Outline

1. BLAS
2. Inner Product
3. Outer Product
4. Matrix-Vector Product
5. Matrix-Matrix Product
Basic Linear Algebra Subprograms (BLAS) are building blocks for many other matrix computations.

BLAS encapsulate basic operations on vectors and matrices so they can be optimized for particular computer architecture while high-level routines that call them remain portable.

BLAS offer good opportunities for optimizing utilization of memory hierarchy.

Generic BLAS are available from netlib, and many computer vendors provide custom versions optimized for their particular systems.
Examples of BLAS

<table>
<thead>
<tr>
<th>Level</th>
<th>Work</th>
<th>Examples</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$O(n)$</td>
<td>daxpy</td>
<td>Scalar $\times$ vector + vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ddot</td>
<td>Inner product</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dnrm2</td>
<td>Euclidean vector norm</td>
</tr>
<tr>
<td>2</td>
<td>$O(n^2)$</td>
<td>dgemv</td>
<td>Matrix-vector product</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dtrsv</td>
<td>Triangular solve</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dger</td>
<td>Outer-product</td>
</tr>
<tr>
<td>3</td>
<td>$O(n^3)$</td>
<td>dgemm</td>
<td>Matrix-matrix product</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dtrsm</td>
<td>Multiple triangular solves</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dsyrk</td>
<td>Symmetric rank-$k$ update</td>
</tr>
</tbody>
</table>

$\gamma_1 \gg \gamma_2 \gg \gamma_3$

BLAS 1 effective sec/flop  BLAS 2 effective sec/flop  BLAS 3 effective sec/flop
Inner Product

- Inner product of two \( n \)-vectors \( \mathbf{x} \) and \( \mathbf{y} \) given by

\[
\mathbf{x}^T \mathbf{y} = \sum_{i=1}^{n} x_i y_i
\]

- Computation of inner product requires \( n \) multiplications and \( n - 1 \) additions

\[
M_1 = \Theta(n), \quad Q_1 = \Theta(n), \quad T_1 = \Theta(\gamma n)
\]

- Effectively as hard as scalar reduction, can be done via binary or binomial tree summation
Parallel Algorithm

Partition

- For $i = 1, \ldots, n$, fine-grain task $i$ stores $x_i$ and $y_i$, and computes their product $x_i y_i$

Communicate

- Sum reduction over $n$ fine-grain tasks
Fine-Grain Parallel Algorithm

\[ z_i = x_i y_i \quad \{ \text{local scalar product} \} \]

reduce \( z_i \) across all tasks \( i = 1, \ldots, n \) \quad \{ \text{sum reduction} \}
Agglomeration and Mapping

Agglomerate

- Combine $k$ components of both $x$ and $y$ to form each coarse-grain task, which computes inner product of these subvectors
- Communication becomes sum reduction over $n/k$ coarse-grain tasks

Map

- Assign $(n/k)/p$ coarse-grain tasks to each of $p$ processors, for total of $n/p$ components of $x$ and $y$ per processor

\[ x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4 + x_5y_5 + x_6y_6 + x_7y_7 + x_8y_8 + x_9y_9 \]
Coarse-Grain Parallel Algorithm

\[ z_i = x^T_i y_i \] \quad \{ \text{local inner product} \}

reduce \( z_i \) across all processors \( i = 1, \ldots, p \) \quad \{ \text{sum reduction} \}

\[ [x[i] \text{ – subvector of } x \text{ assigned to processor } i ] \]
Performance

The parallel costs \((L_p, W_p, F_p)\) for the inner product are given by

- Computational cost \(F_p = \Theta(n/p)\) regardless of network
- The latency and bandwidth costs depend on network:
  - 1-D mesh: \(L_p, W_p = \Theta(p)\)
  - 2-D mesh: \(L_p, W_p = \Theta(\sqrt{p})\)
  - hypercube: \(L_p, W_p = \Theta(\log p)\)
- For a hypercube or fully-connected network time is
  \[ T_p = \alpha L_p + \beta W_p + \gamma F_p = \Theta(\alpha \log(p) + \gamma n/p) \]
- Efficiency and scaling are the same as for binary tree sum
Inner product on 1-D Mesh

- For 1-D mesh, total time is \( T_p = \Theta(\gamma n / p + \alpha p) \)
- To determine strong scalability, we set constant efficiency and solve for \( p_s \)

\[
\text{const} = E_{p_s} = \frac{T_1}{p_s T_p} = \Theta\left(\frac{\gamma n}{\gamma n + \alpha p_s^2}\right) = \Theta\left(\frac{1}{1 + (\alpha/\gamma)p_s^2/n}\right)
\]

which yields \( p_s = \Theta(\sqrt{\gamma/\alpha} n) \)

