Parallel Gaussian Elimination

Prashant Mital
SDS 394C Parallel Comput for Sci & Engr, Spring 2015
University of Texas at Austin

May 10, 2015

Contents

Chapter 1

Problem Description

Gaussian Elimination is an algorithm in Linear Algebra best understood as a series of row operations on the coefficient matrix of a system of linear equations. The method can also be used to calculate the rank, determinant, and inverse of a matrix. The algorithm involves adding multiples of each row to subsequent rows, in order to make the coefficient matrix upper triangular. The resulting system is then solved by back-substitution which is trivial to perform. This algorithm is stated somewhat formally as Algorithm 1 and illustrated in Fig. 1.1.

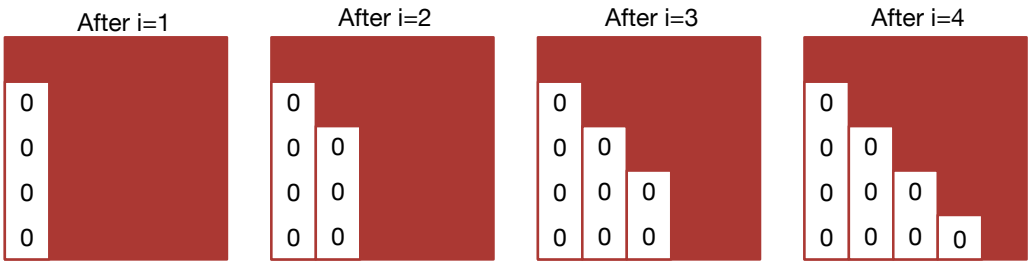

Figure 1.1: An illustration of how Gaussian Elimination zeros out entries below the diagonal

As is self evident from the illustration, computation of the zeros below the pivot elements can be avoided. It is common practice to use the zeroed out entries to store the corresponding multipliers m as they can be used to transform the load vector into the correct form before backward substitution. Despite its ease of implementation, there are some caveats with this particular method of Gaussian Elimination. In some circumstances, this method can fail entirely even if the system is non-singular and has a unique solution.

Algorithm 1 Gaussian Elimination

```
1: procedure SOLVING Ax = b
2:     N ← size of square system
3:     for (i=1; i<=N; ++i) do
4:         for (j=i+1; j<=N; ++j) do
5:             m = A(j,i)/A(i,i)
6:             for (k=i+1; k<=N; ++k) do
7:                 A(j,k) = A(j,k) − m × A(i,k)
8:     return A                                  ▷ A is upper triangular
```

Consider the following 2×2 matrix:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Algorithm 1 will fail in this situation even though $Ax = b$ is trivial. This situation is an extreme case of the numerical instability this algorithm encounters when the pivot element becomes progressively smaller (closer to machine precision). Due to round-off errors, the algorithm can yield completely incorrect answers. Subtraction of two nearly equal numbers is the main culprit that propagates error through the computation. This problem is solved by swapping rows of A so that the pivot element is the heaviest in its corresponding column confined to the remaining submatrix. The formal algorithm is stated as Algorithm 2

Algorithm 2 Gaussian Elimination with Partial Pivoting

```
1: procedure SOLVING Ax = b
2:     N ← size of square system
3:     for (i=1; i<=N; ++i) do
4:         Find k such that A(k,i) = max{A(x,i)}_{x=i+1}^{N}
5:         if (k != i) then
6:             Swap row k with row i
7:         for (j=i+1; j<=N; ++j) do
8:             m = A(j,i)/A(i,i)
9:             for (k=i+1; k<=N; ++k) do
10:                A(j,k) = A(j,k) − m × A(i,k)
11:    return A                                  ▷ A is upper triangular
```

Although systems of linear equations are ubiquitous in scientific applications, solution methods are often chosen after consideration of the structure

and properties of that particular system. For example, numerical methods that solve Partial Differential Equations typically yield vast systems in which most entries of the coefficient matrix are zero. These matrices are called Sparse whereas matrices with mostly nonzero entries are known as Dense (or Full). Sparse matrices require special attention because efficiently storing them relies upon specialized data structures that must be accounted for at the time of computation. Moreover, the mathematical properties of sparse matrices, when combined with the high efficiency and scalability of sparse matrix-vector product algorithms, make iterative solution methods an attractive and more pragmatic solution. Hence direct solution methods like Gaussian Elimination are usually costlier for Sparse matrices than the approximate iterative alternatives. However, for dense matrices – which also show up frequently in scientific computing – an efficient Gaussian Elimination algorithm is extremely useful. In fact, large sparse systems often yield smaller dense problems. The LINPACK benchmark used in ranking the Top 500 tests how fast a computer solves $Ax = b$ where A is dense.

