CSC 458 Assignment 3

Nick Graham

March 10, 2017

Introduction

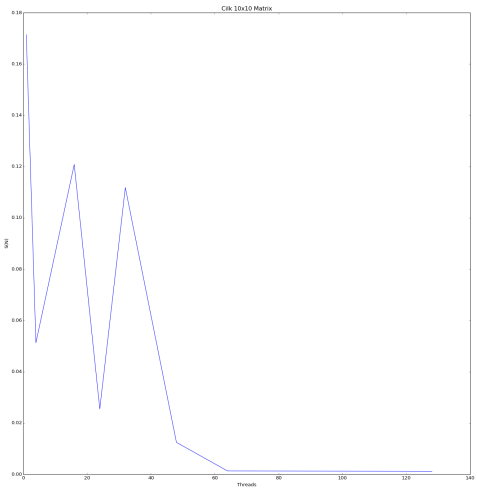
Assignment Three has us take a sequential program for solving systems of equations using gauss's method and make it parallel using both OpenMP and Cilk Plus. The code was tested using two matrices, one that was 10×10 and the other that was 2000×2000. The speedup of each method was then plotted to see how well they performed and to see if they were scalable.

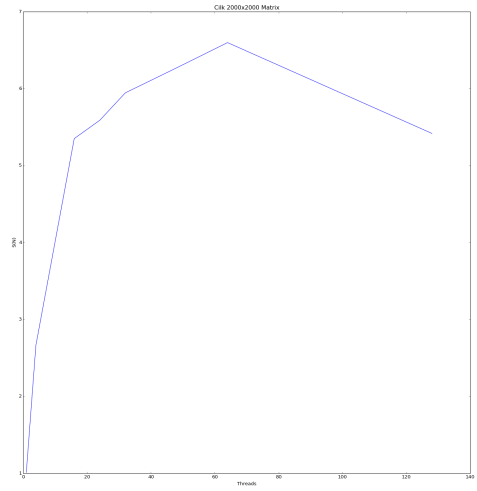
Data

Cilk

The Cilk implementation overall performed better than the sequential version of the code, and as you can see from the graphs it is moderatly scalable. While it is no where close to linear, there is only a dropoff when the number of threads exceeds the number of cores on the machine (visable in the plots for the value recorded with 128 threads). The graphs also show that for small data sets such as the 10x10 matrix the overhead of cilk's parallelization is more then the performace improvements and it has rather erratic and poor behavior.

Cilk did not achieve as high of a speedup as OpenMP did, however Cilk was more consistent in its growth. Because of this consistency I would say that Cilk is more scalable. Not only did it continue to increase in terms of speedup, but it could easily be modeled what adding more threads would do.





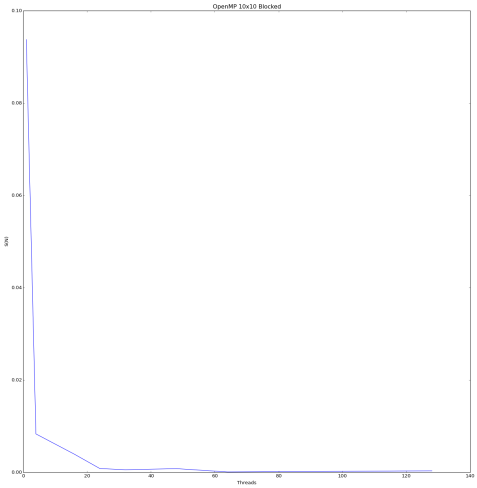
OpenMP

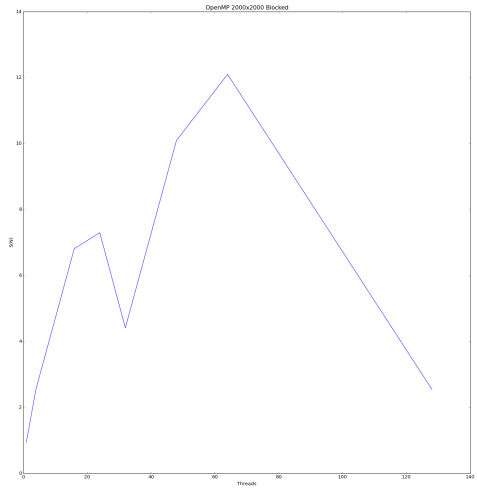
The OpenMP solution appeared to perform better with smaller numbers of threads, as well as being able to achieve the highest speedup. The OpenMP solution was by far less scalable then the Cilk implementation was. One area where OpenMP did perform "better" was that for the lower matrix sizes

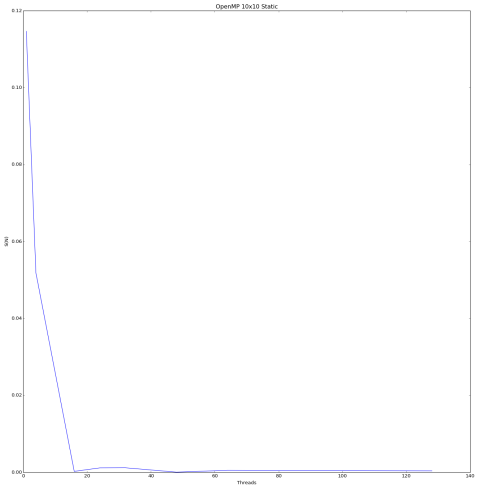
it exibited more consistent behavior, and was more predictable, however, it was still worse then the cilk implementatin in terms of speedup. All three task allocations saw a noticible drop in speedup for max threads numbers between 16 and 32. While the machine should have been able to accomodate those threads, it seems that how OpenMP was managing them caused the overhead for the non-powers of two to be much higher.

