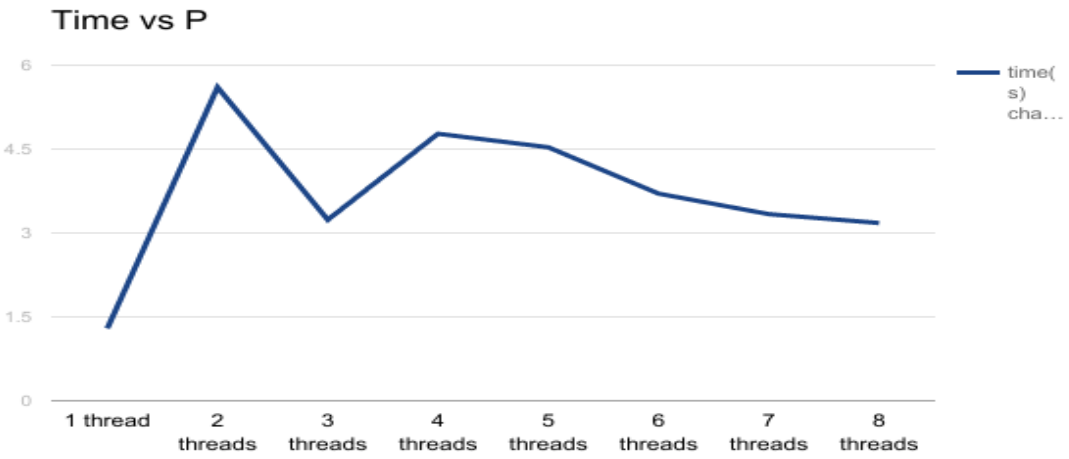
Lab 01 Report
Tianming Xu
2017/2/17

The Exploration and Analysis of Unexpected Performance of Serial program

In this lab I am working on exploring how multi-processing program will help us improve the performance of our original sequential program. As I am using the machine in Haverford computer lab, which has four cores in total, and two threads can work concurrently in one core, I took experiments of program from one thread to eight threads to see the change of performance. Since what I expected to see in my experiment is how much multi-processing program will increase the speed of my program. The result is very unexpected, which is that the multi-processing program works more slowly than sequential one.

Analysis: Times vs number of threads

In the experiment of finding the relation between running time and number of threads working on it, I pick 10^7 times as default running time. Not only it is a fairly large number to see the difference between the time spent by each experiment, but also not that much amount of time that I need to wait a very long time. In the line chart and table below, we can see that when we use 2 threads at the same time, I takes the longest time to run. It is very abnormal since in the textbook, it mentions that using 2 threads is the most efficient way for computer to compute. And when I use three threads at the same time, the time drop quickly and goes up when I use four threads. Interestingly, using three threads is more efficient than using eight threads. But the most surprising thing is that, the sequential program (only use one thread) work most efficiently! That is not what I expected and why it happens makes no sense to me.



Times vs P (10^7 times for each test)

	(threads) 1	2	3	4	5	6	7	8
10^7	1.293162s	5.60515s	3.23595s	4.77442s	4.53226s	3.7038s	3.33701s	3.178327s

After analyzing the times vs P chart, I calculate the speedup and efficiency of each experiment. Now we can see more clearly that for each experiment, when we use multiple threads to run our program, it works more slowly than we run it sequentially. And the efficiency just continuously goes down when we use increasing amount of threads.

1).speedup

(number of threads) 2	3	4	5	6	7	8
0.2307	0.399623	0.270853	0.285323	0.349144	0.387521	0.315321

