Oriah Ulrich
Assignment 2
Project Report

Description of partitioning schemes:

During the first iteration of the design of the program, I had thought about trying to parallelize the vector product in different ways depending on the order of partial vector products. However, after careful analysis, I realized that it would be computationally slower than another approach by a factor of log(p). I decided to go with my other approach, which is more straight-forward and i think would factor into less communication overhead. The approach is to scatter the first matrix and broadcast the second matrix. The first matrix would be broken up into equal blocks of rows, so that each process will process an equal amount of data. The second matrix is broadcast out to each process so that each processor can handle performing a partial matrix multiply.

Matrix multiplies can happen in several forms: ijk, ikj, or kij. Each describes the order in which the matrices are iterated over in order to gather data. The order in which the iteration occurs affects the runtime performance of a program. When the CPU detects that it needs to fetch data from memory, it first looks in the cache. If the data is in the cache, then the CPU fetches it from the cache and moves on. Otherwise it has to fetch the data from memory each time. The cache hit time is magnitudes faster than accessing RAM. I took this in consideration when constructing the ijk forms. Each of the ijk forms accesses the data in the matrix a little bit differently

So, for the ijk form for example, when i read in the second matrix ‘B’ i store it in transposed form so that i can iterate over Transpose(B)’s rows. I did this in order to avoid many cache misses. Doing so improved performance by some positive factor. Testing the program i finished a test run in half the time in ijk form. With ikj and kij form i simply added some local variables so i don't have to keep fetching the data from memory. With KIJ i was thinking about sending each processor a column wise partition of the transpose of A in order to iterate over the elements in each row of A rather than choosing to iterate over every element in the column of A. Again, this would maximize cache efficiency.

Run Description:

Running on my home computer was pretty interesting, and much quicker than running on jaguar. I ran the matrixmult on a matrix of size N = 4800 and it took less than 90 seconds on ijk form and around 120 seconds on kij form. Interestingly, running on 4 core maximized the efficiency. Any more was only slightly less efficient, less than that the performance dropped significantly.

Timing Tables:

The following timing tables are tables describing the times collected from running the program on the lab machines from jaguar. the minimum time was strangely on processor on one computer. As the number of computers increased, the more time it took for a run to complete. However, decreasing the number of machines in the hostfile but increasing the slots used seemed to have speed up the processing. The timing table reflect the times of consistent runs using the hostfiles included in the zip. There are 3 timing tables, one for each of the forms of ijk. Each timing table describes the times that the program took to complete on the given number of processors at each run stage. The minimum times are highlighted for each processor’s set of runs. Using the minimum times of each set of runs, I generated speedup and efficiency tables and graphs.

Form:	ijk:							
		1	4	8	12	16	20	Processor Count
	1	291.45	634.82	825.52	1768.16	1736.67	1315.56	
	2	211.72	734.14	901.23	803.02	2003.66	1207.38	
	3	201.01	565.95	823.57	887.98	1902.17	1527.82	
	4	199.22	776.63	775.15	794.62	1738.75	1423.26	
	5	166.07	783.2	810.69	837.64	1723.28	1297.55	
	Run #							
Form:	ikj:							
		1	4	8	12	16	20	Processor Count
	1	241.18	778.43	825.52	887.98	1902.17	1527.82	
	2	261.72	694.23	901.23	794.62	1738.75	1423.26	
	3	227.13	693.84	823.57	837.64	1723.28	1297.55	
	4	189.92	390.99	812.17	817.2	1534.28	1127.36	
	5	172.16	393.14	810.69	698.24	1681.1	1397.86	
	Run #							
Form:	kij:							
		1	4	8	12	16	20	Processor Count
	1	372.69	911.43	1129.77	1271.33	1000.98	720.01	
	2	290.96	614.23	824.89	1023.26	1001.13	1235.85	
	3	398.56	829.58	922.6	1108.99	1011.86	1072.6	
	4	394.2	1117.22	817.2	926.12	870.6	893.26	

