Assignment
Solving System of Linear Equations Using MPI

Phạm Trần Vũ
Assignment (1)

- Develop an MPI program to solve system of linear equations using MPI

- Requirements:
  - The program must be able to solve various systems with different numbers of variables
  - Parallelization strategy must be able to run on different numbers of processors

- Due date: 31 May 2010
Assignment 1 (2)

- Submission:
  - Report on:
    » parallelization strategy used in program
    » Theoretical speed up of the strategy used in program (ignore the cost of message passing)
    » Practical speed up measured by experiments on the MPI program and the sequential version
    » Calculation of theoretical and practical efficiency
  - Source code of the program
  - Demonstration of the program in the lab
System of Linear Equations

- A general linear system of $m$ equations and $n$ unknown variables

\[
\begin{align*}
  a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
  a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
  \vdots \quad \vdots \quad \vdots \quad \vdots \\
  a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m
\end{align*}
\]

- Usually expressed as $Ax = b$, where

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

- We are interested in systems with $n$ equations and $n$ unknown variables ($m=n$)
Solving Systems of Linear Equations

- Solution of a linear system is an assignment of values to variables $x_1, x_2, \ldots, x_n$ that satisfies the system.

- Two classes of methods for solving linear systems:
  - Direct
    - Backward substitutions
    - Gaussian elimination algorithm
  - Indirect
    - By approximation
    - Jacobi algorithm
Backward Substitution

- Used to solve the system $Ax = b$ where $A$ is a upper triangular matrix

- Example
  
  \[
  \begin{align*}
  x_1 + x_2 - x_3 + 4x_4 &= 8 \\
  -2x_2 - 3x_3 + x_4 &= 5 \\
  2x_3 - 3x_4 &= 0 \\
  2x_4 &= 4
  \end{align*}
  \]

- The time to solve a linear system using backward substitution is $O(n^2)$
Backward Substitution Algorithm

n: size of system
a[1..n][1..n]: matrix A
b[1..n]: vector b
x[1..n]: vector x

begin
    for i = n down to 1 do
        x[i] = b[i]/a[i][i]
        for j = 1 to i – 1 do
            b[j] = b[j] – x[i]*a[j][i]
        end for
    end for
end
Parallelizing Backward Substitution(1)

\[ \begin{align*}
1x_1 + 1x_2 - 1x_3 + 4x_4 &= 8 \\
-2x_2 - 3x_3 + 1x_4 &= 5 \\
2x_3 - 3x_4 &= 0 \\
2x_4 &= 4 \\
\end{align*} \]

begin
for i = n down to 1 do
\[ x[i] = b[i]/a[i][i] \]
for j = 1 to i - 1 do
\[ b[j] = b[j] - x[i]*a[j][i] \]
end for
end for
end
A processor can be assigned with a number of equations.

Once a variable is solved, it is broadcasted to other processors to calculate unsolved variables.

A good parallelization strategy is the one that can divide the load on each processor equally and reduce the overhead of message passing.
Gaussian Elimination (1)

- Reduce a general $Ax = b$ system to $Tx = c$ system, where $T$ is an upper triangular matrix.
- Using principle: a row can be replaced by the sum of that row and a non-zero multiple of any row of the system.
- The selected row for multiplication is called a pivot row.
- Then, apply Backward substitution algorithm to solve the system.

