Matt Kwong

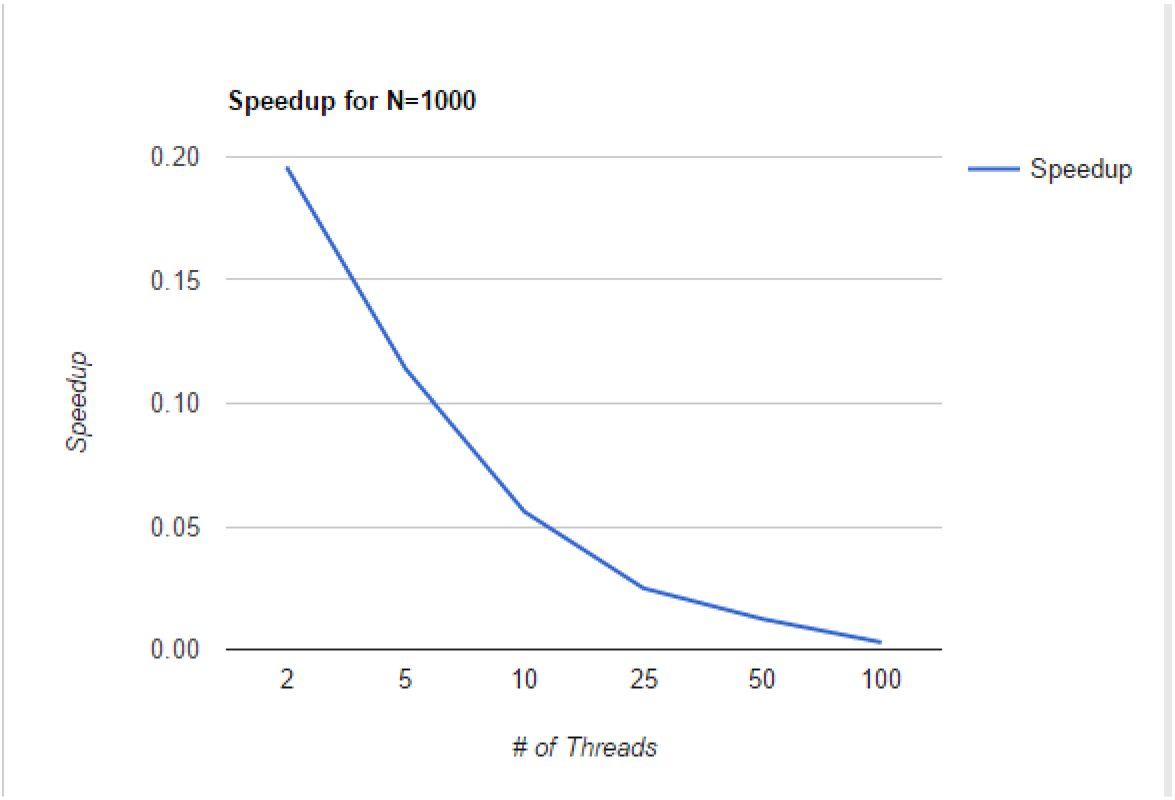
Parallel Computing Lab 2

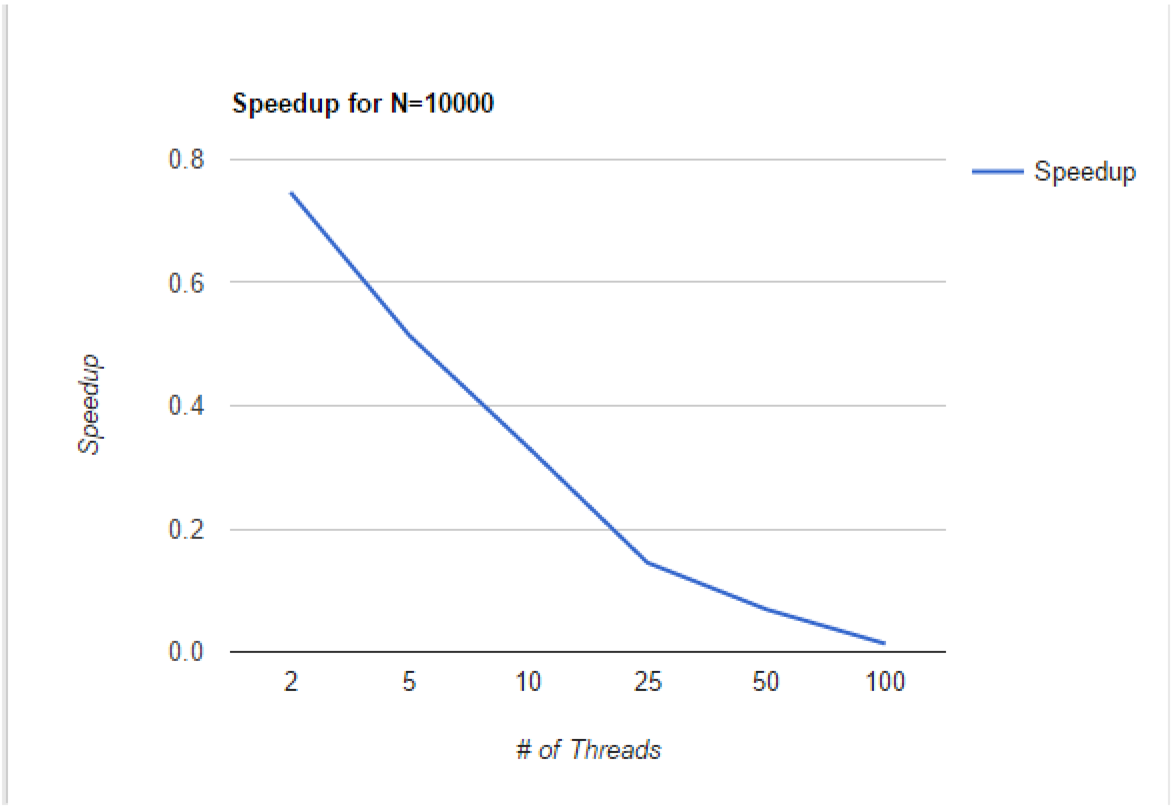
Spring 2016

Note: I compiled my program with the following command on crunchy6: “gcc –g – lm –Wall –fopenmp -o genprime genprime.c” – I added –lm to link the math header to use the sqrt() function.

	T=1	**T=2**	**T=5**	**T=10**	**T=25**	**T=50**	**T=100**
N=1000	.000032	.000163	.000281	.000568	.001284	.002589	.010981
Speedup	1	0.19631902	0.113879	0.05633803	0.02492212	0.01235998	0.00291412
N=10000	.000233	.000312	.000453	.000701	.001611	.003375	.017631
Speedup	1	0.74679487	0.51434879	0.33238231	0.14463066	0.06903704	0.01321536

*Times are averaged over 5 runs





1000 vs 10000

Both graphs exhibit a very similar graph shape with the main exception being the speedup for 2 threads is .747 for N = 10,000 vs .196 for N=1,000, which is a huge difference in performance. As with most parallel programs, we have overhead for setting up threads which is relatively fixed around the # of threads we initialize, so this causes our poor performance compared to 1 thread.

Our speedup can be estimated by comparing the overhead to how much work we parallelized and how well the load is balanced. With N=1,000, we have relatively a low amount of work that can be parallelized, so minimizing the overhead by using 1 thread leads to best performance. This is also the case for N=10,000, but here N begins to approach the threshold where using multiple thread leads to better performance. I have tested my code on N=100,000 and N=1,000,000 and experienced significant speedups, so we just need to test on an N large enough that our parallelized code saves more time than our overhead cost to illustrate the benefits of threading.

Algorithm

I adhered to using the algorithm described in the lab even though there are more efficient algorithms for generating prime numbers that may also be better parallelizable. Here are the steps I followed:

1) I allocated a global array that supports generating primes up to 1,000,000,000 (setting N this large generates a large .txt file so test with care)

2) Start with 2 and start setting all indexes of the array with a multiple of 2 and <= N to -1 to indicate not prime
3) Get next number that is still set to 0 (3 in this case) and do the same as step 2
4) Do the same thing with 5 because index 4 was set to -1
5) I iterate all the way to floor((N+1)/2), but I implement a check so that if I go past sqrt(N), I do not iterate through the array to remove all multiples of that number because all multiples of that number less than N were already removed (respective index was set to -1). Therefore, once we pass this point, the only multiples of this number that we have not yet marked non-prime exceed N and is not significant for our purpose.
6) I then can generate all of the primes by iterating through the array starting at index 2 and ending at index N and considering all indexes not set to -1 to be prime.

