Operating Systems White Paper

Scott Madeux
CS 345 – Brother Comeau
4/5/19

Purpose: To determine how many threads should be used for a given program.

Executive Summary

With multi-core processors becoming main stream these days it's hard to find a device that has only one core. Multiple CPUs can give computers several benefits, one of which being that some programs have the ability to get work done more quickly by performing tasks in parallel. With a single-core system you are limited by the speed of the CPU because tasks can only be performed one at a time. However, with a multi-core system you only limited by the number of cores and the percentage of your program that can be parallelized.

One method of giving your code the ability to run in parallel is by creating threads. Threads are very light weight compared to other methods such as forking a process. Threads can help you split up your process into lot of smaller tasks that are all independent of each other and can be run whenever a CPU has time to run them. This is great because you can potentially have 4 threads from the same process running at the same time on a 4-core system instead of that same process only being able to use one core because everything must be performed serially. Hence, you can get work done 4 times faster by taking full advantage of all your cores.

A few questions start to arise however, when we begin using threads. How many threads should I use? What factors affect how many threads I should be using? These are a couple of the things I want to discuss. I want to go through what some experts have said about the best ways to use threads when parallelizing your code and some of my own research on the topic. By the end of this white paper I hope that you'll be able to gain some knowledge and confidence in knowing how many threads you need in your program.

Body

You've probably heard of Amdahl's Law, but in case you haven't, it is a way of estimating the speedup of a process based on the number of processors being used and how much of the process and be run in parallel. Amdahl's is given by the following formula where P is the percentage of the code that can be run in parallel and N is the number of processors being used[1]:

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

So according to Amdahl's law, two factors that play a huge role in speeding up out process are the percentage of that process that can be run in parallel and the number of cores we have. We have to find a good balance between the two though. If 95% of our code can run in parallel, but we only have two processors, we won't get much out of having all of that parallel code. The same can also be said for having a high number of processors, but a low percentage of parallel code. The graph below illustrates what happens to speedup as you change these two factors[2].

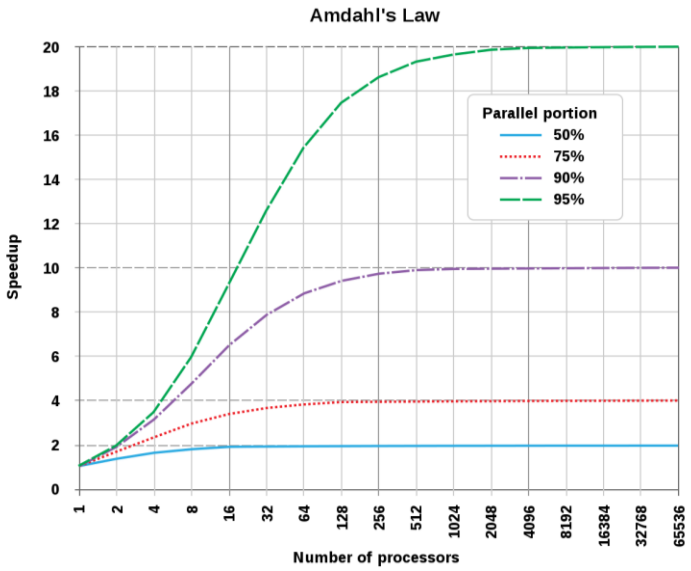


Figure 1: Speedup based on parallel proportion of code and number of processors using Amdahl's Law.

Now that we've talked a bit about what factors can affect the speedup of a program, let talk about threads and how we can know how many threads we should be using to take full advantage of your parallel code.

Not all programs you will be writing are the same and there is a very important distinction to make when calculating how many threads to use. This distinction is whether your program is uses more CPU time (CPU bound) or whether it uses more I/O (I/O bound). This will make a difference because I/O bound programs only use the CPU for short bursts of time in between I/O, which let's you use more threads since they constantly being switched out for each other. CPU bound programs on the other hand require much more time on the CPU. This means that threads aren't switched out as frequently and the optimal number of threads is lower.

For CPU bound tasks, calculating the optimal number of threads is fairly simple. According to Brain Goetz[3] the optimal number of threads for a program like this is[4]:

$$Number\ of\ Threads = Number\ of\ cores + 1$$

This way you will always have all of your cores busy with your program threads and one in the ready queue waiting to be swapped out.

For I/O bound tasks things are slightly more complicated. In this case we will need to know (or estimate) the ratio of wait time to service time. In other words, the amount of time spent waiting on I/O (w) over the amount of time spent using the CPU (s). With this fraction, we can calculate the optimal number of threads using the following formula[4]:

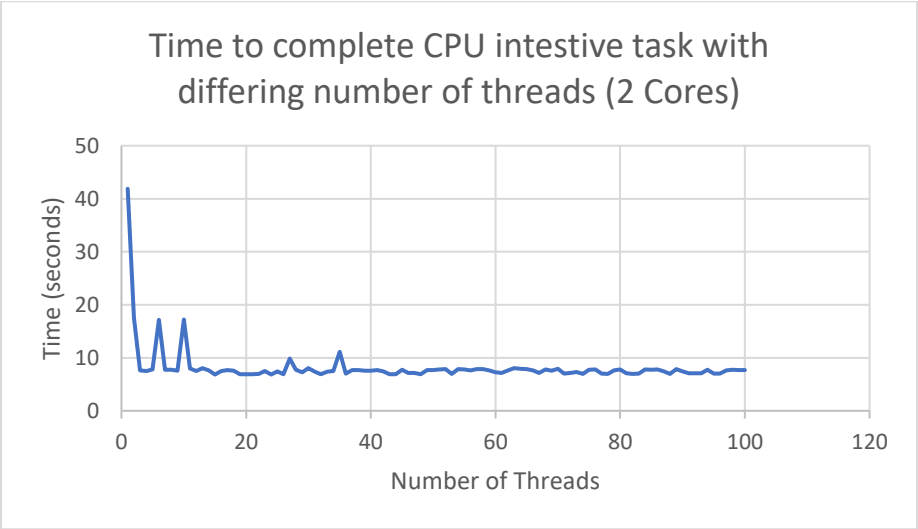
$$Number\ of\ Threads = Number\ of\ cores \times (1 + \frac{w}{s})$$

Using this formula, we should have enough threads to keep all of the cores occupied while the other threads are waiting for I/O.