- 1-D mesh weakly scalable to \( p_w = \Theta((\gamma/\alpha) n) \) processors:

\[
E_{p_w}(p_w n) = \Theta\left(\frac{1}{1 + (\alpha/\gamma)p_w^2/(p_w n)}\right) = \Theta\left(\frac{1}{1 + (\alpha/\gamma)p_w/n}\right)
\]
Inner product on 2-D Mesh

- For 2-D mesh, total time is $T_p = \Theta(\gamma n/p + \alpha \sqrt{p})$
- To determine strong scalability, we set constant efficiency and solve for $p_s$

$$\text{const} = E_{p_s} = \frac{T_1}{p_s T_{p_s}} = \Theta\left(\frac{\gamma n}{\gamma n + \alpha p_s^{3/2}}\right) = \Theta\left(\frac{1}{1 + (\alpha/\gamma)p_s^{3/2}/n}\right)$$

which yields $p_s = \Theta((\gamma/\alpha)^{2/3}n^{2/3})$

- 2-D mesh weakly scalable to $p_w = \Theta((\gamma/\alpha)^n)$, since

$$E_{p_w}(p_w n) = \Theta\left(\frac{1}{1 + (\alpha/\gamma)p_w^{3/2}/(p_w n)}\right) = \Theta\left(\frac{1}{1 + (\alpha/\gamma)\sqrt{p_w}/n}\right)$$
Outer Product

- Outer product of two \( n \)-vectors \( \mathbf{x} \) and \( \mathbf{y} \) is \( n \times n \) matrix 
  \[
  \mathbf{Z} = \mathbf{x} \mathbf{y}^T 
  \]
  whose \((i, j)\) entry \( z_{ij} = x_i y_j \)

- For example,

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 
\end{bmatrix}
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 
\end{bmatrix}
^T
 = 
\begin{bmatrix}
  x_1 y_1 & x_1 y_2 & x_1 y_3 \\
  x_2 y_1 & x_2 y_2 & x_2 y_3 \\
  x_3 y_1 & x_3 y_2 & x_3 y_3 
\end{bmatrix}
\]

- Computation of outer product requires \( n^2 \) multiplications,

\[
M_1 = \Theta(n^2), \quad Q_1 = \Theta(n^2), \quad T_1 = \Theta(\gamma n^2)
\]

(in this case, we should treat \( M_1 \) as output size or define the problem as in the BLAS: \( \mathbf{Z} = \mathbf{Z}_{\text{input}} + \mathbf{x} \mathbf{y}^T \))
Parallel Algorithm

Partition

- For , , ,  fine-grain task stores , yielding 2-D array of fine-grain tasks
- Assuming no replication of data, at most fine-grain tasks store components of and , say either
  - for some , task stores and task stores , or
  - task stores both and , 

Communicate

- For , task that stores broadcasts it to all other tasks in th task row
- For , task that stores broadcasts it to all other tasks in th task column
Fine-Grain Tasks and Communication
Fine-Grain Parallel Algorithm

broadcast $x_i$ to tasks $(i, k), \; k = 1, \ldots, n$ \{ horizontal broadcast \}

broadcast $y_j$ to tasks $(k, j), \; k = 1, \ldots, n$ \{ vertical broadcast \}

$z_{ij} = x_i y_j$ \{ local scalar product \}
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

- Each task that stores portion of $x$ must broadcast its subvector to all other tasks in its task row.
- Each task that stores portion of $y$ must broadcast its subvector to all other tasks in its task column.
2-D Agglomeration

\[
\begin{array}{ccc}
\underline{x_1y_1} & x_1y_2 & x_1y_3 \\
\underline{x_2y_1} & x_2y_2 & x_2y_3 \\
x_3y_1 & x_3y_2 & x_3y_3 \\
x_4y_1 & x_4y_2 & x_4y_3 \\
x_5y_1 & x_5y_2 & x_5y_3 \\
x_6y_1 & x_6y_2 & x_6y_3 \\
\end{array}
\]
1-D Agglomeration

- If either $x$ or $y$ stored in one task, then broadcast required to communicate needed values to all other tasks.
- If either $x$ or $y$ distributed across tasks, then multinode broadcast required to communicate needed values to other tasks.
1-D Column Agglomeration
1-D Row Agglomeration

\[ x_1 y_1 \quad x_1 y_2 \quad x_1 y_3 \quad x_1 y_4 \quad x_1 y_5 \quad x_1 y_6 \]

\[ x_2 y_1 \quad x_2 y_2 \quad x_2 y_3 \quad x_2 y_4 \quad x_2 y_5 \quad x_2 y_6 \]

\[ x_3 y_1 \quad x_3 y_2 \quad x_3 y_3 \quad x_3 y_4 \quad x_3 y_5 \quad x_3 y_6 \]

\[ x_4 y_1 \quad x_4 y_2 \quad x_4 y_3 \quad x_4 y_4 \quad x_4 y_5 \quad x_4 y_6 \]