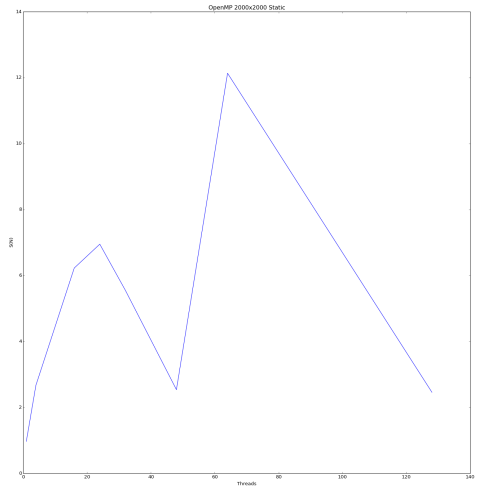
All three task allocations were very similar in terms of both performance and speedup, however in this case the blocked allocation appeared to work the best. The static allocation was the next best, and the dynamic was the worse. This was not what I had expected. I had expected the static allocation to outperform the blocked allocation as the static allocation assigned the array rows in a cyclic manor, so throught the algoritm each thread was still doing work. The dynamic allocation being the worst makes sense as it doles out work to the threads as they finished, as opposed to staticlly assigning a set ammount of work per thread, and this adds significant overhead. This overhead being so high seems reasonable based on the performace of OpenMP on small data sets. With the small data set a mojority of the time is spent on the overhead as apposed to the computation, and in my test, the performance of OpenMP on the 10X10 matrix decreased as more threads were added.

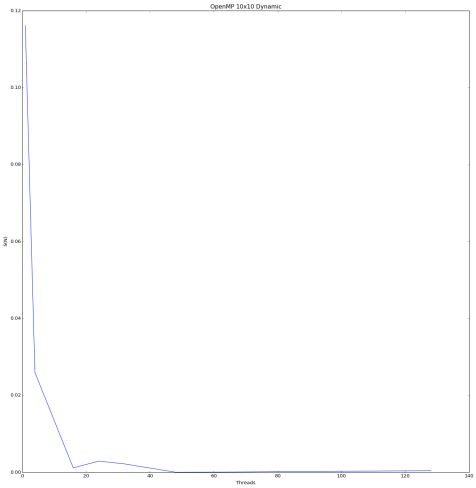
My choice of saying that blocked allocation performed better then static was based off of the fact that it did not suffer as much between 16 and 32 threads. The two implementations otherwise had very similar performance. The results from running the OpenMP code on the IBM machine were more inline with I had expected, and so I would like to run this test again at somepoint to see if my results are repeatable.

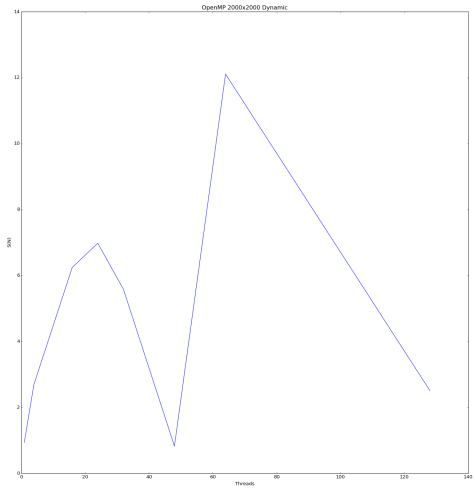

OpenMP 10x10 Blocked
SPKI
Threads
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100
120
140


OpenMP 2000x2000 Blocked
SPKI
Threads
14
12
10
8
6
4
2
0
0
20
40
60
80
100
120
140


OpenMP 10x10 Static
SPU
Threads
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100
120
140


