Lecture Notes 8: Gaussian Elimination—Sequential and Basic Parallel Algorithms

Shantanu Dutt
ECE Dept., UIC
Acknowledgements

• Parallel Implementations of Gaussian Elimination, Vasilije Perovi´c, Western Michigan University
• Parallel Gaussian Elimination, Univ. of Frankfurt
Parallel Implementations of Gaussian Elimination

Vasilije Perović

Western Michigan University

vasilije.perovic@wmich.edu

January 27, 2012
Linear systems of equations

General form of a linear system of equations is given by

\[
\begin{align*}
 a_{11}x_1 & + \cdots + a_{1n}x_n = b_1 \\
a_{21}x_1 & + \cdots + a_{2n}x_n = b_2 \\
& \vdots \quad \vdots \\
a_{m1}x_1 & + \cdots + a_{mn}x_n = b_m \\
\end{align*}
\]

where \( a_{ij} \)'s and \( b_i \)'s are known and we are solving for \( x_i \)'s.
More compactly, we can rewrite system of linear equations in the form

\[ \mathbf{Ax} = \mathbf{b} \]

where

\[ A = \begin{bmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & a_{22} & \cdots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix}
    b_1 \\
    b_2 \\
    \vdots \\
    b_m
\end{bmatrix} \]
Why study solutions of $Ax = b$

1. One of the most fundamental problems in the field of scientific computing
2. Arises in many applications:
Why study solutions of $Ax = b$

1. One of the most fundamental problems in the field of scientific computing
2. Arises in many applications:
   - Chemical engineering
   - Interpolation
   - Structural analysis
   - Regression Analysis
   - Numerical ODEs and PDEs
Methods for solving $Ax = b$

1. Direct methods
2. Iterative methods
Methods for solving $Ax = b$

1. **Direct methods** – obtain the exact solution (in real arithmetic) in finitely many operations

2. **Iterative methods** – generate sequence of approximations that converge in the limit to the solution
Methods for solving $Ax = b$

1. **Direct methods** – obtain the exact solution (in real arithmetic) in finitely many operations
   - Gaussian elimination ($LU$ factorization)
   - $QR$ factorization
   - $WZ$ factorization

2. **Iterative methods** – generate sequence of approximations that converge in the limit to the solution
   - Jacobi iteration
   - Gauss-Seidal iteration
   - SOR method (successive over-relaxation)
Methods for solving $Ax = b$

1. *Direct methods* – obtain the exact solution (in real arithmetic) in finitely many operations
   - Gaussian elimination (*LU factorization*)
   - QR factorization
   - WZ factorization

2. *Iterative methods* – generate sequence of approximations that converge in the limit to the solution
   - Jacobi iteration
   - Gauss-Seidel iteration
   - SOR method (successive over-relaxation)
When solving $Ax = b$ we will assume throughout this presentation that $A$ is non-singular and $A$ and $b$ are known

\[
\begin{align*}
    a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\
    a_{21}x_1 + \cdots + a_{2n}x_n &= b_2 \\
    \vdots & \quad \vdots \\
    a_{n1}x_1 + \cdots + a_{nn}x_n &= b_n .
\end{align*}
\]
Gaussian Elimination

Assuming that $a_{11} \neq 0$ we first subtract $a_{21}/a_{11}$ times the first equation from the second equation to eliminate the coefficient $x_1$ in the second equation, and so on until the coefficients of $x_1$ in the last $n-1$ rows have all been eliminated. This gives the modified system of equations

$$
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1
$$

$$
a_{22}x_2 + \cdots + a_{2n}x_n = b_2
$$

$$
a_{32}x_2 + \cdots + a_{3n}x_n = b_3
$$

$$
a_{42}x_2 + \cdots + a_{4n}x_n = b_4
$$

$$
a_{n-1,2}x_2 + \cdots + a_{n-1,n}x_n = b_{n-1}
$$

$$
a_{n,n-1}x_{n-1} + a_{nn}x_n = b_n,
$$
Assuming that $a_{11} \neq 0$ we first subtract $a_{21}/a_{11}$ times the first equation from the second equation to eliminate the coefficient $x_1$ in the second equation, and so on until the coefficients of $x_1$ in the last $n - 1$ rows have all been eliminated. This gives the modified system of equations

\[
\begin{align*}
    a_{11}x_1 + & \quad a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
    a_{22}x_2 + & \quad \cdots + a_{2n}x_n = b_2^{(1)} \\
    \vdots & \quad \vdots \\
    a_{n1}x_1 + & \quad \cdots + a_{nn}x_n = b_n^{(1)},
\end{align*}
\]

where

\[
\begin{align*}
    a_{ij}^{(1)} &= a_{ij} - a_{1j} \frac{a_{i1}}{a_{11}} \\
    b_i^{(1)} &= b_i - b_1 \frac{a_{i1}}{a_{11}}
\end{align*}
\]
i, j = 1, 2, \ldots, n.
Gaussian Elimination \textit{(Forward Reduction)}

