ECE 420

Lab 3 Report

Lab Section: H41

March 15th, 2017

By: Quentin Lautischer and Raghav Vamaraju

Description of Implementation:

Our serial implementation of Gauss-Jordan Elimination with partial pivoting was based on the "serialtester.c" code provided on eClass. We then added OpenMP parallelizations on top of this. The code for our parallel Gauss-Jordan implementation can be found in "main_omp.c". The details are as follows:

1. The NxM matrix was loaded from file using the libraries provided in "Lab3IO.c". The size of this matrix was then used to allocate memory for an arbitrary vector (using "CreateVec()").
2. A function "report_timing()" was created as helper in fetching and printing the elapsed time of the program. This made use of the "GET_TIME()" function in the "timer.h" library.
3. The function "Solve()" contained the Gauss-Jordan algorithm, based on the provided "serialtester.c" code. This function made use of the OpenMP ("pragma") calls. The algorithm required multiple for-loops to iterate over and manipulate the matrix by multiplying rows by constant factors and rearranging rows. The goal of the algorithm was to obtain a diagonal row of values, and zeros everywhere else. The "pragma" (parallelization) statement was added to the final portion of the Gaussian algorithm, right before entering the Jordan section. This was the only point in the code at which OpenMP was used.
4. Upon completing the algorithm, the elapsed time was reported (using "report_timing()"), and the resulting matrix was saved to file using the provided "Lab3SaveOutput()" function.
5. The memory allocated for the matrix and vector were then destroyed.

Testing and Verification:

We used the ***serialtester*** that was supplied in the Lab3 development kit. We generated test data of sizes 3, 10, 100, and 1000, and would run our implementation of the algorithm, followed by serialtester to verify that our answer matched within a given error range. We found that our implementation consistently produced a negligible error amount, as well as a significant speedup. We also used multiple "printf()" statements to ensure that the parallelizations were running as expected and correctly. We found that the provided serialtester program and our simple "printf()" debug statements were enough to verify correctness of the program.

Performance Discussion:

- After applying various optimizations all over the code using "omp pragma", we came to the conclusion that it may be better to first identify what the largest bottlenecks of our algorithm were.

- We began our performance review by logging the time taken to execute specific blocks of the algorithm:
    - Init Block (t~0.000003s)
    - Gaussian Block (t~1.5s)
    - Jordan Block (t~0.013s)
    - End Calculation Block (t~0.00001s)
- After viewing this we discovered that the bottleneck of the algorithm was the Gaussian block. Within the Gaussian block, we evaluated the timing of blocks within.
    - The Pivot block (single for-loop)
    - The Calculation Block (for-loop nested within another for-loop)
- The calculation block proved to be the true bottleneck within the Gaussian block.
- This was the code block that required the most optimization work and so we decorated it with an "omp pragma" as seen in the image below.

```
# pragma omp parallel for shared(A) private(temp, i, j)
for (i = k; i < size; i++){
  temp = A[index[i]][k] / A[index[k]][k];
  for (j = k; j < size + 1; j++){
    A[index[i]][j] -= A[index[k]][j] * temp;
  }
}
```

- After compiling the new code containing the "omp pragma" optimization, we found a speedup of about 3x, where our execution time went from ~1.5s to ~0.5s.
- We then began experimenting with adding pragma decorators to other code blocks within our algorithm but found that the majority of them either had critical code sections **or** would actually have an overall slowdown due to the omp environment setup cost.