	5	360.55	634.67	849.13	1048.67	1022.1	928.78	
	Run #							

Speedup Table:

ijk:							
	1	**4**	**8**	**12**	**16**	**20**	**Processor Count**
	1	**0.205**	**0.214**	**0.208**	**0.096**	**0.127**	**Speedup Factor**
ikj:							
	1	**4**	**8**	**12**	**16**	**20**	**Processor Count**
	1	**0.44**	**0.21**	**0.211**	**0.099**	**0.152**	**Speedup Factor**
kij:							
	1	**4**	**8**	**12**	**16**	**20**	**Processor Count**
	1	**0.474**	**0.356**	**0.314**	**0.334**	**0.402**	**Speedup Factor**

Efficiency Table:

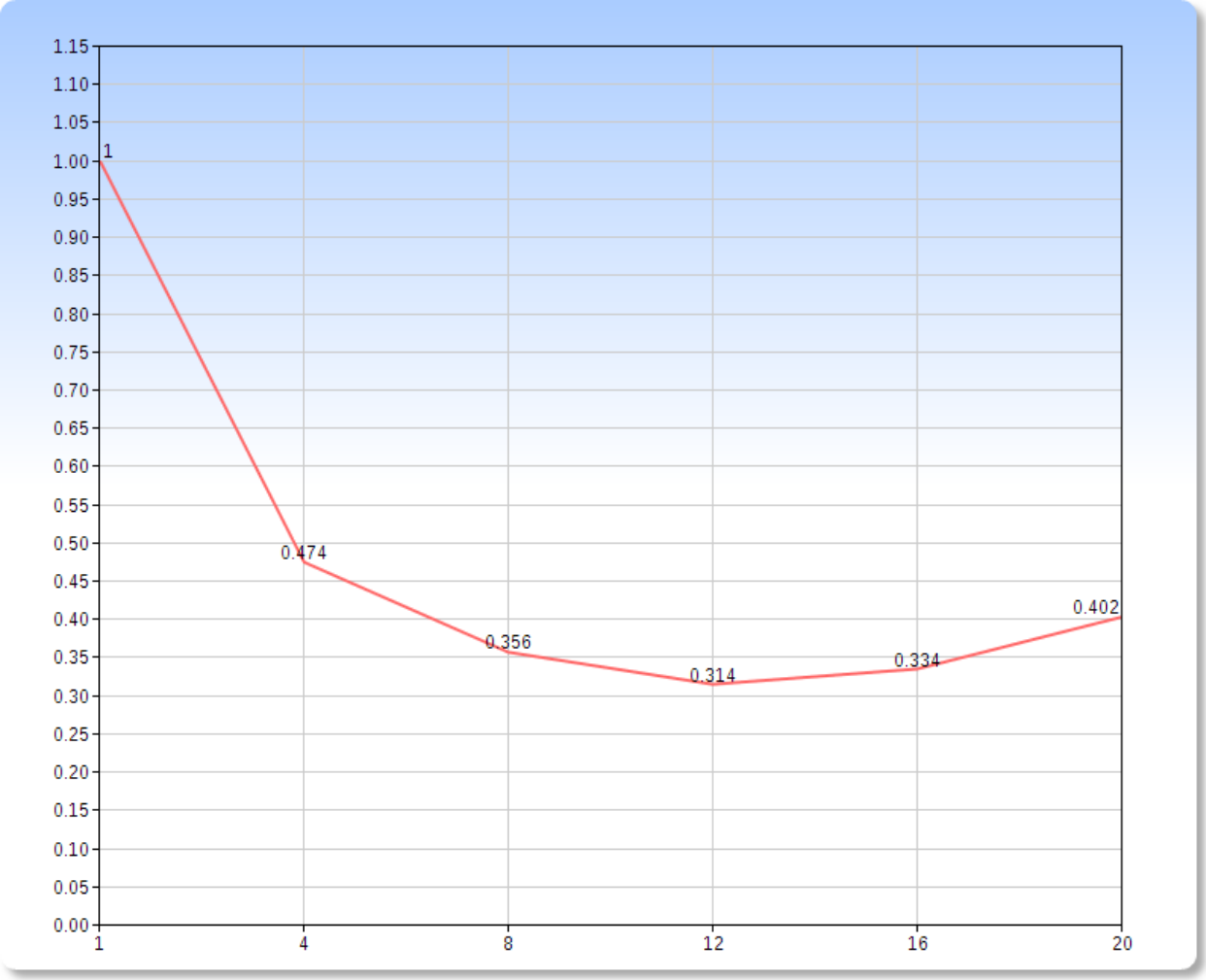
ijk:							
	1	**4**	**8**	**12**	**16**	**20**	**Processor Count**
	1	**0.05125**	**0.02675**	**0.0173**	**0.006**	**0.00635**	**Speedup Factor**
ikj:							
	1	**4**	**8**	**12**	**16**	**20**	**Processor Count**
	1	**0.11**	**0.02625**	**0.01758**	**0.0061875**	**0.0076**	**Speedup Factor**
kij:							
	1	**4**	**8**	**12**	**16**	**20**	**Processor Count**
	1	**0.1185**	**0.0445**	**0.02617**	**0.02088**	**0.0201**	**Speedup Factor**