OpenMP 2000x2000 Static
SPU
Threads
14
12
10
8
6
4
2
0
0
20
40
60
80
100
120
140

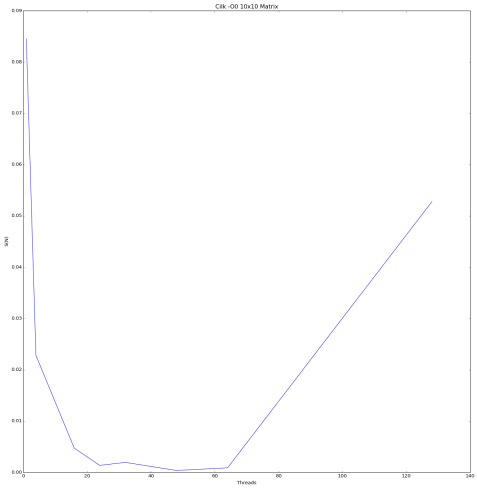


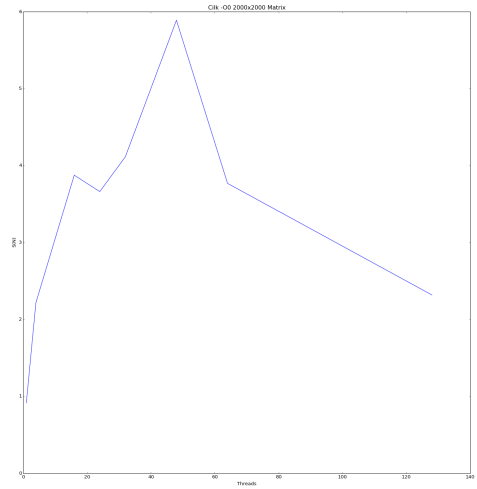


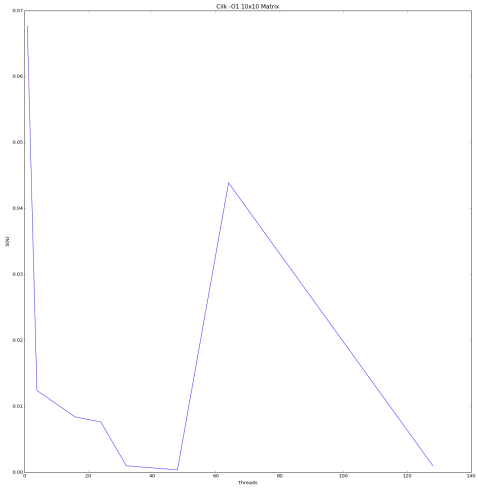
1 Optimization Levels

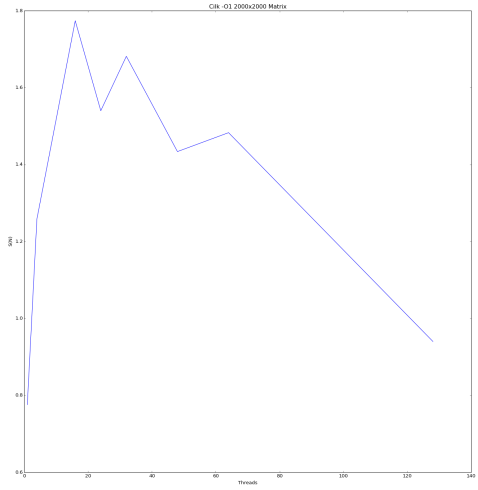
I also tested each algorithm compiled at varying optimization levels. Overall the more optimization I allowed the compiler to do, the less speedup was seen in the parallel implementations. At the highest optimization level, the sequential algorithm actually ran better than any of the Cilk runs. OpenMP

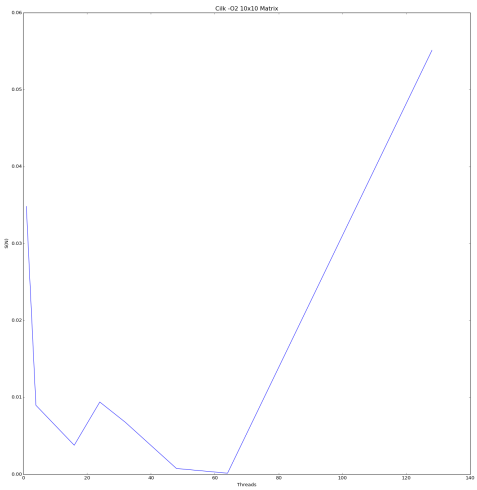
still ran about 0.9 seconds faster.

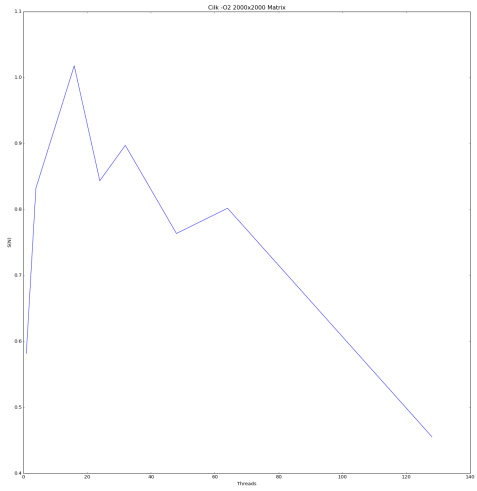


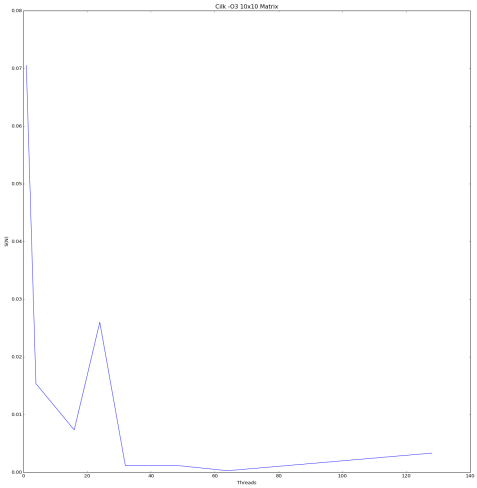


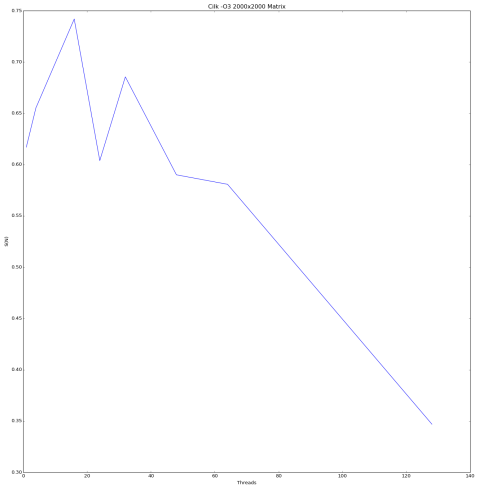

Cilk -O1 10x10 Matrix
SpU
Threads
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0
20
40
60
80
100
120
140

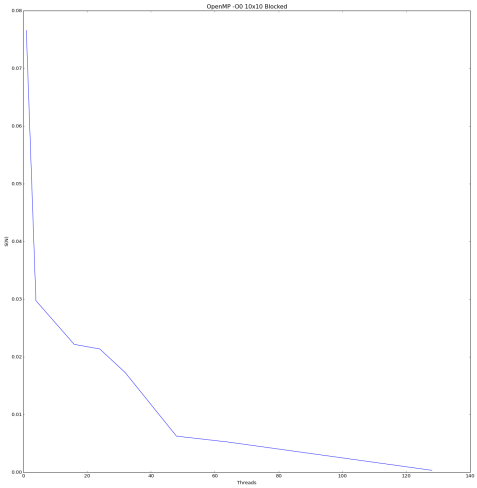

Cilk -O1 2000x2000 Matrix
SpU
Threads
0.6
0.8
1.0
1.2
1.4
1.6
1.8
0
20
40
60
80
100
120
140