Chapter 2

Implementation

The main challenge in parallelizing Gaussian Elimination on a distributed memory machine is that the calculation of each row requires the calculation of all rows that have come before it and therefore concurrency of operations becomes the overriding issue. In this work, the implementation is performed using C with Message Passing Interface (MPI). A common complication that arises in linear algebra is the need to store and operate on extremely large matrices which might not be possible on a shared memory systems. This makes an MPI implementation more pliable than the OpenMP alternative. The MVAPICH2 implementation of MPI was used along with the Intel C compiler.

2.1 Mapping

Mapping refers to the method of dividing up a matrix between the processors in a geometric manner. Striped mapping is used throughout our implementation. An entire row or group of rows are issued to each processor. In striped mapping, a row is an atomic unit. As a result, there cannot be more processors than rows in the coefficient matrix. In this work, I did not implement the checkerboard mapping due to the significantly higher complexity of indexing and addressing. Moreover, as the algorithm proceeds in a row-by-row manner, the striped topology logically follows from its structure. One important advantage of a checkerboard map (or a 2D block-cyclic map) is the ability to use more processors than there are rows in the coefficient matrix.
The topology which offers a more balanced workload between processors - the cyclic mapping - is the more likely candidate to be the optimal approach.

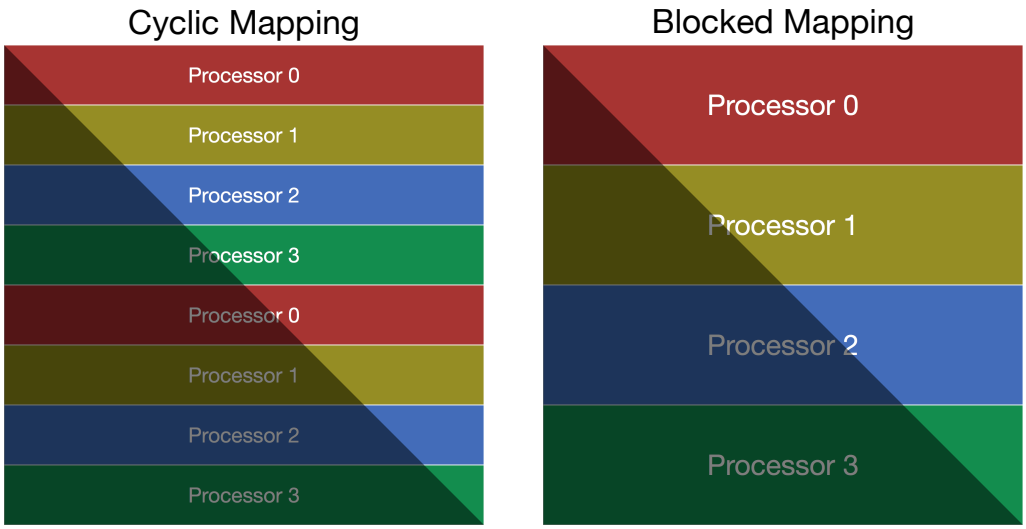

Figure 2.1: A visual representation of the two mapping schemes. The shadow indicates the computational load. It can be seen that processor 3 is responsible for a disproportionately large chunk of the computational load in the blocked mapping while the cyclic mapping distributes work more equally and should be the preferred topology for better load balancing.

2.2 Parallel Algorithm

The algorithm is fairly straightforward, with each step only requiring the communication of pivot row elements to the remaining team. As our mapping scheme considers the row to be an atomic unit, there is a one-to-one correspondence between a row on a particular processor and a row within the system matrix. Both mapping schemes guarantee that given two rows owned by the same processor, the row with the larger local index would also have the the larger global index. Global column numbers are also the local column numbers as per this map, which simplifies things significantly. This means that for each processor, we only need to determine the starting row to apply Gaussian Elimination and then apply it all the way to the end. The algorithm is more formally stated as Algorithm 3.