\[ x_5 y_1 \quad x_5 y_2 \quad x_5 y_3 \quad x_5 y_4 \quad x_5 y_5 \quad x_5 y_6 \]

\[ x_6 y_1 \quad x_6 y_2 \quad x_6 y_3 \quad x_6 y_4 \quad x_6 y_5 \quad x_6 y_6 \]
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
2-D Agglomeration with Block Mapping

\[ \begin{align*}
&x_1y_1 & x_1y_2 & x_1y_3 & x_1y_4 & x_1y_5 & x_1y_6 \\
&x_2y_1 & x_2y_2 & x_2y_3 & x_2y_4 & x_2y_5 & x_2y_6 \\
&x_3y_1 & x_3y_2 & x_3y_3 & x_3y_4 & x_3y_5 & x_3y_6 \\
&x_4y_1 & x_4y_2 & x_4y_3 & x_4y_4 & x_4y_5 & x_4y_6 \\
&x_5y_1 & x_5y_2 & x_5y_3 & x_5y_4 & x_5y_5 & x_5y_6 \\
&x_6y_1 & x_6y_2 & x_6y_3 & x_6y_4 & x_6y_5 & x_6y_6
\end{align*} \]
1-D Column Agglomeration with Block Mapping
1-D Row Agglomeration with Block Mapping
Coarse-Grain Parallel Algorithm

broadcast $x[i]$ to $i$th process row \{ horizontal broadcast \}

broadcast $y[j]$ to $j$th process column \{ vertical broadcast \}

$$Z[i][j] = x[i] y^T[j]$$ \{ local outer product \}

$[Z[i][j]]$ means submatrix of $Z$ assigned to process $(i, j)$ by mapping
The parallel costs \((L_p, W_p, F_p)\) for the outer product are

- **Computational cost** \(F_p = \Theta(n^2/p)\) regardless of network
- **The latency and bandwidth costs can be derived from the cost of broadcast/allgather**
  - **1-D agglomeration**: \(L_p = \Theta(\log p), W_p = \Theta(n)\)
  - **2-D agglomeration**: \(L_p = \Theta(\log p), W_p = \Theta(n/\sqrt{p})\)

For 1-D agglomeration, execution time is
\[
T_p^{1-D} = T_p^{\text{allgather}}(n) + \Theta(\gamma n^2/p) = \Theta(\alpha \log(p) + \beta n + \gamma n^2/p)
\]

For 2-D agglomeration, execution time is
\[
T_p^{2-D} = 2T_p^{\text{bcast}}(n/\sqrt{p}) + \Theta(\gamma n^2/p) = \Theta(\alpha \log(p) + \beta n/\sqrt{p} + \gamma n^2/p)
\]
1-D agglomeration is strongly scalable to

\[ p_s = \Theta\left(\min\left(\frac{\gamma}{\alpha}n^2 / \log\left(\frac{\gamma}{\alpha}n^2\right), \frac{\gamma}{\beta}n\right)\right) \]

processors, since

\[ E_{p_s}^{1-D} = \Theta\left(\frac{1}{1 + \left(\frac{\alpha}{\gamma}\right) \log\left(p_s p_s / n^2 + \left(\frac{\beta}{\gamma}\right)p_s / n\right)}\right) \]

2-D agglomeration is strongly scalable to

\[ p_s = \Theta\left(\min\left(\frac{\gamma}{\alpha}n^2 / \log\left(\frac{\gamma}{\alpha}n^2\right), \frac{\gamma}{\beta}^2n^2\right)\right) \]

processors, since

\[ E_{p_s}^{2-D} = \Theta\left(\frac{1}{1 + \left(\frac{\alpha}{\gamma}\right) \log\left(p_s p_s / n^2 + \left(\frac{\beta}{\gamma}\right)\sqrt{p_s / n}\right)}\right) \]
Outer Product Weak Scaling

- 1-D agglomeration is weakly scalable to
  \[ p_w = \Theta\left(\min\left(2^{(\gamma/\alpha)n^2}, (\gamma/\beta)^2 n^2\right)\right) \]
  processors, since
  \[ E_{p_w}^{1-D}(\sqrt{p_w}n) = \Theta\left(1/(1 + (\alpha/\gamma) \log(p_w)/n^2 + (\beta/\gamma) \sqrt{p_w}/n)\right) \]

- 2-D agglomeration is weakly scalable to
  \[ p_w = \Theta\left(2^{(\gamma/\alpha)n^2}\right) \]
  processors, since
  \[ E_{p_w}^{2-D}(\sqrt{p_w}n) = \Theta\left(1/(1 + (\alpha/\gamma) \log(p_w)/n^2 + (\beta/\gamma)/n)\right) \]
Memory Requirements