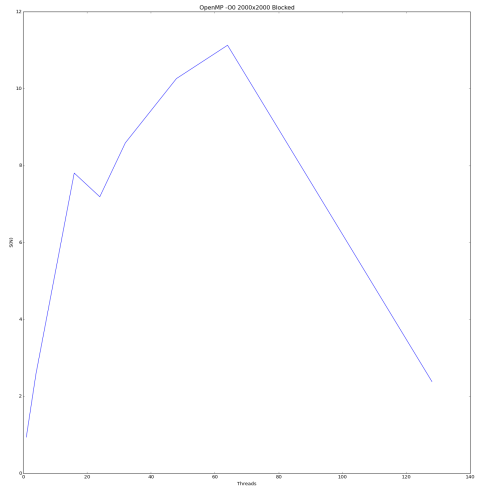

Cilk -O2 10x10 Matrix
0.06
0.05
0.04
0.03
0.02
0.01
0.00
SNI
0
20
40
60
80
100
120
140
Threads

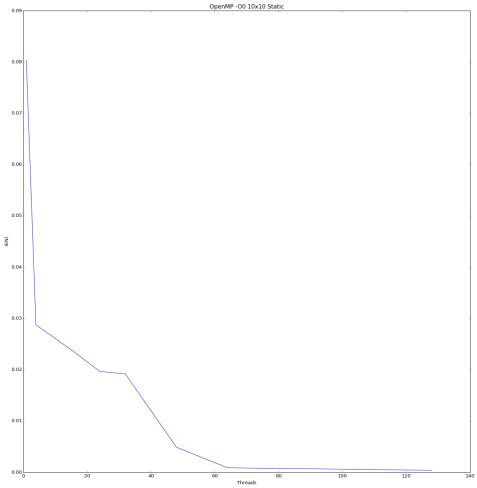

Cilk -O2 2000x2000 Matrix
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
SNI
0
20
40
60
80
100
120
140
Threads

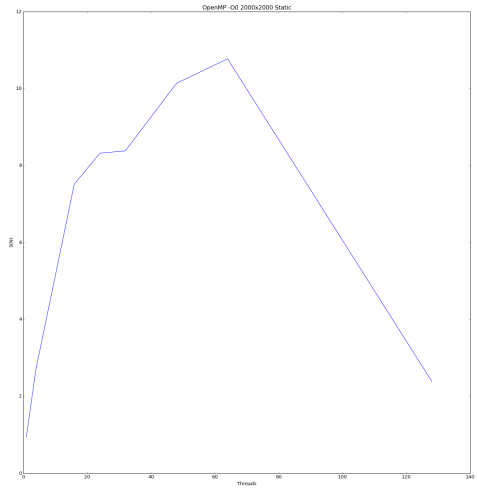

Cilk -O3 10x10 Matrix
Time(s)
Threads
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0
20
40
60
80
100
120
140

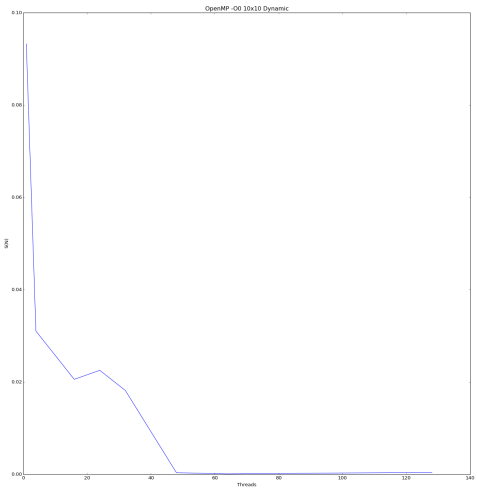

Cilk -O3 2000x2000 Matrix
Time(s)
Threads
0.30
0.35
0.40
0.45
0.50
0.55
0.60
0.65
0.70
0.75
0
20
40
60
80
100
120
140

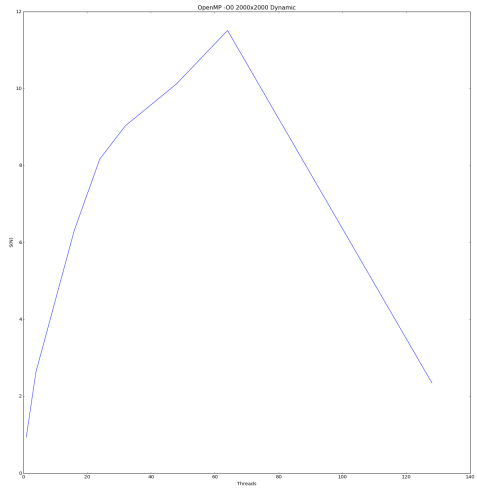

OpenMP -O0 10x10 Blocked
SPU
Threads
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
20
40
60
80
100
120
140

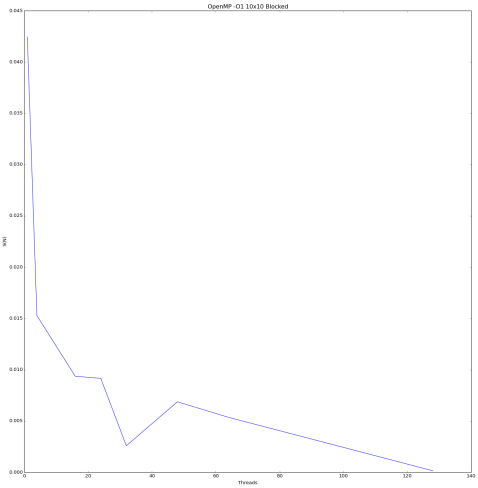

OpenMP -O0 2000x2000 Blocked
SPU
Threads
12
10
8
6
4
2
0
0
20
40
60
80
100
120
140

