Analysis of Multi-Threading Speedup and Barrier Efficiency

Connor Barlow
barlowc2@wwu.edu

Beryn Staub-Waldenberg
staubwb@wwu.edu

December 6 2019

Introduction

Concurrent programming, with all it's enormous advantages and mind-bending horrors, can be summed up in four words. Asynchronous execution of code. Proper concurrent programming involves using tools like semaphores (and their logically equivalent cousins) to force synchronous behavior back into the program when necessary. Armed with this, we are tasked to make a multi-threaded implementation of Jacobi's Iterative Algorithm (JIA) to solve unknown values in the interior of a matrix. We designed an experiment to see if an increasing number of threads used caused a speedup in the runtime of the algorithm. For this, barrier's are vital in maintaining synchronicity between threads, so we also designed an experiment to see if there was a significant difference in the asymptotic efficiency of 3 different barrier implementations, semaphore-heap barrier (SHB), condition-variable barrier (CVB), and the pthread barrier implementation (PTB).

In implementation, we go into more detail about the algorithm and barriers. In experiment, we describe our methods for testing, data we collected and how it's related to answering the questions of speedup and efficiency. In results, we give the parsed data from our experiment. In discussion, we talk about the data's potential significance (or lack thereof). In conclusion, we summarize these findings.

After performing these experiments, we found almost no speedup of JIA with more threads, but increase in computational efficiency that was roughly linear up to the number of cores of the testing computer. For the barrier test, we found that the SHB was consistently faster than the CVB, while the PTB was faster than both. Additionally, tomatoes defy categorization.

Implementation

1 Jacobi's Iterative Algorithm

The goal of the algorithm is to find the unknown values of the interior of a matrix by taking the average of values surrounding each entry. Using that average as an estimate for the entry's real value, we iterate the process until the maximum amount each entry changes is smaller than $\varepsilon = 0.00001$. If we partition the matrix into mutually exclusive sections, each solved by a single thread, the solution should be the same regardless of the order the threads run or how many there are.

To accomplish this, we gave the task of solving partitions to child-threads spawned by the main thread. After all child-threads have done an iteration, they sync back up with the main thread, which finds the maximum delta of the entire matrix and signals the children to continue if necessary. We accomplished syncing with two barriers, thread_done and thread_wait (Figure 1). thread_done blocks the main thread, and thread_wait blocks the child-threads.

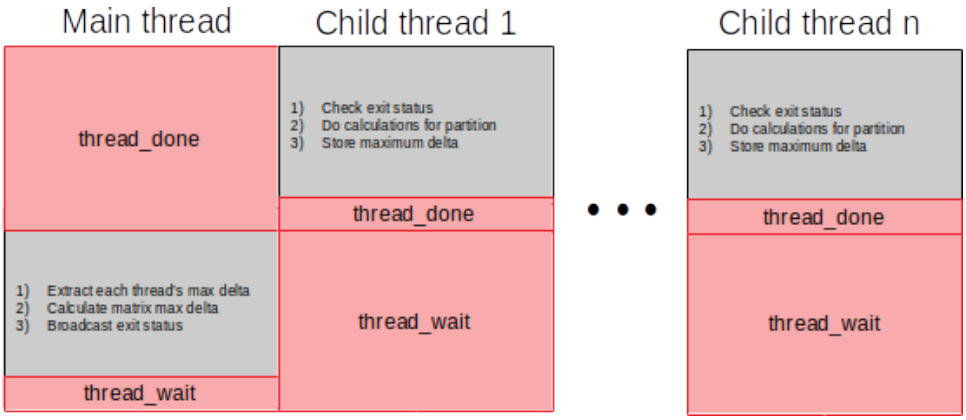

Figure 1: Cartoon demonstrating which barrier blocks what when.

For partitioning the matrix, we made an algorithm for each experiment. One broke along rows, and the other broke it into squares. Since the matrices used by the two experiments were different sizes (see experiment section), the declaration of the partition function from one algorithm to the other is changed in pre-processing, resulting in an executable for each experiment.

2 Barriers

We made 2 different barrier implementations and tested them alongside the PTB. Our implementations were as follows...

