Introduction

A positive integer N is a perfect number if the sum of all its factors excluding itself is N, where a factor is a perfect divisor of N. Let's examine the following examples.

- Given number 6, I know its factors are 1, 2, and 3 excluding itself. Since 1 + 2 + 3 = 6, 6 is a perfect number.
- Given number 28, its factors are 1, 2, 4, 7, and 14 excluding itself, and since 1+2+4+7+14 = 28, 28 is also a perfect number.
- Given number 496, its factors are 1, 2, 4, 8, 16, 31, 62, 124, and 248 excluding itself, and since 496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248, 496 is too a perfect number.
- Given number 30, its factors are 1, 2, 3, 5, 6, 10, and 15 excluding itself, since 1 + 2 + 3 + 5 + 6 + 10 + 15 = 42 ≠ 30, 30 isn't a perfect number.

Introduction (continued)

Two different programs are created to check whether or not a number a perfect number, and compare the two programs.

1. A serial program which does the work in the program serially.
2. A parallel program which can run on one or multiple threads. This program can split the computation tasks to multiple parts, each thread can do different parts, and at last, the calling thread (parent thread) combines the results from different threads and get a conclusion about whether a number is a perfect number.

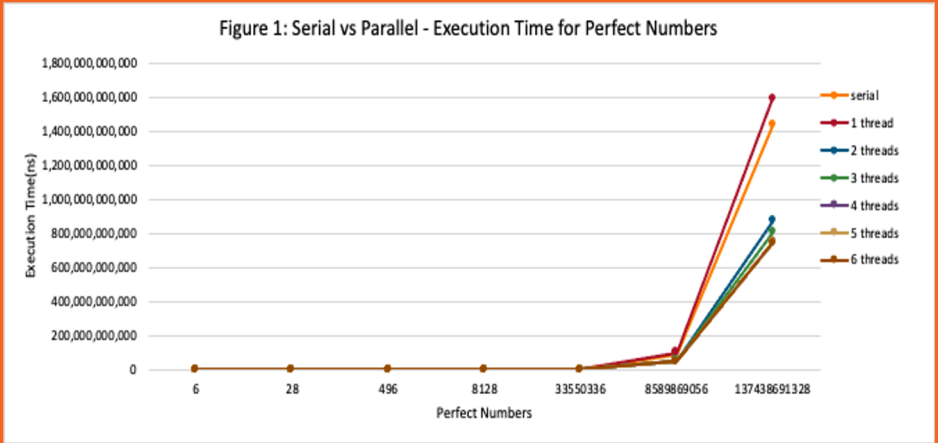
Introduction (continued)

In the parallel program, I use Pthread to create an array of threads. Because the threads run concurrently, different threads can add their factors almost simultaneously to the sum which is a critical section. This could cause race condition. I avoid race condition by creating mutex lock. In this way, only one thread can run critical section at one time.

Experiment

1. Serial vs Parallel for perfect numbers

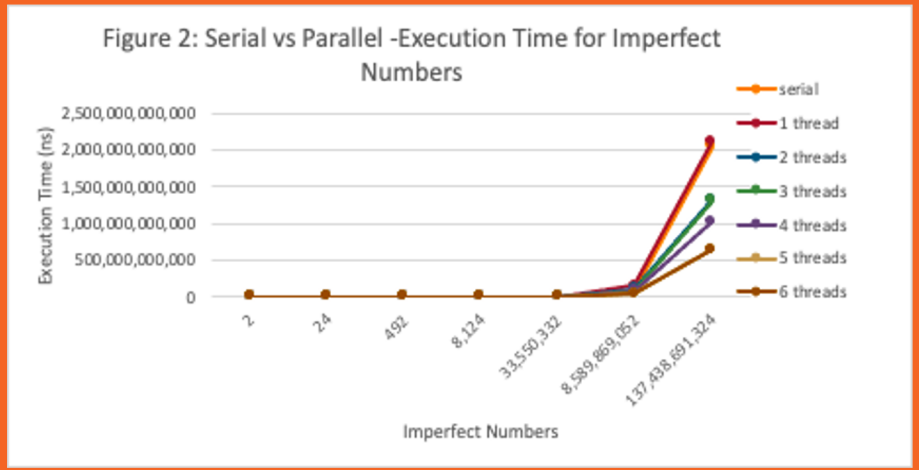
I run the serial program and parallel program that are created with C++. I check the execution time for some perfect numbers, they are 6, 28, 496, 8128, 33550336, 8589869056, 137438691328. Each number is checked with different number of threads. The result is shown in below Figure 1.



Experiment (continued)

2.Serial vs Parallel for imperfect numbers.

