Multi-threaded Linear Algebra Algorithms Performance Analysis (OpenMP)

Justin Newman, Andrew Fleming

[a]*James Madison University Department of Computer Science, Harrisonburg, 22801, VA, United States*

Abstract

OpenMP (Open Multi-Processing) is a popular method for adding parallelism to C programs. It is a set of compiler directives and library functions that allowed us to specify which parts of the program can be executed in parallel by multiple threads. By distributing the work among multiple threads, we significantly speed up the execution of computationally-intensive applications on the JMU cluster. We discuss the number of threads, the data sharing between threads, and the synchronization of threads to avoid race conditions and ensure correctness. We elaborate on the process we used to achieve our multi-threaded performance results.

Keywords: Linear Algebra, Parallelism, C, OpenMP

1. Introduction

To better understand the performance of our parallel code, we first measured the runtime for multiple runs of the same input size on the serial code. As the input size of the matrix increased, we observed an increase in the time required to complete the Gaussian elimination. This helped us determine the scalability and efficiency of the parallel code and gave us a baseline to compare with.

Nthreads	ERR	INIT	GAUS	BSUB	N
1	2.3e-14	0.0114	0.4798	0.0010	1000
1	2.7e-14	0.0218	1.3508	0.0020	1414
1	4.2e-14	0.0500	7.1733	0.0053	2000
1	4.4e-14	0.0997	25.4612	0.0107	2828
1	6.1e-14	0.1972	76.6564	0.0185	4000
1	6.4e-14	0.3948	217.8954	0.0340	5656

Table 1: Results showing the serial time in seconds for initialization, Gaussian elimination, and backwards substitution.

We noticed the accuracy of the approximation improved as the input size increased. As the input size increases, the number of computations required to solve the system also increases, and this generally results in greater accuracy.

2. Initialization

We noticed a near-perfect linear scaling improvement during the initialization phase due to the parallelization of the random system generation using OpenMP. By using OpenMP to parallelize the outer loop of the random matrix generation process, the program was able to make use of multiple processors in order to facilitate an improvement in performance.

Parallelization is a powerful tool for increasing performance because it allows multiple processors to work on the same task simultaneously. In the case of the random system generation the shared memory architecture of OpenMP allowed threads to share information about the size of the matrix and the pseudo-random number generator seed, while each thread operated on a distinct portion of the matrix. The use of OpenMP to parallelize the random system generation thus led to the near-perfect linear scaling improvement observed during the initialization phase, demonstrating the power of parallelization in improving program performance.

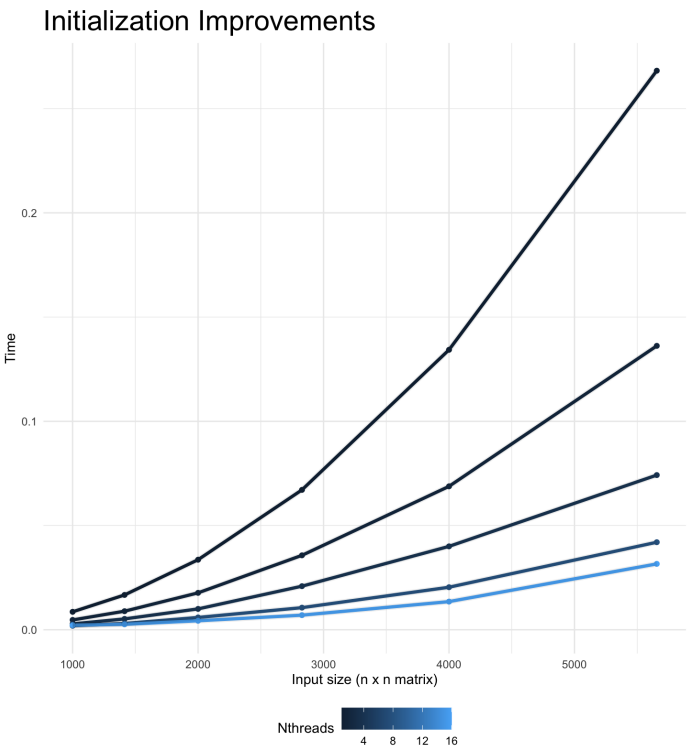

Figure 1: Shows relatively linear scaling.

