Parallel Gaussian Elimination Using MPI

Alessandro Febretti

Introduction

In this project, we had to implement a parallel solver for linear equation systems, using a technique known as *Gaussian elimination (GE).* As with many other algorithms for solving linear equation systems, GE is performed on the matrix representation of the system, A**x** = **b**, where A is the coefficient matrix and **b** is the vector of known values. GE works by applying a set of elementary row operations (swapping rows, multiplying a row by a non-zero number, adding a multiple of one row to another) in order to turn the coefficient matrix into upper triangular form.

The matrix is turned into upper triangular form through an interative process called *forward elimination:* on the first step of this process, we take the first row of the [A|b] matrix and subtract a multiple of it from rows below it, in order to eliminate (i.e. set to zero) all coefficients in column 1. We repeat the process on row 2 to eliminate column 2 coefficients, and so on. An important concern here is to guarantee numerical stability and correctness of the new coefficients on actual implementations with limited number precision. A solution for this is *partial pivoting*: in step i, instead of using row i, we choose a row j>i whose element (j,i) is the biggest in the entire column i. We then exchange rows i and j, and proceed with forward elimination. Partial pivoting guarantees that coefficients used for row elimination are not too extreme. A discussion of the entire algorithm would take too much space here but can be easily found online (http://en.wikipedia.org/wiki/Gaussian_elimination). To recap, pseudocode for sequential forward elimination looks as follows:

```
for k = 1 ... m:
   // Find pivot for column k:
   i_max := argmax (i = k ... m, abs(A[i, k]))
   if A[i_max, k] = 0
     error "Matrix is singular!"
   swap rows(k, i_max)
   Do for all rows below pivot:
   for i = k + 1 ... m:
     // Do for all remaining elements in current row:
     for j = k + 1 ... n:
       A[i, j] := A[i, j] - A[k, j] * (A[i, k] / A[k, k])
     // Fill lower triangular matrix with zeros:
     A[i, k] := 0
```

When the coefficient matrix is upper triangular, the equation in the last row is reduced to a single unknown, and can therefore be used to find the value of x_n, where n is the matrix size. This value can be used to solve row n-1 for x_{n-1}, and so on, in a process called *back substitution*.

1.1. Parallel Gaussian Elimination

The main part of GE that is suitable for parallel execution is forward elimination. If the augmented matrix [A|b] is split between multiple processors, the processor owning the current pivot row can broadcast it to other processors, which then proceed to recalculate their local rows. If we distribute rows in a smart way (for instance, cyclically) we can also limit processor idling. Back substitution can be optimized following a similar principle. If present, reading of the initial data can be optimized too, since all nodes can read in parallel only the rows that they need to access. Pivot row identification is difficult to optimize, since the required message passing usually consumes more time than computing a maximum over a set of elements (at least for systems with just a few hundred or thousand equations).

The general structure of the parallel GE can be similar to this:

```
All nodes:
        Read a subset of input data, [A|b], size N
Forward elimination:
        For each row r[j], j in [1, N]:
                // Find pivot row
                rp = find_pivot(j)
                swap_rows(rp, r[j])

                // eliminate column j
                Node owning r[j]:
                        Broadcast r[j]:
                All nodes:
                        Receive r[j]
                        For each local row r[i], i>j:
                                Recompute row r[i] using r[j] (elem[i, j] will be 0 after this)
Back substitution:
        For each row r[j], j in [N, 1]:
                Node owning r[j]:
                        Compute x[j] using partial solution p[j]
                        Broadcast x[j]
                All nodes:
                        Receive x[j]
                        Compute partial solution p[j] using x[j]…x[N]
End:
        Root node:
                Receive x elements computed by each node, assemble solution vector x.
```