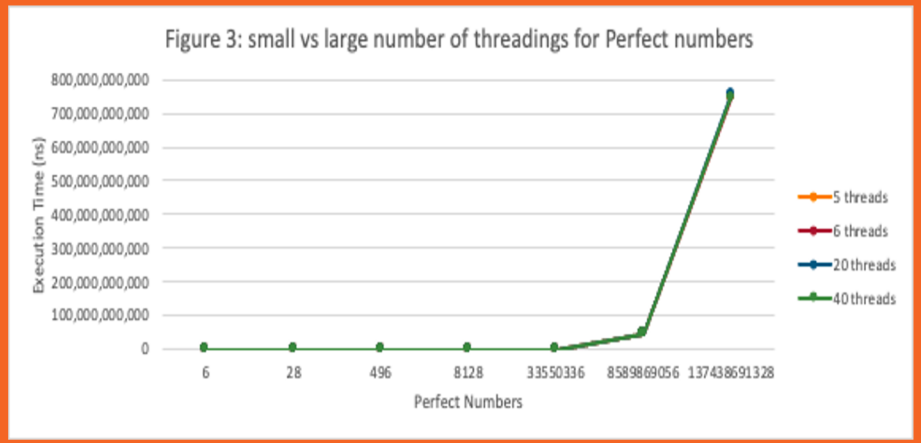
Besides perfect numbers, I also ran the two programs for some imperfect numbers with different number of threads . The results are shown as below in Figure 2.



Experiment (continued)

3.Less threads vs more threads for perfect numbers

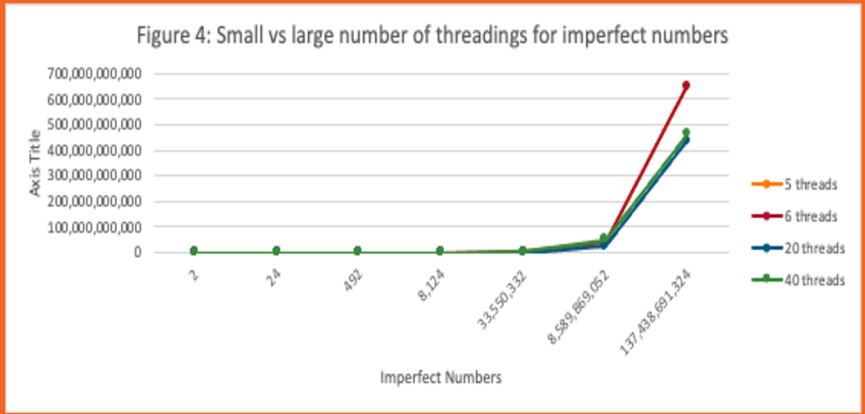
I compare the execution time of parallel programs with a few threads and that with many threads. The numbers are perfect. It is shown in Figures 3.



Experiment (continued)

4.Less threads vs more threads for imperfect numbers

For the imperfect numbers, I also compare the execution time of parallel programs with a few threads vs that with many threads. The output is shown in Figure 4.



Experiment (continued)

5. Race Condition

When I remove mutex lock from the finished parallel program, I did not observe race condition. I did it for both perfect numbers and imperfect numbers. The outputs are all correct. This indicate that the race condition occurs very rare.

Experiment (continued)

6. speed-up

I get the speed-up factor of the parallel programs to the serial program:

Speed-up factor - parallel program to serial program								
Numbers	1 thread / serial	2 threads / serial	3 threads / serial	4 threads / serial	5 threads /serial	6 threads / serial	20 threads / serial	40 threads /serial
6	3.60	4.43	5.11	5.49	6.46	6.79	15.74	22.48
28	3.35	4.24	4.51	4.52	6.36	7.10	13.94	23.79
496	3.24	3.52	3.55	4.37	4.92	5.97	11.93	19.11
8128	2.18	2.26	2.08	1.38	2.29	2.83	4.16	6.70
33550336	1.93	1.03	0.93	0.86	0.85	0.84	0.84	0.84
8589869056	1.09	0.60	0.56	0.51	0.51	0.51	0.51	0.52
137438691328	1.11	0.61	0.56	0.52	0.52	0.52	0.53	0.52

Experiment (continued)

7. Estimating s

The number of computer processor cores is 2. With Amdahl's Law and speed-up factor (serial program to parallel program), I are able to estimate s, which is the percentage of the work in the program that is executed in serial.