Example:

\[
\begin{align*}
  x_1 + 2x_2 + 2x_3 &= 2 & L_1 \\
  x_1 + 3x_2 - 2x_3 &= -1 & L_2 \\
  3x_1 + 5x_2 + 8x_3 &= 8 & L_3
\end{align*}
\]
Gaussian Elimination (2)

- Original system

\[
\begin{align*}
    x_1 + 2x_2 + 2x_3 &= 2 & L_1 \\
    x_1 + 3x_2 - 2x_3 &= -1 & L_2 \\
    3x_1 + 5x_2 + 8x_3 &= 8 & L_3
\end{align*}
\]

- Step 1

\[
\begin{align*}
    x_1 + 2x_2 + 2x_3 &= 2 & L_1 \\
    x_2 - 4x_3 &= -3 & L_2 \leftarrow L_2 - L_1 \\
    -x_2 + 2x_3 &= 2 & L_3 \leftarrow L_3 - 3L_1
\end{align*}
\]

- Step 2

\[
\begin{align*}
    x_1 + 2x_2 + 2x_3 &= 2 & L_1 \\
    x_2 - 4x_3 &= -3 & L_2 \\
    -2x_3 &= -1 & L_3 \leftarrow L_3 + L_2
\end{align*}
\]
Gaussian Elimination (3)

- Complexity of Gaussian Elimination is $O(n^3)$
- To have good numerical stability, partial pivoting is used
  - At step $i$ (drive to zero all nonezero values of column $i$ of rows below row $i$).
  - Select the row from row $i$ upward that has the largest absolute value at column $i$
  - Swap selected row with row $i$
i := 1
j := 1

while (i ≤ n and j ≤ n) do
    Find pivot in column j, starting in row i:
    maxi := i
    for k := i+1 to n do
        if abs(A[k,j]) > abs(A[maxi,j]) then
            maxi := k
        end if
    end for
    if A[maxi,j] ≠ 0 then
        swap rows i and maxi, but do not change the value of i
        divide each entry in row i by A[i,j]
        for u := i+1 to n do
            subtract A[u,j] * row i from row u
        end for
        i := i + 1
    end if
    j := j + 1
end while
Parallelize Gaussian Elimination

- Each processor can be assigned with a number of rows of the system
- If partial pivoting is used
  - The selection of pivoting row has to be done across processors
  - The pivot row needs to be broadcasted to all other processors
- Assignment of rows to processors should be done in a way that backward substitution algorithm can be used straight away without re-allocating the work
Jacobi Algorithm

- An iterative method by estimating the values of variables after a number of iterations.
- At iterative $t + 1$, variable $x_i$ is estimated by the following equation:

$$x_i(t+1) = \frac{1}{a_{i,i}} \left( b_i - \sum_{i \neq j} a_{i,j} x_i(t) \right)$$

- Stop iterating when the greatest difference of newly estimated values of variables and the old values is smaller than some threshold.
- If the calculation does not converge, there is no solution found.
Sequential Implementation of Jacobi Algorithm (1)

- **Input**
  - $n$: size of the system
  - $\epsilon$: convergence threshold
  - $a[1..n][1..n]$: matrix $A$
  - $b[1..n]$: vector $b$

- **Output**
  - $x[1..n]$: old estimate of solution vector
  - $\text{newx}[1..n]$: new estimate of solution vector
  - $\text{diff}$: maximum difference after one iteration
Sequential Implementation of Jacobi Algorithm (2)

begin
  for i=1 to n do
    x[i] = b[i]/a[i][i] //initial estimation
  end for
  do
    diff = 0
    for i=1 to n do
      newx[i] = b[i]
      for j =1 to n do
        if j !=i then
          newx[i] = newx[i] – a[i][j]*x[j]
        end if
      end for
      newx[i] = newx[i]/a[i][i]
    end for
    for i=1 to n do
      diff = max(diff, abs(x[i] – newx[i])
      x[i] = newx[i]
    end for
    while diff > epsilon
  end
end
Parallelize Jacobi Algorithm

- Each processor can be assigned with a number of variables for estimation
- After each iteration, newly estimated values need to be broadcasted to all processors
Conclusion

- For the assignment, either of direct or iterative methods can be implemented.
- Corresponding sequential algorithm has to be implemented to calculate speed up and efficiency.
- Read Chapter 9: Solving Linear Systems of “Parallel Computing: Theory and Practice” of Michael J. Quinn for more detail.