Michael Spagon MRS2555

Tiffany Tillett TAT573

Parallel Algorithms – Summer 2017

Solving Linear Systems and Find Determinants Using MPI

Introduction:

We implemented algorithms to calculate determinants and solve linear systems of equations. This paper will walk through the mathematical concepts of both problems as well as the algorithms that we used to solve them. The main focus will be on applying Gaussian Elimination to both problems. However, we will also consider an additional algorithm for calculating determinants. We will provide runtime data for both problems and discuss tradeoffs for different implementations.

Gaussian Elimination:

Gaussian Elimination is a mathematical procedure which has the ability to transform a matrix into upper and/or lower triangular form. For purposes of this project, we will focus on upper triangular form. We will later see how the upper triangular form can be beneficial for calculating determinants and solving linear systems.

$$\begin{bmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & \dots & u_{1,n-1} \\ & & \dots & \\ 0 & 0 & \dots & u_{n-1,n-1} \end{bmatrix}$$

Figure 1: Upper Triangular Form of a Matrix

Gaussian Elimination allows for certain operations to be performed on a matrix without affecting the properties of that matrix. These operations are: multiplication of a matrix row by any nonzero constant, permutation of matrix rows and addition of any matrix row to any other matrix row. We combine these three operations to transform the matrix into upper triangular form.

```
for $k = 1, n-1$
    for $i = k+1, n$

    $\ell_{ik} = \frac{a_{ik}}{a_{kk}}$ (assuming $a_{kk} \neq 0$)

    for $j = k, n$
        $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$
```

Figure 2: Pseudocode for Sequential Gaussian Elimination

Original Matrix:

$$\begin{bmatrix} 1 & 3 & 2 \\ 2 & 7 & 5 \\ 1 & 4 & 6 \end{bmatrix}$$

Subtract 2* row 1 from row 2

$$\begin{bmatrix} 1 & 3 & 2 \\ 0 & 1 & 1 \\ 1 & 4 & 6 \end{bmatrix}$$

Subtract row 1 from row 3

$$\begin{bmatrix} 1 & 3 & 2 \\ 0 & 1 & 1 \\ 0 & 1 & 4 \end{bmatrix}$$

Subtract row 2 from row 3

$$\begin{bmatrix} 1 & 3 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 3 \end{bmatrix}$$

K = 1

I = 2
A[2][1] = 2
A[1][1] = 1
Scale = 2 / 1 = 2

I = 3
A[3][1] = 1
Scale = 1 / 1 = 1

K = 2
I = 3
A[3][2] = 1
A[2][2] = 1
Scale = 1 / 1 = 1

Figure 3: Gaussian Elimination Example

We can make Gaussian Elimination parallel by computing all iterations of i in parallel. For example, both the 2nd and 3rd rows can simultaneously compute their scale factor relative to row 1 and update their rows accordingly. However, we must add some handling in order for MPI to support this. Because each processor has its own copy of the matrix, we must broadcast the finished copy of each matrix row to all other processors. The steps are:

```
1)  If I am assigned to the kth row, broadcast it to all other processors
2)  Else receive the updated kth row
3)  For all rows greater than k in parallel
        a)  Calculate scale factor relative to the kth row
        b)  Subtract the scaled multiple of the kth row from my row
```

Figure 4: Pseudocode for Parallel Gaussian Elimination

Determinants:

The determinant of a matrix is a useful mathematical operation. It can be used to calculate things such as the area of a triangle or an equation of a line. For a 2x2 matrix, the calculation is straightforward:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$$

Figure 5: 2x2 Determinant Definition

For matrices larger than 2x2, however, we need to take some extra steps.

$$\begin{aligned} |A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} &= a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &= aei + bfg + cdh - ceg - bdi - afh. \end{aligned}$$

Figure 6: 3x3 Determinant Definition

This procedure can be extended to any $n \times n$ matrix.