Applying the same process the last $n-1$ equations of the modified system to eliminate coefficients of $x_2$ in the last $n-2$ equations, and so on, until the entire system has been reduced to the \textit{(upper)} triangular form

\[
\begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
a_{22} & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\
  \vdots & \vdots & \ddots & \vdots \\
a_{n-1}^{(n-1)} & \cdots & a_{n-1}^{(n)} & a_{nn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
x_2 \\
  \vdots \\
x_n
\end{bmatrix}
= \begin{bmatrix}
  b_1 \\
b_2^{(1)} \\
  \vdots \\
b_n^{(n-1)}
\end{bmatrix}.
\]
Gaussian Elimination (Forward Reduction)

Applying the same process the last \( n - 1 \) equations of the modified system to eliminate coefficients of \( x_2 \) in the last \( n - 2 \) equations, and so on, until the entire system has been reduced to the (upper) triangular form

\[
\begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{22} & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{nn}^{(n-1)} & \cdots & a_{nn}^{(n-1)} & a_{nn}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_n
\end{bmatrix}
= \begin{bmatrix}
b_1 \\
b_2^{(1)} \\
\vdots \\
b_n^{(n-1)}
\end{bmatrix}.
\]

• The superscripts indicate the number of times the elements had to be changed.
Gaussian Elimination – Example

• Perform the \textit{forward reduction} the the system given below

\[
\begin{bmatrix}
4 & -9 & 2 \\
2 & -4 & 4 \\
-1 & 2 & 2
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix} =
\begin{bmatrix}
2 \\
3 \\
1
\end{bmatrix}
\]
Gaussian Elimination – Example

• Perform the *forward reduction* the the system given below

\[
\begin{bmatrix}
4 & -9 & 2 \\
2 & -4 & 4 \\
-1 & 2 & 2
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
= 
\begin{bmatrix}
2 \\
3 \\
1
\end{bmatrix}
\]

We start by writing down the augmented matrix for the given system:

\[
\begin{bmatrix}
4 & -9 & 2 & 2 \\
2 & -4 & 4 & 3 \\
-1 & 2 & 2 & 1
\end{bmatrix}
\]
Gaussian Elimination – Example

We perform appropriate row operations to obtain:

\[
\begin{bmatrix}
4 & -9 & 2 & | & 2 \\
2 & -4 & 4 & | & 3 \\
-1 & 2 & 2 & | & 1
\end{bmatrix}
\longrightarrow
\]
Gaussian Elimination – Example

We perform appropriate row operations to obtain:

\[
\begin{bmatrix}
4 & -9 & 2 & 2 \\
2 & -4 & 4 & 3 \\
-1 & 2 & 2 & 1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
4 & -9 & 2 & 2 \\
0 & 0.5 & 3 & 2 \\
0 & -0.25 & 2.5 & 1.5
\end{bmatrix}
\]
We perform appropriate row operations to obtain:

\[
\begin{bmatrix}
4 & -9 & 2 & 2 \\
2 & -4 & 4 & 3 \\
-1 & 2 & 2 & 1
\end{bmatrix}
\]

\[R_2 - \left( \frac{2}{4} \right) R_1 \rightarrow R_2\]

\[R_3 - \left( \frac{-1}{4} \right) R_1 \rightarrow R_3\]

\[R_3 - \left( \frac{-0.25}{0.5} \right) R_2 \rightarrow R_3\]

\[
\begin{bmatrix}
4 & -9 & 2 & 2 \\
0 & 0.5 & 3 & 2 \\
0 & 0 & 4 & 2.5
\end{bmatrix}
\]
Gaussian Elimination – Example