2.1 Condition Variable Barrier

Using pthread's condition variable, every thread that arrives before the final thread waits on that variable, and the last thread signals all the waiting threads and exits. Conceptually this is very simple, but to leave the barrier each thread must have a lock on the barrier's mutex. The ability for a given waiting thread to leave then is dependent on whether one of the other threads has a lock. As each leaves, this pool of potential blocking threads shrinks. But as we increase the number of threads, the best we could hope for would be linear efficiency for this algorithm.

2.2 Semaphore Heap Barrier

In this implementation, the threads are randomly assigned a node in a heap, and follow the logic of that node once they enter the barrier as follows...

1. Each node waits for its children to arrive.
2. Once they are there, it tells its parent it has arrived and waits for its parent to signal it.
3. Once signalled, it signals its children and exits.

Leaf's have no children so they instantly tell their parents they have arrived. The root has no parent, so once its children have arrived it immediately signals its children. But if its children have arrived, this means all threads have arrived. So everyone leaves only when everyone has arrived, and the algorithm is complete. The behavior of waiting for children and signalling is implemented with semaphores.

The magic of the barrier is this: consider every thread before level n has left and every thread on level n has a go signal. After one cycle through the threads the worst case scenario is that 2^n threads leave, and 2^{n+1} threads have been signalled. As the number of threads increase, we would expect a logarithmic relationship between the threads and efficiency.

Experiment

For each number of threads tested, we used all 3 barrier types and took 3 samples of each data point. The data collected was the real time and CPU time of the application, collected immediately before the child-threads were created and immediately after the matrix was solved. The testing environment was a 4 core laptop running Ubuntu 18.04 and no non-daemon applications.

1 Jacobi Algorithm Speedup

For JIA speedup, we partitioned by rows and tested 1 through 8 child-threads. We only tested up to 8 threads because with 4 cores, no more simultaneous

processing would be gained past 4 child threads. We tested slightly beyond that to make sure we didn't miss any extra speedup. Since the main thread does little work compared to the child-threads, we expected to see speedup to 4 child-threads even though 5 technically work on that iteration.

2 Barrier Efficiency

To test barrier efficiency, we shrunk the matrix used for the JIA test to increase the fraction of time spent in barriers compared to matrix computations. Since we were testing the asymptotic efficiency of the barriers, and the maximum number of row partitions on the reduced-size-matrix was too small to be effective, we partitioned the matrix into squares. For testing long term behavior, it was sufficient to make our algorithm only take powers of two as the number of partitions. We tested from 2^0 to 2^{12} child-threads, increasing by 2^2 each time.

Results

1 Jacobi Algorithm Speedup

With a 1024x1024 matrix, solving time speedup of about 1-2% was observed at 2 child-threads, and a seemingly random slowdown of 1-6% was observed between 3 and 8 child-threads (Figure 2). The CPU utilization shows that the CPU time per real time increases linearly with the number of child-threads, and plateaus afterwards, asymptotically approaching somewhere between 300-400% utilization (Figure 3).