Based on the definition of the determinant, the most straightforward implementation would be to develop a recursive solution with the 2x2 case as the base case.

```
1)  If matrix size = 2
    a)  Directly compute determinant
2)  Else
    a)  Det = 0
    b)  For each element in top row
        i)   Create a sub-matrix excluding the row and column of the current
             element
        ii)  Recursively calculate the determinant on that submatrix and apply
             the correct +/- sign
        iii) Det += element * sub_det
```

Figure 7: Pseudocode for Recursive Determinant Calculation

This algorithm is straightforward, but the performance is poor. Creating a sub-matrix takes $O(n^2)$ time, we need $O(n)$ recursive calls and we execute the entire loop n times. Therefore, this takes $O(n^4)$ time! We can make the outer loop parallel to reduce it to $O(n^3)$ time, but this is still not feasible for a practical application. We did implement this version as an experiment, but we observed timeout issues for fairly small matrices (15x15) due to the poor performance.

In order to improve performance, we applied Gaussian Elimination to calculate the determinant. Once we arrive at the upper triangular matrix generated by Gaussian Elimination, calculating the determinant is straightforward. We simply need to multiply all of the elements of the diagonal. Each processor will sequentially do the multiplication for the rows it is assigned to. We then use MPI_Reduce to get the final result based on the results from each processor.

Gaussian Elimination gives us a significant performance improvement, but there are a few things to consider. The first is that we cannot use integer arithmetic to do Gaussian Elimination since it will often result in obtaining a scale factor of zero. According to the rules of Gaussian Elimination, we can only multiply rows by a non-zero constant, so we must use floating point numbers to avoid this possibility. The switch to floating point alone will have a negative impact on the runtime. In addition, the Gaussian Elimination algorithm as currently implemented can result in numbers within the matrix that are fairly large. These larger numbers further increase the runtime of the floating-point operations. In addition, they cause a significant increase in the amount of time that it takes to broadcast the matrix rows. Because of the size of the numbers, it is essentially impossible to operate on any large matrices as the resulting determinant cannot fit within the range of the C++ double. In addition, the additional time spent on message passing degrades the performance to the point that the sequential algorithm actually performs better. This performance gap increases as we increase the matrix size or the number of MPI processors. Below are charts illustrating the performance of the sequential and parallel algorithms in seconds.

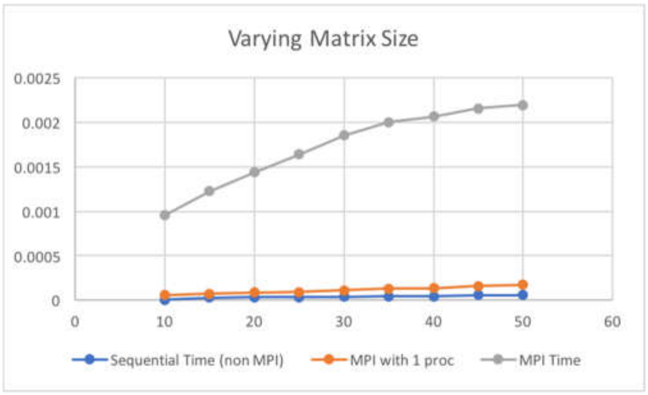


Figure 8: Runtime Results for Varying N with P=16

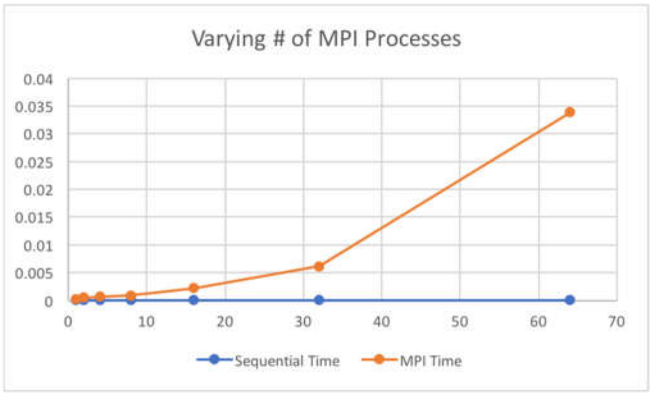


Figure 9: Runtime Results for Varying P with N=50

How Can We Improve?

One of the main reasons that the performance is so poor is that the sizes of the numbers become very large throughout the process of Gaussian Elimination. As previously mentioned, Gaussian Elimination allows for three operations: multiplication of a matrix row by any nonzero constant, permutation of matrix rows and addition of any matrix row to any other matrix row. The previous version of the algorithm only utilized the first and third of those operations. There is a concept called pivoting which allows us to take advantage of the second operation to contain the sizes of the matrix elements.