Algorithm 3 Gaussian Elimination in Parallel using MPI Collectives

1: **procedure** SOLVING $\bar{A}x = b$
2: $N \leftarrow$ *number of rows/columns*
3: $P \leftarrow$ *number of processors*
4: $NROWS \leftarrow$ *rows per processor* N/P
5: *procid* $\leftarrow$ *rank of current processor*
6: $A \leftarrow$ *portion of* $\bar{A}$ *owned by procid*
7: **for** `(i=1; i<=N; ++i)` **do**
8: $x = mapping(i)$
9: $\implies$ get x s.t. $A(x,:) = \bar{A}(i,:)$ if $\bar{A}(i,:)$ lives on *procid*
10: $\implies$ else get $x = min\{j\}$ s.t. $A(x,:) = \bar{A}(j,:)$ and $j > i$
11: **if** $\bar{A}(i,:)$ `lives on` *procid* **then**
12: $pivotrow = A(x,:)$
13: $x = x + 1$
14: $rootid = procid$
15: **MPI_Bcast (Buffer:** *pivotrow* **, Root:** *rootid* **)**
16: **for** `(j=x; j<=NROWS; ++j)` **do**
17: $m = A(j,i)/pivotrow(i)$
18: **for** `(k=i+1; k<=N; ++k)` **do**
19: $A(j,k) = A(j,k) - m \times A(i,k)$
20: **return** $\bar{A}$ ▷ A is upper triangular

Chapter 3

Analytic Performance Model

3.1 Serial Time Complexity

Figure 3.1 visually illustrates the number of floating point operations that are needed in a serial implementation of the algorithm using a single processor. Assuming each operation takes unit time, and we operate on a square system with linear dimension N we can perform the following computation to ascertain the time complexity of the algorithm.

$$C(n) = 2\sum_{i=0}^{N-2}(N-i)(N-i-1)$$

$$C(n) = 2\sum_{i=0}^{N-2}(N(N-1)+i(1-2N)+i^2)$$

$$C(n) = 2N(N-1)^2 + 2\frac{(1-2N)(N-2)(N-1)}{2} + 2\frac{(N-2)(N-1)(2N-3)}{6}$$

$$C(n) = \frac{2}{3}(N^3 - N)$$

When N is sufficiently large, the complexity can be stated as:

$$C(n) = \frac{2}{3}N^3$$
$$C(n) \sim \mathcal{O}(N^3)$$

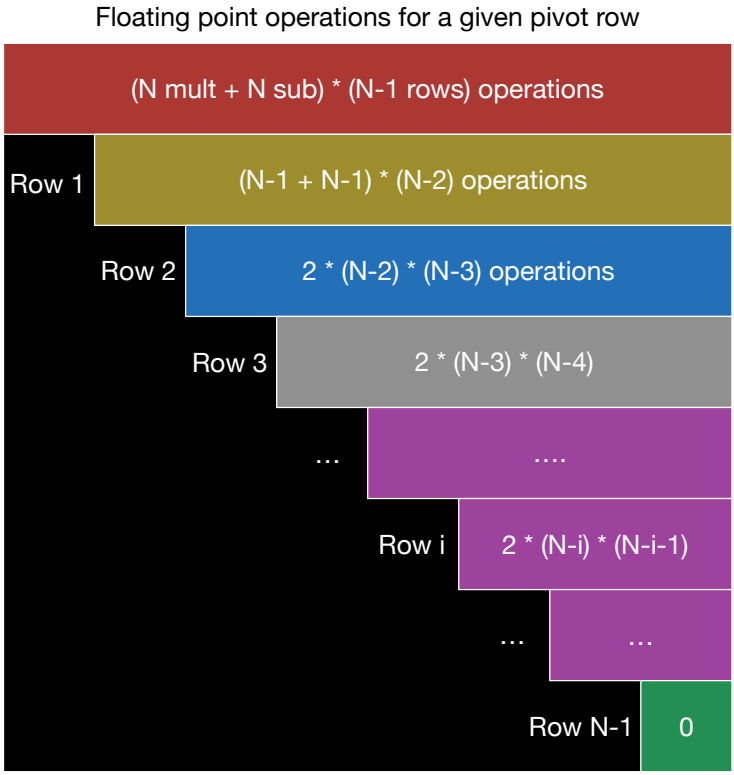

Figure 3.1: Number of floating point operations needed for a given pivot row. Observe that we operate on a progressively smaller submatrix of A as the algorithm progresses.