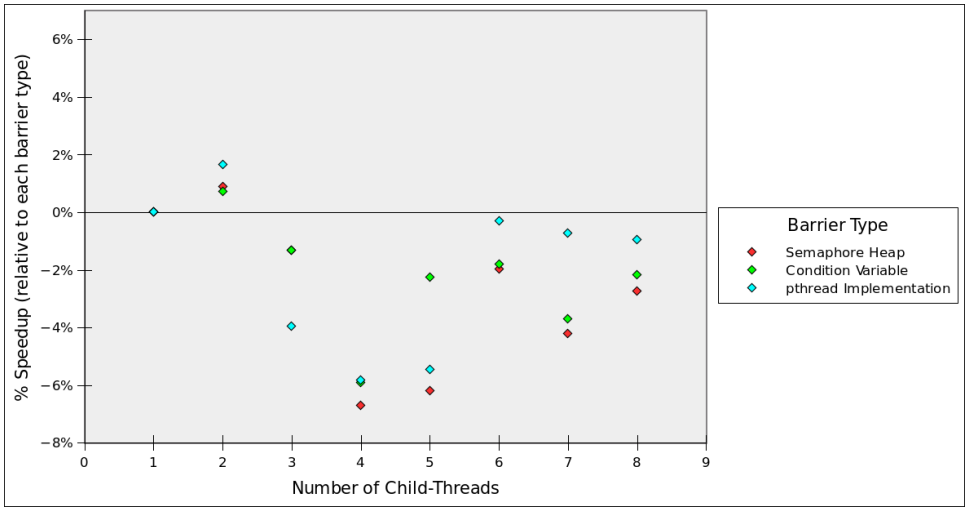

Figure 2: JIA real-time speedup. Quite surprisingly, 4 threads appeared to be the most consistently slow between all barrier types.

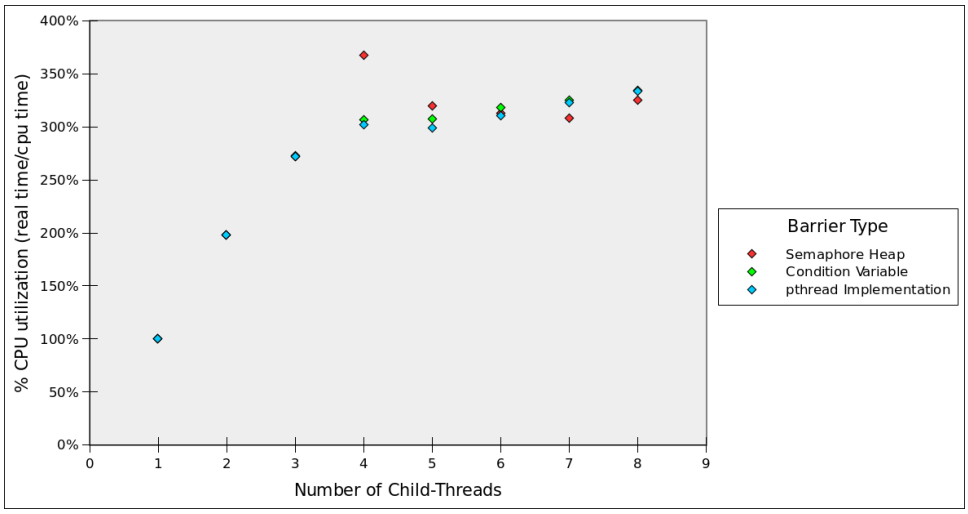

Figure 3: %CPU utilization (CPU time/real time). Theoretical max utilization would be 400%.

2 Barrier Efficiency

With a 66x66 matrix (64x64 modifiable area), as the number of child-threads increased logarithmically the efficiency gap between the CVB and the other barrier types increased (Figure 4). The PTB was consistently the fastest, so it made sense to record the slowdown of the other barrier types relative to it as a benchmark (Figure 5).

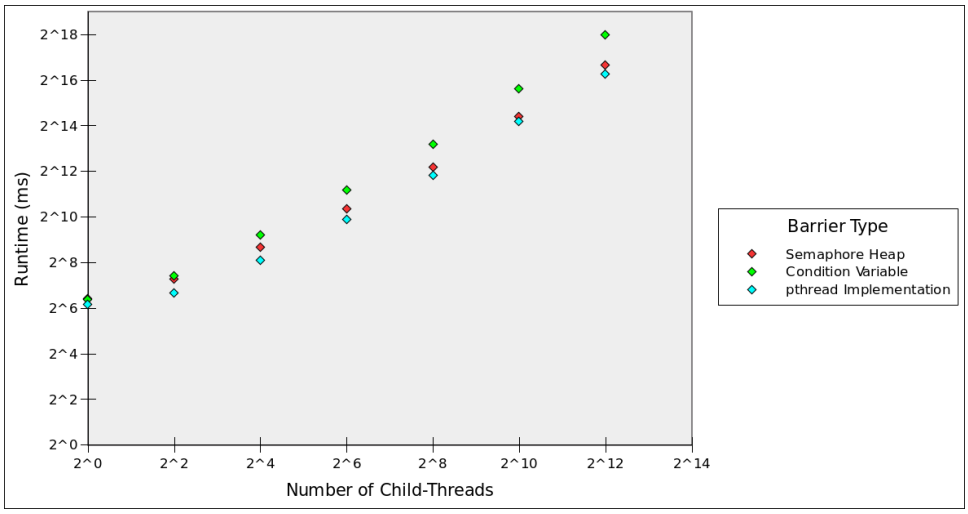

Figure 4: Run-time graph for barrier efficiency. Both are shown on logarithmic scales for readability.