Note that the row operations used to eliminate $x_1$ from the second and the third equations are equivalent to multiplying \textit{on the left} the augmented matrix:

\[
\begin{bmatrix}
1 & 0 & 0 \\
-0.5 & 1 & 0 \\
0.25 & 0 & 1 \\
\end{bmatrix}
\cdot
\begin{bmatrix}
4 & -9 & 2 & 2 \\
2 & -4 & 4 & 3 \\
-1 & 2 & 2 & 1 \\
\end{bmatrix}
= 
\begin{bmatrix}
4 & -9 & 2 & 2 \\
0 & 0.5 & 3 & 2 \\
0 & -0.25 & 2.5 & 1.5 \\
\end{bmatrix}
\]
Gaussian Elimination – Example

Note that the row operations used to eliminate $x_1$ from the second and the third equations are equivalent to multiplying on the left the augmented matrix:

$$
\begin{bmatrix}
1 & 0 & 0 \\
-0.5 & 1 & 0 \\
0.25 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
4 & -9 & 2 & 2 \\
2 & -4 & 4 & 3 \\
-1 & 2 & 2 & 1
\end{bmatrix}
= 
\begin{bmatrix}
4 & -9 & 2 & 2 \\
0 & 0.5 & 3 & 2 \\
0 & -0.25 & 2.5 & 1.5
\end{bmatrix}
$$
Note that the row operations used to eliminate $x_1$ from the second and the third equations are equivalent to multiplying on the left the augmented matrix:

$$
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0.5 & 1
\end{bmatrix}
\cdot
\begin{bmatrix}
1 & 0 & 0 \\
-0.5 & 1 & 0 \\
0.25 & 0 & 1
\end{bmatrix}
\cdot
[A|b] =
\begin{bmatrix}
4 & -9 & 2 & 2 \\
0 & 0.5 & 3 & 2 \\
0 & 0 & 4 & 2.5
\end{bmatrix}
$$
Gaussian Elimination – Example

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0.5 & 1
\end{bmatrix}
\cdot
\begin{bmatrix}
1 & 0 & 0 \\
-0.5 & 1 & 0 \\
0.25 & 0 & 1
\end{bmatrix}
\cdot
[A|b] =
\begin{bmatrix}
4 & -9 & 2 & 2 \\
0 & 0.5 & 3 & 2 \\
0 & 0 & 4 & 2.5
\end{bmatrix}
\]

Thus, we can write

\[(L_2 \cdot L_1) \cdot A = \hat{L} \cdot A = U\]
LU factorization

In general, we can think of row operations as multiplying with matrices on the left, thus the $i^{th}$ elimination step is equivalent as multiplication on the left by

\[
L_i = \begin{bmatrix}
1 & & & & \\
& \ddots & & & \\
& & 1 & & \\
& & -l_{i+1,i} & \ddots & \\
& & & \ddots & \\
& & & & -l_{n,i}
\end{bmatrix}.
\]
Continuing in this fashion we obtain

\[ \hat{L}A\hat{x} = \hat{L}b, \quad \hat{L} = L_{n-1} \cdots L_2 L_1, \]
Continuing in this fashion we obtain

\[ \hat{L}A\hat{x} = \hat{L}b, \quad \hat{L} = L_{n-1} \cdots L_2 L_1, \]

\[ A = \hat{L}^{-1}U = LU. \]
LU factorization

Continuing in this fashion we obtain

\[ A = \hat{L}^{-1} U = LU. \]

Thus, the Gaussian elimination algorithm for solving \( Ax = b \) is mathematically equivalent to the three-step process:

1. Factor \( A = LU \)
2. Solve (forward substitution) \( Ly = b \)
3. Solve (back substitution) \( Ux = y \).
Continuing in this fashion we obtain

\[ A = \hat{L}^{-1}U = LU. \]

Thus, the Gaussian elimination algorithm for solving \( Ax = b \) is mathematically equivalent to the three-step process:

1. Factor \( A = LU \)
2. Solve (forward substitution) \( Ly = b \)
3. Solve (back substitution) \( Ux = y \).

- Order of operations:
  - \( \frac{n^3}{3} + n^2 - \frac{n}{3} \) multiplications/divisions
  - \( \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6} \).
Need for partial pivoting

Apply LU factorization *without pivoting* to

\[
A = \begin{bmatrix}
0.0001 & 1 \\
1 & 1
\end{bmatrix}
\]

in *three-decimal-digit* floating point arithmetic.
Need for partial pivoting

\[ A = \begin{bmatrix} 0.0001 & 1 \\ 1 & 1 \end{bmatrix} \]