The find_pivot step works as follows:

```
Find_pivot(row j):
        Pj = node owning row j
        All nodes:
                Compute local maximum of column j
                Broadcast local maximum to Pj
        Pj:
                Gather local maximums of column j
                Compute global maximum of column j, MCJ
                Broadcast MCJ and row[j]
                Receive row rp, containing MCJ
                Return rp
        All nodes:
                Receive MCJ, row[j]
                If MCJ == local maximum, send local_maximum_row(j) to Pj
                Swap_rows(local_maximum_row(j), row[j])
```

2. Implementation

The algorithm has been implemented in C, using the Message Passing Interface (MPI) API. Most of the code is a straightforward implementation of the previous pseudocode, mapping communication parts to the MPI Send, Recv, Broadcast and Gather primitives. Here is a more detailed breakdown of communication calls (worst case scenario considered: some send/recv pairs are skipped during execution for local rows).

- PV - find pivot and row swap (per row): 1 gather, 1 broadcast, 2 send/recv pair.
- FE - forward elimination (per row): 1 broadcast
- BS - back substitution (per row): 1 send/recv pair for each other node (number of messages equivalent to 1 broadcast).

Therefore, given a GE run for a NxN matrix on P nodes, the total number of messages passed per phase is:

- PV: $2(P – 1) + 2 = 2P$
- FE: $P - 1$
- BS: $P – 1$

giving a total number of N(4P -2) messages for the worst case scenario. The message passing time can be written as tN(P – 1), where t is the time required for a single message pass. Note that here we are taking into account the message number only, not the message size. Given the small size of exchanged messages for GE vs. bandwidth this is an acceptable approximation.

The number of multiplications/divisions done by the algorithm is O(N^3). An estimate of the execution time of the parallel algorithm can therefore be written as:

$$tN(P-1)+\frac{eN^3}{P}$$

Where t is the previously mentioned message passing time, and e is the time required for a single division multiplication. We can expect t >> e.

Given the previous approximative formula, the speedup can be written as:

$$S=\frac{eN^3}{tN(P-1)+\frac{eN^3}{P}}=\frac{ePN^2}{tP^2-tP+eN^2}$$

If we call k = $\frac{t}{e}$ the transmission-to-execution time ratio (t >> 1) we can rewrite the speedup as

$$S=\frac{PN^2}{kP^2-kP+N^2}$$

We expect speedup optimums (i.e. number of processors that gives us best speedup) to change depending on data size and transmission-to-execution ratio.

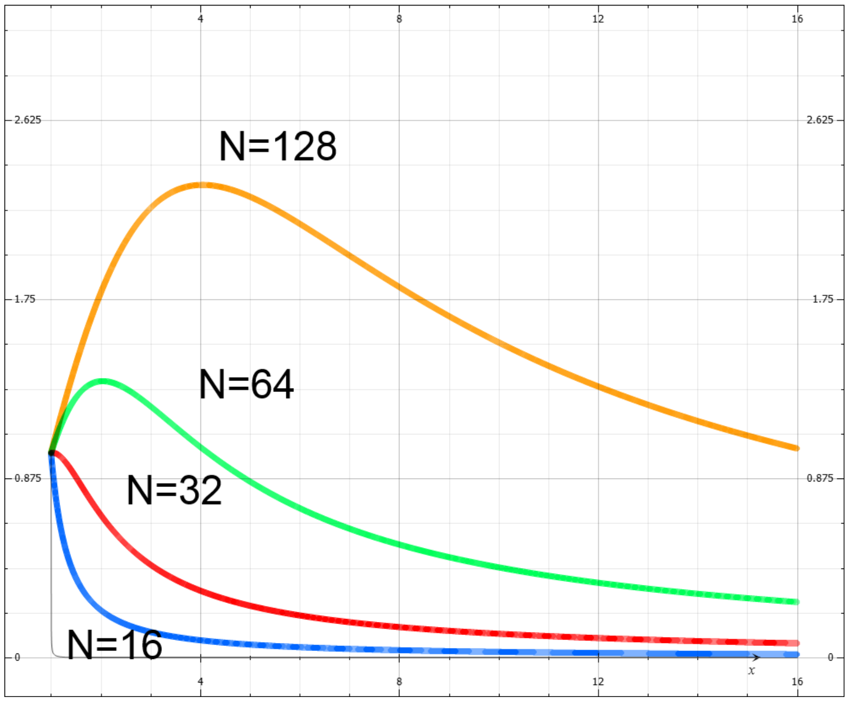


Figure 1. Speedup trends for varying N. For this example, a message transmission is set to be 100x the cost of a division/multiplication (k=100).

