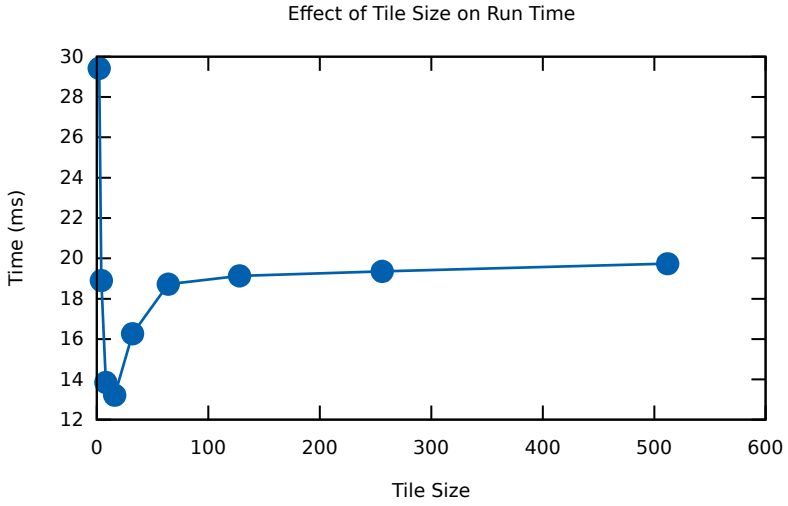
Nathan Nard
CS 484 : Parallel Programming -- Machine Problem 1
University of Illinois Urbana-Champaign
20190214

1. Matrix Transpose

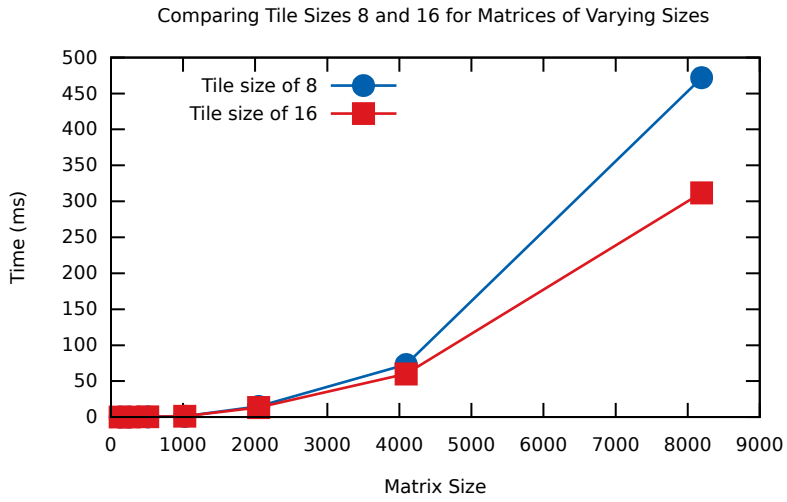
1.1 Identifying Optimal Tile Size



The machine's L1 cache has a size of 32768 B with a cache line size of 64 B, for a total of 512 cache lines, and is 8-way set associative. As our 2D matrix is composed of double precision floating points of size 8 B, a total of 8 'words' can fit on a cache line. In order to maximize cache hits, we want a tile size that is a multiple of 8 'words'. The empirical data appear to reflect this notion, with sizes 8 and above generally giving better results, particularly sizes 8 and 16, though 16 is best overall. Tile size of 2 does not do well as each new tile of data will have 2 cache misses and 2 cache hits. Compare this to tile size of 8's, which has one cache miss for each line for a total of 8 misses, but all other cells will be cache hits, 8 * 8 – 8 = 56 total hits. Thus we see a much better run time for tile size 8 compared to tile size 2 since 8 has a hit ratio of 0.875 while 2 has a hit ratio of 0.5.

Due to the cache line's size, we can see that there's a minimum of 2048 * 2048 / 8 = 524288 cache misses. This minimum can be reached with tile sizes 8 and above. However, while these sizes do better than size 2, sizes 8 and 16 clearly do better compared to 32 and above. It may have something to do with the 8-way set associativity of the L1 cache. For a tile size of 8, it is possible to fit all of the cells into one set. A set of 8 cache lines can hold 8 words each for a total of 64 words and a 8 by 8 tile has a total of 8 * 8 = 64 words. Larger tile sizes would require using more sets of cache lines, which would be accessed inconsistently. Perhaps there's some computational advantage in keeping tile sizes near the size of the cache line sets (More predictability?). Overall though it appears a tile size of 16 is optimal, especially when considering the graph in the next section. As to why 16 is empirically better than 8, I am unsure.

1.2 Varying Matrix Size



Since tile sizes 8 and 16 appeared really close to each other in performance, I benchmarked both of them for varying matrix sizes. As you can see, 16 is clearly the winner as its advantage becomes more apparent in much larger matrix sizes.

The differences in performance for the first few matrix sizes are hard to see due to the scale the last two sizes reach, but they get worse steadily as matrix size increases. It's not until a size of 2048 do we see a noticeable bump on the graph. This is the matrix size in which the number of Bytes exceeds the collective sizes of the three cache levels. Compare 32768 + 262144 + 26214400 = 26,509,312 total bytes for all three caches to 2048 * 2048 * 8 = 33,554,432 total bytes in the matrix. As we get to much larger matrix sizes the more main memory accesses will be required thus causing the steep climb in run times we see for the matrix sizes above N = 2048.

2. Matrix Multiplication

Here are the averages of 8 trials for matrix multiplication for matrices of size 512 :

```
Average execution time of multiply_basic(): 0.364976
Average execution time of multiply_tiled(): 0.205442
Average execution time of multiply_transposed(): 0.126153
```

Note, the time units are the same as those returned by the get_time() method. (seconds?) Clearly both tiled and transposed methods are better than the naive basic method. Comparing the tiled method to the transposed method, clearly the transposed method is better. While the time measurement for the transposed method does not include the execution time of transposing matrix B, the time difference is very small in comparison to those shown above (looking at other transpose test times on 512 sized matrix). So having ruled out that potential oversight, we now look at the two different implementations. The tiled method has 6 nested for loops while the transposed method has only 3 nested for loops, so that will help contribute to its better performance. In terms of cache hits/misses, both methods should have the same number of cache misses: N*N/w. But with the transposed method the elements are all accessed sequentially in the array, so there may be some predictability that the hardware is taking advantage of.

3. Basic OpenMP

Here are the benchmarks for the openmp directives being applied to generating squares and testing primes for an array of size 1024:

```
benchmarking squares_serial()...
squares_serial() completed in 0.000000 seconds
benchmarking squares_parallel() with 4 threads...
squares_parallel() completed in 0.000002 seconds
benchmarking primes_serial()...
primes_serial() completed in 0.000137 seconds
benchmarking primes_parallel() with 4 threads...
primes_parallel() completed in 0.000053 seconds
```

For the squares, there's no discernible difference at this displayed scale of time, though it does appear that the parallelized method is performing worse. Squaring numbers is computationally cheap and openmp's parallelizing methods have overhead due to splits, waits, and joins of threads. It appears that this overhead is creating a bottleneck to the squares method, so it looks like openmp may not be beneficial for computationally cheap tasks like computing squares. However testing whether a series of numbers are prime is a lot more computationally heavy, and as we can see there was some benefit in parallelizing the prime tests. Thus openmp appears to be great only for parallelizing relatively complex procedures. However the difference in performance for the squares was pretty small, so there may not be much to lose in attempting to naively parallelizing tasks.