Matrix Multiplication
Single Threaded vs. Multi-Threaded

Thomas Mease

3-8-2018

Number of logical processors on laptop: 4

1 Algorithms Used

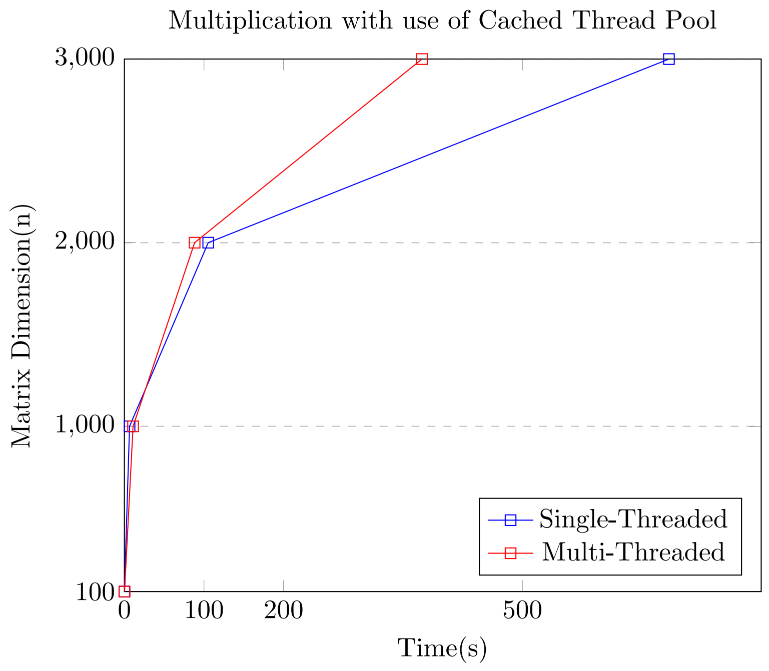
The single-threaded algorithm used can be found on the Matrix Multiplication Algorithm wikipedia page. The multi-threaded algorithm is as follows:

```
input  : Two randomly generated matrices, $a$ of size $n$x$m$ and $b$ of size $m$x$p$
output: The product of the two matrices

Create matrix $c$ of size $n$x$p$
Create list of threads
for $i \leftarrow 0$ to $n$ do
    for $j \leftarrow 0$ to $p$ do
        Create thread of MyMatrixMultiply with $a, b, c, i, j$
        Add thread to list
    end
end
Create ExecutorService executor
for $l \leftarrow 0$ to size of thread list do
    Run thread at $l$
end

Run:
$c[i][j]$ equals the DotProduct$(a, b, i, j)$

DotProduct:
sum $\leftarrow 0$
for $k \leftarrow 0$ to $m$ do
    sum$+ = a[i][k] * b[k][j]$
end
Return sum

Wait for threads to complete
Return $c$
```

