Multithreaded Applications

White Paper

CS345 Operating Systems

How Many Threads Does It Take to Weave a Tapestry of Efficiency?

Andrew Rindfleisch
Brigham Young University – Idaho

Introduction

When a programmer first enters the wonderful world of parallel programming, he/she is sometimes tempted to throw an absurd number of threads at the CPU in an attempt to maximize efficiency. However, after some trial and error, eventually the programmer begins to wonder, “Is there an optimal number of threads that I can utilize to maximize my application’s performance?” This paper aims to answer that question.

Types of Processes

This paper explores the efficacy of threads as they pertain to two separate applications: computational efficiency and IO handling efficiency. Computational efficiency was measured by performing large matrix multiplication using the POSIX threads API (pthread) in C. IO handling efficiency was tested using a program which creates threads and causes them to sleep for a set amount of time, mimicking the time a thread would spend waiting for an IO request. The program uses the POSIX threads API and is written C++.

Workload selection

Matrix multiplication was performed on two 1,000 x 1,000 matrices. The number of threads tested ranged from 1 thread (i.e. no multithreading) to either 9 or 10 threads (depending on the number of processors on the system). This large workload was chosen to prevent the overhead from overshadowing the performance improvement.

IO handling was performed using threads in increments of powers of two (1, 2, 4, 8, etc.). The program itself generates 1000 IO requests, totaling 53,232,819 microseconds (53.233 seconds). The longest IO request is 2,975,927 microseconds (2.976 seconds), and the shortest is 0 microseconds.

Data Collection

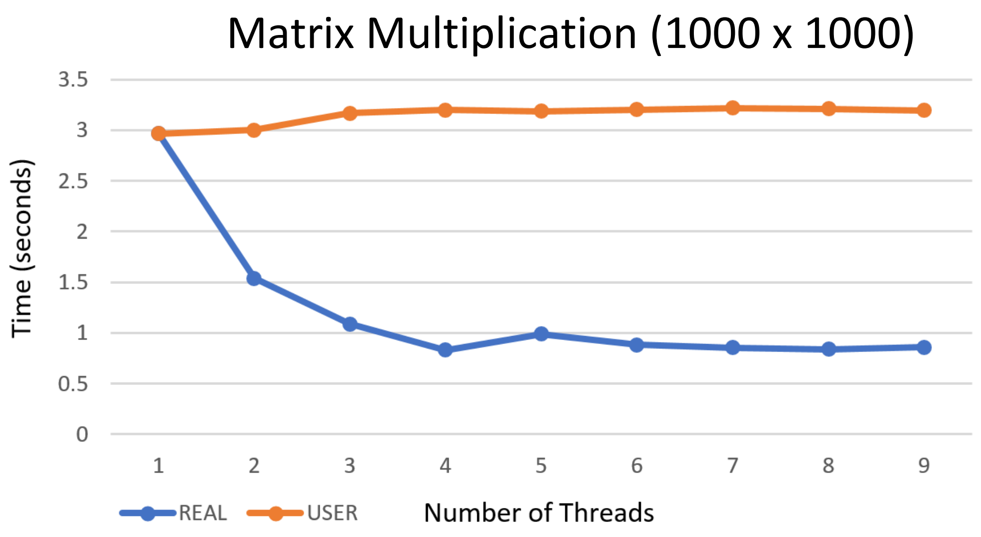
Each of the two program was compiled on the respective system just before beginning each series of tests. There were no other users on the systems while these tests were being run.

Time presented in the graphs comes from timing the processes via the Linux `time` command. Remember, real time is the “wall clock” time. This is the amount of real-world time that passes from start to end of execution. User time is the amount of time that the operating system said that the CPU took. In other words, this is the sum of time that the threads supposedly spent running.

