Parallel Numerical Algorithms

Chapter 3 – Dense Linear Systems
Section 3.3 – Triangular Linear Systems

Michael T. Heath and Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

1. Triangular Systems
2. 1D Algorithms
3. Wavefront Algorithms
4. 2D Algorithms and TRSM

Triangular Matrices

- Matrix $\boldsymbol{L}$ is *lower triangular* if all entries above its main diagonal are zero, $\ell_{ij} = 0$ for $i < j$
- Matrix $\boldsymbol{U}$ is *upper triangular* if all entries below its main diagonal are zero, $u_{ij} = 0$ for $i > j$
- Triangular matrices are important because triangular linear systems are easily solved by successive substitution
- Most direct methods for solving general linear systems first reduce matrix to triangular form and then solve resulting equivalent triangular system(s)
- Triangular systems are also frequently used as preconditioners in iterative methods for solving linear systems

Forward Substitution

For lower triangular system $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{b}$, solution can be obtained by *forward substitution*

$$x_i = \left(b_i - \sum_{j=1}^{i-1} \ell_{ij}\, x_j\right) / \ell_{ii}, \quad i = 1, \ldots, n$$

for $j = 1$ **to** n
  $x_j = b_j / \ell_{jj}$ { compute soln component }
  for $i = j + 1$ **to** n
    $b_i = b_i - \ell_{ij} x_j$ { update right-hand side }
  end
end

Back Substitution

For upper triangular system $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{b}$, solution can be obtained by *back substitution*

$$x_i = \Big(b_i - \sum_{j=i+1}^{n} u_{ij}\, x_j\Big)/u_{ii}, \quad i = n, \ldots, 1$$

for $j = n$ **to** 1
    $x_j = b_j / u_{jj}$ { compute soln component }
    for $i = 1$ **to** $j - 1$
        $b_i = b_i - u_{ij} x_j$ { update right-hand side }
    end
end

Solving Triangular Systems

- Forward or back substitution requires about $n^2/2$ multiplications and similar number of additions, so serial exeuction time is
$$T_1 = \Theta(\gamma n^2)$$
- We will consider only lower triangular systems, as analogous algorithms for upper triangular systems are similar
- The depth of triangular solve is $D = \Theta(n)$, so the maximum speed-up is $T_1/D = \Theta(n)$

Loop Orderings for Forward Substitution

for $j = 1$ **to** n
    $x_j = b_j / \ell_{jj}$
    for $i = j + 1$ **to** n
        $b_i = b_i - \ell_{ij}\, x_j$
    end
end

- right-looking
- immediate-update
- data-driven
- fan-out

for $i = 1$ **to** n
    for $j = 1$ **to** $i - 1$
        $b_i = b_i - \ell_{ij}\, x_j$
    end
    $x_i = b_i / \ell_{ii}$
end

- left-looking
- delayed-update
- demand-driven
- fan-in

Parallel Algorithm

Partition

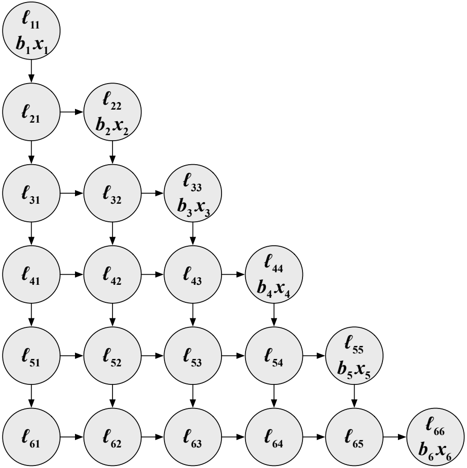
- For $i = 2, \ldots, n$, $j = 1, \ldots, i-1$, fine-grain task (i, j) stores ℓ_{ij} and computes product $\ell_{ij}\, x_j$
- For $i = 1, \ldots, n$, fine-grain task (i, i) stores ℓ_{ii} and b_i, collects sum $t_i = \sum_{j=1}^{i-1} \ell_{ij}\, x_j$, and computes and stores $x_i = (b_i - t_i)/\ell_{ii}$

yielding 2-D triangular array of $n\,(n+1)/2$ fine-grain tasks

Communicate

- For $j = 1, \ldots, n-1$, task (j, j) broadcasts x_j to tasks (i, j), $i = j+1, \ldots, n$
- For $i = 2, \ldots, n$, sum reduction of products $\ell_{ij}\, x_j$ across tasks (i, j), $j = 1, \ldots, i$, with task (i, i) as root