To improve parallel execution and avoid idling, parallel GE is implemented with support for block-cyclic distribution of rows: instead of just passing N/P rows to each node, we split rows in smaller blocks and associate them cyclically to nodes. For instance, given a block cyclic factor R = 2, we can split the data into N/RP blocks. If P = 8, blocks would be distributed as follows:

NODE	Node 1		Node 2		Node 3		Node 4		Node 5		Node 6		Node 7		Node 8	
BLOCK	B1	B9	B2	B10	B3	B11	B4	B12	B5	B13	B6	B14	B7	B15	B8	B16

To simplify implementation of block cyclic distribution, I added three helper functions to my code:

- **`int getRowPID(j):`** returns the id of the node 'owning' row j.
- **`int getLocalRowID(j) :`** given the absolute index of a row, returns its index (or offset) from the local row buffer.
- **`double* getLocalRow(j, localRowBuffer):`** using the previous two functions, retrieves row[j] from the local row buffer. If row j is not owned by this node, returns a NULL pointer.

`getRowPID` and **`getLocalRowID`** use the matrix size N, the number of nodes P, and the block cyclic factor to compute row and node indices as follows:

```
getRowPID(j):
        // compute total number of blocks
        b = RP
        // compute rows per block
        r = N / b
        // find block containing row j
        bj = j / r
        // returns the ID of node containing block bj
        Return mod(bj, P)

getLocalRowId(j):
        // compute total number of blocks
        b = RP
        // compute rows per block
        r = N / b
        // find block containing row j
        bj = j / r
        // find the offset of row j in block
        rbj = mod(j, r)
        // find the local index of block bj
        lbj = bj / P
        // return full row offset
        Return lbj * r + rbj
```

2.1. Data Access and Distribution

Initial data access (i.e. loading in the [A|b] matrix) has been implemented in two ways:

- Distributed read: all nodes open the input file and seek/read rows they own. **`getRowPID`** and **`getLocalRowID`** are used to implement the process efficiently.
- Read and scatter: the head node reads the entire [A|b] matrix and distributes it to nodes. To deal with block cyclic row allocation and avoid useless message passing, the head node rearranges the matrix rows to reflect their distribution, before sending them to nodes. This rearrangement makes all rows assigned to a node contigious, so the entire matrix can be distributed using a single `MPI_Scatter` call.

3. Data Collection

The parallel GE program measured execution times for its for main components: data distribution (DD), Forward Elimination (FE), Pivot row identification (PV), Back substitution (BS). Note that PV is part of FE. Also, measured times for DD and BS **include** the forward elimination time FE. So the full breakdown of times is as follows:

- DD: measures time to read / scatter data and to perform forward elimination with pivoting
- FE: measures time to perform forward elimination with pivoting
- BS: measures time to perform forward elimination with pivoting, followed by back substitution
- PV: measures time to perform pivot row identification and exchange only.

These times have been collected for sequential and parallel runs of the algorithm to generate corresponding speedups. The program has been run varying 4 major parameters: N, P, R and data distribution mode. For N, P, R, the following parameter values have been tested:

- N: 128, 256, 512, 1024, 2048. Matrices were composed of random floating point numbers and were checked for non-singularity.
- P: 1, 2, 4, 8, 16. Processes have been allocated using local cores when possible (up to 4 per machine)
- R: 1, 2, 4, 8, 16, 32, 64, 128. Since N needs to be divisible by RP, not all values of R have been tested for each matrix size.