Why Pivoting is Necessary:

Pivoting is the process of swapping rows in a matrix. It is necessary when the pivot element of a row is zero. Consider the example in figure 10. On the second iteration of Gaussian elimination, the element $a_{2,1} = 0$ must be used to eliminate the 2 below it. We cannot multiply 0 by any constant to achieve -2. Therefore, we must *pivot*, or swap, the second and third rows in order to proceed.

$$\left[\begin{array}{ccc|c} 1 & -1 & 2 & 8 \\ 0 & \enclose{circle}{0} & -1 & -11 \\ 0 & 2 & -1 & -3 \end{array}\right]$$

Figure 10: Gaussian Elimination cannot proceed without pivoting.

Pivoting is also essential for computational accuracy. Without pivoting, Gaussian elimination is unstable and often useless. Consider the example in figure 11. The solutions to the system are $x_1 = 10.00, x_2 = 1.000$, but when the system is solved using Gaussian elimination without pivoting using 4-digit arithmetic the results are $x_1 \approx 9873.3, x_2 \approx 4$. The effects of rounding is compounded in larger cases which explains why our numbers grew to ∞.

$$\left[\begin{array}{cc|c} 0.00300 & 59.14 & 59.17 \\ 5.291 & -6.130 & 46.78 \end{array}\right]$$

Figure 11: Rounding errors when pivot elements are small.

How to Achieve Pivoting:

At each iteration of Gaussian elimination, we must find the maximum of the absolute value of the elements in the current pivot column. During the first iteration of Gaussian elimination, the pivot element should be in the first row and first column. Examining each element in the first column of figure 12, we see that 1 is the maximum and should be the pivot. We swap the first and second rows. *This will minimize rounding errors.*

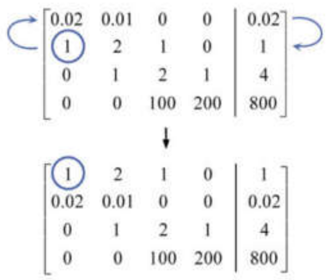
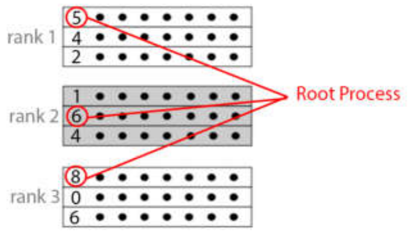


Figure 12: Pivoting during the 1st iteration of Gaussian Elimination.

Parallel Gaussian Elimination using Pivoting:

At each iteration i of Gaussian elimination we must do three things:

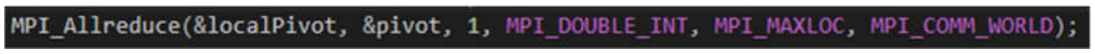
1. **Choosing the pivot row** – Each process looks at all of its local elements to determine a local maximum. Using an **MPI_Allreduce** statement, the root process collects each local maximum value from each of the processes to determine which processor has the global maximum and therefore the new pivot element.



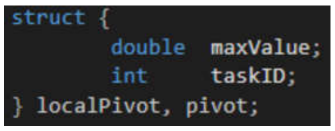
2. **Broadcast pivot row to other processors** – The processor with the global maximum will broadcast its row to all subtasks.

3. **Subtract the pivot row** – Every processor will use the broadcasted row to eliminate the element in that column.

We used the following MPI command for finding the maximum pivot element:


```
MPI_Allreduce(&localPivot, &pivot, 1, MPI_DOUBLE_INT, MPI_MAXLOC, MPI_COMM_WORLD);
```

The data type MPI_DOUBLE_INT is a struct that contains a double and an int and is used for MPI_MAXLOC reductions. The MPI_MAXLOC operation returns the global maximum and which processor rank has the maximum.


```
struct {
        double  maxValue;
        int     taskID;
} localPivot, pivot;
```
Figure 13: Struct used for MPI_Allreduce

Solving Linear Systems using Back Substitution:

Once a triangular matrix is achieved from Gaussian elimination, back substitution can be applied to solve for the unknowns:

$$\begin{aligned} x_0+3x_1+2x_2 &= 1 \\ x_1+\ x_2 &= 16 \\ 3x_2 &= 9 \end{aligned}$$

Figure 14: The results of Gaussian Elimination

$$\begin{aligned} x_0+3x_1+2x_2 &= 1 \\ x_1\qquad &= 13 \\ x_2 &= 3 \end{aligned}$$

Figure 15: The first iteration of back substitution

From the third equation, it is clear that $x_2 = 3$. This value can be used to solve for x_1 in the second equation. We can proceed in a similar fashion until the system is solved. With regards to exposing parallelism, whenever a value is known it can be broadcasted to all other processes and eliminated from other equations in parallel.