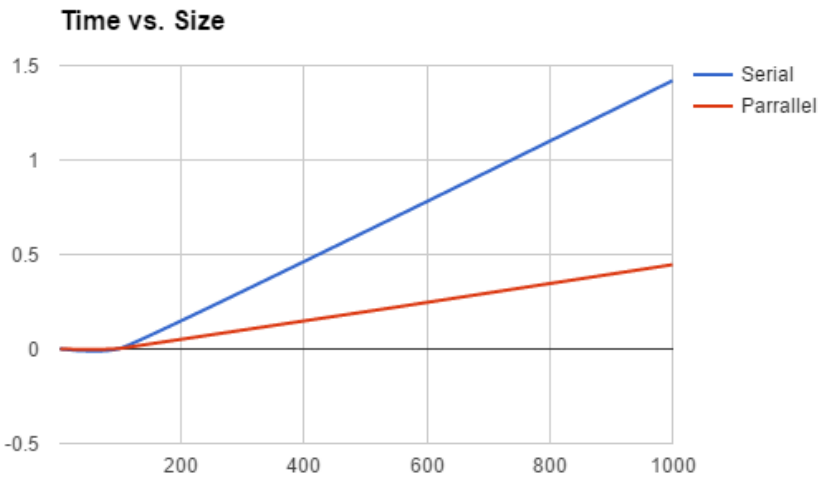
Below is a table containing a summary of various implementations and their timings:

OpenMP Type	***Matrix Size***	***Num Threads***	***Execution Time (s)***
Serial Implementation (no OpenMP)	3	1	0.000002
	100	1	0.002170
	1000	1	1.42
"#pragma omp parallel for" before final for-loop in Gaussian section	3	8	0.000306
	100	8	0.002552
	1000	8	0.446
	1000	6	0.58
	1000	4	0.83
	1000	2	0.87
	1000	1	1.72

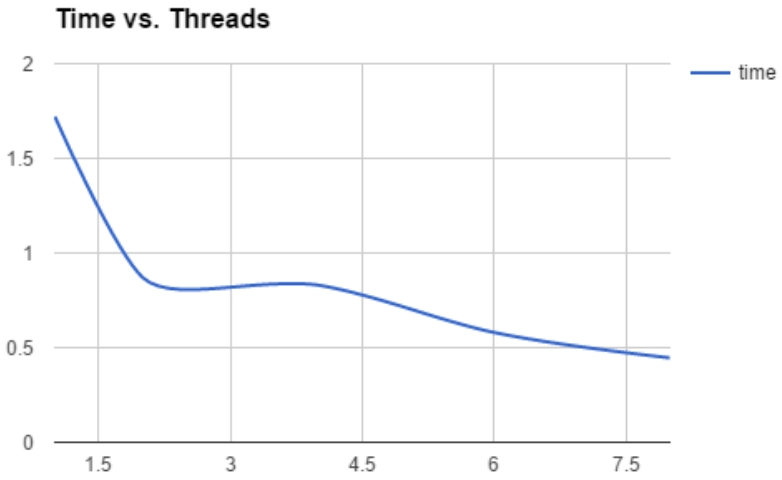
"#pragma omp parallel for" before multiple for-loops earlier in the Gaussian section	1000	8	2.07
	1000	6	1.76
	1000	4	1.64
	1000	2	1.43
	1000	1	1.45

As shown in the table, an inferior implementation (the final dataset) was significantly slower than the serial implementation, even though it utilized OpenMP. This is because the overhead from spawning and joining threads served to decrease the overall speed, since the "pragma" decorators were not placed optimally. This was the case for various other implementations that we tried.

Below is a graph depicting overall execution time (seconds) vs. matrix size for our optimized parallel solution and the serial solution. The parallel solution showed a significant increase in speed (decrease in time) as the matrix size increased.



Below is a graph depicting overall execution time vs. number of threads for our optimized parallel solution. As the number of threads increased, the speed increased (time decreased).



Conclusion and Experience:

This lab demonstrated that a relatively complex algorithm, written serially, can be parallelized in a simple fashion using OpenMP decorators. By placing the "pragma" statement in the optimal place, we achieved a speedup of approximately 3X the original, serial implementation. In addition, we discovered that it is difficult to assess where to place OpenMP optimizations in code, and how exactly it will impact the speed, and if it will produce a correct output. We tried many implementations expecting significant speedup, but found that most of them either slowed down the implementation, or produced a false output due to compromising a critical sections. While OpenMP makes it very easy to add parallelization to serial code one still has to evaluate whether each block actually benefits from the addition and outweighs the overhead. Adding parallelization to code is a relatively fragile process and can be quite difficult to debug.