Name: Yixing Guan

Netid: yg1227

Professor Zahran

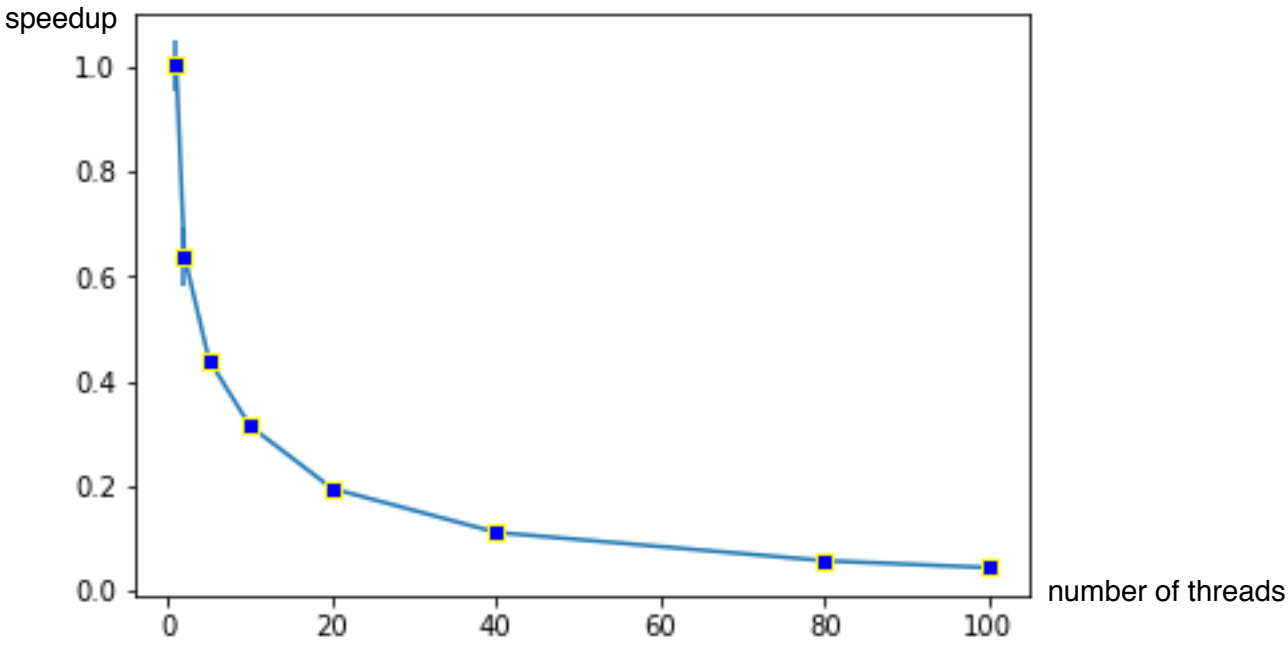
Parallel Computing

9 April 2018

Lab 2 Report

Table 1: speedup relative to the running time with 1 thread (N=10,000)

