Grant Kinsley (gkinsley)
Ethan Duong (duonget)

231P: HW 4

The characterization of the computation progress in a single processor computer, for each of the six mentioned permutations. Add figures as necessary to support your explanation.

At every step in our triply nested for loop, we have that $C_{i,j} \leftarrow C_{i,j} + A_{i,k} * B_{k,j}$ where A, B, and C are 2D matrices (integer arrays organized in row-major order). This means that if integers in a 1000x1000 matrix A (assume size of integers are 4 bytes), $A_{i,k}$ and $A_{i,k+1}$ are four bytes away while $A_{i,k}$ and $A_{i+1,k}$ are four thousand bytes away. This difference and spacing can impact program performance as the system stores sequential blocks of memory in the cache. Generally speaking, it is better to read memory sequentially instead of jumping to best make use of the cache.

(i, j, k) implies that the outermost loop is i and the innermost loop is k.

1. (i, j, k): In this example, k is the fastest, meaning that reading matrix B could bottleneck our program due to cache misses (we jump one row at a time in B ignoring sequential memory)
2. (i, k, j): This example reads and writes memory sequentially making better use of the cache. A is kept static while B and C are read sequentially.
3. (j, i, k): Suffers from the same issue as (i, j, k). Matrix B jumps one row at a time
4. (j, k, i): Matrix A and C both jump one row at a time causing slowdowns.
5. (k, i, j): Memory from C and B are read sequentially while A remains static (in the innermost loop). If for some reason the innermost loop is very small (i.e. B is a very tall matrix), then we will have many read-jumps in matrix A.
6. (k, j, i): Matrix C and A have many jumps in the innermost loop.

Thus, the ordering that makes the best use of the cache is (i, k, j).

Discussion of the mapping of the problem to a n-processors Ring. Add figures as necessary to support your explanation.

We tackle the same problem as before, this time utilizing n processors to compute the multiplication of an n x n matrix. We again seek to maximize cache hits and minimize cache misses.

We parallelize the outermost loop meaning if our order is (i, j, k), every process alpha in {0,...,n-1} calculates (alpha, j, k). Intuitively, this means every process only runs the innermost loops.

$C_{i,j} \leftarrow C_{i,j} + A_{i,k} * B_{k,j}$

We assume every processor gets its own L1 cache meaning that processors generally do not have to compete for cache space. Matrices are stored in the heap and all processors must access the shared heap to access the matrices (assuming there is cache miss).

1. (i, j, k): We hold i constant for each process. We still have many cache misses in B moving k the fastest.
2. (i, k, j): We hold i constant. C, A, and, B are all read sequentially maximizing cache hits
3. (j, i, k): In this scenario, every processor is responsible for calculating a column of C instead of a row since j is held constant. There are frequent cache misses in matrix B since k is the fastest.
4. (j, k, i): We hold j constant. Frequent cache misses in A and C since i is the fastest.
5. (k, i, j): In this example and the next, processes are now able to read and write memory that is read by other processes. Without mutexes, there will be data races. Since we need mutexes, there will be potential slowdowns compared to the previous examples. As an aside, there will be low cache misses due to the ordering as a small bonus
6. (k, j, i): Has the same issues as the previous example but this time also has many cache misses due to jumps in A and C.

In conclusion, we strongly favor programs that do not have processors competing to read and write the same spots in memory. Therefore, the best option is still (i, k, j) as this minimizes cache misses and allows processes to work in parallel without mutexes.

Discussion of the gain in performance of the multi-thread implementation as a function of the number of threads concurrently used. Add plots and figures as necessary to explain your results.

Generally, for each size matrix of size N = { 64, 128, 256, 512, 1024, 2048 }, the multi-thread execution times of matrix multiplication decreases as the number of threads increases.

Below is a plot of the average execution time of matrix multiplication of size N = { 64, 128, 256, 512, 1024, 2048 } as a function of the number of threads used T = { 2, 4, 8, 16 }. The vertical axis is scaled logarithmically. The average execution times were calculated by take the arithmetic mean of 3 trials per combination of N and T.

We can see that for all N that is not size 64, there is a decrease in execution time from using 2 threads to 4 threads, but there is not as much improvement between 4, 8, and 16 threads used. N = 64 is interesting since it is the only case where the average execution time increases as the number of threads increases. This could be that the overhead of using more threads outweighs the gain in performance of using threads to parallize the computations, while in larger dimensions N yields more gain in performance with the same overhead of using threads.