Runtimes of Parallel Algorithm:

The programs were run on TACC Stempede at the University of Texas at Austin. The sequential algorithm was implemented differently which explains the super-linear speedup.

Table 1.0 – Solving Linear Systems Algorithmic Runtimes

Matrix Size	Sequential Algorithm	Parallel Algorithm					
		2 cores	4 cores	8 cores	16 cores	32 cores	64 cores
		Time	Time	Time	Time	Time	Time
1000	**1.73**	0.112	0.070	0.045	0.034	0.126	**0.158**
2000	**14.01**	1.169	0.818	0.667	0.352	0.174	0.158
3000	**48.19**	4.825	3.928	3.571	1.306	0.407	0.335
4000	**112.43**	11.909	9.906	9.175	4.514	1.630	0.585
5000	**222.60**	23.504	19.556	18.100	9.684	4.744	1.468
6000	**383.03**	41.073	33.988	31.381	17.195	9.356	3.564

Adding too many cores relative to the problem size leads to a degradation of runtime due to message passing overhead.

Table 2.0 – Message Passing Overhead

Matrix Size	Parallel Algorithm			
	64 cores	1000 cores	2000 cores	4096 cores
	Time	Time	Time	Time
1000	**0.158**	0.719	-	-
2000	0.158	**1.833**	**2.148**	-
6000	3.564	**2.137**	-	**4.475**
15000	-	**7.137794**	-	**11.774**

Data Distribution:

Using a simple row distribution scheme leads to idle processors and an unequal computational load. When performing Gaussian elimination and back-substitution, processors that have already computed their row will remain idle. Therefore, a cyclical row striping scheme is ideal.

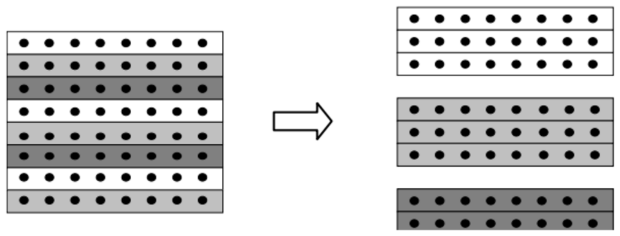


Figure 16: Optimal row distribution amongst processors using cyclical striping.

Conclusion:

Gaussian Elimination is a useful mathematical procedure that can be applied to operations such as solving linear systems of equations and calculating determinants. We outlined a parallel version of the algorithm and accessed running times.

Work Cited

"Determinant." Wikipedia. Wikimedia Foundation, 02 July 2017. Web. 5 July 2017.

Heath, Michael T. "Parallel Numerical Algorithms." (n.d.): n. pag. 2013. Web. 5 July 17. <https://courses.engr.illinois.edu/cs554/fa2013/notes/06_lu_8up.pdf>.

"Iterative Methods for Solving Linear Systems." Texts in Applied Mathematics Numerical Mathematics (n.d.): 125-82. Web. 5 July 17. <http://www.hpcc.unn.ru/mskurs/ENG/DOC/pp09.pdf>.

Jones, James. "6.5 - Applications of Matrices and Determinants." 6.5 - Applications of Matrices and Determinants. Richland Community College, n.d. Web. 04 Aug. 2017. <https://people.richland.edu/james/lecture/m116/matrices/applications.html>.

Karniadakis, George, and Robert M. Kirby. "9 Fast Linear Solvers." Parallel Scientific Computing in C and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation. New York: Cambridge UP, 2008. N. pag. Print.

Quinn, Michael J. “12. Solving Linear Systems.” Parallel Programming in C with MPI and OpenMP, McGraw-Hill Education

Grama, Ananth. “8. Dense Matrix Algorithms.” Introduction to Parallel Computing, Pearson, 2003.