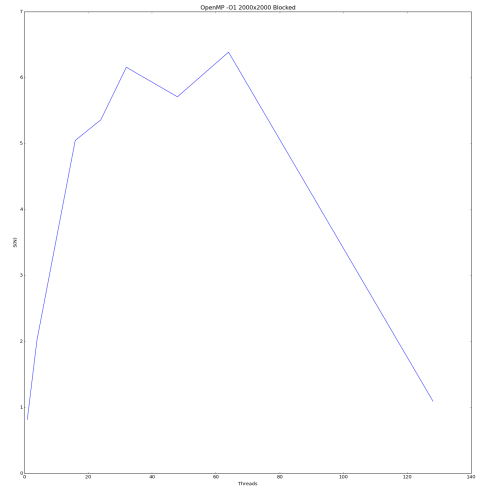

OpenMP -O0 10x10 Static
SPU
Threads
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
20
40
60
80
100
120
140

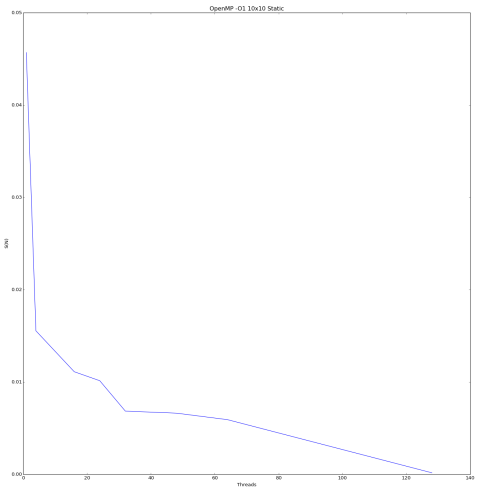

OpenMP -O0 2000x2000 Static
SPU
Threads
12
10
8
6
4
2
0
0
20
40
60
80
100
120
140

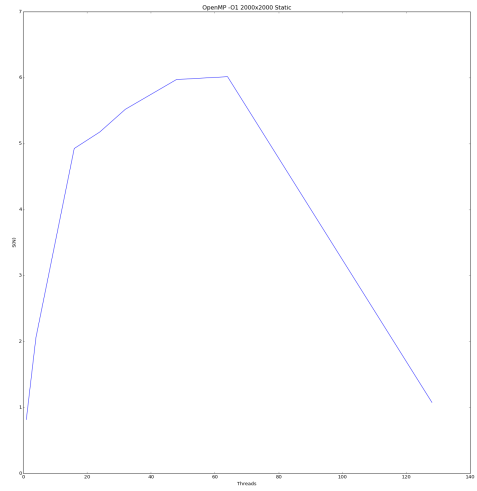

OpenMP -O0 10x10 Dynamic
SPU
Threads
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100
120
140

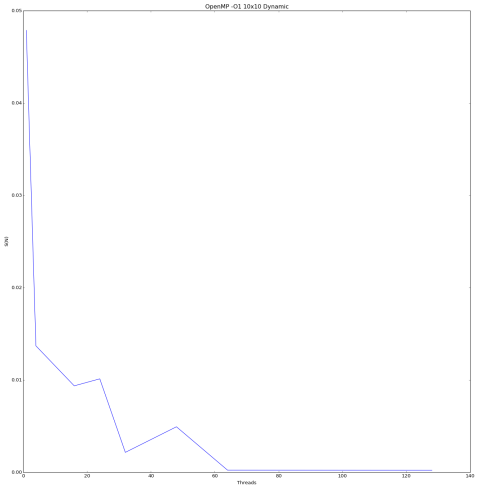

OpenMP -O0 2000x2000 Dynamic
SPU
Threads
12
10
8
6
4
2
0
0
20
40
60
80
100
120
140

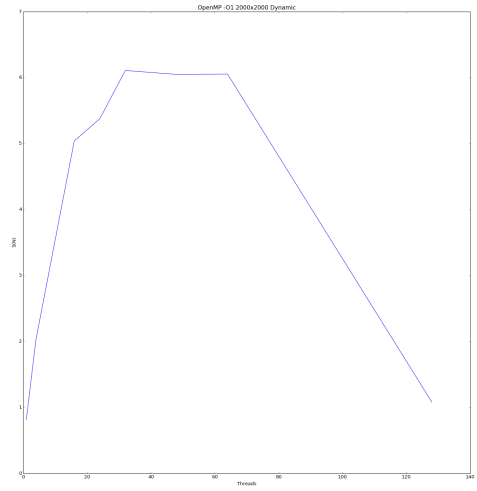

OpenMP -O1 10x10 Blocked
GNS
Threads
0.045
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
20
40
60
80
100
120
140

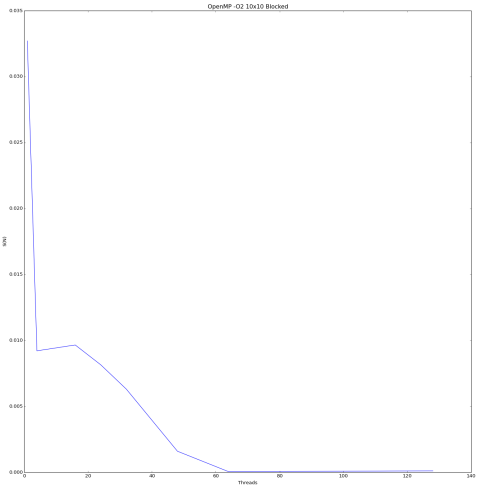

OpenMP -O1 2000x2000 Blocked
SNS
Threads
7
6
5
4
3
2
1
0
0
20
40
60
80
100
120
140

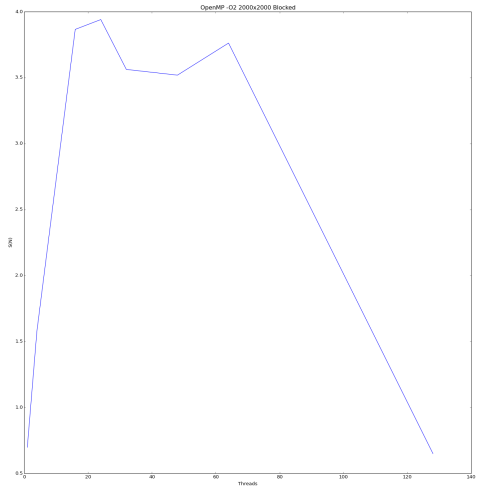

OpenMP -O1 10x10 Static
SPEEDUP
Threads
0.05
0.04
0.03
0.02
0.01
0.00
0
20
40
60
80
100
120
140

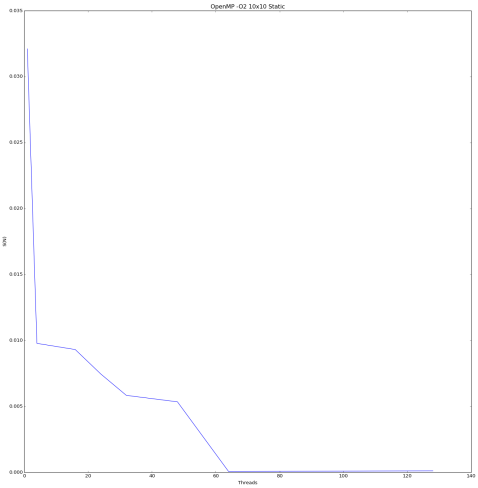

OpenMP -O1 2000x2000 Static
SPEEDUP
Threads
7
6
5
4
3
2
1
0
0
20
40
60
80
100
120
140

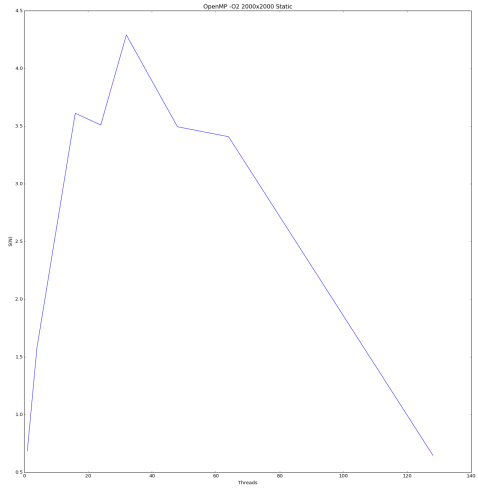

OpenMP -O1 10x10 Dynamic
SPEEDUP
Threads
0.05
0.04
0.03
0.02
0.01
0.00
0
20
40
60
80
100
120
140

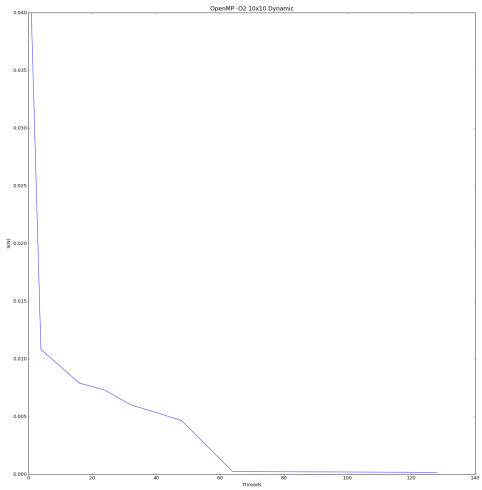

OpenMP -O1 2000x2000 Dynamic
SPEEDUP
Threads
7
6
5
4
3
2
1
0
0
20
40
60
80
100
120
140

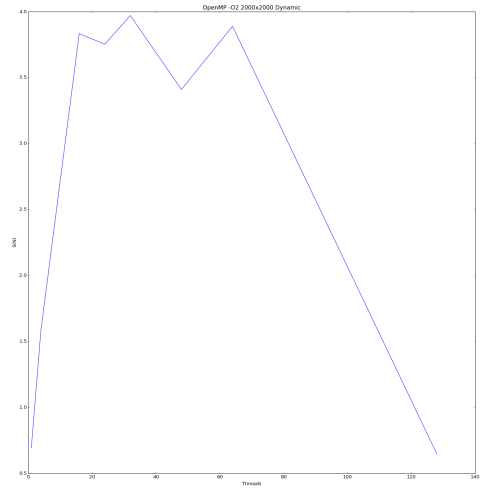

OpenMP -O2 10x10 Blocked
SPEED
Threads
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
20
40
60
80
100
120
140

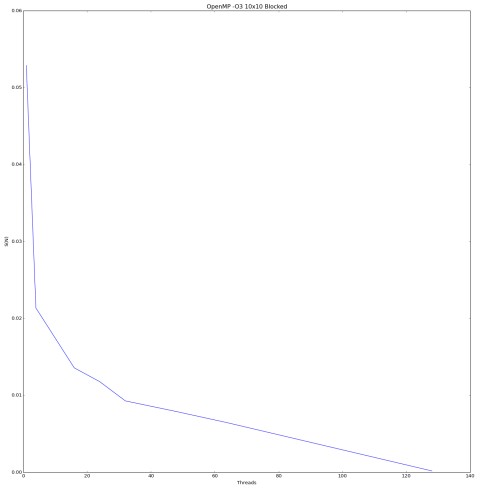

OpenMP -O2 2000x2000 Blocked
SPEED
Threads
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100
120
140

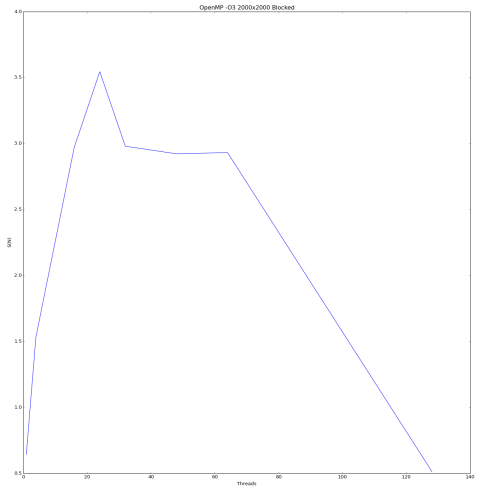

OpenMP -O2 10x10 Static
SPU
Threads
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
20
40
60
80
100
120
140