Nthreads	ERR	INIT	GAUS	BSUB	N
1	2.10E-14	0.0086	0.1209	0.0007	1000
1	2.50E-14	0.0167	0.3585	0.0013	1414
1	3.80E-14	0.0336	1.617	0.0026	2000
1	3.70E-14	0.0671	5.169	0.005	2828
1	5.90E-14	0.1343	14.8516	0.0093	4000
1	5.70E-14	0.2682	41.7842	0.0178	5656
2	1.80E-14	0.0047	0.0637	0.0021	1000
2	2.60E-14	0.0089	0.178	0.0028	1414
2	3.80E-14	0.0177	0.4914	0.0048	2000
2	3.60E-14	0.0357	2.8269	0.0075	2828
2	4.60E-14	0.0688	9.66	0.0115	4000
2	5.70E-14	0.1362	28.5211	0.0186	5656
4	2.20E-14	0.0028	0.0379	0.0036	1000
4	2.60E-14	0.0052	0.0964	0.0056	1414
4	3.20E-14	0.01	0.2509	0.0085	2000
4	4.10E-14	0.0209	0.724	0.0123	2828
4	5.00E-14	0.04	4.8436	0.0177	4000
4	5.80E-14	0.0742	17.2251	0.03	5656
8	2.40E-14	0.0019	0.0237	0.0055	1000
8	2.90E-14	0.003	0.0552	0.0083	1414
8	3.50E-14	0.0059	0.14	0.0132	2000
8	3.60E-14	0.0106	0.3813	0.0183	2828
8	4.70E-14	0.0204	2.5406	0.0308	4000
8	5.50E-14	0.042	12.6346	0.0431	5656
16	2.20E-14	0.0022	0.0215	0.0066	1000
16	3.20E-14	0.0026	0.0468	0.0099	1414
16	2.80E-14	0.0043	0.1048	0.0147	2000
16	3.40E-14	0.007	0.2324	0.0219	2828
16	4.20E-14	0.0135	0.6017	0.0343	4000
16	5.20E-14	0.0316	12.2385	0.0522	5656

Table 2: Results showing the serial time in seconds for initialization, Gaussian elimination, and backwards substitution for multiple threads in non-trianglur mode.

3. Multi-threaded Gaussian elimination algorithm

Gaussian elimination is a method used to solve a system of linear equations by systematically manipulating the equations to simplify them and ultimately arrive at a solution. In linear algebra we first write the system of equations in matrix form, with the coefficients of the variables as the entries of the matrix and the constants on the right-hand side. Next, we perform elementary row operations on the matrix to transform it into row echelon form. These operations include adding a multiple of one row to another row, swapping two rows, and multiplying a row by a nonzero constant.

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right] \rightarrow \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & c_m \end{array}\right]$$

The goal of Gaussian elimination is to simplify the matrix as much as possible so that the back-substitution step is straightforward. The row echelon form of the matrix has the property that the first nonzero entry in each row (called the ”pivot” element) is to the right of the pivot element in the row above it. This makes it easier to solve for the variables in the back-substitution step.

In order to improve the performance of our code for Gaussian elimination, we first attempted to parallelize as much of the Gaussian elimination as we could. This consisted of putting a parallel block of code around the entire method, and then three parallel for loops before the pivot iteration, the row iteration, and the column iteration respectively. We also put a critical section around the innermost assignment of b[row]. We found a significant speedup, but errors increased to the point where our results were considered to be significantly incorrect. This occurred because multiple threads began accessing the same variable in a data race. To amend this, we added the shared clause so that the variables [n, b, A] should be shared among all threads. This means that each thread has its own copy of the row and col variables, but they all access the same [n, b, A]. This allowed us to achieve an implementation of Gaussian elimination that ran faster than the serial implementation and produced accurate results.