Graphs: (in order of speedup and then efficiency for each ijk form..)

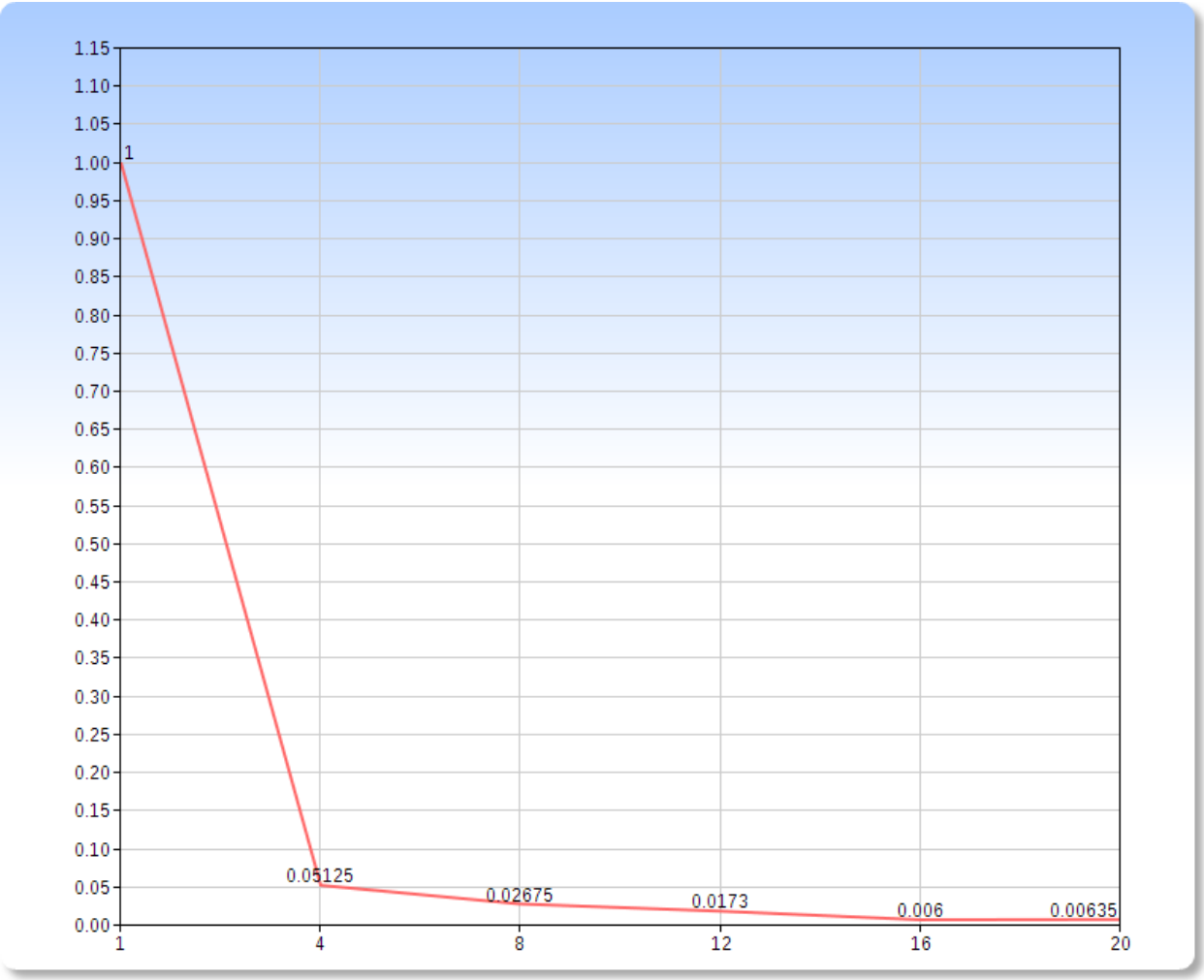
Speedup vs Processors (ijk)