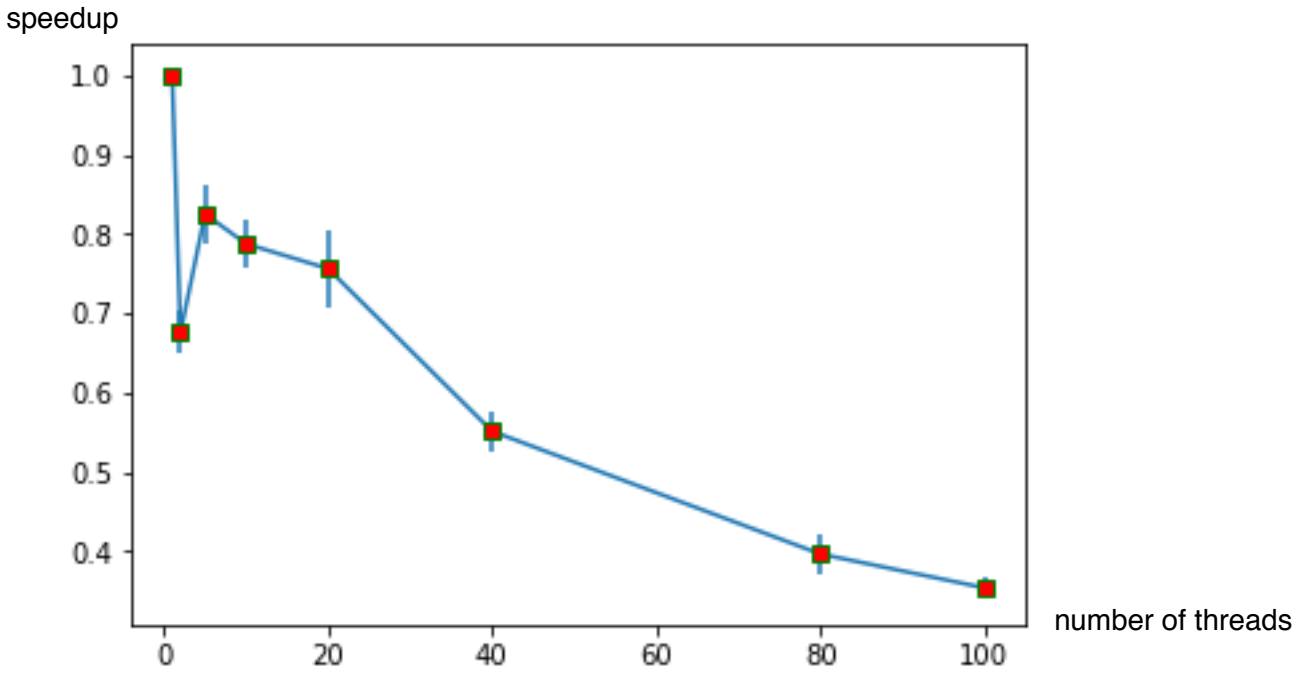
	Speedup							
N	1 thread	2 threads	5 threads	10 threads	20 threads	40 threads	80 threads	100 threads
10000	1	0.6341	0.4400	0.3144	0.1953	0.1115	0.0576	0.0443



Graph 1: speedup graph relative to the running time with 1 thread (N=10,000)
(Note: see Appendix for data used to calculate the standard deviation)

Table 2: speedup relative to the running time with 1 thread (N=100,000)

	Speedup							
N	1 thread	2 threads	5 threads	10 threads	20 threads	40 threads	80 threads	100 threads
100000	1	0.6743	0.8235	0.7863	0.7532	0.5500	0.3950	0.3536



Graph 2: speedup graph relative to the running time with 1 thread (N=100,000)
(Note: see Appendix for data used to calculate the standard deviation)

My Setting:

1) I use the "static, 1" schedule type when parallelizing the main for loop to gain the performance reported in tables, graphs and the appendix. It is the most suitable scheduling type for my program, at least according to my experiments. Although it cannot fully solve the load imbalance issue in my parallelized for-loop, it brings much less overhead than the dynamic schedule type and the guided schedule type.

2) For each number, I use a bit to indicator whether there are any smaller numbers divides it or not instead of removing it from the “prime candidate list”. Since the operations in my parallelized for-loop will only set some bits on, there is no harmful race condition and we don’t need to treat some regions as critical sessions.

Explanation for Table 1 & Graph 1:

1) To fully understand the phenomena in Table 1, Table 2, Graph 1 and Graph 2, I introduce the function of running time: f(p)=a*M/log(p)+b*p, where a, b are two fixed constants, M denote the “true” workload, a constant depend on the value of N, and p is the variable “number of thread”. To grasp the intuition behind this function, note the following:

a) Elementary Number Theory shows that for large N, there are approximately log(N) prime numbers between 2 and N.

b) When we use a large number of threads p to generate the prime number list, some threads are doing meaningless work. For example, take p=50. When thread 1 is marking the multiples of 2 as composite, thread 3 is meaninglessly the multiples of 4 as composite, all of which will eventually marked as composite by thread 1. Hence, following from a), when we use p threads concurrently, only approximately log(p) or even less threads are doing the “real” or “meaningful” work. This should justify the term a*M/log(p), where the choice of the constant a should take into account of this approximation.

c) The term b*p comes from the overhead from creating and joining p threads. Since the overhead increases as the number of threads p increases, for simplicity, we assume that the overhead is linearly increasing by a factor of b.