CPU Bound Threads

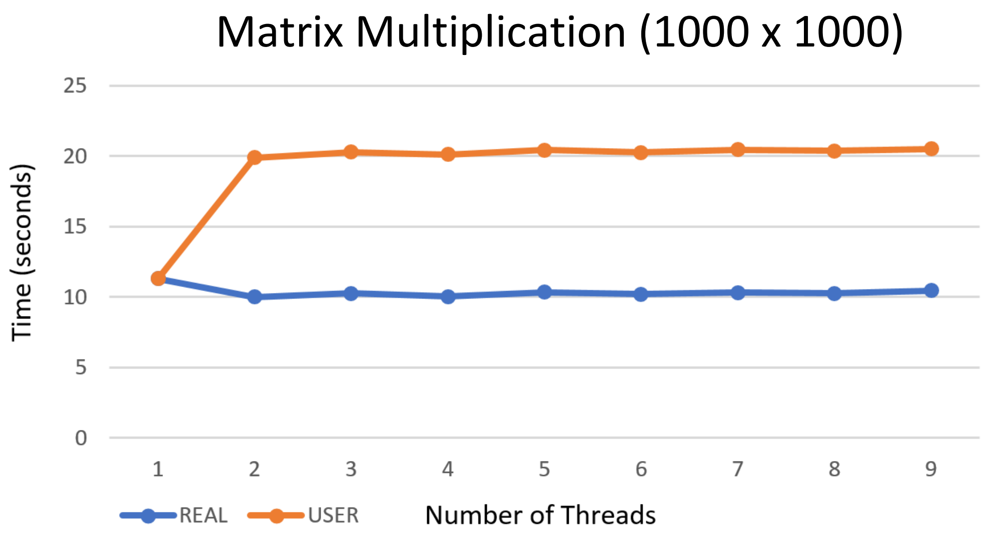
Data was obtained for the matrix multiplication program by running the program on four different systems. Each data point represents a series of 5 tests using the same system and same number of threads. The numbers shown on the chart are the median result of each test series. The horizontal axis represents the number of threads used in each test. The vertical axis represents the time each test took in seconds.

a. System 1: Intel® Core™ i5-3570 CPU, 3.4 GHz, 4 processors, non-hyper-threaded



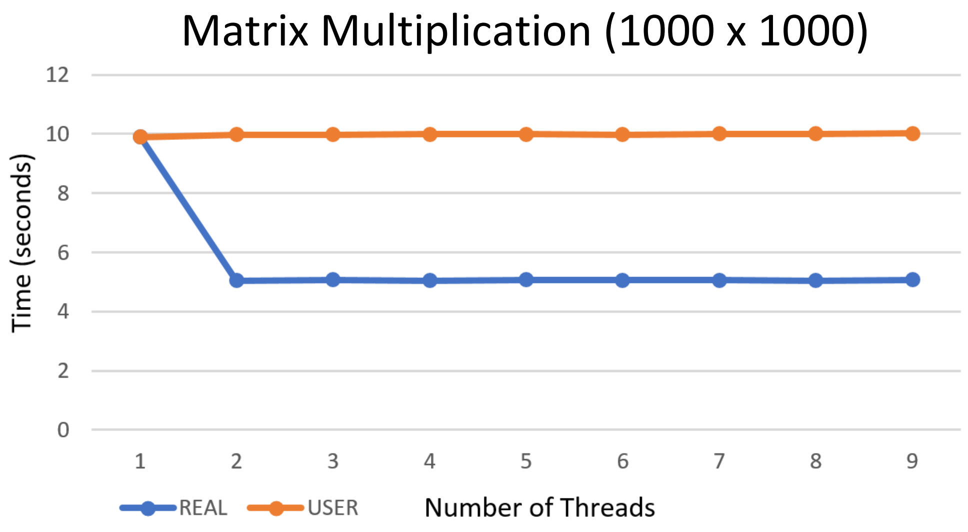
The fastest recorded time was with 4 threads (0.831 seconds). The overhead of using the threads is noticeable at about 3 threads (the overhead averaged to about 0.23 second).

b. System 2: Intel® Pentium® 4 CPU, 3.4 GHz, 1 processor, hyper-threaded



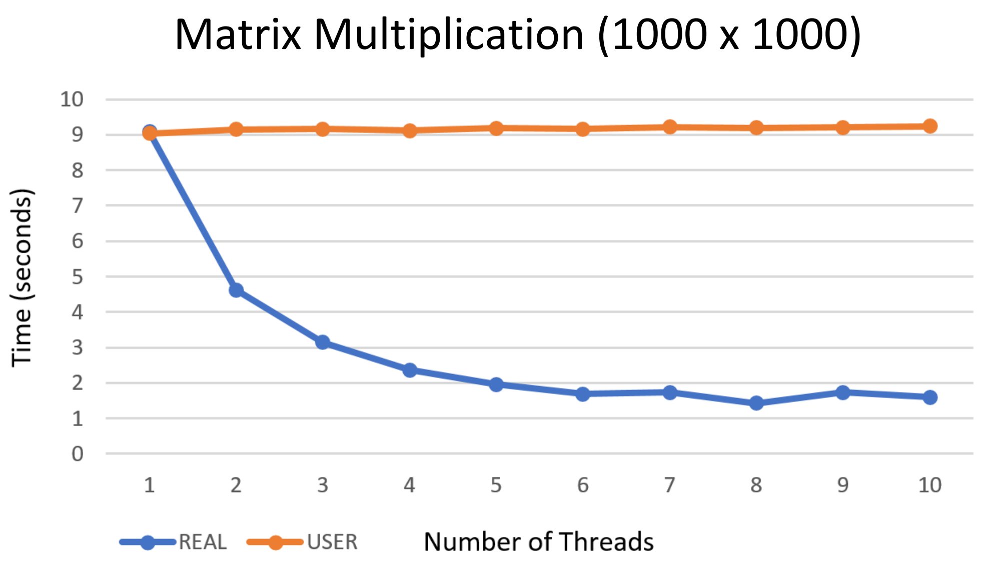
The fastest recorded time was with 2 threads (10.007 seconds). Note how much more time is required by the processor (user time) to complete the program. The increase is not proportional.