Amdahl's Law:

$$S = \frac{1}{s + \frac{1-s}{N}}$$

The speed-up factor of serial program to parallel program, and estimated s for different numbers and different nthread are shown:

Experiment (continued)

Speed-up factor - serial program to parallel program				
Number	Serial / 3 threads	Serial / 4 threads	Serial / 5 threads	Serial / 6 threads
33550336	1.08	1.17	1.18	1.20
8589869056	1.80	1.95	1.96	1.97
137438691328	1.78	1.91	1.92	1.93

Estimated s				
Number	Serial / 3 threads	Serial / 4 threads	Serial / 5 threads	Serial / 6 threads
33550336	0.85	0.71	0.70	0.67
8589869056	0.11	0.02	0.02	0.02
137438691328	0.13	0.04	0.04	0.04

Experiment (continued)

8. sources of speed-up

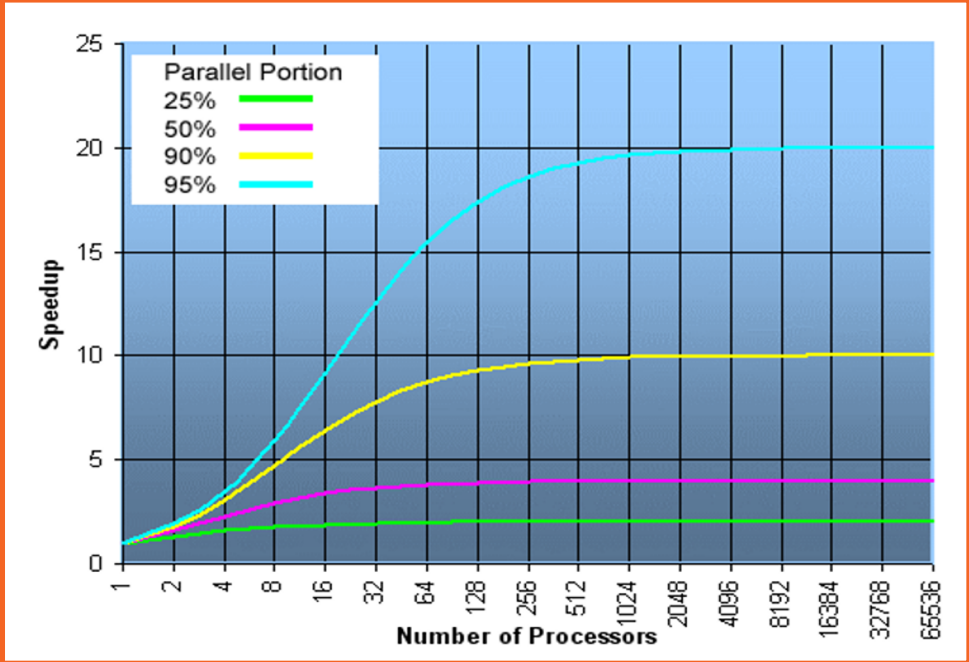
Based on the experiments and the results, to speed up, one of the methods is to increase the number of processors.

But the speed-up has a bottleneck based on research. When the number of processor is greater than a number, even I add more processors, the speed-up keeps almost the same.

Another source to speed up is to increase the portion of the work that can be executed in parallel.

To some extent, the programs with the greater portion of parallel tasks will be more efficient. When the parallel portion is 95 percent, the speed up could be up to 20 based on research. This conclusion is also shown in figure below .

Experiment (continued)



Analysis

1. Serial Program vs Parallel Program

1.1. For small perfect and imperfect numbers (up to 8128 in this project) : Parallel programming doesn't have much advantage in comparison with serial program. The execution time is very close. This is shown in figure 1 and figure 2.

1.2. For very large perfect and imperfect numbers (greater than 33550336):

Analysis (continued)

1. 2.1. When the number of threads is not very large, n_thread = 1, 2, 3, 4, 5, 6 in our project:

There is a trend in the results. The serial program and the one-thread program take the most time. The one-thread program needs slightly more execution time than serial program. But then increasing number of threads decreases the execution time. Our hypothesis is that it takes more time to initialize in parallel program than that in serial program. When the number of threads increases, the extra time to initial in parallel program is finally offset by the time efficiency gained.

Analysis (continued)

1.2.2. When the number of threads becomes very large, n_thread = 20, 40 in this instance:

Then increasing the number of threads doesn't keep decreasing the running time of the parallel program. Therefore, there is bottleneck to increase the efficiency by keeping adding more threads. Our hypothesis is initializing greater amount of threads and distributing more range of numbers consumes more time. This is also shown in figure 3 and figure 4.

Analysis (continued)

2.perfect numbers vs imperfect numbers

I did both perfect number and imperfect numbers, separately. When the value of the numbers is close, the execution time for a imperfect number is mostly less than that for a perfect number. It is may be because more calculation are needed to determine whether or not a number is perfect. If a number is perfect, the program has to check all the factors. If a number is imperfect, since the program checks the factors from greatest to smallest, when the sum of part of the factors is greater than the number itself, then the program won't need check other factors because it already has answer.

Conclusion

When the number of tasks is small, the execution time of the serial program and parallel program is very close. There is no significant correlation.

However, when there are a very large number of tasks to do, the parallel program have a big improvement in efficiency when the number of thread is greater than one and less than some number, let's say X. When the number of threads keeps increasing and exceeds X while the number of tasks keeps the same, then the execution time doesn't change a lot even there are more threads. Therefore, there is a range of number of threads. Within the range, I can decrease the execution time by increasing the number of threads.

Thank you!