prevents perfect linear speedup.

Another possible cause of this nonlinear speedup is that as the number of threads increases, the iteration space distributed to a given thread is decreased. Since Pthreads v1 and v2 use block distribution, all threads get an equal sized chunk of the bodies array to read from and the corresponding chunk from the bodies_new to write to. As more threads are added, the total number of borders between thread chunks increases in the bodies_new array. In short, this means that there will be more chances for false sharing to occur in the bodies_new array as one thread writes to an index in its chunk that is in the same cache line that contains part of another thread's chunk. This false sharing results in cache invalidation and a re-fetching of the cache line, which causes performance loss. As more threads are used, there will be more chunks created and therefore more edges where false sharing occur. This helps explain why the speedup of Pthreads v2 decreases as more threads are added to the equation, resulting in non-linear speedup and reduced performance. This issue would also affect the Pthread v1 version as well, but I believe the overhead of creating and joining the threads dominates the overhead from this issue.

OMP v1 Performance:

The OMP v1 version of the program is like Pthreads v1 in that it spawns the threads at the start of every step and ends them at the end of every step. However, despite this similarity, OMP v1 manages to have much better speedup, performance scaling, and overall efficiency versus the sequential program compared to Pthreads v1. Looking at the graph for OMP v1, it has very good parallel scaling up to 8 threads but then begins to fall off. At 48 threads, OMP v1 has a max speedup of about 18 compared to 11 of the Pthreads v1. This difference in performance can also be seen in the smaller disparity between OMP v1's main thread total step time and the average worker total step time. For example, with 48 threads the difference between the total main thread and average worker step times is 6 seconds, compared to the 20 seconds difference of Pthread v1.

This smaller difference indicates to me that OMP has a different backend scheme for handling the end of a parallel section. In the Pthread v1 version, the spawned threads terminate at the end of one step loop iteration. They then need to be created again at the start of the next step. For OMP, it seems that this full termination at the end of the step function does not happen. Instead OMP most likely puts these threads to sleep at the end of the parallel section and then wakes them up at the start of the next parallel section. I could not readily find a definitive answer over whether this is what really happens, because the official OMP documentation I read was sparse on information about the backend implementation of thread management. Regardless, the OMP version is obviously more efficient at thread management than the Pthreads v1 version, which means less sequential overhead is incurred by the main thread. This results in the better parallel scaling observed in the test.

Even with this better thread management, OMP v1 still doesn't maintain linear scaling and plateaus in speedup from 32 to 48 threads. There could be several reasons for why this happens. The first is that the auto scheduling method I used may not have picked the most effective scheduling technique for the given thread amount and iteration space. For instance, it may have picked a chunk size that was too small for the 32 and 48 thread runs, which would then play into the false sharing issue I mentioned earlier. With smaller chunk sizes, there would need to be more chunks to cover the iteration space and therefore more chances for false sharing to occur when threads write to their chunks in the new_bodies array. This would cause a performance decrease when more threads are added, like when the test goes from 32 to 48 threads. Also, assuming OMP is actually putting the threads to sleep and waking them up, these actions still would incur more overhead than using barriers like I did with OMP v2. Indeed, the results of OMP v2 show that it had better speedup and efficiency than OMP v1. Of course, the time spent by OMP v1's main thread doing the write out every other step also adds sequential overhead, as it increases the time the main thread takes to get to the next iteration of the step loop and start the parallel section. This starting and ending of the parallel region for every step, the potential scheduling issue, and the sequential write out all contribute to the non-linear speedup observed in the OMP v1 tests.

OMP v2 Performance:

Out of all the program versions, OMP v2, performs the best. It has effectively a linear speedup for 2 to 16 threads, or in other words, an efficiency of 1.0 for 2 to 16 threads. However, the speedup versus the sequential program does eventually become non-linear and decreases in rate at 32 and 48 threads. The speedup plateaus at 32 to 48 threads most likely because of similar scheduling issues also experienced by OMP v1. At 48 threads, OMP v2 has a speedup of over 25 times that of the sequential program and an efficiency of 0.53. Like Pthread v2, OMP v2 keeps the threads alive for the entirety of the step loop but instead uses *implicit* barriers and the "single" directive to make sure the pointer operations and write_out() function are done by only one thread after all the calculations for the step are complete. The fact that the parallel section is not ended after each step gives this version a performance advantage over OMP v1. In combination with this, I believe the auto scheduling gives OMP v2 the advantage over Pthread v2, which uses block scheduling. In Pthread v2, the main thread is always tasked with writing out the frame. This will cause the main thread to fall behind the other threads in the next step, and all other threads will have to wait at the barrier for the main thread to catch up. However, with OMP v2, the auto scheduling can potentially allow for the use of a producer/consumer style distribution of the

iteration space so that other threads can pick up the slack of the thread that is doing the write out operation. There is the potential issue of increased false sharing with this scheme as mentioned before, which can explain the eventual nonlinear speedup and performance plateau. However, the results appear to show that this load balancing feature is still able to beat out the block scheduling method.

One thing to note again with auto scheduling is that it is up to the compiler or runtime system to select the schedule method, and it can pick any possible scheme available. This means it is possible for it to have chosen static block scheduling. However, I believe this was not the case because if it did choose block scheduling, the speedup for OMP v2 would be closer to that of Pthread v2's. From my tests, the speedup of OMP v2 was 25 while Pthread v2 was only 20 for 48 threads. This confirms to me that auto is choosing a more effective scheduling scheme in this scenario. Overall, the combination of OMP v2's ability to potentially choose a more efficient scheduling type and not ending the parallel region after each step are the two main reasons that OMP v2 has the best scaling and efficiency out of all the versions I created.