Essentially my algorithm does the same thing as the algorithm described in the lab except for one optimization I made, which does not affect the results or change the processing order of the algorithm; it just avoids unnecessary steps.

Performance vs Thread

With each thread, we add more overhead. This is easily illustrated by a decrease in performance with an increasing number of threads. To get an idea of what an optimal number of threads are, realize I call OpenMP in an inner for loop that iterates through an array and marks the indexes of all multiples of a number to be -1. This for loop is initialized as j * j where j is the number whose multiples will all be marked as non-prime. It iterates as long as j <= N and increments by j. Because of the way this is set up, when j is set to 2, we see the maximum number of iterations, and each subsequent time this for loop is used, there will be less iterations. These later for loops will likely have less iterations than threads (when using numerous threads) and this will also result in a higher ratio of overhead to parallel code.

It's also important to note how much work is involved in each iteration—the only thing in the for loop is an array index assignment, which is a constant time lookup of an array and reassignment. This is great for maintaining load balance as each thread will have about an equal amount of work per iteration and should finish their work with similar timing as other threads. I use the default scheduling setting which attempts to allocate an equal number of iterations for each thread. This should be the best option because each iteration should take a similar amount of time, and there is little overhead associated with this scheduling method as opposed to a scheduling method like dynamic allocation.

Lastly, there is an implicit barrier that stops the program from proceeding past the for loop until all threads have finished. This is important because we want to mark the non-prime numbers before continuing with further iterations of the outer for loop to avoid marking a non-prime number as prime. This implicit barrier is vital for assuring that my program identifies prime numbers correctly, so I cannot override the implicit barrier.

Wrapping Up

Overhead is the bane of getting good performance. The best thing we can do to optimize is to attempt perfect load balancing as well as using a number of threads that is appropriate to the machine's number of cores and less than the number of iterations in the for loop. Too many threads compared to cores will lead to excessive context switching and cache misses, which degrades performance as seen in the graph (more threads than cores can be a good thing if the parallel code has a lot of blocking, but my code does not block). I was able to balance the work in each iteration, but my code does not have a constant # of iterations as the program progresses, so the thread to iteration ratio increases as my algorithm runs, which can be considered the bottleneck of my parallel code.