The data was output as separate files for each run, and as a global comma-separated values (csv) file storing times and speedups for all runs. I collected data for 844 runs in total.

3.1. Analysis

The generated csv file has been imported in Excel and used to generate a set of pivot charts to support data analysis. Pivot charts are particularly useful in this scenario since they allow to quickly aggregate datapoints using various functions (min, max, average, count), and to easily switch chart dimensions.

4. Results

4.1. Global Speedup

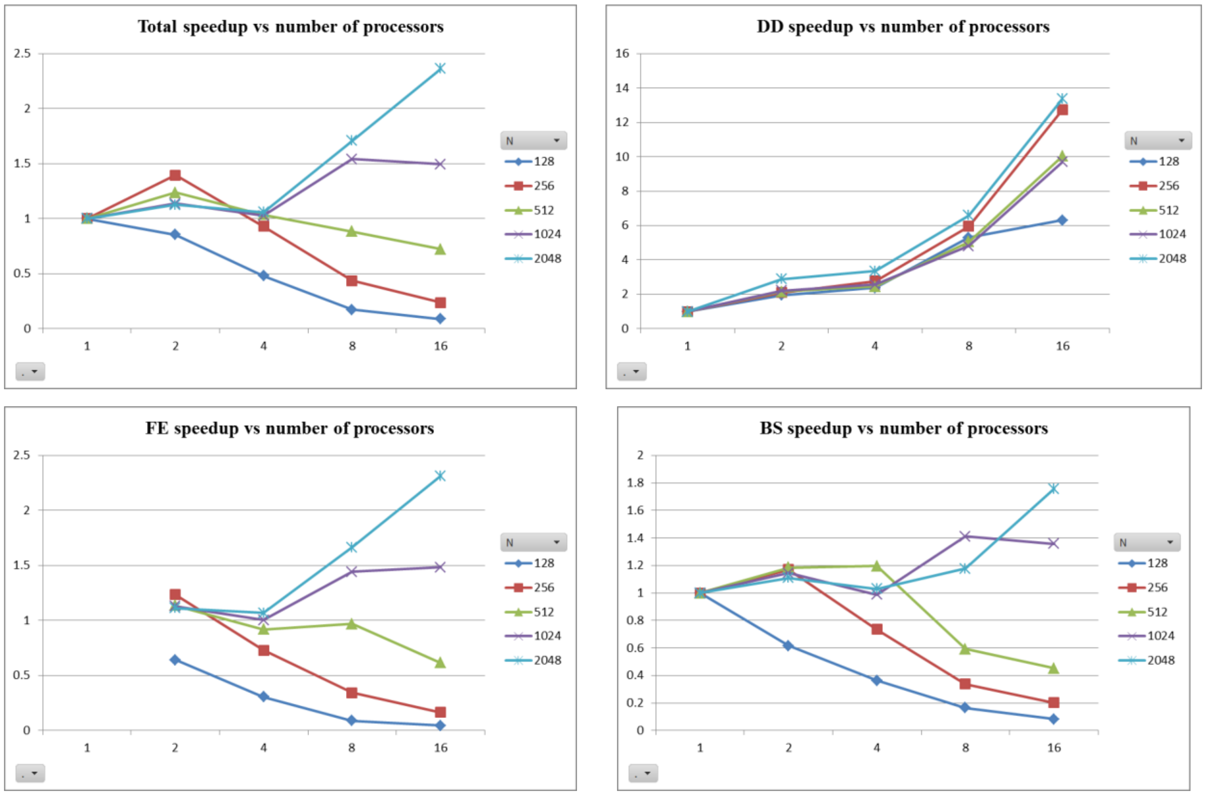


Figure 2 . Speedup trends for different input matrix sizes. Trends show maximum speedup achieved per datapoint (considering optimal R and dat adistribution mode).