**Solution:** \( L \) and \( U \) are easily obtainable and are given by:

\[
L = \begin{bmatrix} 1 & 0 \\ f\ell(1/10^{-4}) & 1 \end{bmatrix}, \quad f\ell(1/10^{-4}) \text{ rounds to } 10^4,
\]

\[
U = \begin{bmatrix} 10^{-4} & 1 \\ 0 & f\ell(1 - 10^4 \cdot 1) \end{bmatrix}, \quad f\ell(1 - 10^4 \cdot 1) \text{ rounds to } -10^4
\]

so \( LU = \begin{bmatrix} 1 & 0 \\ 10^4 & 1 \end{bmatrix} \begin{bmatrix} 10^{-4} & 1 \\ 0 & -10^4 \end{bmatrix} = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 0 \end{bmatrix} \)

but \( A = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix} \).
Need for partial pivoting

\[ A = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix} \quad \text{but} \quad LU = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 0 \end{bmatrix} \]

**Remark 1:** Note that original \( a_{22} \) has been entirely “lost” from the computation by subtracting \( 10^4 \) from it. Thus if we were to use this \( LU \) factorization in order to solve a system there would be no way to guarantee an accurate answer. This is called *numerical instability*.
Need for partial pivoting

\[ A = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix} \quad \text{but} \quad LU = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 0 \end{bmatrix} \]

**Remark 1:** Note that original \( a_{22} \) has been entirely “lost” from the computation by subtracting \( 10^4 \) from it. Thus if we were to use this \( LU \) factorization in order to solve a system there would be no way to guarantee an accurate answer. This is called *numerical instability*.

**Remark 2:** Suppose we attempted to solve system \( Ax = [1, 2]^T \) for \( x \) using this \( LU \) decomposition. The correct answer is \( x \approx [1, 1]^T \). Instead, if we were to stick with the three-digit-floating point arithmetic we would get an answer \( \hat{x} = [0, 1]^T \) which is completely erroneous.
for (k = 0; k < n-1; k++) { /* Forward elimination */
    r = max_col(a,k);
    if (k != r) exchange_row(a,b,r,k);
    for (i=k+1; i < n; i++) {
        l[i] = a[i][k]/a[k][k];
        for (j=k+1; j < n; j++)
            a[i][j] = a[i][j] - l[i] * a[k][j];
        b[i] = b[i] - l[i] * b[k];
    }
}
for (k = n-1; k >= 0; k--) { /* Backward substitution */
    sum = 0.0;
    for (j=k+1; j < n; j++)
        sum = sum + a[k][j] * x[j];
    x[k] = 1/a[k][k] * (b[k] - sum);
}
return x;
Row-Oriented Algorithm

1. Determination of the local pivot element,
2. Determination of the global pivot element,
3. Exchange of the pivot row,
4. Distribution of the pivot row,
5. Computation of the elimination factors,
Remark 1: The computation of the solution vector $\mathbf{x}$ in the backward substitution is inherently serial, since the values of $x_k$ depend on each other and are computed one after another. Thus, in step $k$ the processor $P_q$ owning row $k$ computes the value of $x_k$ and sends the value to all other processors by a single broadcast operation.
Row-Oriented Algorithm

Remark 1: The computation of the solution vector $x$ in the backward substitution is inherently serial, since the values of $x_k$ depend on each other and are computed one after another. Thus, in step $k$ the processor $P_q$ owning row $k$ computers the value of $x_k$ and sends the value to all other processors by a single broadcast operation.

Remark 2: Note that the data distribution is not quite adequate. For example, suppose we are at the $i^{th}$ step and that $i > m \cdot n/p$ where $m$ is a natural number. In that case, processors $p_0, p_1, \ldots, p_{m-1}$ are idle since all their work is done until the back substitution step at the very end. Hence there is an issue with load balancing.
Remark 2: Note that the data distribution is not quite adequate. For example, suppose we are at the $i^{th}$ step and that $i > m \cdot n/p$ where $m$ is a natural number. In that case, processors $p_0, p_1, \ldots, p_{m-1}$ are idle since all their work is done until the back substitution step at the very end. Hence there is an issue with load balancing.

