Solution of Linear Equations

$$
\begin{array}{ccccccccc}
a_{00}x_0 & + & a_{01}x_1 & + & \ldots & + & a_{0(n-1)}x_{(n-1)} & = & b_0 \\
a_{10}x_0 & + & a_{11}x_1 & + & \ldots & + & a_{1(n-1)}x_{(n-1)} & = & b_1 \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
a_{(n-1)0}x_0 & + & a_{(n-1)1}x_1 & + & \ldots & + & a_{(n-1)(n-1)}x_{(n-1)} & = & b_{(n-1)}
\end{array}
$$

- $\mathbf{Ax} = \mathbf{b}$ in matrix form
  - $\mathbf{A}$ is an $n \times n$ matrix
  - $\mathbf{b}$ and $\mathbf{x}$ are vectors of length n
- Small systems can be solved by *Gaussian elimination*
  - expensive – $\Theta(n^3)$
  - hard to parallelise

Upper/lower triangular form

We can write $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$ where:

- $\mathbf{L} = \{l_{ij} \mid 0 \leq i, j \leq n-1\}$ is lower-triangular

$$l_{ij} = \begin{cases} a_{ij} & j < i \\ 0 & j \geq i \end{cases}$$

- $\mathbf{D} = \{d_{ij} \mid 0 \leq i, j \leq n-1\}$ is diagonal

$$d_{ij} = \begin{cases} a_{ii} & \\ 0 & j \neq i \end{cases}$$

- $\mathbf{U} = \{u_{ij} \mid 0 \leq i, j \leq n-1\}$ is upper-triangular

$$u_{ij} = \begin{cases} a_{ij} & j > i \\ 0 & j \leq i \end{cases}$$

Assumption

- We assume $d_{ii} \neq 0$ for all i
  - if not, we can permute the variables of $\mathbf{x}$ or the sequence of equations
  - but there is no solution if this is not possible

Jacobi's Method

- Matrix equation can be written as:

$$\mathbf{x} = \mathbf{D}^{-1}\left(\mathbf{b} - (\mathbf{L} + \mathbf{U})\,\mathbf{x}\right)$$

- Jacobi iteration is simply defined by:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)}$$

  where $k \geq 0$ and $x^{(0)}$ is an initial "guess"

- Or, in terms of elements:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}) \quad \text{for } i, j = 0, 1, \ldots, n-1$$

Convergence of Jacobi

- Sufficient condition for convergence is:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad \text{for } 0 \leq i, j \leq n-1$$

($\mathbf{A}$ is *strictly diagonally dominant*)

- A common check in practical implementations is:

$$\frac{||x^{(k+1)} - x^{(k)}||_\infty}{||x^{(k+1)}||_\infty} < \varepsilon$$

where $||x||_\infty = \max_i |x_i|$ (the *infinity-norm*) and ε is a pre-defined threshold (e.g. 10^{-8} or less)

Parallel Jacobi

- Parallel implementation is straightforward:
  - $\mathbf{D}^{-1}\mathbf{b}$ is a constant vector
  - $(\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{x}^{(k)})$ is evaluated by parallel matrix-vector multiplication
  - $\mathbf{x}^{(k)}$ is distributed according to the partition of $\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})$, e.g. :
    - on row-processor for striping
    - on "diagonal" processor for checkerboarding
    - or according to some other partitioning scheme (more on this later!)

Gauss-Seidel Method

- Improve rate of convergence of Jacobi by using *up-to-date information*
  - if components of $\mathbf{x}^{(k+1)}$ are calculated in increasing order of subscript, $\mathbf{Lx}$ can use $\mathbf{x}^{(k+1)}$ instead of $\mathbf{x}^{(k)}$
  - but $\mathbf{Ux}$ can't
- Gauss-Seidel iteration is defined by:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}\mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{D}^{-1}\mathbf{U}\mathbf{x}^{(k)}$$

or

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right)$$

Convergence of Gauss-Seidel

- Same sufficient condition for convergence (strict diagonal dominance of $\mathbf{A}$) as Jacobi
- Also the same practical test for numerical convergence
- If $\mathbf{A}$ is symmetric, then $\mathbf{A}$ positive-definite is a *necessary and sufficient condition*

Parallel Gauss-Seidel

- Parallelisation is harder for Gauss-Seidel because of the sequentiality of the update process...
- ...although for some sparse matrices you may be able to reorder the equations to allow computation to be done in parallel

Gaussian elimination

- To solve a system of linear equations $A.\mathbf{x} = \mathbf{b}$ in matrix form, Gaussian elimination successively eliminates the variable x_k from equations $k+1, \ldots n-1$ for $k = 0, \ldots, n-2$ by subtracting appropriate multiple of (normalised) equation j from equation $k, k < j \leq n-1$
- usual “school" method
- assume that the diagonal elements are always positive and sufficiently large, but no problem to include “pivoting"