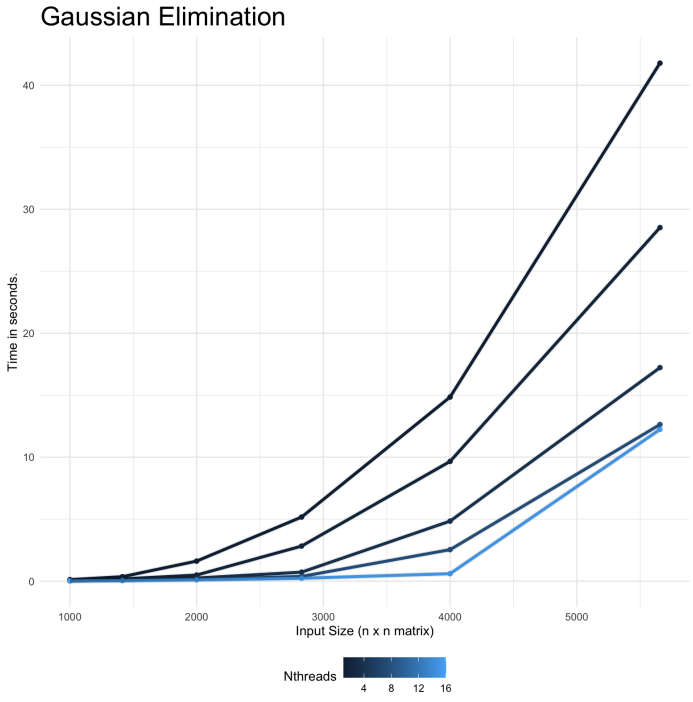

Figure 2: Shows relatively linear scaling.

4. Multi-threaded backwards substitution algorithms

Speeding up the row-oriented backwards substitution functions involved a trial and error approach to achieve the best results for our particular implementation. At first, we tried adding pragma omp for parallel to the outer loop of the code to parallelize the execution. However, this introduced some errors in the computation. Eventually, we decided to move the parallelization pragma to the inner loop and add a reduction clause to ensure the correct calculation of the temporary variable. To parallelize the column-oriented backward substitution outer loop we tried adding a parallel for pragma and specifying the shared variables between the threads.

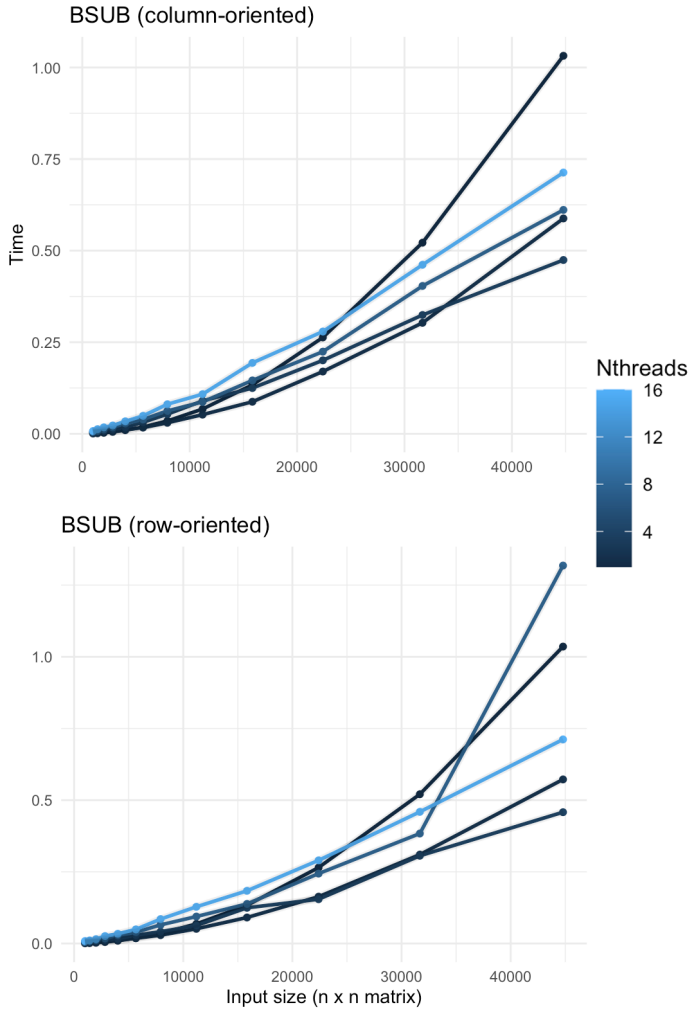

Figure 3: Shows relatively linear scaling.