*block-cyclic row distribution*
Remark 3: We could also consider column-oriented data distribution. In that case, at the $k^{th}$ step the processor that owns $k^{th}$ column would have all needed data to compute the new pivot. On the one hand, this would reduce communication between processors that we had when considering row-oriented data distribution. On the other hand, with column orientation pivot determination is all done serially. Thus, in case when $n >> p$ (which is often case) row oriented data distribution might be more advantageous since the pivot determination is done in parallel.
Parallel Gaussian Elimination
Gaussian Elimination

- We work with the rowwise decomposition:
  - Each processor receives \( n/p \) rows of the extended matrix \((A,b)\)

- We parallelize the individual pivoting steps
  - In step \( i \) we find the maximum \( A(j,i) \) for \( i \leq j \leq n \)
  - Swap row \( j \) with row \( i \)
  - Broadcast row \( i \) to processors holding rows \( i+1 \ldots n \)
  - Each processor computes the new row \( j = \text{row}_j - \frac{A(j,i)}{A(i,i)} \cdot \text{row}_i \)
Gaussian Elimination

- Time: Finding the maximum \( A(j,i) \) and \( j \) can be done by tournaments in time \( O(n/p + \log p) \)

- Broadcasting the pivot row takes communication time \( O(n \cdot \log p) \) (or \( O(nP) \) for a linear array)

- Computation time for the remaining operations is \( O(n^2/p) \) per phase

- In \( n \) phases the computation time is thus \( O(n^3/p) \) and the communication time is \( O(n^2 \log p) \) (or \( O(n^2P) \) for a linear array)

- Hence the algorithm has constant efficiency once \( n = \Omega(p \log p) \)
Gaussian Elimination

- The disadvantage of this algorithm is that during the broadcasting step no computations take place, so effectively processors are idle during these communications, for time $O(n^2p)$ for a linear array and $O(n^2 \log P)$ for a hypercube.
- Need to independently reduce comm. time, as due to much larger constants than computation time, it can dominate or at least significantly affect parallel time.
A different algorithm

- Again we work with the rowwise decomposition
- In the previous algorithm we were using tournaments to find the maximum $A(j,i)$
- Consequently we couldn’t predict the pivot row $j$ ahead of time
- And must then broadcast that row
A different algorithm

- Again we parallelize the pivoting step
- This time we always use row i in phase i
- Find the maximum A(i,j)
- We then want to eliminate all A(k,j) for k>i
- When we do this for i=1,...,n the resulting matrix is a columnwise permuted triangular matrix
- So we can afterwards solve via backsubstitution

Based on the fact that a pivot is picked from every row, except one, exactly once
The big advantage is that now we can organize computation and communication more efficiently:

- Processors are arranged as a linear array.
- We know that row i is the pivoting row.
- The processor holding row i sends row i to its nearest neighbor in the array.
- Every processor immediately passes the row on after reception.
- Then starts to replace row_k = row_k - A(k,j)/A(i,j) \cdot row_i for all rows k in its possession.

So a new set of col elim. + row operations done every n + n^2/P time units (of which n time units is idling due to commun. delay) after an initial wait for n(P-1) time unit. When pivot oper. is included for its rows, computation done every 2n + n^2/P time units (idling is again n time units). This is the worst-case stage delay.

Thus worst case stage delay is 2n + n^2/P.

So total time = n(P-1) + (n-1)(2n + n^2/P) \Rightarrow Comp. time = \Theta(n^3/P); Commun. time is only \Theta(nP + n^2) [initial pipeline fill time due to commun. only, and subsequent commun. delay per stage of n (causing idling) \Rightarrow n(P-1)+(n-1)n time
A different algorithm

- **Time:** The computation time is again $O(n^3/p)$
- **The communication time:**
  - The last processor can start working after $(p-1)n$ communication time
  - However, this delay happens only once, when the computation time $n^2/p$ per phase exceeds $p n$, i.e., if $n=\Omega(p^2)$, because
  - In that case the overall extra communication time is only $O(p n)$
  - This effect is called pipelining

So total time = $n(P-1) + (n-1)(2n + n^2/p)$
$\implies \text{Comp. time} = \Theta(n^3/P)$; \text{Commun. time is only } \Theta(n P + n^2) \text{ [initial pipeline fill time due to commun. only, and subsequent commun. delay per stage of } n \text{ (causing idling)} \implies n(P-1)+(n-1)n$ time