CMSE 401 - Homework 2

Jack Hamel

February 1, 2019

1 Implementation

My matrix transpose code is written in C++ and uses only standard C++ libraries. I wrote three different ways to compute the transpose, but only submitted the fastest one. My first attempt used C-style arrays, but I ran into memory issues so I abandoned this approach and switched to C++ vectors. My submitted version of the code is the most direct approach to taking the transpose. I attempted to make it fast by reducing the size of my loops by half and doing two copy operations per loop iteration, but it actually increased the run time of the transpose. The obvious way I can foresee getting more speed ups to my transpose algorithm would be to implement a blocked matrix approach like in part 2 or the cache-oblivious approach like in part 3. It may be possible to see a speedup also by storing my matrix as a 1-D array rather than a 2-D one.

2 Questions

Would it help to block separately for each level of cache? Why or why not?

Yes. Ideally, each block would use all of the memory of one cache at a time to masximize effieciency. This would minimize the number of loading operations of blocks into the cache.

In this exercise you used square blocks. Would rectangular blocks be better? If so, what shape and why? How would you test this?

I think the answer to this question depends on the storage layout of the matrices in memory. My understanding of the reason we use blocks to solve this transpose problem is that the layout of the original and transpose matrices are in such a way that blocks of the matrices have more locality to each other in memeory than when moving through the entire matrices at once. With this said, if the matrices are stored one-dimensionally and in parallel to each other, then I think blocking the matrix in n-length (one-dimensional) strips might be better because the strides to access elements of each strip will be smaller.

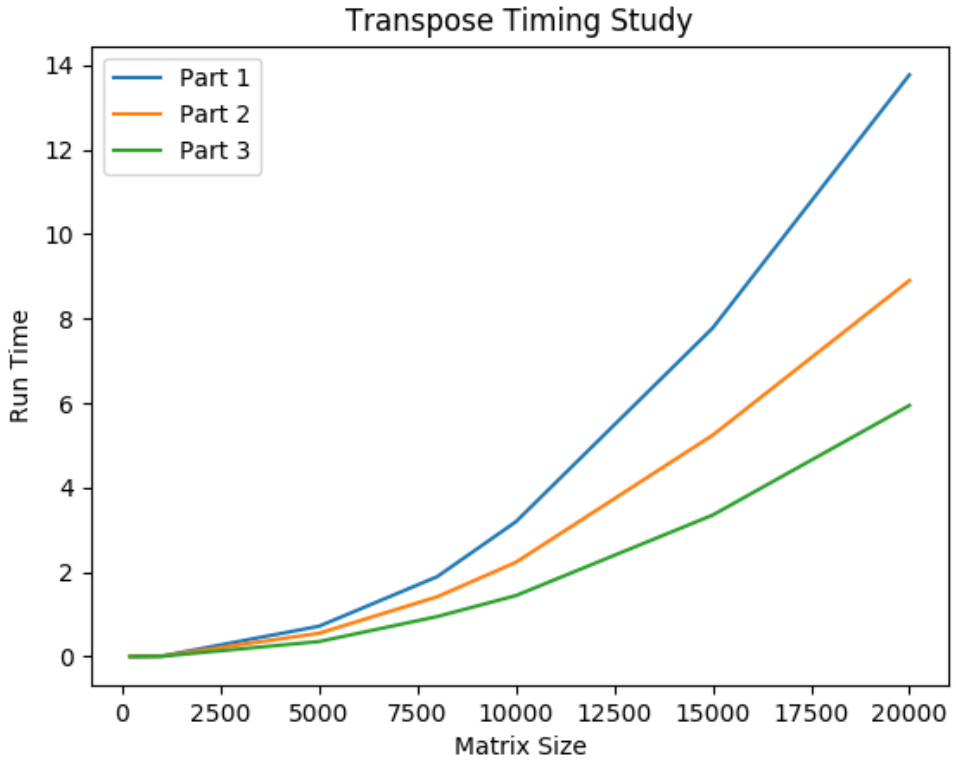
This could be tested by using my code from part 2 and creating 2 separate block dimensions; one for rows and one for columns. Then I would loop through different combinations of dimensions and time the transpose duration for each. I could then see which block size is most efficient.

Can you use the cache oblivious performance model to estimate the matrix size where the transpose performance changes abruptly?

To answer this question I found the second derivative of the run time vs. matrix size curve. Where the magnitude of this derivative is maximized, I interpreted it as the most abrupt change in performance. This occurs when matrix size is 10000.

3 Timing Study

The study was done by running each part of the assignment three times for each matrix size and averaging the times. The plot below shows how each algorithm scales with matrix size. The second figure shows a table of the exact times. The running of the codes and generating of the plot was all done using a bash script submitted to the HPC called `submission.sh`.



Matrix Size	Basic	Blocked	Cache Oblivious
200	0	0	0
1000	0.01	0.01	0.004
5000	0.717	0.55	0.354
8000	1.89	1.413	0.945
10000	3.19	2.23	1.442
15000	7.777	5.24	3.348
20000	13.773	8.903	5.944
40000	97.213	40.143	n/a

Figure 1: Run time per matrix size measured in seconds

The cache oblivious algorithm clearly performed fastest for all matrix sizes up to 20,000 and likely for 40,000 as well as made evident by the scaling in the plot above. When attempting to run the cache oblivious code with a matrix size of 40,000, I continually had failures on the HPC and I have conceded defeat in obtaining that single data point.

The Computer Specifications used during this study are as follows:
Name: dev-intel16
Processor: Intel Xeon CPU E5-2680 v4 @ 3.00GHz (1 core used)
Memory: 40 GB
OS: Linux 64-bit

4 Instructions

My code is written in C++ and because it does not depend on any external libraries, should be compilable using GCC without any additional includes. To compile part 1 use: `g++ part1.cpp`; to run part 1: `./a.out MatSize`. If no MatSize is given, then the default size is 4000. To compile part 2: `g++ part2.cpp`; execution is the same as part 1. You wrote part 3 so I hope you can compile it (hint: if not just do the same as with the other two files) :)