Benjamin Hansen

Professor Luc Comeau

CS 345

July 11, 2020

Parallelism: Considerations for the Number of Threads Needed

Executive Summary

Today, parallelization of programs boasts huge performance gains that businesses desire for their software to increase the efficiency of the user using their software. With a busy world becoming more and more dependent on software solutions to real world problems, the typical user is becoming more and more impatient with his/her technology, expecting it to run fast like other programs he/she is familiar with. If it does not meet expectations, the user will leave. Looking to multithreading as an answer to this problem is a good step, but how many threads will actually guarantee desired performance gains?

How many threads are needed?

Chadha et al. describes the paradox between too many threads and too few threads. If an application has too few threads, then we underutilize a CPU's cores that are designed for parallel processing [1]. Take a serial program as an example and consider a scheduling of a CPU's cores such that when a user-program is running, the CPU is bound to that one program. If the program only spawns one thread (the main thread), then any other cores the multicore CPU has are wasted, spinning for work to do. On the other hand, if too many threads are spawned passing the threshold for how many threads can actually be served by CPU cores, the threads wait, fighting for CPU resources, thereby decreasing performance [1][2].

Having more threads than there are cores leads to threads busy-waiting for locks to release, counters to decrement, and threads to finish work before continuing (like at a barrier). Consider a scenario posed by Moore and Childers [2]. The Operating System (OS) context-switches a process (and thereby all of the threads it spawned, called child threads) out and serves some other process. This process is no longer running even though we want it to! Suppose, also, that many of the child threads were waiting for a lock held by one of the threads to become available. The great overhead and decrease in speedup then comes when the OS context-switches in the suspended process (the one we want to be running), and the scheduler prioritizes the thread which owns the lock. If the thread has to wait for the other threads to be served, all CPU time is wasted, doing no useful work, until that thread is served and releases the lock.

Let us go back to the simple example of a multicore CPU serving a serial (not multithreaded) program like we introduced earlier. The problem here was the CPU's power was wasted. Even though this CPU contains multiple cores, only one is used on the serial thread. How to we increase utilization of the CPU's other cores? Chadha et al. says, "A common solution is to set the number of threads equal to the number of available cores. [1]" Sample data of a CPU-bound program shows us the reasoning behind this rule of thumb. The program performs two random 100x100 matrix multiplications on cells filled with random integers from 0-99.

Models

The following models and analyses of those models were generated from programs ran on a Dual-Core Intel Core i5, two-core CPU running at 2.7 GHz on a MacBook Pro (macOS 10.15.5). We will refer to this as our example machine.

Data were collected on the real-time (total elapsed time from execution to completion) of a CPU-bound and an IO-bound application. As we look into how increasing the number of threads used on these example programs affects performance, we discover patterns that will help us answer our question for how many threads are needed.

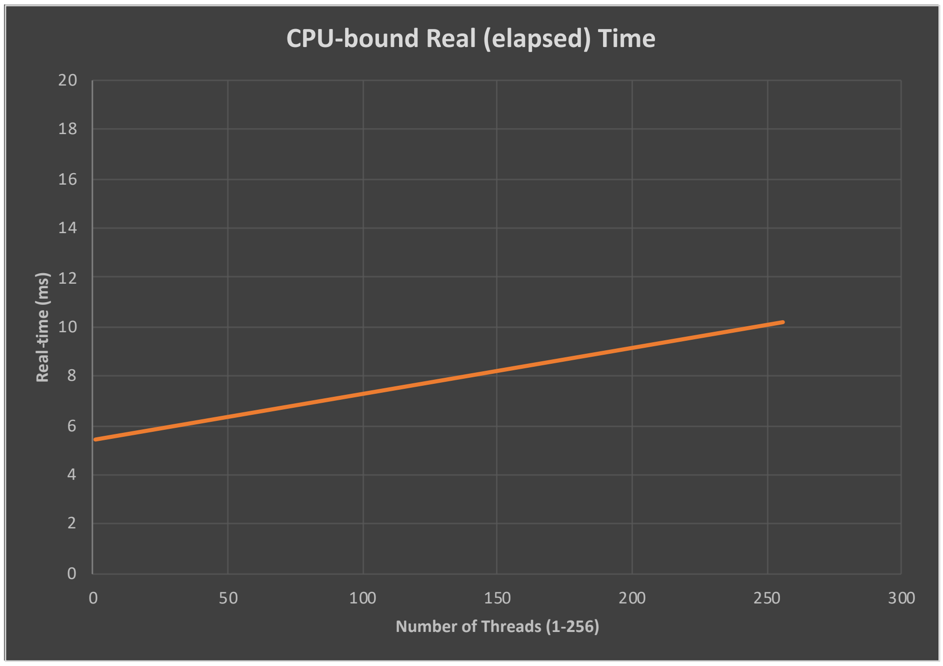

Figure 1: Linear fit line of a CPU-bound program (100x100 matrix multiplications) real-time data points in milliseconds versus the number of threads used (1-256).