Fine-Grain Tasks and Communication



Fine-Grain Parallel Algorithm

if $i = j$ **then**
  $t = 0$
  if $i > 1$ **then**
    recv sum reduction of t across tasks (i, k), $k = 1, \ldots, i$
  end
  $x_i = (b_i - t)/\ell_{ii}$
  broadcast x_i to tasks (k, i), $k = i + 1, \ldots, n$
else
  recv broadcast of x_j from task (j, j)
  $t = \ell_{ij}\, x_j$
  reduce t across tasks (i, k), $k = 1, \ldots, i$
end

Fine-Grain Algorithm

- If communication is suitably pipelined, then fine-grain algorithm can achieve $\Theta(n)$ execution time, but uses $\Theta(n^2)$ tasks, so it is inefficient
- If there are multiple right-hand-side vectors $\boldsymbol{b}$, then successive solutions can be pipelined to increase overall efficiency
- Agglomerating fine-grain tasks yields more reasonable number of tasks and improves ratio of computation to communication

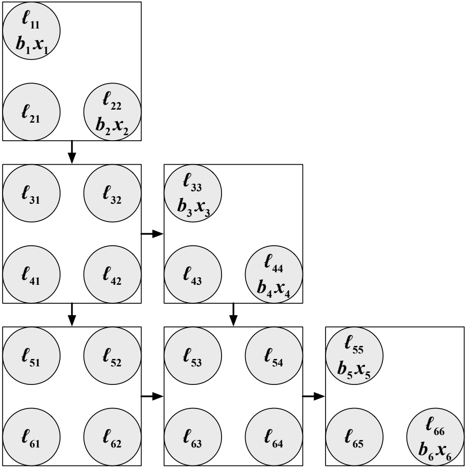
Agglomeration

Agglomerate

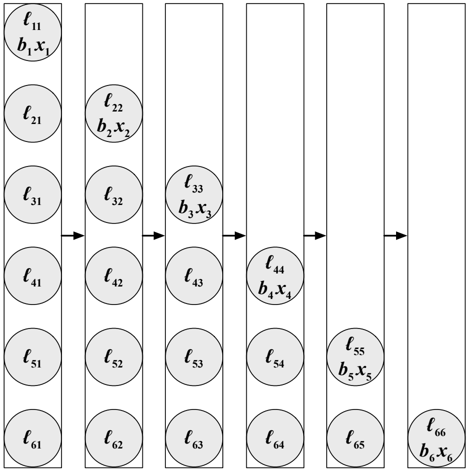
With $n \times n$ array of fine-grain tasks, natural strategies are

- 2-D: combine $k \times k$ subarray of fine-grain tasks to form each coarse-grain task, yielding $(n/k)^2$ coarse-grain tasks
- 1-D column: combine n fine-grain tasks in each column into coarse-grain task, yielding n coarse-grain tasks
- 1-D row: combine n fine-grain tasks in each row into coarse-grain task, yielding n coarse-grain tasks