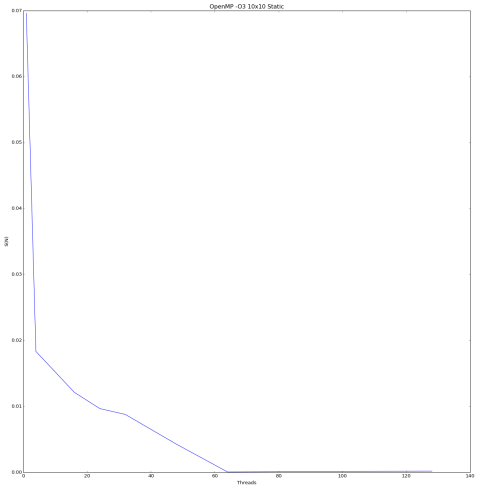

OpenMP -O2 2000x2000 Static
SPU
Threads
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100
120
140

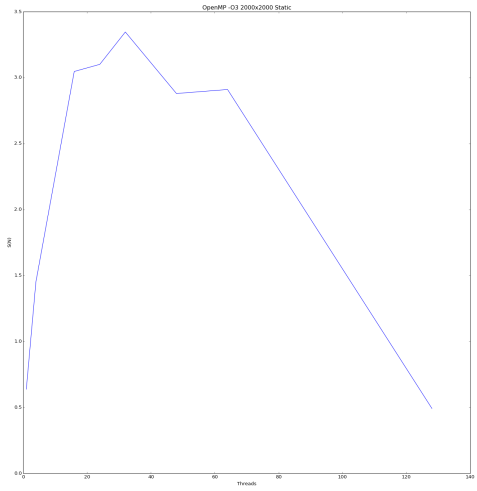

OpenMP -O2 10x10 Dynamic
Sec
Threads
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
20
40
60
80
100
120
140

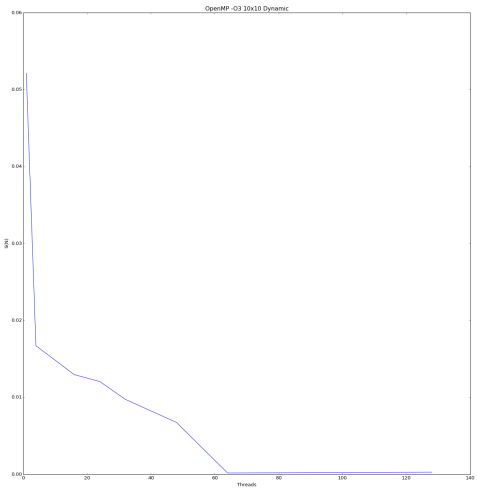

OpenMP -O2 2000x2000 Dynamic
SpU
Threads
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100
120
140

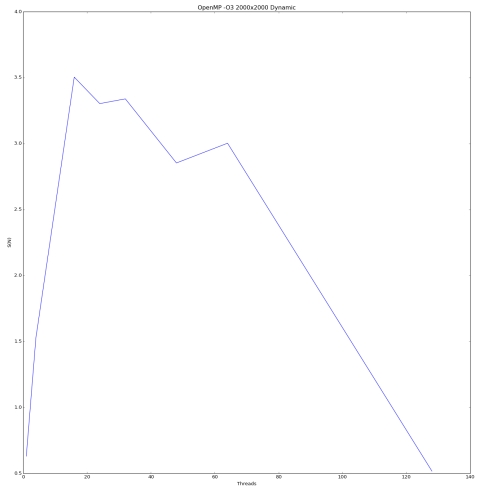

OpenMP -O3 10x10 Blocked
SPU
Threads
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0
20
40
60
80
100
120
140


OpenMP -O3 2000x2000 Blocked
SPU
Threads
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
20
40
60
80
100
120
140


OpenMP -O3 10x10 Static
Secs
Threads
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
20
40
60
80
100
120
140


OpenMP -O3 2000x2000 Static
Secs
Threads
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
20
40
60
80
100
120
140