- The memory requirements are dominated by storing $Z$, which in practice means the outer-product is a poor primitive (local flop-to-byte ratio of 1).
- If possible, $Z$ should be operated on as it is computed, e.g. if we really need
\[ v_i = \sum_j f(x_i y_j) \quad \text{for some scalar function } f \]
- If $Z$ does not need to be stored, vector storage dominates
- Without further modification, 1-D algorithm requires
  \[ M_p = \Theta(np) \quad \text{overall storage for vector} \]
- Without further modification, 2-D algorithm requires
  \[ M_p = \Theta(n\sqrt{p}) \quad \text{overall storage for vector} \]
Matrix-Vector Product

- Consider matrix-vector product
  
  \[ y = Ax \]

  where \( A \) is \( n \times n \) matrix and \( x \) and \( y \) are \( n \)-vectors

- Components of vector \( y \) are given by inner products:
  
  \[ y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \ldots, n \]

- The sequential memory, work, and time are
  
  \[ M_1 = \Theta(n^2), \quad Q_1 = \Theta(n^2), \quad T_1 = \Theta(\gamma n^2) \]
Parallel Algorithm

**Partition**

- For $i, j = 1, \ldots, n$, fine-grain task $(i, j)$ stores $a_{ij}$ and computes $a_{ij} x_j$, yielding 2-D array of $n^2$ fine-grain tasks.
- Assuming no replication of data, at most $2n$ fine-grain tasks store components of $x$ and $y$, say either:
  - for some $j$, task $(j, i)$ stores $x_i$ and task $(i, j)$ stores $y_i$, or
  - task $(i, i)$ stores both $x_i$ and $y_i$, $i = 1, \ldots, n$.

**Communicate**

- For $j = 1, \ldots, n$, task that stores $x_j$ broadcasts it to all other tasks in $j$th task column.
- For $i = 1, \ldots, n$, sum reduction over $i$th task row gives $y_i$. 
Fine-Grain Tasks and Communication

\[ a_{11}x_1 + y_1 \]
\[ a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5 + a_{16}x_6 \]
\[ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5 + a_{26}x_6 \]
\[ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 + a_{36}x_6 \]
\[ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5 + a_{46}x_6 \]
\[ a_{51}x_1 + a_{52}x_2 + a_{53}x_3 + a_{54}x_4 + a_{55}x_5 + a_{56}x_6 \]
\[ a_{61}x_1 + a_{62}x_2 + a_{63}x_3 + a_{64}x_4 + a_{65}x_5 + a_{66}x_6 \]
Fine-Grain Parallel Algorithm

broadcast $x_j$ to tasks $(k, j), \ k = 1, \ldots, n$ \quad \{ \text{vertical broadcast} \} \\

$y_i = a_{ij} x_j$ \quad \{ \text{local scalar product} \} \\

reduce $y_i$ across tasks $(i, k), \ k = 1, \ldots, n$ \quad \{ \text{horizontal sum reduction} \}
Agglomeration

\textit{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

- Subvector of $x$ broadcast along each task column
- Each task computes local matrix-vector product of submatrix of $A$ with subvector of $x$
- Sum reduction along each task row produces subvector of result $y$
2-D Agglomeration

\[
\begin{align*}
& a_{11}x_1 & a_{12}x_2 \\
& y_1 & a_{22}x_2 \\
& a_{21}x_1 & y_2 \\
& a_{31}x_1 & a_{32}x_2 \\
& y_3 & a_{43}x_3 \\
& a_{41}x_1 & a_{42}x_2 \\
& a_{51}x_1 & a_{52}x_2 \\
& y_5 & a_{63}x_3 \\
& a_{61}x_1 & a_{62}x_2 \\
& a_{54}x_4 & y_6 \\
& a_{64}x_4 \\
& a_{14}x_4 & a_{15}x_5 \\
& a_{16}x_6 \\
& a_{25}x_5 & a_{26}x_6 \\
& a_{35}x_5 & a_{36}x_6 \\
& a_{45}x_5 & a_{46}x_6 \\
& a_{55}x_5 \\
& a_{65}x_5 \\
& a_{56}x_6 \\
& a_{66}x_6 \\
\end{align*}
\]
1-D Agglomeration

1-D column agglomeration

- Each task computes product of its component of \( x \) times its column of matrix, with no communication required
- Sum reduction across tasks then produces \( y \)