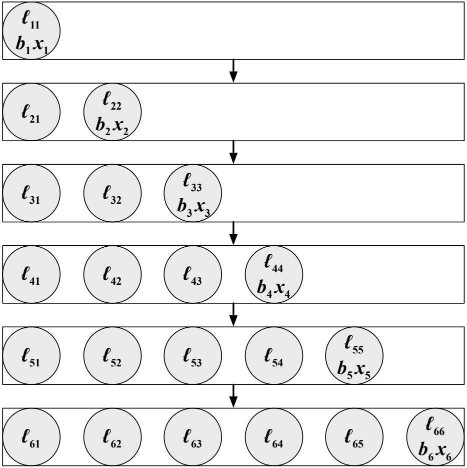
2-D Agglomeration



1-D Column Agglomeration



1-D Row Agglomeration

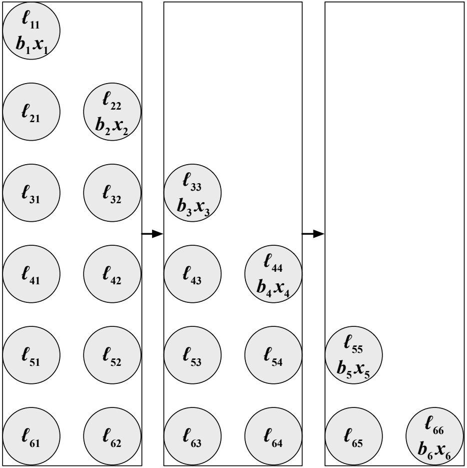


Mapping

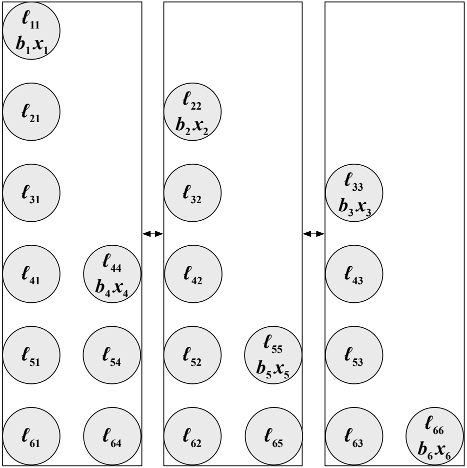
Map

- 2-D: assign $(n/k)^2/p$ coarse-grain tasks to each of p processors using any desired mapping in each dimension, treating target network as 2-D mesh
- 1-D: assign n/p coarse-grain tasks to each of p processors using any desired mapping, treating target network as 1-D mesh

1-D Column Agglomeration, Block Mapping



1-D Column Agglomeration, Cyclic Mapping



1-D Aggregation with Block-Cyclic Mapping Cost

- With block-size b, 1D partitioning
  - requires n/b broadcasts of b items for row-agglomeration
  - requires n/b reductions of b items for column-agglomeration
  - in both cases $O(nb/p + b^2)$ work must be done to solve for b entries of x between each of the n/b collectives
- The overall execution time is

$$T_p(n, b) = \Theta\Big(\alpha(n/b)\log(p) + \beta n + \gamma(n^2/p + nb)\Big)$$

- Selecting block-size $b = n/p$, parallel execution time is

$$T_p(n, n/p) = \Theta\Big(\alpha p \log(p) + \beta n + \gamma n^2/p\Big)$$

1-D Block-Cyclic Algorithm Communication Cost

To determine strong scalability limit, we wish to determine when $T_p(n, n/p)$ is dominated by the term $\gamma n^2/p$, we have

$$T_p(n, n/p) = \Theta\Big(\alpha p \log(p) + \beta n + \gamma n^2/p\Big)$$

- The bandwidth cost yields the bound

$$p_s = O\Big((\gamma/\beta)n\Big)$$

- The latency cost yields the bound

$$p_s = O\Big((\sqrt{\gamma/\alpha})n/\sqrt{\log(\sqrt{(\gamma/\alpha)}n)}\Big)$$

1-D Block-Cyclic Algorithm Weak Scalability

- The efficiency of the block-cyclic algorithm is

$$E_p(n) = \Theta\bigg(1/\Big(1 + (\alpha/\gamma)p^2\log(p)/n^2 + (\beta/\gamma)p/n\Big)\bigg)$$

- Weak scaling, corresponds to p processors and $n = \sqrt{p_w}n_0$ elements (input size per processor is $M_1/p = (n_0\sqrt{p})^2/p = n_0^2$)

$$E_{p_w}(n_0\sqrt{p_w}) = \Theta\bigg(1/\Big(1+(\alpha/\gamma)p_w\log(p_w)/n_0^2+(\beta/\gamma)\sqrt{p_w}/n_0\Big)\bigg)$$

- Therefore, weak scalability is possible to

$$p_w = \Theta\Big(\min[(\gamma/\alpha)n_0^2/\log((\gamma/\alpha)n_0^2), (\gamma/\beta)^2n_0^2]\Big) \quad \text{processors}$$

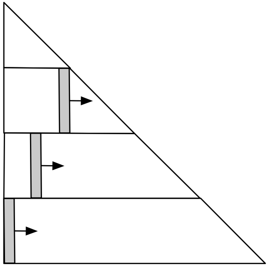
Wavefront Algorithms

- Naive fan-out and fan-in algorithms derive their parallelism from inner loop, whose work is partitioned and distributed across processors, while outer loop is serial
- Conceptually, fan-out and fan-in algorithms work on only one component of solution at a time, though successive steps may be pipelined
- Wavefront algorithms exploit parallelism in outer loop explicitly by working on multiple components of solution simultaneously

1-D Column Wavefront Algorithm

- Naive 1-D column fan-out algorithm seems to admit no parallelism: after processor owning column j computes x_j, resulting updating of right-hand side cannot be shared with other processors because they cannot access column j
- Instead of performing all such updates immediately, however, process owning column j could complete only first s components of update vector and forward them to processor owning column $j+1$ *before* continuing with next s components of update vector, etc.
- Upon receiving first s components of update vector, processor owning column $j+1$ can compute x_{j+1}, begin further updates, forward its own contributions to next process, etc.