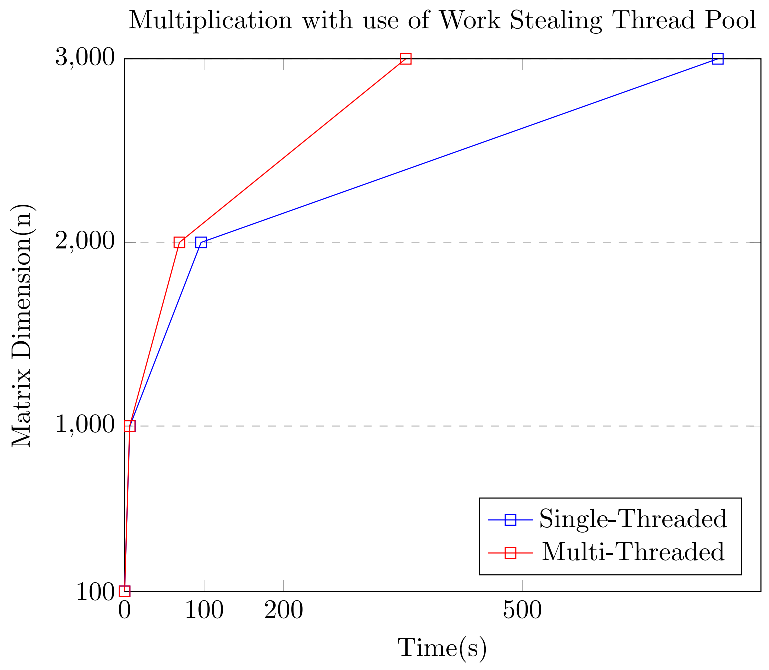
2 Exprimental Results

For the experimental testing, I used square matrices and ran the program three times for each size(100, 1000, 2000, 3000). I then graphed the average time vs. the size. All experiments were run on an AMD A10-9620P Radeon R5, 2.50GHz processor with minimal programs running to ensure that the program could run with as many resources as possible.

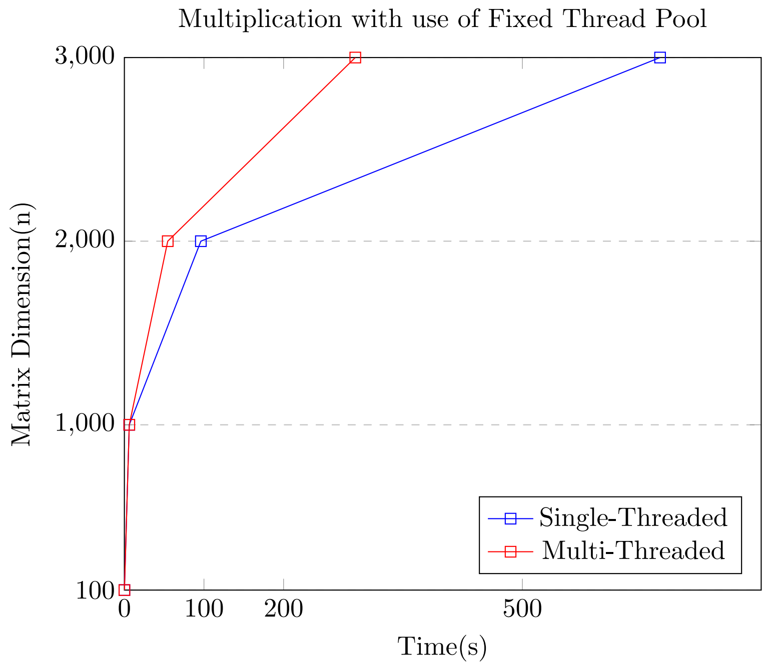
1. **Cached Thread Pool**



2. **Work Stealing Thread Pool**



3. **Fixed Thread Pool**



Out of the three different types of thread pools, I found that fixed thread pool worked the fastest. All three types yielded minimally slower results for 100x100 matrices, which was expected due to the smaller size. For 1000x1000, the only thread pool that was quicker than a single thread was the Work Stealing pool. As the matrix size jumped up to 2000x2000, an improvement in speed could be seen, with all three types yielding $20 - 40$ seconds faster results than the single thread. In each case, the 3000x3000 multiplication yielded $\sim$ 2x as fast as the single threaded.

3 Challenges and Solutions

The only challenge that I really encountered was my first version of the multi-threaded multiplication was running slower than the single-threaded multiplication. Initially, I just generated and executed the threads inside the double for-loop. In order to speed up the process, I instead generated the threads and added them to a list. Then after all the threads were generated, I executed them all in a for-loop.

I assume that this worked because of the strength of my system and processors. With the first iteration, I was executing the threads right away, so I probably lost some time waiting for threads to finish executing before be able to add a new one to the thread pool. With the version that was faster, I instead generated all the threads, then sent them all into the queue at once, allowing for much quicker execution