c. System 3: Intel® Core™ 2 6400 CPU, 2.13 GHz, 2 processors, non-hyper-threaded



The fasted recorded time was with 2 threads (5.040 seconds). The recorded user time remains practically constant regardless of the number of threads used.

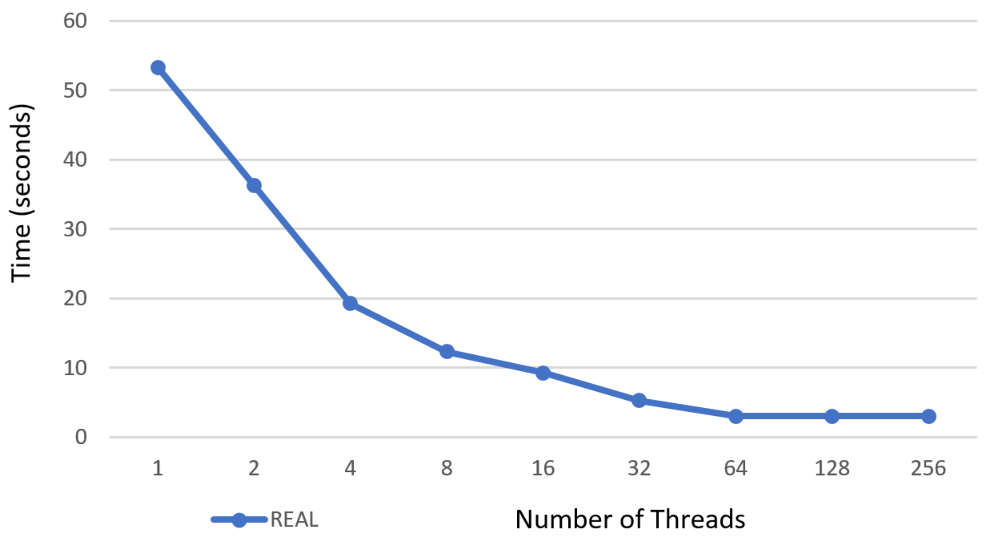
d. **System 4: Intel® Xeon® E5345 CPU, 2.3 GHz, 8 processors, non-hyper-threaded**



The fastest recorded time was with 8 threads (1.424 seconds). The recorded user time remains practically constant regardless of the number of threads used.

IO Bound Threads

The IO program was run on each system, but all systems performed approximately the same. The data in the chart below was collected from the Intel® Core™ i5-3570 CPU, 3.4 GHz, 4 processors, non-hyper-threaded system. The horizontal axis represents the number of threads used in each test. The vertical axis represents the time each test took in seconds.



After about 64 threads, the increase in speed was nominal, regardless of the system used. The lowest recorded time was 2.981 seconds.

Rules of Thumb

Graphing the data makes it easy to spot a couple of curious patterns. First, **for CPU bound threads, the optimal number of threads is the number of processors on the system**. The exception for this is systems with only one processor which are hyperthreaded. There is a slight performance boost when using two threads. Second, **for IO bound processes, increasing the number of threads will increase performance until all IO processes can be completed in the time that it takes the longest IO process to finish**.

Conclusion

The temptation to go into a program “threads blazing” results in unnecessary overhead and concerned looks from colleagues. You can save yourself a lot of trouble by understanding what kind of system you are working on, the nature of your application (computationally heavy vs. IO heavy), and the size of the workload (sometimes the overhead caused by the threads just aren’t worth it!). A bit of experimentation with your application and a handful of knowledge can aid you tremendously your quest for computational efficiency.