1-D Column Wavefront Algorithm



To formalize wavefront column algorithm we introduce

- z : vector in which to accumulate updates to right-hand-side
- *segment* : set containing at most s consecutive components of z

1-D Column Wavefront Algorithm

for $j \in$ *mycols*
    for $k = 1$ **to** # segments
        recv *segment*
        if $k = 1$ **then**
            $x_j = (b_j - z_j)/\ell_{jj}$
            segment $=$ *segment* $- \{z_j\}$
        end
        for $z_i \in$ *segment*
            $z_i = z_i + \ell_{ij}\, x_j$
        end
        if $|$*segment*$| > 0$ **then**
            send *segment* to processor owning column $j + 1$
        end
    end
end

1-D Column Wavefront Algorithm

- Depending on segment size, column mapping, communication-to-computation speed ratio, etc., it may be possible for all processors to become busy simultaneously, each working on different component of solution
- Segment size is adjustable parameter that controls tradeoff between communication and concurrency
- "First" segment for given column shrinks by one element after each component of solution is computed, disappearing after s steps, when next segment becomes "first" segment, etc.

1-D Column Wavefront Algorithm

- At end of computation only one segment remains and it contains only one element
- Communication volume declines throughout algorithm
- As segment length s increases, communication start-up cost decreases but computation cost increases, and vice versa as segment length decreases
- Optimal choice of segment length s can be predicted from performance model

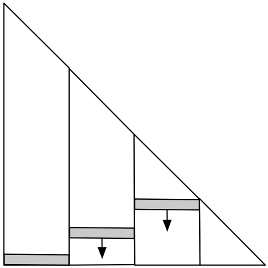
1-D Row Wavefront Algorithm

- Wavefront approach can also be applied to 1-D row fan-in algorithm
- Computation of ith inner product cannot be shared because only one processor has access to row i of matrix
- Thus, work on multiple components must be overlapped to attain any concurrency
- Analogous approach is to break solution vector x into segments that are pipelined through processors

1-D Row Wavefront Algorithm

- Initially, processor owning row 1 computes x_1 and sends it to processor owning row 2, which computes resulting update and then x_2
- This processor continues (serially at this early stage) until s components of solution have been computed
- Henceforth, receiving processors forward any full-size segments *before* they are used in updating
- Forwarding of currently incomplete segment is delayed until next component of solution is computed and appended to it

1-D Row Wavefront Algorithm



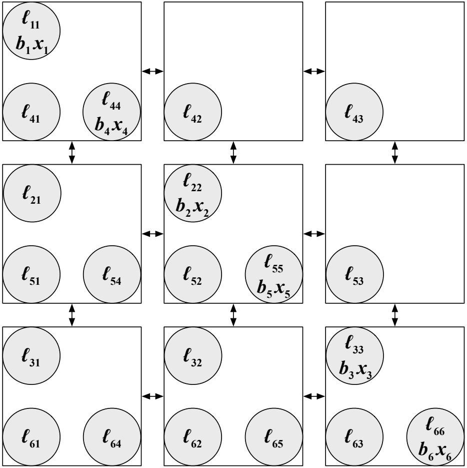
1-D Row Wavefront Algorithm

```
for i ∈ myrows
    for k = 1 to # segments − 1
        recv segment
        send segment to processor owning row i + 1
        for x_j ∈ segment
            b_i = b_i − ℓ_ij x_j
        end
    end
    recv segment        /* last may be empty */
    for x_j ∈ segment
        b_i = b_i − ℓ_ij x_j
    end
    x_i = b_i / ℓ_ii
    segment = segment ∪ {x_i}
    send segment to processor owning row i + 1
end
```

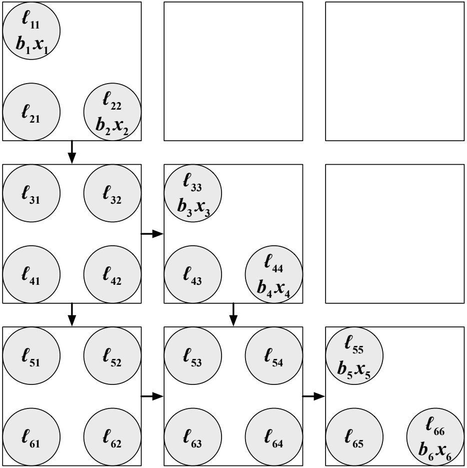
1-D Row Wavefront Algorithm

- Instead of starting with full set of segments that shrink and eventually disappear, segments appear and grow until there is a full set of them
- It may be possible for all processors to be busy simultaneously, each working on different segment
- Segment size is adjustable parameter that controls tradeoff between communication and concurrency, and optimal value of segment length s can be predicted from performance model