3.2 Parallel Time Complexity

This section will use the following notation:

$$\alpha \leftarrow \textit{message latency}$$
$$\beta \leftarrow \textit{time per word}$$
$$\gamma \leftarrow \textit{time per floating point operation}$$

3.2.1 Blocked Mapping

In the blocked mapping approach illustrated in Figure 2.1, we perform a similar analysis to find time complexity of the algorithm. On the most heavily loaded processor (processor 3 in Figure 2.1) in a team of P processors, communication involves making N broadcasts each taking $\alpha \log_2 P$ time and

sending N words in βN time. For the computational complexity, we consider the two cases - one where the pivot row belongs to a previous processor, and the second where it belongs to the last processor.

$$C(n) = \alpha \log_2 P + \beta N + \sum_{i=0}^{\frac{(P-1)N}{P}} 2(N-i)\frac{N}{P} + \gamma \sum_{i=\frac{(P-1)N}{P}}^{N} 2(N-i)(N/P-i-1)$$

As we are looking for asymptotic estimates of performance, we consider the case of large P and see that the limits on the final sum in the above equation reduce to 0 and we see no contribution from it. This intuitively makes sense as the operational submatrix would have significantly diminished in size if the pivot row were to reside on the final processor. It is therefore assumed that the contribution of those computations is negligible for the purposes of this analysis.

$$C(n) = \alpha \log_2 P + \beta N + \gamma \sum_{i=0}^{N} 2(N-i)\frac{N}{P}$$

$$C(n) = \alpha \log_2 P + \beta N + \gamma \sum_{i=0}^{N} \left(\frac{2N^2}{P} - 2\frac{iN}{P}\right)$$

$$C(n) = \alpha \log_2 P + \beta N + 2\gamma\frac{N^3}{P} - 2N\frac{N(N+1)}{2p}$$

$$C(n) = \alpha \log_2 P + \beta N + \gamma\frac{2N^3 - N^3 - N^2}{P}$$

$$C(n) = \alpha \log_2 P + \beta N + \gamma\frac{N^3 - N^2}{P}$$

For sufficiently large N, this reduces to:

$$C(n) = \alpha \log_2 P + \beta N + \gamma\frac{N^3}{P}$$

Using the conventional definition of speedup, S and efficiency, E we arrive at the following relations:

$$S(N,P) = \frac{T_{serial}}{T_{parallel}} = \frac{2}{3}\frac{1}{\left(\frac{\alpha \log_2 P}{N^3} + \frac{\beta}{N^2} + \frac{\gamma}{P}\right)}$$

$$E(N,P) = \frac{S}{P} = \frac{2}{3}\frac{1}{\left(\frac{P\alpha \log_2 P}{N^3} + \frac{P\beta}{N^2} + \gamma\right)}$$

Strong Scaling

In strong scaling we keep the problem size fixed and increase the number of processors. The asymptotic speedup and efficiency are found to be:

$$S(P) \sim \mathcal{O}(\log_2^{-1} P)$$
$$E(P) \sim \mathcal{O}(P^{-1})$$

Hence, we see that this approach is not scalable as efficiency declines rapidly when the number of processors is large. We proceed now to analyzing the weak scaling behavior which is a more useful metric in most situations.

Weak Scaling

In weak scaling we keep the **problem size per processor** fixed while increasing the size of the problem as well as the number of processors. This is often a more useful test of the usefulness of an algorithm than strong scaling. The memory per processor defined as $M = N^2/P$ is maintained constant.

$$\begin{aligned} M = \frac{N^2}{P} &= constant \\ \implies E(N,P) &= \frac{2}{3}\frac{1}{\left(\frac{\alpha \log_2 P}{N \cdot M} + \frac{\beta}{M} + \gamma\right)} \\ \implies E(N,P) &= \frac{2}{3}\frac{1}{\left(\frac{\alpha \log_2 P}{P^{\frac{1}{2}} \cdot M^{\frac{3}{2}}} + \frac{\beta}{M} + \gamma\right)} \\ \implies E(P) &\sim \mathcal{O}\left(\frac{\sqrt{P}}{\log_2 P}\right) \end{aligned}$$

Which is an increasing function of P. Hence the approach does not scale asymptotically in the weak sense.