Figure 1 shows the average linear growth in elapsed time of a CPU program as we increase the number threads used. Instead of an average decrease in elapsed time, which is what we want, we see an average increase in time. This happens because the threads start to contend for the CPU's time.

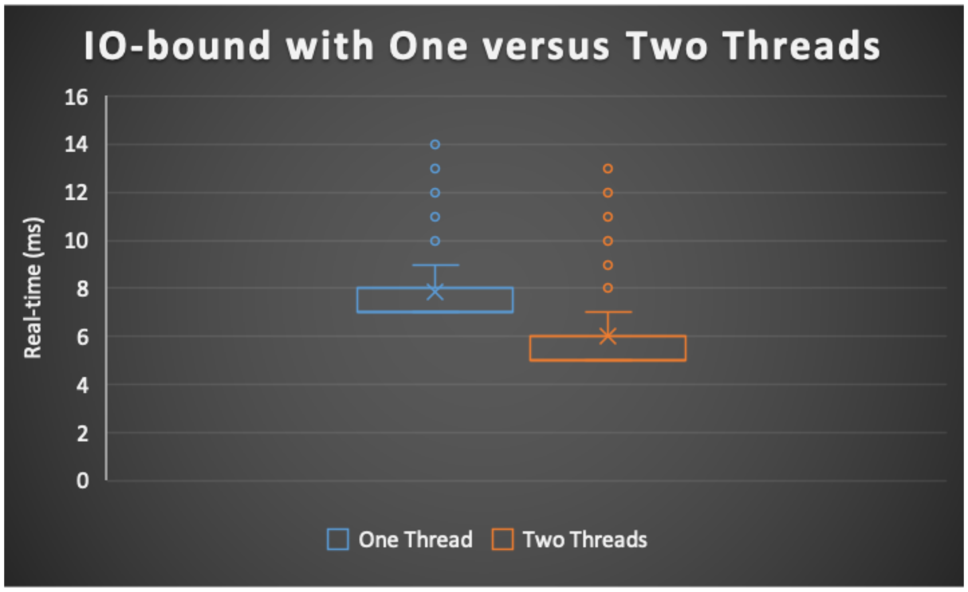

Figure 2: Box plots a CPU-bound program's real-times using one thread versus two threads. This shows the decrease in real-time using two threads.

Data Analysis

To analyze the behavior at the beginning of our model, where the number of threads has not exceeded the number of cores, an Independent (Two) Samples t-test was carried out on data collected from our example machine. Real-times of the same CPU-bound program as analyzed earlier were collected, testing 100 trials each of the program using one thread and two threads. Since the sample size is large, the data will be approximately normal. Our hypothesis was that the elapsed times of the program ran with one thread would be slower (longer) than similar elapsed times ran with two threads. From Figure 2, we see the average times of the program with one thread are greater than those with two threads. The computed P-value for the hypothesis test was 1.9809e-13. Thus, with most reasonable alpha significance levels, there is significant evidence to suggest that the difference in the means of elapsed times of these two data trials is not zero. We can conclude that, on a two-core machine, a CPU-bound program will run faster with two threads than with one. We see that the general rule of thumb is verified: a CPU-bound program running on an n-core machine should use n threads or less.

What are the implications of these data? We can see in the elapsed (real) time of the CPU-bound program a decrease in time when using two threads instead of just one. But as soon as we use more than two threads, we start an upward slope in elapsed time. This is because matrix multiplication is CPU-intensive (i.e. it requires as much of the CPU's computational power as possible to compute the multiplications). As Moore and Childers also observe, the increase in elapsed time of a CPU-bound program when the number of threads surpasses the number of cores is due to the threads contending for a core's computational attention. Hence, the rule of thumb is sufficient for initial production of a given software. In the best case, each CPU core will serve each thread and thereby achieve the maximum amount of performance. As more time and money is available, data analysis can help the programmer find the right balance.