1-D row agglomeration

- If \( x \) stored in one task, then broadcast required to communicate needed values to all other tasks
- If \( x \) distributed across tasks, then multinode broadcast required to communicate needed values to other tasks
- Each task computes inner product of its row of \( A \) with entire vector \( x \) to produce its component of \( y \)
1-D Column Agglomeration
1-D Row Agglomeration

\[
\begin{align*}
\left( a_{11}x_1, y_1 \right) & \quad \left( a_{12}x_2 \right) & \quad \left( a_{13}x_3 \right) & \quad \left( a_{14}x_4 \right) & \quad \left( a_{15}x_5 \right) & \quad \left( a_{16}x_6 \right) \\
\left( a_{21}x_1 \right) & \quad \left( a_{22}x_2, y_2 \right) & \quad \left( a_{23}x_3 \right) & \quad \left( a_{24}x_4 \right) & \quad \left( a_{25}x_5 \right) & \quad \left( a_{26}x_6 \right) \\
\left( a_{31}x_1 \right) & \quad \left( a_{32}x_2 \right) & \quad \left( a_{33}x_3, y_3 \right) & \quad \left( a_{34}x_4 \right) & \quad \left( a_{35}x_5 \right) & \quad \left( a_{36}x_6 \right) \\
\left( a_{41}x_1 \right) & \quad \left( a_{42}x_2 \right) & \quad \left( a_{43}x_3 \right) & \quad \left( a_{44}x_4, y_4 \right) & \quad \left( a_{45}x_5 \right) & \quad \left( a_{46}x_6 \right) \\
\left( a_{51}x_1 \right) & \quad \left( a_{52}x_2 \right) & \quad \left( a_{53}x_3 \right) & \quad \left( a_{54}x_4 \right) & \quad \left( a_{55}x_5, y_5 \right) & \quad \left( a_{56}x_6 \right) \\
\left( a_{61}x_1 \right) & \quad \left( a_{62}x_2 \right) & \quad \left( a_{63}x_3 \right) & \quad \left( a_{64}x_4 \right) & \quad \left( a_{65}x_5 \right) & \quad \left( a_{66}x_6, y_6 \right)
\end{align*}
\]
1-D Agglomeration

Column and row algorithms are dual to each other

- Column algorithm begins with communication-free local vector scaling ($\text{daxpy}$) computations combined across processors by a reduction
- Row algorithm begins with broadcast followed by communication-free local inner-product ($\text{ddot}$) computations
**Mapping**

*Map*

- 2-D: Assign \((n/k)^2/p\) coarse-grain tasks to each of \(p\) processes 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 \(p\) processes using any desired mapping, treating target network as 1-D mesh
2-D Agglomeration with Block Mapping
1-D Column Agglomeration with Block Mapping
1-D Row Agglomeration with Block Mapping

\[ a_{11}x_1 \]
\[ y_1 \]
\[ a_{12}x_2 \]
\[ a_{13}x_3 \]
\[ a_{14}x_4 \]
\[ a_{15}x_5 \]
\[ a_{16}x_6 \]
\[ a_{21}x_1 \]
\[ a_{22}x_2 \]
\[ a_{23}x_3 \]
\[ a_{24}x_4 \]
\[ a_{25}x_5 \]
\[ a_{26}x_6 \]
\[ a_{31}x_1 \]
\[ a_{32}x_2 \]
\[ a_{33}x_3 \]
\[ a_{34}x_4 \]
\[ a_{35}x_5 \]
\[ a_{36}x_6 \]
\[ a_{41}x_1 \]
\[ a_{42}x_2 \]
\[ a_{43}x_3 \]
\[ a_{44}x_4 \]
\[ a_{45}x_5 \]
\[ a_{46}x_6 \]
\[ a_{51}x_1 \]
\[ a_{52}x_2 \]
\[ a_{53}x_3 \]
\[ a_{54}x_4 \]
\[ a_{55}x_5 \]
\[ a_{56}x_6 \]
\[ a_{61}x_1 \]
\[ a_{62}x_2 \]
\[ a_{63}x_3 \]
\[ a_{64}x_4 \]
\[ a_{65}x_5 \]
\[ a_{66}x_6 \]
Coarse-Grain Parallel Algorithm

\[
\begin{align*}
\text{broadcast } x_{[j]} \text{ to } j\text{th process column} & \quad \{ \text{vertical broadcast} \} \\
y_{[i]} &= A_{[i][j]} x_{[j]} & \quad \{ \text{local matrix-vector product} \} \\
\text{reduce } y_{[i]} \text{ across } i\text{th process row} & \quad \{ \text{horizontal sum reduction} \}
\end{align*}
\]
The parallel costs \((L_p, W_p, F_p)\) for the matrix-vector product are

- Computational cost \(F_p = \Theta(n^2/p)\) regardless of network
- Communication costs can be derived from the cost of collectives
  - 1-D agglomeration: \(L_p = \Theta(\log p), W_p = \Theta(n)\)
  - 2-D agglomeration: \(L_p = \Theta(\log p), W_p = \Theta(n/\sqrt{p})\)

For 1-D row agglomeration, perform allgathere,
\[
T_p^{1-D} = T_p^{\text{allgathere}}(n) + \Theta(\gamma n^2/p) = \Theta(\alpha \log(p) + \beta n + \gamma n^2/p)
\]

For 2-D agglomeration, perform broadcast and reduction,
\[
T_p^{2-D} = T_\sqrt{p}^{\text{bcast}}(n/\sqrt{p}) + T_\sqrt{p}^{\text{reduce}}(n/\sqrt{p}) + \Theta(\gamma n^2/p)
= \Theta(\alpha \log(p) + \beta n/\sqrt{p} + \gamma n^2/p)
\]
Matrix-Matrix Product

- Consider matrix-matrix product
  \[ C = A \cdot B \]
  where \( A, B, \) and result \( C \) are \( n \times n \) matrices
- Entries of matrix \( C \) are given by
  \[ c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \quad i, j = 1, \ldots, n \]
- Each of \( n^2 \) entries of \( C \) requires \( n \) multiply-add operations, so model serial time as
  \[ T_1 = \gamma n^3 \]
Matrix-matrix product can be viewed as
- $n^2$ inner products, or
- sum of $n$ outer products, or
- $n$ matrix-vector products

and each viewpoint yields different algorithm

One way to derive parallel algorithms for matrix-matrix product is to apply parallel algorithms already developed for inner product, outer product, or matrix-vector product

However, considering the problem as a whole yields the best algorithms
For $i, j, k = 1, \ldots, n$, fine-grain task $(i, j, k)$ computes product $a_{ik} b_{kj}$, yielding 3-D array of $n^3$ fine-grain tasks.

Assuming no replication of data, at most $3n^2$ fine-grain tasks store entries of $A$, $B$, or $C$, say task $(i, j, j)$ stores $a_{ij}$, task $(i, j, i)$ stores $b_{ij}$, and task $(i, j, k)$ stores $c_{ij}$ for $i, j = 1, \ldots, n$ and some fixed $k$.

We refer to subsets of tasks along $i$, $j$, and $k$ dimensions as rows, columns, and layers, respectively, so $k$th column of $A$ and $k$th row of $B$ are stored in $k$th layer of tasks.
Parallel Algorithm

Communicate

- Broadcast entries of $j$th column of $A$ horizontally along each task row in $j$th layer
- Broadcast entries of $i$th row of $B$ vertically along each task column in $i$th layer
- For $i, j = 1, \ldots, n$, result $c_{ij}$ is given by sum reduction over tasks $(i, j, k), k = 1, \ldots, n$
Fine-Grain Algorithm

broadcast $a_{ik}$ to tasks $(i, q, k), \ q = 1, \ldots, n$ \hspace{1cm} \{ horizontal broadcast \}

broadcast $b_{kj}$ to tasks $(q, j, k), \ q = 1, \ldots, n$ \hspace{1cm} \{ vertical broadcast \}

$c_{ij} = a_{ik}b_{kj}$ \hspace{1cm} \{ local scalar product \}

reduce $c_{ij}$ across tasks $(i, j, q), \ q = 1, \ldots, n$ \hspace{1cm} \{ lateral sum reduction \}
Agglomeration

Agglomerate

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

- 3-D: Combine \( q \times q \times q \) subarray of fine-grain tasks
- 2-D: Combine \( q \times q \times n \) subarray of fine-grain tasks, eliminating sum reductions
- 1-D column: Combine \( n \times 1 \times n \) subarray of fine-grain tasks, eliminating vertical broadcasts and sum reductions
- 1-D row: Combine \( 1 \times n \times n \) subarray of fine-grain tasks, eliminating horizontal broadcasts and sum reductions
Mapping

Corresponding mapping strategies are

- **3-D:** Assign \(\frac{(n/q)^3}{p}\) coarse-grain tasks to each of \(p\) processors using any desired mapping in each dimension, treating target network as 3-D mesh.

- **2-D:** Assign \(\frac{(n/q)^2}{p}\) 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 \(\frac{n}{p}\) coarse-grain tasks to each of \(p\) processors using any desired mapping, treating target network as 1-D mesh.
Agglomeration with Block Mapping

- 1-D row
- 1-D col
- 2-D
- 3-D
Coarse-Grain 3-D Parallel Algorithm

broadcast $A_{[i][k]}$ to $i$th processor row  \{ horizontal broadcast \}

broadcast $B_{[k][j]}$ to $j$th processor column  \{ vertical broadcast \}

$C_{[i][j]} = A_{[i][k]} B_{[k][j]}$  \{ local matrix product \}

reduce $C_{[i][j]}$ across processor layers  \{ lateral sum reduction \}
Agglomeration with Block Mapping

2-D:

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= \begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix}
\]

1-D column:

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= \begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix}
\]

1-D row:

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= \begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix}
\]
Coarse-Grain 2-D Parallel Algorithm

\[ \text{allgather } A_{[i][j]} \text{ in } i\text{th processor row} \]
\[ \text{allgather } B_{[i][j]} \text{ in } j\text{th processor column} \]
\[ C_{[i][j]} = 0 \]
\[ \text{for } k = 1, \ldots, \sqrt{p} \]
\[ C_{[i][j]} = C_{[i][j]} + A_{[i][k]} B_{[k][j]} \]
\[ \text{end} \]
Algorithm just described requires excessive memory, since each process accumulates $\sqrt{p}$ blocks of both $A$ and $B$.