3.2.2 Cyclic Mapping

In the cyclic mapping approach illustrated in Figure 2.1, we do not encounter the load-balancing problem. The communication overhead however, doesn't improve as we still need to communicate the pivot row to all other processors in the team. In a team of P processors, communication involves making N broadcasts each taking $\alpha log_2 P$ time and sending N words in βN time. The time complexity of the serial algorithm can be utilized to trivially quote the overall complexity for sufficiently large N and P as:

$$C(n) = \alpha \log_2 P + \beta N + \gamma \frac{2}{3} \frac{N^3}{P}$$

Using the conventional definition of speedup, S and efficiency, E we arrive at the following relations:

$$S(N,P) = \frac{T_{serial}}{T_{parallel}} = \frac{2}{3} \frac{1}{\left(\frac{\alpha log_2 P}{N^3} + \frac{\beta}{N^2} + \frac{2\gamma}{3P}\right)}$$
$$E(N,P) = \frac{S}{P} = \frac{2}{3} \frac{1}{\left(\frac{P\alpha \log_2 P}{N^3} + \frac{P\beta}{N^2} + \frac{2\gamma}{3}\right)}$$

Strong Scaling

The asymptotic speedup and efficiency are found to be the same as in the blocked mapping case.

$$S(P) \sim \mathcal{O}(\log_2^{-1} P)$$
$$E(P) \sim \mathcal{O}(P^{-1})$$

Weak Scaling

Weak scaling behavior is also found to asymptotically mimic the behavior in the blocked mapping case.

$$M = \frac{N^2}{P} = constant$$
$$\implies E(P) \sim \mathcal{O}\left(\frac{\sqrt{P}}{\log_2 P}\right)$$

3.3 Comparison

As we showed in the previous sections, use of different topological schemes does not change the asymptotic scaling behavior of the Gaussian Elimination algorithm. However since cyclic mapping schemes are considerably more difficult to implement, proper justification must be made in their favor. This involves a straightforward comparison of the efficiency of the two algorithms and we find that:

$$E_{cyclic}(N,P) > E_{blocked}(N,P) \quad \forall (N,P) \in Z_+ \times Z_+$$

Chapter 4

Results

A single version of the code was setup for the purposes of testing. The executable was built using MVAPICH2 and the Intel 13 C compiler with level 2 compiler optimization. A second version of the code with support for partial pivoting was also written but was not used for testing. This code can be found in the repository under the `final_pivoting` directory.

4.1 Blocked vs. Cyclic Mapping

As mentioned in Chapter 1, a major challenge with implementing an efficient parallel G-E scheme is ensuring good load balancing between the processors. To analyze this behavior, the program is tested with a variable number of contiguous rows assigned to each processor at a time. Two problems were used to test this behavior, the parameters for which are listed below.

		Values	
Parameter	Name	Problem 1	Problem 2
Number of processors	NPROC	32	64
Number of rows	NROWS	1024	2048
Number of columns	NCOLS	1024	2048
Number of cycles	NREPS	32, 16, 8, 4, 2, 1	32, 16, 8, 4, 2, 1

Here number of cycles $NREPS$ determines how each block of $\frac{NROWS}{NPROC}$ rows is divided among the processor team. A processor gets $\frac{NROWS}{NPROC \cdot NREPS}$ contiguous rows from each block. Consequently, the algorithm with NREPS=1 is an implementation of a pure blocked mapping. The other extreme is attained at $NREPS = \frac{NROWS}{NPROC}$ which means that each processor gets only one

row from the $\frac{NROWS}{NPROC}$ submatrices. Varying $NREPS$ between these two extremes tweaks the granularity of parallelism in our algorithm. The execution times are plotted in 4.1 and the performance gained from a cyclic map is clearly visible. The cyclic mapping with $NREPS = \frac{NROWS}{NPROC}$ is selected as the optimal scheme and this is the topology that is used for all remaining analyses.