Since we've looked at some of the different factors that can be tweaked to optimize our code, let's look at some real life experiments and see if these ideas hold up on a actual machine.

For this experiment, I chose to run two different programs (one CPU-bound and the other I/O bound). I ran them each 100 times on a 2-core machine starting by using 1 thread all the way to 100 threads. Next, I ran them each 100 more times, but this time on a 4-core machine.

Here is a graph of the data I got by running a CPU bound task on a 2-core machine:



As you can see the time it takes to run the program drops drastically by only adding a few threads. After that, time levels off and stays fairly consistent except for a few small spikes which could have been caused by any number of things. According to the formula that we talked about above, the optimum number of threads for a program like this should be 3 (2 cores + 1). If we look at the numbers, we can see the time it took to run flattens out once we go beyond three threads.

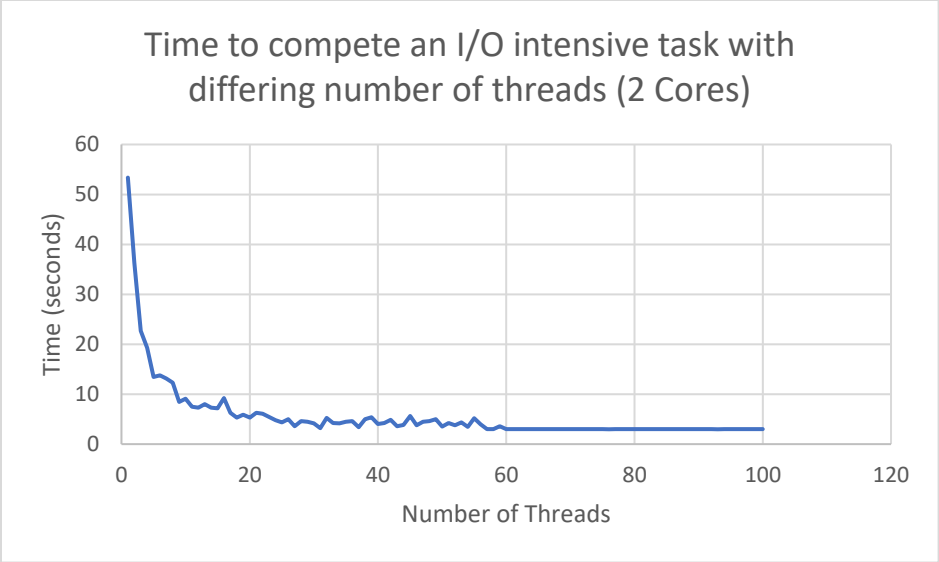
Number of Threads	Time (Seconds)
1	41.87
2	17.52
3	7.64
4	7.53

Now let's look briefly at the same test run on a 4-core machine to see if our formula still holds up:

Number of Threads	Time (Seconds)
1	9.93
2	5.12
3	3.49
4	2.67
5	2.77

Things are not as definitive as with the test on the 2-core machine, but we can see by the table above that the time levels off after around the number of cores plus one.

So what about I/O intensive tasks? Well, according to what we discussed earlier, they should generally use more threads before the time taken to complete the task levels off. But how many more? Let's look at the data from a test I ran with an I/O intensive program on a 2-core machine.



As you can see in the graph, it takes more threads for before the time taken to complete the task stops getting smaller. In fact, the line doesn't completely flatten out until it gets to around 60 threads. Now to calculate the optimum number of threads we would have to go a little more in depth and figure out the time spent waiting for I/O verses the time spent on the CPU. But, it's safe to say that this I/O bound program can take advantage of more threads than the CPU bound program. I also ran this same test on a 4-core machine and got almost exactly the same results to those I got on the 2-core.

Conclusion

There are many factors that play a role in how well code can be sped up by being run in parallel. In this paper I was only able to touch on a couple of things like changing the number of processors, threads, and the amount of code that can be run in parallel. Other factors include caching, different libraries used to parallelize your code, and complier optimizations[5]. I feel like the things discussed here are a great base for learning how to analyze your parallel programs and get the most out of using threads to speed up your code. There are many things you can do to whittle down the time it takes to run your code, but one of the most important is knowing the right number of threads that will help your code take full advantage to all of that multiprocessing power.

Sources

1. Amdahl, G. M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*(Tech.). Sunnyvale, CA: IBM. website: http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf
2. D. (2008, April 13). AmdahlsLaw.svg [Digital image]. Retrieved April 5, 2019, from https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg
3. Brian Goetz: Author of Java Concurrency in Practice https://nofluffjuststuff.com/conference/speaker/brian_goetz.html
4. Goetz, B. (2002, July 1). *Thread pools and work queues*(Tech.). Retrieved April 5, 2019, from IBM website: https://www.ibm.com/developerworks/java/library/j-jtp0730/index.html
5. Kukanov, A. (2008, March 4). *Why a simple test can get parallel slowdown*(Tech.). Retrieved from https://software.intel.com/en-us/blogs/2008/03/04/why-a-simple-test-can-get-parallel-slowdown