Figure 2 shows speedup trends for different parts of the GE algorithm. For small N, message passing cost dominates computation: for N=128 parallelization seems to be always inconvenient, while with N=256 a positive speedup is achieved only for a 2 processor parallelization. Interestingly, N=512,1024,2048 show very similar speedups on P=2,4. This may be due to the tested cyclic distribution variations being more efficient on specific data sizes. For N=1024, speedup appear to reach an optimum at 8 processors, while for N=2048 speedup is still increasing at 16 processors, suggesting that increasing parallelization would still be advantageous. Data distribution speedup trends are pretty much consistent across data sizes and are quite big: Altough not shown in the graph, these speedups correspond to the distributed read version of the data read mode. With some minor variation, FE and BS speedups are consistent with the global trend.

4.2. Data distribution

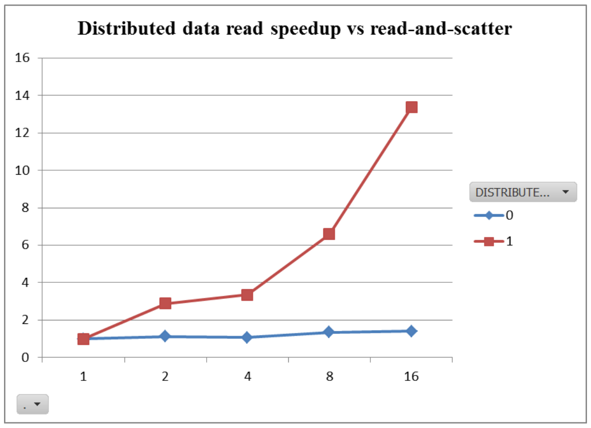
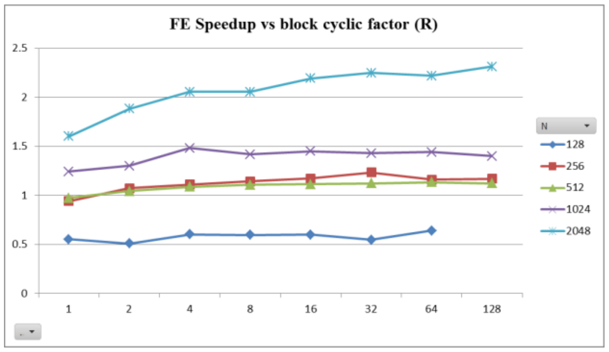


Figure 3. Distributed data read speedup almost matches P.

To confirm the DD speedup observations made in the previous section, we can plot the speedup trends for both data distribution modes. In Figure 3 it is clear how distributing reads is much more efficient than doing a read-and-scatter from a single node. In fact, the speedup of distributed read almost patches the number of processors, underscoring how data access is highly parallelizable (given enough bandwidth).

4.3. Block Cyclic Factor



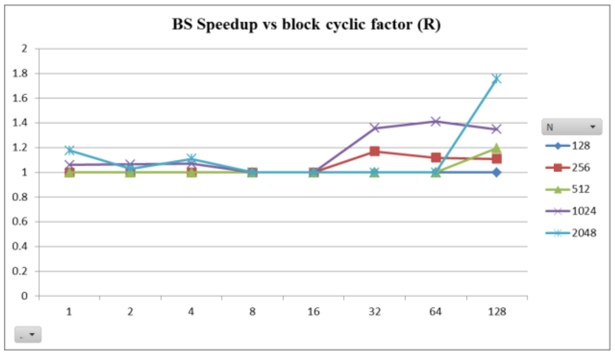


Figure 4. Block cyclic factor speedup trends for Forward Elimination (FE) and Backward Substitution (BS).

Figure 4 shows how the block cyclic factor R has a major effect only on bigger matrix sizes (note: here like in previous figures we are considering best speedups, so for each data point we show the speedup for the best combination of R and number of processors). For smaller systems, R has a smaller effect on both forward elimination and back substitution. FE speedups show linear trends on varying R, while speedups for back substitution look less predictable and are probably influenced more by general message passing patterns.