The basic algorithm

For $k = 0, \ldots, n-1$:

  For $j = k, \ldots, n-1$:

α $a[k,j] := a[k,j]/a[k,k]$

  (i.e. divide row k of A and $b[k]$ by $a[k,k]$

  EndFor j

α' $b[k] := b[k]/a[k,k]$

  For $i = k+1, \ldots, n-1$:

    For $j = k, \ldots, n-1$:

β $a[i,j] := a[i,j] - a[i,k] \times a[k,j]$

    (i.e. subtract scaled row k from row j)

    EndFor j

β' $b[i] := b[i] - a[i,k] \times b[k]$

  EndFor i

EndFor k

Parallel implementation

Use single-row-striping (one row and vector element per processor)

- Processor P_i initially holds b_i and $\{a_{ij} \mid 0 \leq j \leq n-1\} \quad (0 \leq i \leq n-1)$
- Division step α is a serial computation step, performed in each processor P_k at the beginning of each iteration $k, 0 \leq k \leq n-1$
- Elimination step β requires one-to-all broadcast
  - of elements b_k and a_{kj} for $j > k$, i.e. $k \downarrow a_k$
  - to processors $P_{k+1}, \ldots, P_{n-1}$

Computation time

In step $k \geq 0$, processor k performs a division step (α) and processors $i = k+1, \ldots, n-1$ perform an elimination step (β):

- $a[k,j] := a[k,j]/a[k,k] : n-k-1$ divisions on processor P_k
- $a[i,j] := a[i,j] - a[i,k] \times a[k,j] : \ n-k-1$ multiplications and subtractions on processor P_i in parallel, $k < i < n$

Parallel run time

- No overlap amongst computation steps within or between iterations, so total computation time is:

$$3\sum_{k=0}^{n-1} n-k-1 = 3n(n-1)/2$$

- Communication at kth iteration has message length $n-k-1$ so latency on a hypercube is

$$(t_s + t_w(n-k-1))\log n$$

Parallel run-time (2)

- Total communication latency is therefore
  $\sum_{k=0}^{n-1}(t_s + t_w(n-k-1))\log n$
- Thus, parallel run-time is:
  $T_p = 3n(n-1)/2 + (t_s + t_w(n-1)/2)n\log n$
- Cost is $C_p = \Theta(n^3 logn)$
- **Not cost-optimal**

Pipelining

Above algorithm is synchronous, i.e. the n iteration steps run sequentially with no overlap

- Greater potential for speed if they do overlap
- $\Rightarrow$ *asynchronous* or *pipelined* Gaussian Elimination
- no need to wait to do elimination step β after communication in step α
- similarly, no need for the next processor in the outer loop to wait to
  - do its division step (α) for the next iteration
  - start its one-to-all broadcast

Pipelining algorithm

The asynchronous algorithm now becomes "data driven", or "lazy": each processor executes thus:

1. Send any data destined for other processors – i.e. a part-row;
2. Perform any computation for which it has sufficient data – i.e. in steps α or β;
3. Otherwise wait to receive data

Performance

The time elapsed between *initiation* of iterations k and $k+1$ (cf. cut-through argument) comprises:

- time for division in part-row
  - $n-k-1$ operations
- time to communicate $n-k$ elements from processor k to processor $k+1$
  - *single* hop time is $t_s + t_w(n-k-1)$
- time for elimination in processor $k+1$,
  - $2(n-k-1)$ operations, as in serial algorithm

Performance (2)

- Total time between iterations $k, k+1$ is therefore

$$t_s + (t_w + 3t_a)(n - k - 1)$$

  where t_a is the time for an arithmetic operation – latency $O(n)$

- Total parallel run time is therefore

$$\begin{aligned} T_p &= t_a + (n-1)t_s + (t_w + 3t_a)\sum_{k=0}^{n-2} n - k - 1 \\ &= t_a + (n-1)t_s + n(t_w + 3t_a)(n-1)/2 \end{aligned}$$

- Cost, $C_p = \Theta(n^3) \implies$ *Cost-optimal!*

Load imbalance

As the iteration number increases

- lower numbered processors become idle
- one may be partially loaded with active rows
- the rest are fully loaded
- after a fraction x of the iterations are complete, only a fraction $1 - x$ (roughly) of the processors are busy

Load imbalance (2)

- load imbalance $\Rightarrow$ *limited efficiency* (about $2/3$)
- much efficiency can be regained by *cyclic striping*
- maximum difference between processor loadings in any step is then $O(n)$ operations, corresponding to one row's difference in the partition
- further efficiency gains by block row-striping