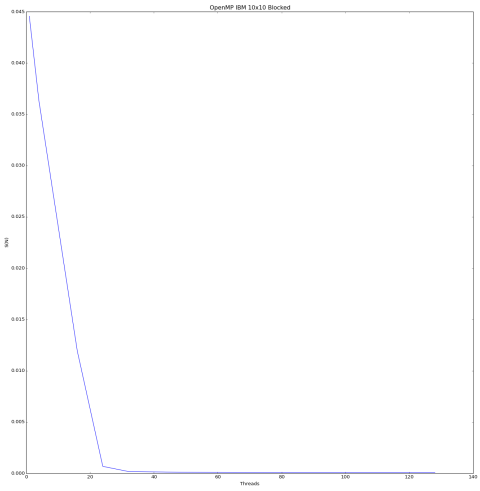


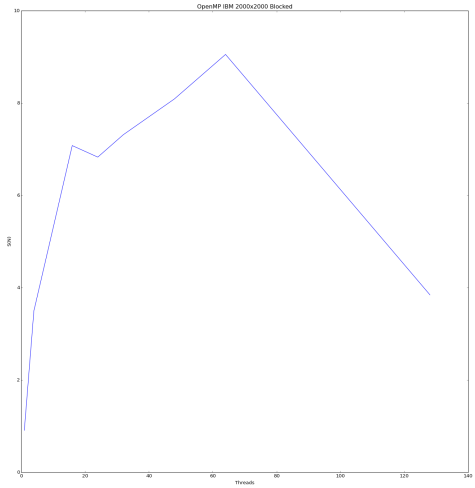
2 IBM

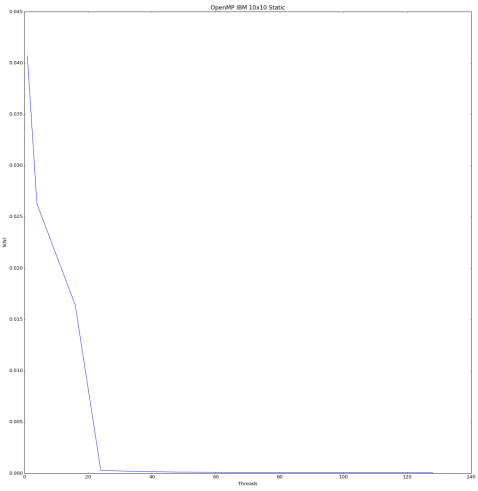
I tested my OpenMP implementation on the IBM machine as well and the results from that were in line with what I expected. These results show that the dynamic allocation had the best performance, closely followed by the static allocation and the blocked allocation had the worst performance. In

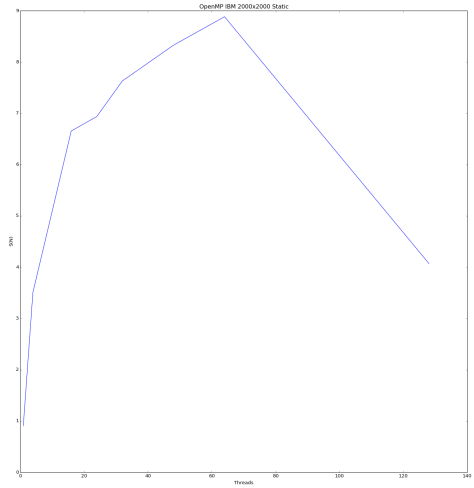
this case the overhead of the dynamic allocation was not enough to reduce the benefits, and with that it makes sense that it would be the fastest. Then the other two were as expected. The static performed better because the data was cyclicly doled out to each thread, so they all kept working up until the end, as opposed to the blocked allocation where the lower number threads would finish working before the higher number threads would.

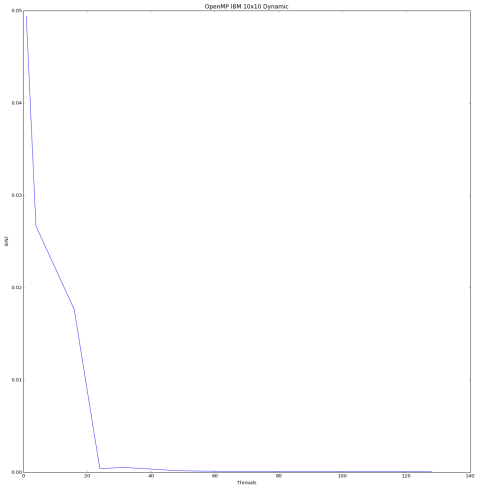
Possible reasons for why the IBM machine performed better would be how the memory is layed out. If the threads are distributed such that memory does not have to travel far on the bus, then there will be better performance.

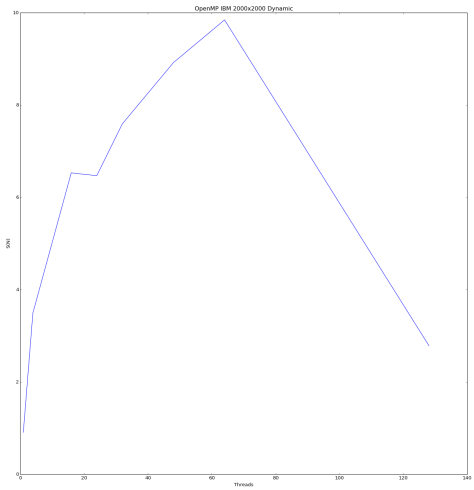



OpenMP IBM 2000x2000 Blocked
SPEED
Threads
0
2
4
6
8
10
0
20
40
60
80
100
120
140


OpenMP IBM 10x10 Static
SPEED
Threads
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
0.040
0.045
0
20
40
60
80
100
120
140


OpenMP IBM 2000x2000 Static
SNS
Threads
0
1
2
3
4
5
6
7
8
9
0
20
40
60
80
100
120
140


OpenMP IBM 10x10 Dynamic
SNS
Threads
0.00
0.01
0.02
0.03
0.04
0.05
0
20
40
60
80
100
120
140



3 Conclusions

Overall the OpenMP implementation provided the highest speedup of just less than 12 at 64 threads. While it provided the best speedup, the OpenMP was much less scalable with sharp irregularities in the curve around 16 to 24 threads. The Cilk Plus implemtation provided a "smoother" curve, with less sharp jumps between thread counts. This results in Cilk Plus being more scalable, even though the speedup is not linear, and is worse than the OpenMP implementation. The max speedup I was able to get with Cilk Plus was around 6.5 at 64 threads.

In both implementations there was a drastic drop in performance when the number of threads exceeded the number of cores, which makes sense, as now there is contention on resources when some of threads are prempted from the processor holding locks.

There was a major difference between the IBM machine and node2x18a for the OpenMP implementation. The node2x18a machine performed much different then expected, and had large performance hits for 24 between 16 and 24 threads, while the IBM was a bit more scalable, athough it did not have quite as large of a speedup. I was unable to test the Cilk implementation on the IBM Machine as it did not have the Cilk header files installed, but I would be interested to see how it performs on the IBM machine as well.

Overall I would say that OpenMP provides the best overall speedup, but only in select cases where select numbers of threads are available. If the number of available threads is known, or can be controlled to stay towards one of the more acceptable values, then it is the better option. If the program needs to be fully scalable to run on many different machines with varying numbers of threads, Cilk will provide a more scalable implementation at the cost of less speedup.