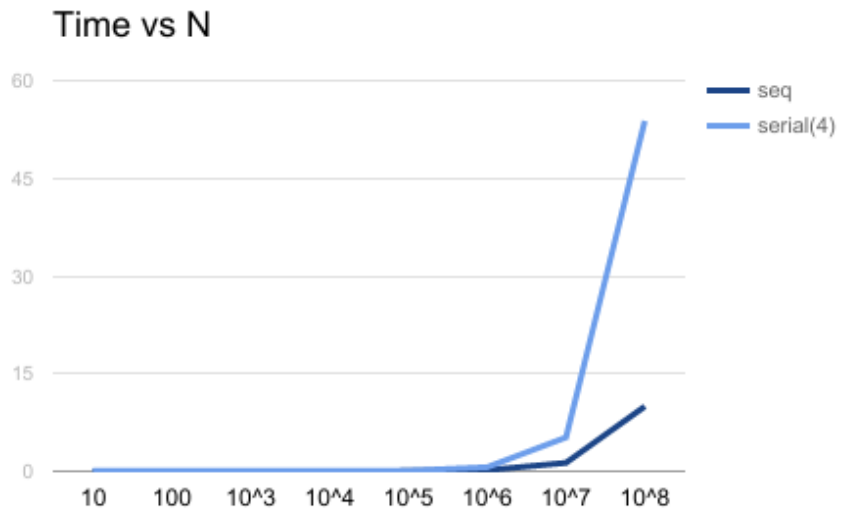
2).efficiency

(number of threads) 2	3	4	5	6	7	8
11.53%	13.321%	6.7713%	5.70646%	5.8191%	5.53601%	3.94151%

Analysis: Time vs number of throwing

After the shocking experiment of time vs P, there are more interesting things happening when I test the relation between running time and number of test. Since the machine has four cores, I choose 4 threads as default amount of threads to compare with my sequential program. Then I find that even we just run program for 10 times, which is done instantaneously in our sequential program, it takes weigh more time in multi-processing program. We can see that it takes 100 times more time than sequential one. Also, there is a quite large change for running time after 10^3 for sequential program, it does not change that much for multi-processing program before 10^6. So I decided to test it again for 10^5(and the largest result is in parenthese), it still does not change that much (less than two times) comparing to the difference between 10^4 and 10^5 of sequential program (nearly 7 times). But it goes well after 10^6. I am really curious about why this happens.

2. Times vs N (1 thread and 4 threads comparing)



	(thread number)1	4
10	0.000337s	0.048050s
10^2	0.000426s	0.055658s
10^3	0.000919s	0.082632s
10^4	0.006260s	0.083952s
10^5	0.040887s	0.081987s(0.152619s)
10^6	0.168011s	0.627177s

10^7	1.293162s	5.197017s
10^8	9.958798s	53.815107

Difference between using Parallel time function and default time function

I actually made a mistake when I did the experiment before. I used the clock function, which I used in my sequential program. However, I found that it actually does not take that amount of time it showed on screen. And I used stopwatch to track its actual time and it really does not take that long. Hence, there is something wrong in my codes.

Finally, I found that since we used sequential clock function to calculate total time and there are actually 4 threads, so it probably count it for four times. Hence, I choose to use the openMP time function to calculate the total time. And this time, luckily, I found a positive relation between time and number of threads (from 2 to 8), though single core is faster than 5 threads, it is slower than 8 threads. Hence, I concluded that the time function used for sequential dart program has some redundant computation when we change it to parallel one.

Data: time vs P by using omp_get_wtime() (10^7 times for each)

	(threads) 1	2	3	4	5	6	7	8
10^7	1.075432s	3.11159s	1.11079s	1.55920s	1.23273s	1.02521s	0.926039s	0.782864s

Now we can see a positive relation of number of threads and speedup, which is what i expected. However, it does not become more efficient. It is really interesting to see that using three threads is the most efficient way to run program.

1).speedup

(number of threads) 2	3	4	5	6	7	8
0.345621	0.968168	0.689732	0.872398	1.04899	1.161324	1.373714

2).efficiency

(number of threads) 2	3	4	5	6	7	8
17.28%	32.2722%	17.2433	17.4479	17.48316%	16.5903%	17.1714%

Conclusion of multi-processing program experiment

For this lab, I think the reason that it does not reach my expectation is that the communication and distribution time between each thread take a lot of time in our

program. But this can not explain why after I change my time function, the eight threads work faster than 2 threads. Also, it can not explain why using 2 threads is the slowest way to run program. Hence, I think there are still a lot of things for us to further study multi-processing program.