So when would having more threads than cores be a good idea? “If a programmer expects that some threads could block for any reason, then she might create more threads than the number of available processor cores in the hope that the operating system scheduler would help her achieve higher performance. [1]” Here we see the opposite of a degradation in the real-time performance like we saw with a CPU-bound program. The more threads we use the faster the program runs, up to a certain plateau spot, where we can no longer make any real-time gains and instead the system and user times increase with the real-time staying fixed. Figure 3 shows the exponential decrease in elapsed time for an IO-heavy program as we increase the number of threads. This happens because as one thread given an input waits for the output to return, another thread can be started by the CPU with another input needing to be served.

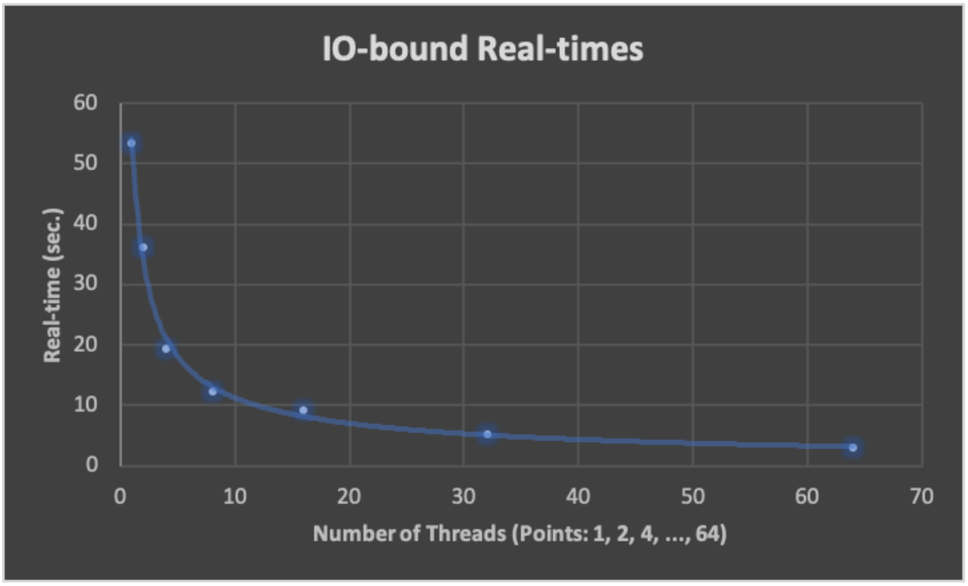

Figure 3: Scatter plot with fit line of an IO-bound program real-times (elapsed time) in seconds using 1, 2, 4, …, 64 concurrent threads. The real-times have been rounded to the nearest second.