2-D Agglomeration, Cyclic Mapping



2-D Agglomeration, Block Mapping



2-D Algorithm

- For 2-D block mapping with $(n/\sqrt{p}\,) \times (n/\sqrt{p}\,)$ fine-grain tasks per process, both vertical broadcasts and horizontal sum reductions are required to communicate solution components and accumulate inner products, respectively
- However, almost half the processors perform no work
- For 1-D block mapping with $n \times n/p$ fine-grain tasks per process, vertical broadcasts are no longer necessary, but horizontal broadcasts send much larger messages, and work is still unbalanced

2-D Algorithm

- Cyclic assignment of rows and columns to processors yields provides each processor with at least $(n/\sqrt{p})(n/\sqrt{p}-1)/2$ entries
- But obvious implementation, computing successive components of solution vector x and performing corresponding horizontal sum reductions and vertical broadcasts, still has limited concurrency

Triangular Solve with Many Right-Hand Sides

- The triangular solve is a BLAS-2 operation
  - $\Theta(1)$ flop-to-byte ratio (operations per memory access)
  - $Q_1 = n^2$ and $D = n$, so degree of concurrency is $\Theta(n)$
- Solving many systems at a time, i.e. determining $\boldsymbol{X} \in \mathbb{R}^{n \times k}$ so that

$$\boldsymbol{A}\boldsymbol{X} = \boldsymbol{B}$$

  where degree of concurrency is $\Theta(nk)$ and flop-to-byte ratio can be as high as $\Theta(k)$
- Triangular solve with multiple equations *TRSM* can also achieve better parallel scaling efficiency

Triangular Inversion

- A different way to solve a triangular linear system is to
  - Invert the triangular matrix $\boldsymbol{S} = \boldsymbol{L}^{-1}$, then perform a
  - Matrix vector multiplication $\boldsymbol{x} = \boldsymbol{S}\boldsymbol{y}$

  This method requires $Q_1 = \Theta(n^3)$ work to solve a single linear system of equations, but has logarithmic depth
  - For k linear systems (TRSM), $Q_1 = \Theta(n^3 + n^2 k)$ may be ok
  - Lower depth evident from decoupling of recursive equations

$$\begin{bmatrix} \boldsymbol{L}_{11} & \\ \boldsymbol{L}_{21} & \boldsymbol{L}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{S}_{11} & \\ \boldsymbol{S}_{21} & \boldsymbol{S}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & \\ & \boldsymbol{I} \end{bmatrix}$$

where we deduce that $\boldsymbol{S}_{11} = \boldsymbol{L}_{11}^{-1}$ and $\boldsymbol{S}_{22} = \boldsymbol{L}_{22}^{-1}$ are independent, while $\boldsymbol{S}_{21} = \boldsymbol{S}_{22}\boldsymbol{L}_{21}\boldsymbol{S}_{11}$ can be done with matrix multiplication which has $D = \Theta(\log(n))$

References

- R. H. Bisseling and J. G. G. van de Vorst, Parallel triangular system solving on a mesh network of Transputers, *SIAM J. Sci. Stat. Comput.* 12:787-799, 1991
- S. C. Eisenstat, M. T. Heath, C. S. Henkel, and C. H. Romine, Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors, *SIAM J. Sci. Stat. Comput.* 9:589-600, 1988
- M. T. Heath and C. H. Romine, Parallel solution of triangular systems on distributed-memory multiprocessors, *SIAM J. Sci. Stat. Comput.* 9:558-588, 1988
- N. J. Higham, Stability of parallel triangular system solvers, *SIAM J. Sci. Comput.* 16:400-413, 1995

References

- G. Li and T. F. Coleman, A parallel triangular solver for a distributed-memory multiprocessor, *SIAM J. Sci. Stat. Comput.* 9:485-502, 1988
- G. Li and T. F. Coleman, A new method for solving triangular systems on distributed-memory message-passing multiprocessors, *SIAM J. Sci. Stat. Comput.* 10:382-396, 1989
- C. H. Romine and J. M. Ortega, Parallel solution of triangular systems of equations, *Parallel Computing* 6:109-114, 1988
- E. E. Santos, On designing optimal parallel triangular solvers, *Information and Computation* 161:172-210, 2000