**Gaussian Elimination with OpenMP**

When choosing how to layout my data, I was presented with two options:

1. A contiguous layout for the array.
2. Represent the array as a vector of n pointers to n-element data vectors.

I elected to choose option 2, which was to represent the array as a vector of n pointers to n-element data vectors. The reason I chose to layout my data this way was because I knew that potentially, we may need to execute n-1 row swaps at worst case during the pivot phase and I knew that if we were to represent the data as a contiguous layout, then these swaps would be costly. On the other hand, representing it with pointers allows us to simply exchange pointer values and that results in a successful swap. This gives us the advantage of having our swap operations finish in constant time AKA O(1).

When choosing how to partition my data, I was presented with 5 options:

1. Block partitioning.
2. Static cyclic with large chunks
3. Static cyclic with small chunks
4. Dynamic
5. Guided

To test which option would be the best for my program, I ran 30 runs with different chunk sizes and different scheduling types, but with the same exact data. I tested the scheduling types against a problem size of n = 4000 with 5 threads, 10 threads, and 20 threads on Jaguar and I used the following options to get a better idea of which scheduling type works best for my program:

-Block -Static with n/(num\_threads \* 2) size chunks -Static with n/(num\_threads) size chunks -Static with n/(num\_threads \* num\_threads) size chunks -Static with the size of the chunks as 1 -Dynamic with n/(num\_threads \* 2) size chunks -Dynamic with n/(num\_threads) size chunks -Dynamic with n/(num\_threads \* num\_threads) size chunks -Dynamic with the size of the chunks as 1 -Guided

The outcomes showed that scheduling the for loops with static partitioning using “n/(num\_threads \* 2)” size chunks provided the fastest results, so that was the scheduling choice I used in my program.

My program takes advantage of parallelism wherever possible, except for the initialization of the matrices/vectors and when calculating which row contains the largest absolute value during the pivoting phase. The reason I did not choose to partition the work in these two cases was because

1. We were asked not to time the initialization of the matrices and vectors, so I did not feel the need to parallelize this portion of the program.
2. Calculating the largest value in a column using multiple threads would take a lot of overhead and seemed like too much work. Also, parallelizing this portion of the program did not seem like it would reap much benefit due to all the communication that would be needed between the threads.

My program does take advantage of parallelism in the following phases of Gaussian elimination:

**Forward elimination:**

In the forward elimination phase, each thread is given its designated iterations and rows to operate on. It then calculates the multiplier for the current iteration and uses this value to update the values in the current row. Once this is done, it also updates the B vector using the same multiplier and the same row number.

**Back substitution:**

During the back substitution phase, each thread is given its designated iterations and rows to operate on. It then proceeds to multiply its current A matrix value by the newly calculated X value and then take this result and subtract it from the corresponding B vector value.

**Multiply A matrix by X vector:**

During this phase, each thread is given its designated iterations and rows to operate on. It then calls a function during the parallel region called “cross\_product,” which is also parallelized.

**Cross product:**

During this phase, each thread is given its designated iterations and cross product portions to calculate. Once the threads calculate their individual portions of the cross product, they perform a “+” reduction using the OpenMP reduction clause, and then return the result to the calling function (the function above).

**Subtract Vectors:**

During this phase, each thread is given its designated iterations and subtraction result to calculate. The result of this calculation is passed to the “l2\_norm” function, which then utilizes parallelization to calculate the l2-norm.

**l2 norm:**

During this phase, each thread is given its designated iterations and vector values to square and then perform a “+” reduction on the result using the OpenMP reduction clause.

Throughout my program, I only use the implicit OpenMP synchronization mechanisms. I did research on whether or not I should try to add some synchronized areas in my code, and I came to the undisputed conclusion that performance is at its most optimal speed when synchronization is minimized. After coming to this realization, I made it my goal to write my code in a way that avoided the need to use any explicit synchronization mechanisms. Therefore, the only synchronization points in my program are the implicit ones that OpenMP provides by default (the ones at the beginning and at the end of parallel constructs).

Timing Results and L2-Norm Table