The fact that OMP v2 can maintain an essentially linear performance scaling up to 16 threads also confirms that the sequential code before and after the step loop have little impact on the overall runtime of the program. OMP v2 is, however, not able to maintain this perfect scaling after 16 threads. As with all other versions, the sequential aspect of writing the frames out can still have some penalty on the parallel performance, even with dynamic scheduling. There is still also the issue of false sharing with the new_bodies array. These issues together prevent OMP v2 from having perfect scaling performance with an increase in threads.

Versions Compared:

In the second column of the graph spreadsheet, I made a plot of the runtimes for the four program versions. Pthreads v1 has the slowest execution time out of the 4, followed by Pthreads v2 being considerably faster. I was somewhat surprised to see that OMP v1 almost matched the performance of Pthreads v2, considering that OMP v1 uses the same parallel section scheme as Pthreads v1. However, after analyzing the results, this makes sense because OMP v1 has a more efficient thread management system than Pthreads v1 and has the potential benefit of the auto scheduling. These two features allow OMP v1 to reduce the overhead and slowdown from various sources enough to perform as well as Pthreads v2. Finally, OMP v2 is the best performer of the four. At 48 threads, OMP v2 has more than double the speedup of Pthreads v1 (~25 vs. ~11). This difference emphasizes that both the parallel library and the way in which you use it can have a major effect on the scaling of a sequential program to a parallel one.

Compiler Flags Test:

After this main testing, I ran a small experiment to see how much impact the sequential write out has on the overall parallel scaling of each version. I did this by compiling each version with the -DNO_OUT flag to turn off the creation of the .gif. This means there would be no write out calls during the step function. I did this test for all four versions running at 48 threads and the results are in column three of the graph spread sheet. The speedup and efficiencies of these versions are relative to the sequential version also compiled with -NO_OUT. Unsurprisingly, all versions with the write out turned off increased in performance (increased speedup and efficiency) compared to the normal versions. The frame writing represented a sequential bottleneck for each

version, so by eliminating it, the parallel scaling of each version was expected to improve. However, this flag did not change the fact that barriers or the start and ending of parallel sections would still occur in each version. This is why we see a speedup and efficiency increase, but the programs are still not able to get a linear speedup as other sequential, synchronization, and scheduling bottlenecks still exist even without the write out.

I also ran the test again with the -NO_OUT and the gcc optimization flag of -O3. For all previous tests, I had specified the -O0 flag to better demonstrate the pure scaling performance of each version. In this test, I wanted to see if compiler optimizations would further increase performance. Surprisingly, the programs had worse performance compared to the -O0 version with no output. In the case with OMP v2, the optimized version with no output did worse than both the non-optimized output/no-output versions. I'm not confident on why this performance drop occurred, but one theory I have is that gcc could have done optimizations on the array writing accesses that would have increased performance for a single threaded program but increased the amount of false sharing for a parallel program. I did not do any further experiments to prove this because this was not the focus of my project. I wanted to include these results because I thought it was interesting, and it shows that optimizations for sequential programs may not also be beneficial for parallel ones.

Switched Parameters Test:

As a final experiment, I wanted to prove in a different way that the time spent in the sequential parts of the step loop and the size of the iteration space distributed to each thread has an impact on the speedup. To do this, I switched around the base parameters for the number of bodies and steps so that there would now be 3000 steps in the simulation, but with only 2000 bodies. I defined this version as the "switched" version and the previous versions with nsteps = 2000 and bodies = 3000 as the normal version. In this experiment, I expected that the switched versions would have worse speedup compared to the normal versions. When there are a smaller number of bodies, each thread will be given a smaller chunk to work with (assuming block distribution). With smaller chunk sizes and a smaller number of bodies total, the work should be done faster per each step since less interactions need to be calculated. However, with the number of steps increased over the regular version, there would be more calls to and therefore more time spent in the sequential write out section where the other threads must wait. I did this experiment for 32 and 48 threads and the results can be seen in the fourth column of the graph spread sheet.

As can be seen in the graphs, the regular versions had better speedup in all cases compared to the switched versions with the same thread count. Even though each thread had less work to do per step in the switched version, the increase in the number of steps resulted in more calls to the sequential section, which ended up controlling the performance with its bottleneck. In the end, this made the overall speedup for each program decrease. This proves that when the amount of sequential code is increased, the overall speedup of the parallel program decreases.

Conclusion:

Overall, the results of all these timing tests mostly matched what I expected. The improved Pthread and OMP versions each had better speedup and scaling than their less efficient, previous versions. I also expected that none of the versions would achieve perfect linear speedup for all

thread counts as I became aware of potential bottlenecks in parallel performance while I made the program versions. Through this project, I learned how to use parallel enabling and optimizing techniques and some of the inner workings of Pthreads and OMP. I also gained firsthand experience on which methods are more effective at increasing speedup. Finally, I learned how to analyze the performance of parallel timing tests and how to make sense of them given the parallel libraries and techniques used. I am sure the knowledge I gained from this project will be useful for me in the future.

Final note:

At the last moment of this project when writing this report, I realized I may have overlooked one aspect of the parallel machine I used, which is that it has 48 threads, but only 24 physical cores total. Looking back at my results, I believe this core count may be more important to performance than the total thread count made possible by Hyper Threading. If I had more time, I would have gone back and analyzed the performance at 24 threads for each parallel program version to see if this is the case, but I have run out of time. I believe the analysis I gave to explain the non-linear speedup for each version is still accurate, but this threads per core issue may have played a bigger role in the scaling drop off from 16 to 32 and 48 threads.