This allowed the loop to be executed in parallel by multiple threads, which significantly reduced the overall execution time and kept our errors low. The next step was to optimize the inner loop of the code, which involved the calculation of x[col]. To ensure that only one thread could execute this section at a time, a critical pragma was added.This ensured that the critical section was executed correctly and efficiently in the multithreaded environment. Finally, the last part of the code was optimized for parallel execution by adding another ”pragma omp parallel for” to the inner loop. This allowed the loop to be executed in parallel by multiple threads, resulting in further reduction of the overall execution time.

Nthreads	ERR	BSUB Col	BSUB Row	N
1	8.50E-14	0.1335	0.1331	15838
1	9.90E-14	0.263	0.2649	22398
1	1.20E-13	0.5217	0.5206	31676
1	1.40E-13	1.032	1.0356	44796
2	8.70E-14	0.0874	0.0907	15838
2	9.70E-14	0.1699	0.1634	22398
2	1.30E-13	0.3029	0.3101	31676
2	1.60E-13	0.5879	0.5723	44796
4	8.30E-14	0.1248	0.1245	15838
4	8.70E-14	0.2005	0.1544	22398
4	1.30E-13	0.3243	0.306	31676
4	1.40E-13	0.4744	0.458	44796
8	8.20E-14	0.1457	0.1381	15838
8	8.60E-14	0.2242	0.2438	22398
8	1.30E-13	0.4036	0.3835	31676
8	1.30E-13	0.6112	1.3182	44796
16	8.00E-14	0.1935	0.1838	15838
16	8.60E-14	0.2791	0.29	22398
16	1.10E-13	0.4614	0.4592	31676
16	1.30E-13	0.7132	0.7118	44796

Table 3: Results showing the serial time in seconds for initialization, Gaussian elimination, and backwards substitution for multiple threads in non-triangular mode.

5. Summary and conclusions

Our optimization efforts resulted in significant improvements in both initialization times, Gaussian elimination times, and in both row and column-oriented backward substitution functions. We were able to achieve these improvements by leveraging the OpenMP library and using a step-by-step process of optimization and experimentation to identify the most effective strategies for parallelization.

Our approach included identifying the most time-consuming and critical parts of the code and optimizing them for parallel execution. We were able to parallelize both the Gaussian elimination and the backward substitution loops, allowing for the calculations to be executed concurrently across multiple threads. Additionally, we utilized OpenMP constructs such as critical sections to ensure the accuracy of the solution set.

The optimizations that we implemented had a profound effect on the speed of the code, resulting in a substantial speedup of the entire program. By leveraging parallel processing techniques and optimizing for concurrency, we were able to achieve a more efficient and scalable solution to the problem at hand.

In conclusion, our approach highlights the importance of identifying the critical sections of code and utilizing the most appropriate parallelization strategies to achieve an optimal solution. Through our efforts, we were able to achieve both an

accurate solution set and a significant reduction in the execution time, demonstrating the benefits of parallel processing in solving complex computational problems.

Acknowledgements

Special thanks to Dr. Lam for introducing us to OpenMP and its benefits. Learning about this powerful tool has been an eye-opening experience that has greatly enhanced our understanding of parallel programming. His guidance and expertise has been instrumental in helping us understand the intricacies of this technology. Through his clear and concise explanations, we have gained a deep appreciation for the efficiency and scalability that OpenMP can provide. We are thrilled to continue exploring the possibilities of OpenMP and CUDA then applying this knowledge to our future projects.

References

https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf

https://w3.cs.jmu.edu/lam2mo/cs470/cluster.html

https://www.sciencedirect.com/book/9780128046050/an-introduction-to-parallel-programming

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial