Outline

1. Triangular Systems
2. Parallel Algorithms
3. Wavefront Algorithms
Matrix $L$ is *lower triangular* if all entries above its main diagonal are zero, $\ell_{ij} = 0$ for $i < j$

Matrix $U$ 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.
For lower triangular system \( Lx = b \), solution can be obtained by \textit{forward substitution}:

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

\[
\begin{align*}
    &\text{for } j = 1 \text{ to } n \\
    &\quad x_j = b_j / \ell_{jj} \\
    &\quad \text{for } i = j + 1 \text{ to } n \\
    &\quad\quad b_i = b_i - \ell_{ij} x_j \\
    &\quad \text{end} \\
    &\text{end}
\end{align*}
\]

\{ compute soln component \}

\{ update right-hand side \}
Back Substitution

For upper triangular system $Ux = b$, solution can be obtained by *back substitution*

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

for $j = n$ to 1

$$x_j = b_j / u_{jj}$$

for $i = 1$ to $j - 1$

$$b_i = b_i - u_{ij} x_j$$

{ compute soln component }

{ update right-hand side }

end

end

Forward or back substitution requires about $n^2/2$ multiplications and similar number of additions, so serial execution time is

$$T_1 = \Theta(\gamma n^2)$$

We will consider only lower triangular systems, as analogous algorithms for upper triangular systems are similar.
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

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

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

- 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 $\ell_{ij}$ and computes product $\ell_{ij} x_j$
- For $i = 1, \ldots, n$, fine-grain task $(i, i)$ stores $\ell_{ii}$ 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

\[
\begin{align*}
\ell_{11} & \quad b_1 x_1 \\
\ell_{21} & \quad b_2 x_2 \\
\ell_{31} & \quad \ell_{32} \quad b_3 x_3 \\
\ell_{41} & \quad \ell_{42} \quad \ell_{43} \quad b_4 x_4 \\
\ell_{51} & \quad \ell_{52} \quad \ell_{53} \quad \ell_{54} \quad b_5 x_5 \\
\ell_{61} & \quad \ell_{62} \quad \ell_{63} \quad \ell_{64} \quad \ell_{65} \quad b_6 x_6
\end{align*}
\]
The Fine-Grain Parallel Algorithm is defined as follows:

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

\[
\begin{align*}
\ell_{11} &\quad b_1 x_1 \\
\ell_{21} &\quad \ell_{22} &\quad b_2 x_2 \\
\ell_{31} &\quad \ell_{32} &\quad \ell_{33} &\quad b_3 x_3 \\
\ell_{41} &\quad \ell_{42} &\quad \ell_{43} &\quad \ell_{44} &\quad b_4 x_4 \\
\ell_{51} &\quad \ell_{52} &\quad \ell_{53} &\quad \ell_{54} &\quad b_5 x_5 \\
\ell_{61} &\quad \ell_{62} &\quad \ell_{63} &\quad \ell_{64} &\quad \ell_{65} &\quad b_6 x_6 \\
\end{align*}
\]
1-D Column Agglomeration
1-D Row Agglomeration

\[ \ell_{11} \quad b_1 x_1 \]

\[ \ell_{21} \quad \ell_{22} \quad b_2 x_2 \]

\[ \ell_{31} \quad \ell_{32} \quad \ell_{33} \quad b_3 x_3 \]

\[ \ell_{41} \quad \ell_{42} \quad \ell_{43} \quad \ell_{44} \quad b_4 x_4 \]

\[ \ell_{51} \quad \ell_{52} \quad \ell_{53} \quad \ell_{54} \quad \ell_{55} \quad b_5 x_5 \]

\[ \ell_{61} \quad \ell_{62} \quad \ell_{63} \quad \ell_{64} \quad \ell_{65} \quad b_6 x_6 \]
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
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 $O(b^2)$ work must be done to solve for $b$ entries of $x$ between each of the $n/b$ collectives

The overall execution time is

$$T_p(n, b) = \Theta\left(\alpha\left(\frac{n}{b}\right) \log(p) + \beta n + \gamma\left(\frac{n^2}{p} + nb\right)\right)$$

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

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

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

- The bandwidth cost yields the bound

$$p_s = O\left((\gamma/\beta)n\right)$$

- The latency cost yields the bound

$$p_s = O\left((\sqrt{\gamma/\alpha})n/\sqrt{\log(\sqrt{\gamma/\alpha})n}\right)$$
The efficiency of the block-cyclic algorithm is

$$E_{p}(n) = \Theta\left(1/\left(1 + (\alpha/\gamma)p^2 \log(p)/n^2 + (\beta/\gamma)p/n\right)\right)$$

Weak scaling, corresponds to $p$ processors and

$n = \sqrt{p_w}n_0$ elements (input size per processor is

$M_1/p = (n_0\sqrt{p})^2/p = n_0^2$)

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

Therefore, weak scalability is possible to

$$p_w = \Theta\left(\min[(\gamma/\alpha)n_0^2/\log((\gamma/\alpha)n_0^2), (\gamma/\beta)^2n_0^2]\right)$$ processes
2-D Agglomeration, Cyclic Mapping
2-D Agglomeration, Block Mapping

\[
\begin{array}{cccc}
\ell_{11} & b_1 x_1 \\
\ell_{21} & & \\
\ell_{31} & \ell_{32} & b_2 x_2 \\
\ell_{41} & \ell_{42} & & \\
\ell_{51} & \ell_{52} & \ell_{53} & \ell_{54} \\
\ell_{61} & \ell_{62} & \ell_{63} & \ell_{64} \\
& & \ell_{55} & b_3 x_3 \\
& & \ell_{43} & b_4 x_4 \\
& & \ell_{65} & b_5 x_5 \\
& & \ell_{56} & \ell_{66} \\
\end{array}
\]
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.
Cyclic assignment of rows and columns to processors yields provides each processor with at least 
\[
\frac{n}{\sqrt{p}} \left( \frac{n}{\sqrt{p}} - 1 \right)/2
\]
entries.

But obvious implementation, computing successive components of solution vector \( x \) and performing corresponding horizontal sum reductions and vertical broadcasts, still has limited concurrency because computation so long as every component is mapped onto a processor column.
Each step of resulting algorithm has four phases

1. Computation of next $b$ solution components by processors in lower triangle using 2-D fine-grain algorithm
2. Broadcast of resulting solution components vertically from processors on diagonal to processors in upper triangle
3. Computation of resulting updates (partial sums in inner products) by all processors
4. Horizontal sum reduction from processors in upper triangle to processors on diagonal
Wavefront Algorithms

- 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 to some degree.

- Wavefront algorithms exploit parallelism in outer loop explicitly by working on multiple components of solution simultaneously.
1-D Column Wavefront Algorithm

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.
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 $k = 1$
            $x_j = (b_j - z_j) / \ell_{jj}$
            segment = segment $-$ \{z\}
        end
    end
    for $z_i \in$ segment
        $z_i = z_i + \ell_{ij} x_j$
    end
    if $|segment| > 0$
        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 an adjustable parameter that controls the tradeoff between communication and concurrency.

- “First” segment for a given column shrinks by one element after each component of the solution is computed, disappearing after \( s \) steps, when the 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 the 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.
Wavefront approach can also be applied to 1-D row fan-in algorithm

Computation of $i$th 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 $i \in \text{myrows}$
  for $k = 1$ to $\# \text{ segments} - 1$
    recv segment
    send segment to processor owning row $i + 1$
    for $x_j \in \text{segment}$
      $b_i = b_i - \ell_{ij} x_j$
    end
  end
  recv segment  /* last may be empty */
  for $x_j \in \text{segment}$
    $b_i = b_i - \ell_{ij} x_j$
  end
  $x_i = b_i / \ell_{ii}$
  segment = segment $\cup \{x_i\}$
  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 Wavefront Algorithm

- We can stagger the broadcasts in the 2-D block-cyclic algorithm to turn broadcasts into shofts.
- Reduces latency cost by a factor of $\Theta(\log(p))$.
- Wavefront-based approaches are also viable in dense matrix factorizations and as parallelism paradigms in general.
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 $X \in \mathbb{R}^{n \times k}$ so that

$$AX = B$$

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 $S = L^{-1}$, then perform a
  - Matrix vector multiplication $x = Sy$

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), $Q_1 = \Theta(n^3 + n^2k)$ may be ok
- Lower depth evident from decoupling of recursive equations

\[
\begin{bmatrix}
L_{11} & L_{21} \\
L_{21} & L_{22}
\end{bmatrix}
\begin{bmatrix}
S_{11} & S_{21} \\
S_{21} & S_{22}
\end{bmatrix}
= \begin{bmatrix}
I & I
\end{bmatrix}
\]

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


