Justin Bahr
NUID: 001592707

High Performance Computing
Extra Credit Homework 1

Explanation of results for mat-mat-mul-openMP.c:
In this C file. The first round of matrix multiplication is done serially without loop blocking, the second round is done in parallel without loop blocking, and the third is done in parallel with loop blocking. Only the third round of matrix multiplication obtains a speedup. The reason for this is because matrix multiplication is memory bound rather than compute bound. Each row of the first matrix must operate with the entire second matrix, which is far too large to store in the cache. Even though arithmetic operations can be done concurrently for the second round, variables are displaced in the cache, then pulled back in repeatedly, which is orders of magnitude slower than any execution speedup. The multiple threads cannot execute on operands before they are brought to the register, so no speedup can be obtained. In fact, multiple threads will be competing for the same cache blocks, which could worsen trashing, even slowing the program down. In the third round, however, effective loop blocking chunks the work into slices that can fit in the cache, meaning variables will stay in the cache while they are being operated on and will only be replaced when the thread moves to the next chunk. This is especially effective for utilizing the L1 and L2 cache, which are core-specific. Threads can store their respective chunk in the L1 or L2 cache and ignore data not included in their chunk, leading to much faster memory latency. To prove this, I wrote a few C++ programs and performed two experiments. The first experiment compares small and large matrix behavior, and the second examines cache faults/sec from the performance monitor. All

Experiment 1:
Based on the C file discussed above, I altered the process for C++ and added a second run through using a smaller 32x32 matrix, which could easily fit entirely in the cache, eliminating the trashing issue. For the large matrices, using OpenMP alone did not provide a speedup, but OpenMP plus loop tiling gave a speedup of 1.93. For the small matrices, OpenMP alone provided a speedup of 1.18, and OpenMP plus loop tiling gave a speedup of 1.43. This demonstrates that when the cache is large enough to eliminate the need for replacement, OpenMP alone can provide a slight speedup for matrix multiplication, more similar to when combined with loop blocking. Based on that result, I can conclude that the large volume of cache misses and memory latency bottleneck are the reason for the results seen in mat-mat-mul-openMP.c. A screenshot of my results is shown below:

```
[jbahr@rho Loop Blocking Extra Credit]$ ./MM_openMP
Check c[511][511] = 1024
Check c[511][511] = 1024
Check c[511][511] = 1024
Check c1[15][15] = 64
Check c1[15][15] = 64
Check c1[15][15] = 64

Time to execute large, single-threaded: 786136650 ns
Time to execute large, multi-threaded: 788099824 ns
Time to execute large, multi-threaded, with loop blocking: 406506721 ns
Large matrix parallel speedup: 0.997509
Large matrix combined speedup: 1.93388

Time to execute small, single-threaded: 119884 ns
Time to execute small, multi-threaded: 101604 ns
Time to execute small, multi-threaded, with loop blocking: 83765 ns
Small matrix parallel speedup: 1.17991
Small matrix combined speedup: 1.43119
```

Figure 1. Experiment 1 Results

Experiment 2:
I wrote a blocked, non-blocked, and non-blocked OpenMP parallel version with an even larger 1024x1024 matrix. Not only did both non-blocked versions take much longer to run (about 26x as long), the parallel blocked version took even longer than the single thread trial. Runtimes are shown below:

```
PS C:\Users\justb\EECE5640_HighPerformanceComputing\Loop Blocking Extra Credit> ./MM_block
Check c[511][511] = 2048
Time to execute with loop blocking: 2330062100 ns
PS C:\Users\justb\EECE5640_HighPerformanceComputing\Loop Blocking Extra Credit> ./MM_no_block
Check c[511][511] = 4096
Time to execute without loop blocking: 61152254300 ns
PS C:\Users\justb\EECE5640_HighPerformanceComputing\Loop Blocking Extra Credit> g++ -o MM_no_block_ope
PS C:\Users\justb\EECE5640_HighPerformanceComputing\Loop Blocking Extra Credit> ./MM_no_block_openMP
Check c[511][511] = 4096
Time to execute without loop blocking: 62261327500 ns
PS C:\Users\justb\EECE5640_HighPerformanceComputing\Loop Blocking Extra Credit> |
```

Figure 2. Experiment 2 Results

Additionally, I ran all executables on my local machine with other applications closed and the performance monitor opened. The blocked version maintained a low Cache Faults/sec reading, while the other two non-blocked applications (regardless of parallelization) had high spikes over 1,400 Cache Faults/sec. The results are seen below:

\\DESKTOP-NKRFL4C	
Memory	
% Committed Bytes In Use	48.873
Available MBytes	18,585.000
Cache Faults/sec	1.004

Figure 3. Cache Faults with Blocking

\\\\DESKTOP-NKRFL4C

Memory

% Committed Bytes In Use	48.585
Available MBytes	18,670.000
Cache Faults/sec	1,443.861

\\\\DESKTOP-NKRFL4C

Memory

% Committed Bytes In Use	48.835
Available MBytes	18,604.000
Cache Faults/sec	341.531

Figure 4. Cache Faults without Blocking (serial)

\\\\DESKTOP-NKRFL4C

Memory

% Committed Bytes In Use	48.767
Available MBytes	18,782.000
Cache Faults/sec	1,785.568

\\\\DESKTOP-NKRFL4C

Memory

% Committed Bytes In Use	48.991
Available MBytes	18,738.000
Cache Faults/sec	270.766

Figure 5. Cache Faults without Blocking (Parallel)