One way to reduce memory requirements is to:
- broadcast blocks of $A$ successively across processor rows
- broadcast blocks of $B$ successively across processor cols

\[
\begin{align*}
C[i][j] &= 0 \\
\text{for } k &= 1, \ldots, \sqrt{p} \\
\text{broadcast } A[i][k] &\text{ in } i\text{th processor row} \\
\text{broadcast } B[k][j] &\text{ in } j\text{th processor column} \\
\end{align*}
\]

{ horizontal broadcast }  
{ vertical broadcast }  
{ sum local products }
SUMMA Algorithm
Another approach, due to Cannon (1969), is to circulate blocks of $B$ vertically and blocks of $A$ horizontally in ring fashion.

Blocks of both matrices must be initially aligned using circular shifts so that correct blocks meet as needed.

Requires less memory than SUMMA and replaces line broadcasts with shifts, lowering the number of messages.
Cannon Algorithm

Abbreviations:
- BLAS: Basic Linear Algebra Subprograms
- Inner Product
- Outer Product
- Matrix-Vector Product
- Matrix-Matrix Product

Parallel Algorithm
- Agglomeration Schemes
- Scalability

Michael T. Heath and Edgar Solomonik
Parallel Numerical Algorithms 63 / 77
It is possible to mix techniques from SUMMA and Cannon’s algorithm:

- circulate blocks of $B$ in ring fashion vertically along processor columns step by step so that each block of $B$ comes in conjunction with appropriate block of $A$ broadcast at that same step

This algorithm is due to Fox et al.

Shifts, especially in Cannon’s algorithm, are harder to generalize to nonsquare processor grids than collectives in algorithms like SUMMA
Execution Time for 3-D Agglomeration

- For 3-D agglomeration, computing each of $p$ blocks $C_{[i][j]}$ requires matrix-matrix product of two $(n/\sqrt[3]{p}) \times (n/\sqrt[3]{p})$ blocks, so
  
  $$F_p = (n/\sqrt[3]{p})^3 = n^3/p$$

- On 3-D mesh, each broadcast or reduction takes time
  
  $$T_{p}^{\text{bcast}} ((n/p^{1/3})^2) = O(\alpha \log p + \beta n^2 / p^{2/3})$$

- Total time is therefore
  
  $$T_p = O(\alpha \log p + \beta n^2 / p^{2/3} + \gamma n^3 / p)$$

- However, overall memory footprint is
  
  $$M_p = \Theta(p(n/p^{1/3})^2) = \Theta(p^{1/3}n^2)$$
The 3-D agglomeration efficiency is given by

\[ E_p(n) = \frac{pT_1(n)}{T_p(n)} = O \left( \frac{1}{1 + (\alpha/\gamma)p \log p/n^3 + (\beta/\gamma)p^{1/3}/n} \right) \]

For strong scaling to \( p_s \) processors we need \( E_{p_s}(n) = \Theta(1) \), so 3-D agglomeration strong scales to

\[ p_s = O(\min((\gamma/\alpha)n^3/\log(n), (\gamma/\beta)n^3)) \] processors
Weak Scalability of 3-D Agglomeration

- For weak scaling (with constant input size / processor) to $p_w$ processor, we need $E_{p_w}(n\sqrt{p_w}) = \Theta(1)$, which holds.

- However, note that the algorithm is not memory-efficient as $M_p = \Theta(p^{1/3}n^2)$, and if keeping memory footprint per processor constant, we must grow $n$ with $p^{1/3}$.

- Scaling with constant memory footprint processor ($M_p/p = \text{const}$) is possible to $p_m$ processors where $E_{p_m}(np_m^{1/3}) = \Theta(1)$, which holds for $p_m = \Theta(2^{(\gamma/\alpha)n^3})$ processors.

- The isoefficiency function is $\tilde{Q}(p) = \Theta(p \log(p))$. 