2) Now let us see what happens when N=10,000. We cannot get any speedup in all cases. This can be explained by the fact that if N is not sufficiently large, the term a*M/ log(p) is relatively small and does not decrease much as the number of threads p increases. Compared to the second term b*p, it might even be lower by more than one order of magnitude. Hence, the value of f(p) is largely determined by b*p, the overhead from creating and joining p threads. This explains why in Graph 1, the speedup seems to be inversely proportional to the number of threads p.

Explanation for Table 2 & Graph 2:

1) Similar to what happens when N=10,000, in all cases, we cannot get any speedup. Again, this can be explained by the fact that since N=100,000 is still not a sufficiently large number, the value of f(p) is largely determined by b*p, the overhead from creating and joining p threads. The decrease in the first term a*M/log(p), which indicates the time saved when using p threads to concurrently solve the problem, does not match the increase in the second term b*p at all.

2) However, as shown in Graph 2, we obtain a local minimum when we use 2 threads. The speedup goes up temporarily after this point. I believe the larger problem size N=100,000 and the property of f(p) when $p<e$ both contributes this phenomenon. First, the larger problem size N=100,000 makes the first term a*M/log(p) no longer negligible when compared to the second term b*p. The value of f(p) now is determined by both terms. Second, when $0<p<e$, $0<\log(p)<1$. Hence, the first term actually becomes larger when we use two threads instead of a single thread to solve the problem, which means, in some sense,

we somehow slightly “increase” the total workload when we use 2 threads. After taking a closer look at what happens when p=2, we will find that it is indeed the case: when p=2, since we use “static, 1” schedule type, thread 1 examines all factors of the form 2k, while thread 2 examines all factors of the form 2k+1. Hence, almost all work done by thread 1 is useless, since examining all factors of the form 2k is essentially examine whether the tested number is even or not. As a result, only thread 2 is actually making real progress and it has to bear the overhead from “static, 1” schedule type, let alone the overhead from creating and joining two threads. This problem is less severe when we use three or more threads. Therefore, we observe the local minimum at p=2.

3) In addition, we obtain a local maximum when we use 3 threads. This phenomenon can be interpreted as the impact of problem size on speedup. Since the first term of f(p), a*M/log(p), is decreasing as p increases, while the second term b*p is linearly increasing, there is naturally a minimal program running time for some thread number p0. Nevertheless, if N is not sufficiently large, p0 might be close to 0 and we cannot achieve this minimal program running time at all. Then as p increases from p0 to infinity, f(p) will seem to be more and more dominated by the second term b*p, which is exactly what happens in Graph 1. Now that we have a larger problem size N=100,000, p0 increases and in the case of Graph 2, is near p=3. Hence, we see the local maximum in Graph 2 at p=3, and I expect that as N increases, this local maximum might be more visible, and possibly becomes a global maximum, though more experiments have to be run in order to test this claim.

Appendix

Table 3

	program running time when N=10000 (schedule:(static,1))					
thread count	case 1	case 2	case 3	case 4	case 5	average
1	0.000215	0.000233	0.000212	0.000202	0.000209	0.0002142
2	0.000358	0.000291	0.000332	0.000371	0.000337	0.0003378
5	0.000481	0.000489	0.000493	0.000459	0.000512	0.0004868
10	0.000698	0.000682	0.000684	0.000698	0.000644	0.0006812
20	0.001088	0.001108	0.001173	0.001023	0.001101	0.0010986
40	0.001899	0.002041	0.001868	0.001866	0.001928	0.0019204
80	0.003668	0.003609	0.003829	0.003684	0.003788	0.0037156
100	0.005386	0.004674	0.004888	0.004638	0.004554	0.004828

Table 4

	program running time when N=100000 (schedule:(static,1))					
thread count	case 1	case 2	case 3	case 4	case 5	average
1	0.002170	0.002163	0.002139	0.002161	0.002149	0.0021564
2	0.003046	0.003111	0.003344	0.003362	0.003126	0.0031978
5	0.002549	0.002784	0.002497	0.002528	0.002735	0.0026186
10	0.002657	0.002695	0.002814	0.002635	0.002911	0.0027424
20	0.003006	0.002704	0.003019	0.002601	0.002984	0.0028628
40	0.004109	0.003775	0.004184	0.003813	0.003723	0.0039208
80	0.006046	0.005526	0.005070	0.005576	0.005081	0.0054598
100	0.006320	0.005928	0.006392	0.005813	0.006039	0.0060984