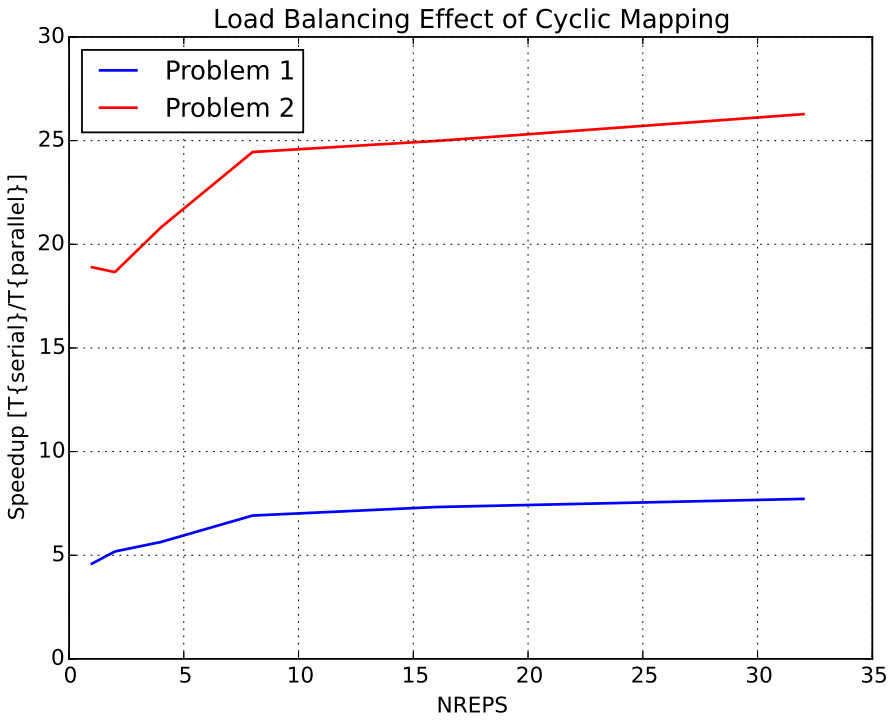

Figure 4.1: NREPS=1 corresponds to a pure Blocked Mapping whereas the other extreme corresponds to a Cyclic Map. It can be seen that performance increases monotonically with increase in NREPS. This is because Cyclic mapping improves the granularity of the parallelism and gives a more equitable workload to all processes.

4.2 Strong Scaling

To analyze the strong scaling behavior of the algorithm, two problems similar to those in the previous section were chosen. For strong scaling, the problem size was kept constant and the number of processors was varied. The MPI executable using collectives is set up in such a way that it can be run on a

single processor. This was used to obtain the serial execution time in both cases. In all cases, the optimal matrix mapping was utilized.

		Values	
Parameter	Name	Problem 1	Problem 2
Number of processors	NPROC	2, 4, 8, 16, 32	2, 4, 8, 16, 32, 64
Number of rows	NROWS	1024	2048
Number of columns	NCOLS	1024	2048
Number of cycles	NREPS	1024/NPROC	2048/NPROC

Table 4.1: Strong scaling analysis parameters

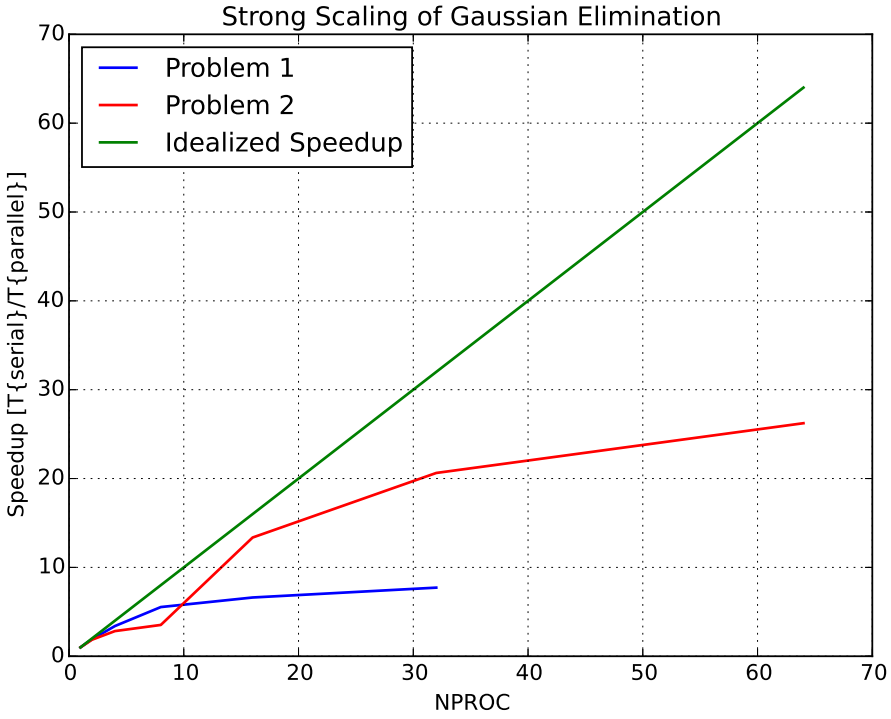

Figure 4.2: The strong scaling behavior of the algorithm is plotted. We see that the speedup drops off significantly as NPROC is increased. This is consistent with the asymptotic estimates that were formulated in Chapter 3.