---

Michael T. Heath and Edgar Solomonik

Parallel Numerical Algorithms
Execution Time for 2-D Agglomeration

For 2-D agglomeration (SUMMA), computation of each block $C_{i,j}$ requires $\sqrt{p}$ matrix-matrix products of $(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks, so

$$F_p = \sqrt{p} \left(\frac{n}{\sqrt{p}}\right)^3 = \frac{n^3}{p}$$

For broadcast among rows and columns of processor grid, communication time is

$$2\sqrt{p}T_{\text{broadcast}}^{\text{bcast}} \left(\frac{n^2}{p}\right) = \Theta(\alpha \sqrt{p} \log(p) + \beta n^2 / \sqrt{p})$$

Total time is therefore

$$T_p = O(\alpha \sqrt{p} \log(p) + \beta n^2 / \sqrt{p} + \gamma n^3 / p)$$

The algorithm is memory-efficient, $M_p = \Theta(n^2)$
Strong Scalability of 2-D Agglomeration

- The 2-D agglomeration efficiency is given by
  \[ E_p(n) = \frac{pT_1(n)}{T_p(n)} = O\left(\frac{1}{1 + (\alpha/\gamma)p^{3/2} \log p/n^3 + (\beta/\gamma) \sqrt{p/n}}\right) \]

- For strong scaling to \( p_s \) processors we need 
  \( E_{p_s}(n) = \Theta(1) \), so 2-D agglomeration strong scales to 
  \[ p_s = O\left(\min\left((\gamma/\alpha)n^2 / \log(n)^{2/3}, (\gamma/\beta)n^2\right)\right) \] processors

- For weak scaling to \( p_w \) processors with \( n^2/p \) matrix elements per processor, we need 
  \( E_{p_w}(\sqrt{p_w}n) = \Theta(1) \), so 2-D agglomeration (SUMMA) weak scales to 
  \[ p_w = O\left(2^{(\gamma/\alpha)n^3}\right) \] processors

- Cannon’s algorithm achieves unconditional weak scalability
Scalability for 1-D Agglomeration

For 1-D agglomeration on 1-D mesh, total time is

\[ T_p = O(\alpha \log(p) + \beta n^2 + \gamma n^3 / p) \]

The corresponding efficiency is

\[ E_p = O\left(\frac{1}{1 + (\alpha/\beta)p \log(p)n^3 + (\beta/\gamma)p/n}\right) \]

Strong scalability is possible to at most \( p_s = O((\gamma/\beta)n) \) processors

Weak scalability is possible to at most \( p_w = O(\sqrt{(\gamma/\beta)n}) \) processors
Rectangular Matrix Multiplication

If $C$ is $m \times n$, $A$ is $m \times k$, and $B$ is $k \times n$, choosing a 3D grid that optimizes volume-to-surface-area ratio yields bandwidth cost...

$$W_p(m, n, k) = O\left( \min_{p_1p_2p_3=p} \left[ \frac{mk}{p_1p_2} + \frac{kn}{p_1p_3} + \frac{mn}{p_2p_3} \right] \right)$$
Communication vs. Memory Tradeoff

- Communication cost for 2-D algorithms for matrix-matrix product is optimal, assuming no replication of storage.
- If explicit replication of storage is allowed, then lower communication volume is possible via 3-D algorithm.
- Generally, we assign $\frac{n}{p_1} \times \frac{n}{p_2} \times \frac{n}{p_3}$ computation subvolume to each processor.
- The largest face of the subvolume gives communication cost, the smallest face gives minimal memory usage.
  - can keep smallest face local while successively computing slices of subvolume.
Leveraging Additional Memory in Matrix Multiplication

- Provided $\bar{M}$ total available memory, 2-D and 3-D algorithms achieve bandwidth cost

$$W_p(n, \bar{M}) = \begin{cases} 
\infty & : \bar{M} < n^2 \\
n^2/\sqrt{p} & : \bar{M} = \Theta(n^2) \\n^2/p^{2/3} & : \bar{M} = \Theta(n^2 p^{1/3}) 
\end{cases}$$

- For general $\bar{M}$, possible to pick processor grid to achieve

$$W_p(n, \bar{M}) = O(n^3/(\sqrt{p}\bar{M}^{1/2}) + n^2/p^{2/3})$$

- and an overall execution time of

$$T_p(n, \bar{M}) = O(\alpha(\log p + n^3\sqrt{p}/\bar{M}^{3/2}) + \beta W_p(n, \bar{M}) + \gamma n^3/p)$$
Strong Scaling using All Available Memory

- The efficiency of the algorithm for a given amount of total memory $\bar{M}_p$ is

$$E_p(n, \bar{M}_p) = O\left(\frac{1}{1 + (\alpha/\gamma)(p \log p/n^3 + p^{3/2}/\bar{M}_p^{3/2})} + (\beta/\gamma)(\sqrt{p}/\bar{M}_p^{1/2} + p^{1/3}/n)\right)$$

- For strong scaling assuming $\bar{M}_p = p\bar{M}_1$, we need

$$E_{ps}(n, p_s\bar{M}_1) = p_sT_1(n, \bar{M}_1)/T_{ps}(n, p_s\bar{M}_1) = \Theta(1)$$

- In this case, we obtain

$$p_s = \Theta(\min((\alpha/\gamma)n^3 / \log(n), (\beta/\gamma)n^3))$$

as good as the 3-D algorithm, but more memory-efficient
References


References

References