Speedup vs Processors (ikj)

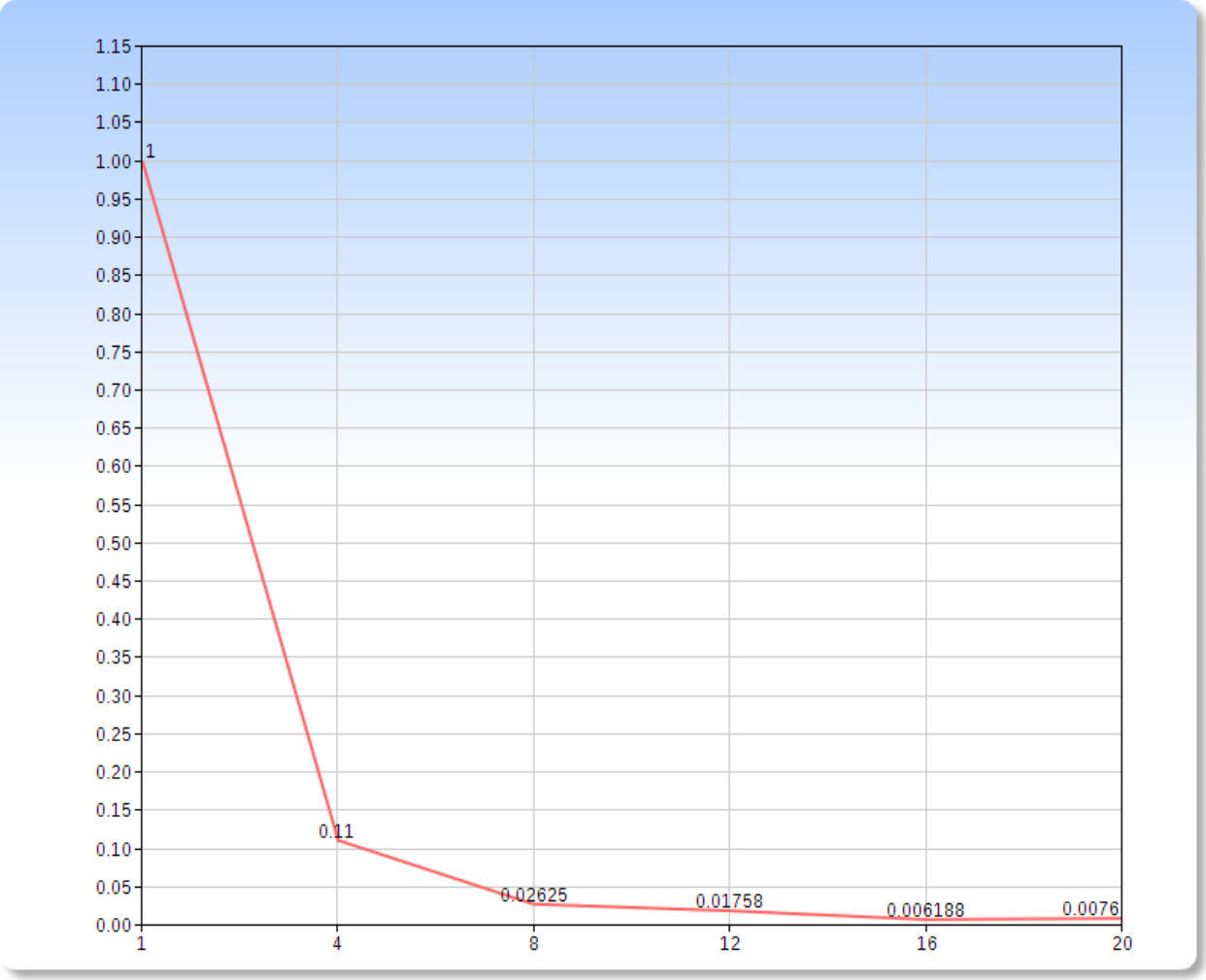
Speedup vs Processors (kij)



