Jake Christensen

CS 322 Assignment 4

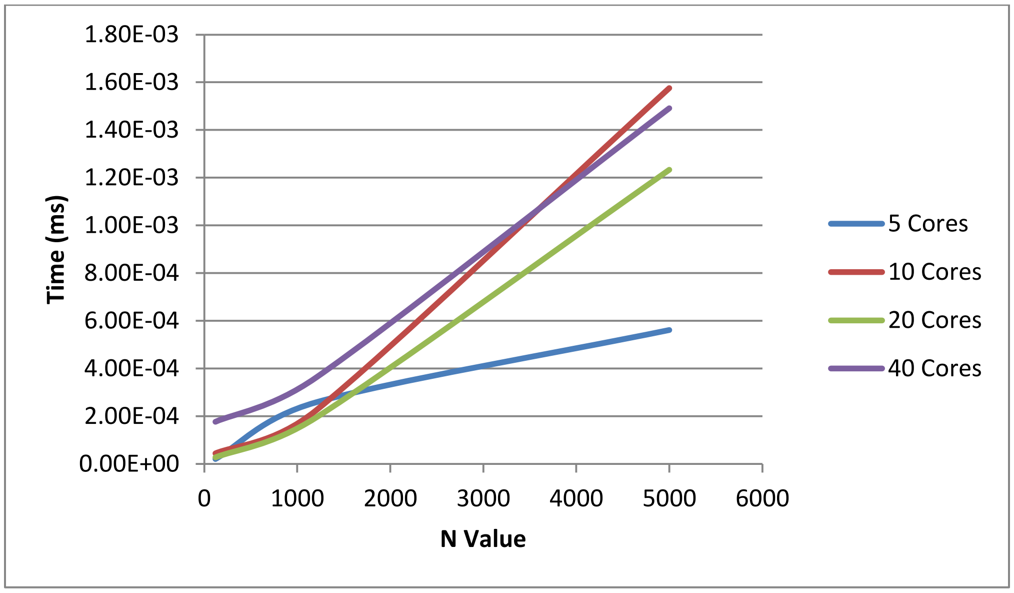
In our latest exercise I experimented with MPI once again. This time I was using MPI to execute a parallel merge sort in which each thread sorts a small array. Similar to parallel addition, this merge sort then uses a tree scheme to send and merge all the arrays together, sort them, and pass them further down the tree until we reach zero. My initial assumptions for this experiment were fairly strait forward, I would see an increase by using MPI when I had a number of cores close to that which matched the hardware I was using. As the number of cores went up, the performance would degrade, as more processing power would be spent on managing thread overhead.

After getting my parallel merge sort algorithm set up I began my experimentation. The table on the next page illustrates the average time to execute a merge sort given a starting value for *n*. This data definitely supports my hypothesis that the higher number of cores used by MPI, the less effective the merge sort becomes. Notice how much slower the rate of growth is for a five core run as opposed to more. This is especially apparent on the jump between 1200 and 5000 where the percentage growth only 54% compared with 86% for 10 cores. If we were to scale the numbers up even further the percentage change would continue to grow. This trend is highlighted very well by the graph on the next page, in which the 5 core runs break away from the pack significantly.

Another interesting metric was the low volatility in the runtimes. I made an attempt at calculating the standard deviation to see how much fluctuation there was in runtime. What I found was that the numbers for the standard deviation became so small that they could not be stored in a double. This suggests that I was getting an extremely consistent runtime at each core number.

This experiment definitely highlights the need to know your hardware system before parallelizing. The performance cost of overshooting the optimal number of cores for a system can cause significant slowdowns on performance. This is definitely a point that OpenMP has going for it, the complicated algorithms it uses to find the optimal number of threads certainly would be appealing to avoid situations like that illustrated here.

	Nvalue		
#Cores	**120**	**1200**	**5000**
5	2.094E-05	2.58E-04	5.617E-04
10	4.454E-05	2.240E-04	1.576E-03
20	2.793E-05	1.941E-04	1.234E-03
40	1.770E-04	3.617E-04	1.491E-03



Notes on running the program

To run the makefile simply type “make merge”. This will compile and create an object called paramerge for you to run. There is some diagnostic output that shows as the command line runs, array values and such. I’ve also included a Java project called stats_hw. This program reads the files generated by my merge code, and lets me run some statistical analysis on it. You can compile it using the **javac** [**options**] [**sourcefiles**] [**@argfiles**] command. Running the program won't do too much though, only recopy everything in the AVGresults_A4.csv file back into it.