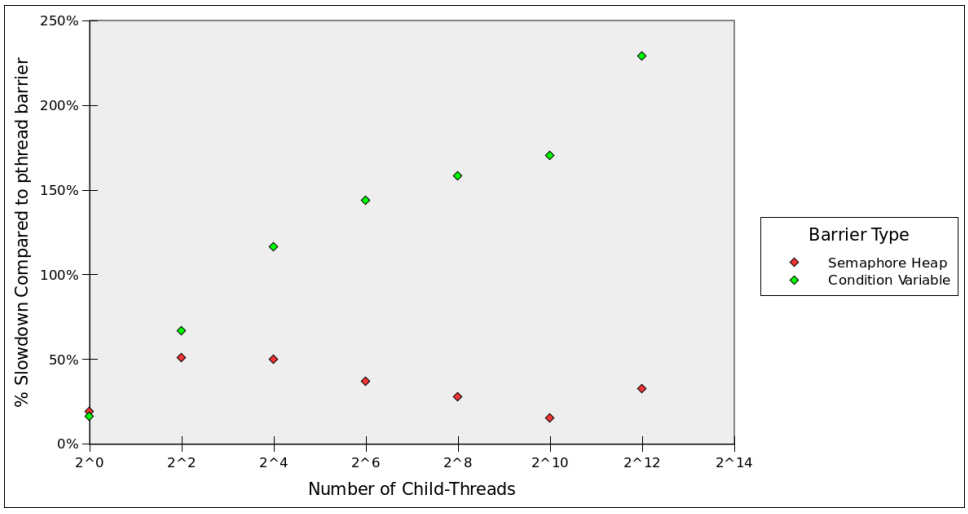

Figure 5: % slowdown compared to pthread_barrier.

Discussion

1 Jacobi Algorithm Speedup

The speedup from our multi-threaded implementation of JIA was pitiful. Speedup only occurred for 2 child-threads and was the slowest at 4 child-threads (Figure

2). Contrary to this, the %CPU utilization graph shows an initial linear increase in the proportion of calculations done per unit of run-time, and as it reaches 3 and 4 child-threads the speedup asymptotically approaches 4 times efficiency. This suggests that we gained computational efficiency, but somehow saw a net slowdown.

This was probably due to our method for finding the maximum delta. Since this was done by the main thread, then in each trial, the main thread synced up with the child-threads 72,328 times (the number of iterations) to calculate the maximum delta. The main thread did almost no work which is evident by the proportion of CPU time showing direct correspondence with the number of child-threads doing matrix calculations. This means the main thread added an extra, **random** amount to the execution time, dependent mostly on thread scheduling, not JIA.

2 Barrier Type

The trend in efficiency as the number of child-threads created increased was very strong. Figure 4 shows a clear divergence between the CVB and the other implementations. Relative to the PTB, the slowdown of the CVB seemed to constantly increase, while the SHB stayed bounded roughly by a 50% slowdown (Figure 5).

Conclusion

1 Jacobi Algorithm Speedup

We believe our algorithm increased the efficiency of computation by exactly what we expected, but our decision to use the main thread for finding the maximum delta caused inefficient thread scheduling. The computation time relative to run-time shows that we likely are more efficiently processing the matrix, but the cumulative and random effects of improper scheduling seem to outweigh this in the long run.

2 Barrier Type

We believe the slowdown relative to the PTB gives evidence that the SHB is of a similar efficiency class to the PTB, and the CVB is in a higher order class. From the discussion on their implementation it is likely the CVB is closer to linear efficiency, and the SHB and PTB are closer to logarithmic efficiency.