Expected behavior is observed in the strong scaling analysis and the algorithm performs fairly well. The decline in speedup is a sign of communication time beginning to dominate the program execution time.

4.3 Weak Scaling

The weak scaling performance of the algorithm is measured by keeping $\frac{NROWS \times NCOLS}{NPROC}$ constant while simultaneously increasing the problem size and number of processors. This ensures that in each run, all processors operate on the same amount of memory. For convenience, we choose $NROWS = NCOLS = N$ and increase $NPROCS = P$ as perfect squares. Once again, the optimal mapping is utilized.

$NPROC(P)$	$NROWS = NCOLS(N)$	N^2/P	$NREPS$
1	512	512^2	1
4	1024	512^2	256
16	2048	512^2	128
64	4096	512^2	64
256	8192	512^2	32
1024	16384	512^2	16

Table 4.2: Weak scaling analysis parameters

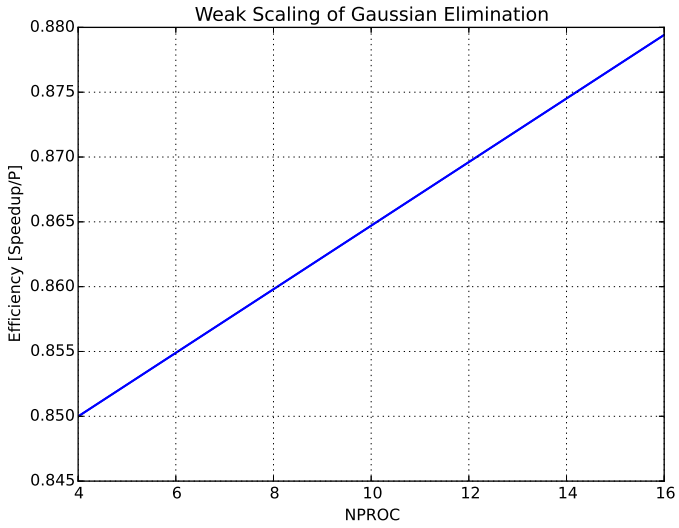

Figure 4.3: The weak scaling behavior of the algorithm is plotted. We see that the efficiency is maintained as the problem size is increased. Due to difficulties with the code and shortage of time, more scaling data could not be collected which makes it difficult to conclude with certainty on the scalability of the approach adopted in this project.

Due to paucity of time I was unable to sort out a bug in the code which resulted in runtime errors when the weak scaling analysis was run with the parameters given in the table above. As a result, a conclusive determination on the weak scaling behavior of the implementation is not possible. Please note that the weak scaling analysis ran successfully for 4 and 16 processors only.

Chapter 5

Conclusions

Providing a parallel algorithm for Gaussian Elimination proves to be a tricky problem due to communication dependencies as well as the load imbalance problem. Our analysis clearly shows how cyclic mapping reduces the load-imbalance significantly; though it is not entirely eliminated. A better mapping technique would be the checkerboard mapping which is also called a 2D block cyclic map in which each processor is assigned a small 2D block of matrix entries. While the implementation is significantly more involved, the payoff is good as load can be balanced even better and more processors can be used than there are rows/columns in the system.

One key observation is the fact that with the use of collectives, the communication overhead for the algorithm is well optimized. Moreover, overlapping communication and computation does not follow naturally which means that no extra performance can be exploited using that technique. Consequently, it becomes self-evident that one method of aggressively pursuing performance improvements in the current framework would be to focus on instruction level parallelism for floating point operations. This would be achieved through the use of optimized libraries like BLAS instead of C-arithmetic. A quick survey of the literature revealed that there has been considerable interest in expressing Gaussian Elimination inner loops as matrix-vector products. This allows algorithms to use BLAS3 matrix-vector multiplication routines which are much faster than BLAS1 and BLAS2 operations.

This implementation provides a good parallel algorithm for Gaussian Elimination. Scaling behavior is reasonable and the code also provides a framework for partial pivoting. However there are still some issues with the implementation such as the one encountered in the weak scaling analysis. With additional testing and by using some of the improvements described here, the

current implementation could be developed into a more robust and scalable parallel implementation of Gaussian Elimination.