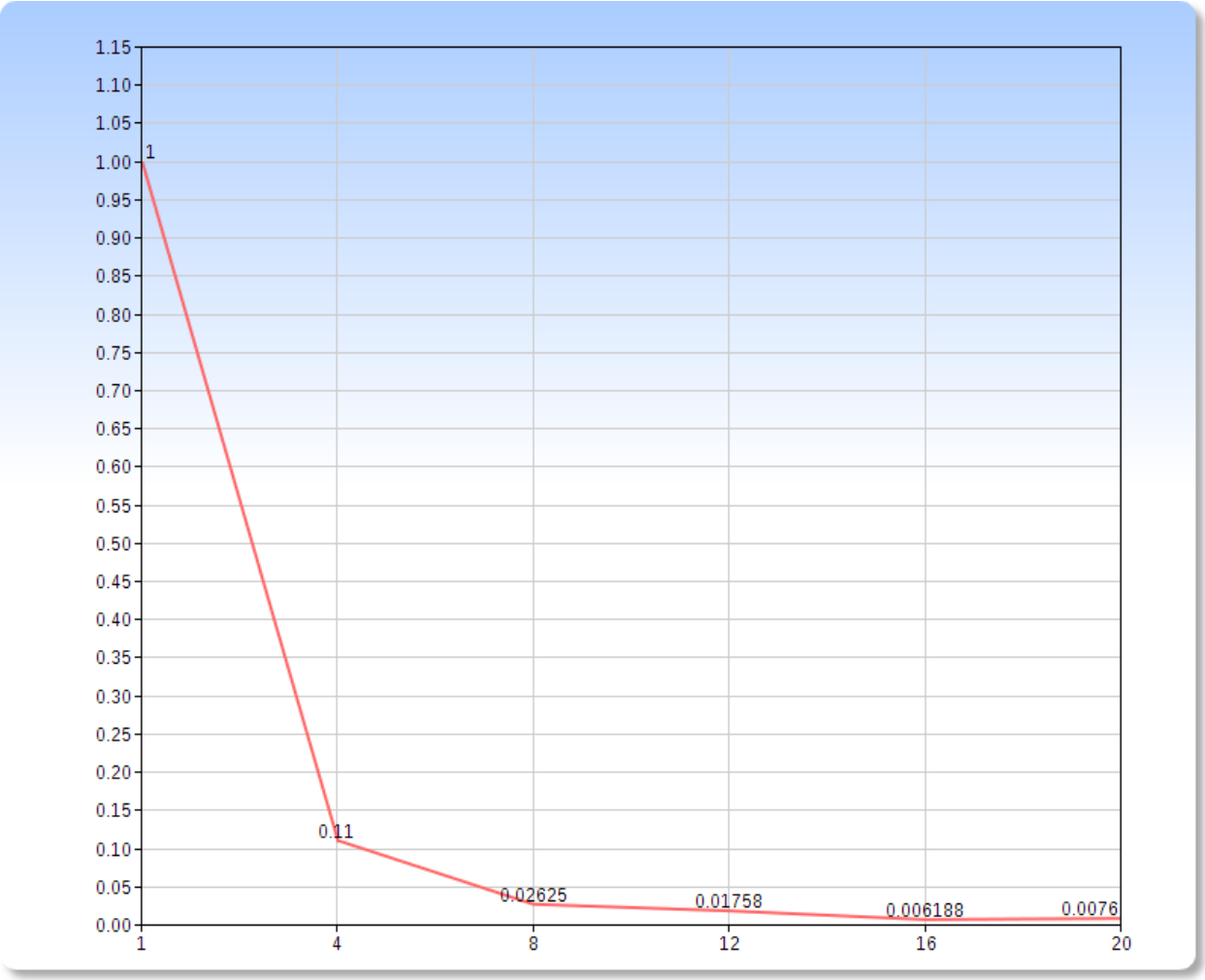
Efficiency vs Processors (ijk)



Efficiency vs Processors (ikj)



Efficiency vs Processors (kij)



Observations and Conclusions:

I have come to the conclusion that parallelization would be good if communication was reduced to a minimum between processors. If the data is partitioned correctly and each processor maximizes it's use on each data block then the program is parallelized as much as possible. However I would not have guessed that this program would perform worse when parallelized. I still do not believe that I implemented an effective matrix multiply program since it logically makes sense that a matrix multiply would run faster in parallel. Perhaps the parallelization would be best focused at some reduced form of the multiplication. Maybe parallelization would be best when applied to the second matrix in pieces and then performing a reduce? Also, perhaps a reduce at each vector multiply would turn out to be a better solution since B would not have to be shared? However, i do not believe the latter to be true since I timed a broadcast of matrix B and it came out to roughly half a second on ten processes on my personal computer, vs the 150 seconds it would take to multiply the matrices. Also, the big O of parallelizing vector products converges to something around O(k/p* n^2 * logp) where as data partitioning Matrix A converges to O(k/p * n^2) The data partitioning of Matrix A should be done first. Then the vector product should be parallelized. But that would not be possible unless we have at least 4800 processors, at least 1 processor for each row and perhaps more to parallelize the vector dot product reduction. With that said, i think that I maximized the the efficiency of parallelizing the matrix multiplication with the given number of processors. Although I seriously still wonder why the multiplications perform less efficiently on jaguar. I also seriously wonder why the speedup converges to some value less than 1, when it seemed at first as if it should have converged so some value greater than or equal to 1.

- Parallelizing vector product is less efficient than simply partitioning the data itself for each processor to work on.
- Maximizing cache efficiency is big deal. Keeping in mind spatial/temporal locality
- Parallel overhead dramatically increases as the number of processors increases and therefore substantially increases the amount of time it takes for a program to finish.

Post Project Analysis:

Looking at the speedup and efficiency graphs one last time I realized that they seriously should be increasing at least some of the time. I ran some verification runs on my personal laptop to show myself that the times decreased as the number of processors increased. I tested 2, 4, and 8 cores. However, what I found was that the time increased as the number of processors increased. I turned off compiler optimization flags after hearing from a friend that they were slowing down his times. I ran the verification runs again on my laptop, again on 2, 4, then 8 cores. Finally, some sign of speedup! The times were getting faster as I increased the processor count. However, the difference between the minimum run-time on a run *without* optimizations and the minimum run-time on a run *with* optimizations decreased as the number of processors increased. As the processors increased, the time seemed to have converged to a specific value, with or without compiler optimizations. I thought this was interesting, since optimizations seem to affect the rate of change of efficiency. Using optimizations seemed to make the "more-serialized" type of code faster, while a more parallel run was slower. Using optimizations affects the speedup graph because it alters the magnitude of time it takes to run *serially.*

Visually, the graph of time when running *with* optimizations increased with respect to # of processors at a decreasing rate. However, the graph of time when running *without* optimizations decreased with respect to # processors at a decreasing rate. (they both sort of converged to some time it seemed). It is a guess, but I'm

thinking that the function is negative and will eventually decrease after throwing in certain number of processors greater than 30 or 40 or something. This depends on the magnitude of overhead.

With all of that said, I was interested in how this would effect my timing values and therefore speedup graphs that I had generated from data collected from the lab machines. Andreas and I were talking through this, since he and I had been considering optimizations from the beginning to ensure the run-times are minimal, however at this point we started to believe optimizations were a detriment to the speedup graphs. So we tested the same set of runs as well as a set of runs on 16 cores. We found that it didn't matter whether the code was optimized or not, we still got speedup less than 1. We expected to find that we would get positive speedup greater than 1 when testing without optimizations, just as we had gotten when testing on my computer, however this was not the case.

This tells us that something about the system. Either I wrote bad code or the jaguar presents enough overhead that the time is positively cushioned in a way that magnifies the amount of time it takes overall, essentially stretches the graph of time vs processors, increasing the amount of time it takes for a run. Running on a distributed parallel system clearly presents greater overhead on parallelization than does a shared memory parallel system.

Final Questions:

I wonder if the rate of increase of overhead as processors increase will eventually allow the time graph to decrease once again. Maybe the magnitude of the second derivative of parallel overhead is too big and therefore would overwhelm any benefits of having a distributed parallel system with such latency.