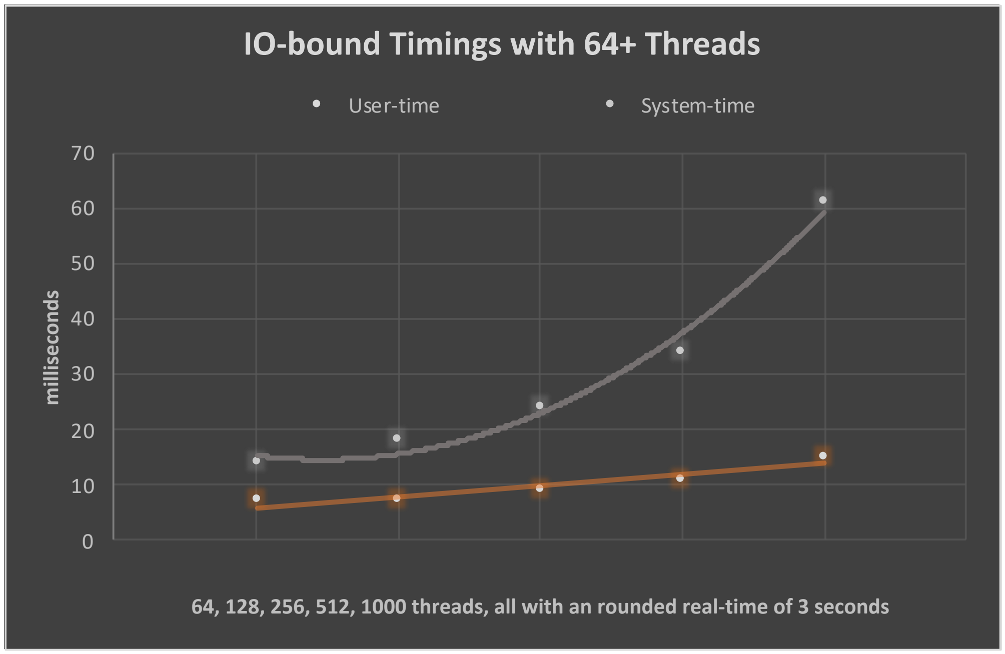

Figure 4: Plot of real-, user-, system-time using 64, 128, 256, 512, 1000 which is ordered on each consecutive line from the left to the right. Fit curves show the growth of these timings. The times all have a correlated rounded real-time of 3 seconds.

To analyze the behavior of the elapsed times of IO-bound programs as the real-time has plateaued even when the number of threads are increased, we graph the user-time (time the CPU spends on the user's actual program code) and the system-time (time the CPU spends doing OS system calls, which are more time consuming than user code). What we notice is that our increase in performance like shown in Figure 3 plateaus and the time the CPU spends doing system calls increases. That is because of the threads contending for time to be served. The threads IO is completing faster than the CPU can serve all the threads in a single iteration. Thus, some threads wait until the CPU can serve them, having already received the desired output from the IO call. There is a limit on each machine and each program on how beneficial adding more threads to the problem will be. We see, in general, that the more threads we add after we have hit that limit will only increase the performance time, rather than our desired decrease in time.

Although the rule of thumb is to have as many threads as there are cores, this does not take into account many other variables that are a consequence of the portability of a program. Not all resources the program will use have the same amount of power available. Thus distributing the available power of an embedded system to all available cores might decrease overall speedup versus giving more of that power to a few cores running a CPU intensive program. Several other variables like degree of parallelism, bandwidth, cache utilization, hardware configurations, and the variableness of inputs into the program, cause this rule of thumb to breakdown.

How to cope with all these variables

So how can one determine how many threads to spawn during a given run of the program?

Use a trusted library. As Moore and Childers point out, “Manual coordination of the overall thread count (among all programs) is burdensome and too slow to react to quickly changing workloads. [2]” Having to manually decide every time a new configuration comes about (which is fairly often for most modern systems) is too expensive. Dynamically deciding the number of threads needed is an inherent problem to solve for any scalable application. [2] suggests using a program utility model in addition to the programmer considering the degree of scalability available. Both human and computer evaluation of program performance are necessary to achieving the right balance in how many threads are needed.

Conclusion

It becomes the programmers great task to determine how to use the select threads available. “On a machine with support for only a finite number of threads, we must have a strategy to prioritize among the available threads. [3]” If the programmer divides the work to be done up too much, the threads contend for the CPU’s execution power. If the programmer divides the work up only a little, then the threads are required to take on more work when the CPU could serve more threads at once. A balance must be found. If there is time and money to experimentation, then modeling the change in performance time given the amount of threads like we have analyzed is the best approach to finding that balance. As Chadha et al. note, the number of threads needed for a given application “can also change at runtime making it infeasible for the programmer to determine this number. [1]” If such analyses are too expensive, then a good runtime library should be sought which will give the program the right amount of threads to, on average, achieve maximum performance. Because there are so many variables that can affect the effectiveness of a multithreaded application, as Chadha et al. sums it up best, “It is important to have a runtime mechanism that can adapt the number of concurrent threads to attain high performance. [1]”

Works Cited

[1] G. Chadha, S. Mahlke, and S. Narayanasamy, “When less is more (LIMO): controlled parallelism for improved efficiency,” *CASES '12: Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems,* pp. 141-150, Oct. 2012. Available: https://dl.acm.org/doi/abs/10.1145/2380403.2380431

[2] R. W. Moore and B. R. Childers, "Using utility prediction models to dynamically choose program thread counts," *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, New Brunswick, NJ, 2012, pp. 135-144, doi: 10.1109/ISPASS.2012.6189220. Available: https://ieeexplore-ieee-org.byui.idm.oclc.org/document/6189220

[3] J. T. Oplinger, D. L. Heine and M. S. Lam, "In search of speculative thread-level parallelism," *1999 International Conference on Parallel Architectures and Compilation Techniques* (Cat. No.PR00425), Newport Beach, CA, USA, 1999, pp. 303-313, doi: 10.1109/PACT.1999.807576. Available: https://ieeexplore-ieee-org.byui.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=807576&isnumber=17466