Outline

1. Inner Product
2. Outer Product
3. Matrix-Vector Product
4. 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>saxpy</td>
<td>Scalar $\times$ vector $+$ vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sdot</td>
<td>Inner product</td>
</tr>
<tr>
<td></td>
<td></td>
<td>snrm2</td>
<td>Euclidean vector norm</td>
</tr>
<tr>
<td>2</td>
<td>$O(n^2)$</td>
<td>sgemv</td>
<td>Matrix-vector product</td>
</tr>
<tr>
<td></td>
<td></td>
<td>strsv</td>
<td>Triangular solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sger</td>
<td>Rank-one update</td>
</tr>
<tr>
<td>3</td>
<td>$O(n^3)$</td>
<td>sgemm</td>
<td>Matrix-matrix product</td>
</tr>
<tr>
<td></td>
<td></td>
<td>strm</td>
<td>Multiple triang. solutions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ssyrk</td>
<td>Rank-$k$ update</td>
</tr>
</tbody>
</table>
For problem of dimension $n$ using $p$ processes, assume $p$ (or in some cases $\sqrt{p}$) divides $n$

For 2-D mesh, assume $p$ is perfect square and mesh is $\sqrt{p} \times \sqrt{p}$

For hypercube, assume $p$ is power of two

Assume matrices are square, $n \times n$, not rectangular

Dealing with general cases where these assumptions do not hold is straightforward but tedious, and complicates notation

Caveat: your mileage may vary, depending on assumptions about target system, such as level of concurrency in communication
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

- For simplicity, model serial time as
  \[ T_1 = t_c n \]

where $t_c$ is time for one scalar multiply-add operation
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

\[ z = x_i y_i \]

reduce \( z \) across all tasks

\{ local scalar product \}

\{ 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 $p$ processes, for total of $n/p$ components of $x$ and $y$ per process

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

\[ z = x^T_i y_i \]  \{ local inner product \}

reduce \( z \) across all processes  \{ sum reduction \}

\([x_i]\) means subvector of \( x \) assigned to process \( i \) by mapping
Time for computation phase is

\[ T_{\text{comp}} = t_c \frac{n}{p} \]

regardless of network

Depending on network, time for communication phase is

- 1-D mesh: \( T_{\text{comm}} = (t_s + t_w)(p - 1) \)
- 2-D mesh: \( T_{\text{comm}} = (t_s + t_w)2(\sqrt{p} - 1) \)
- hypercube: \( T_{\text{comm}} = (t_s + t_w)\log p \)

For simplicity, ignore cost of additions in reduction, which is usually negligible
Scalability for 1-D Mesh

- For 1-D mesh, total time is

\[ T_p = t_c \frac{n}{p} + (t_s + t_w) (p - 1) \]

- To determine isoefficiency function, set

\[ T_1 \approx E(p T_p) \]
\[ t_c n \approx E(t_c n + (t_s + t_w) p (p - 1)) \]

which holds if \( n = \Theta(p^2) \), so isoefficiency function is \( \Theta(p^2) \), since \( T_1 = \Theta(n) \)
Scalability for 2-D Mesh

- For 2-D mesh, total time is

\[ T_p = \frac{t_c n}{p} + (t_s + t_w) 2(\sqrt{p} - 1) \]

- To determine isoefficiency function, set

\[ t_c n \approx E \left( t_c n + (t_s + t_w) p 2(\sqrt{p} - 1) \right) \]

which holds if \( n = \Theta(p^{3/2}) \), so isoefficiency function is \( \Theta(p^{3/2}) \), since \( T_1 = \Theta(n) \)
Scalability for Hypercube

- For hypercube, total time is
  \[ T_p = t_c \frac{n}{p} + (t_s + t_w) \log p \]

- To determine isoefficiency function, set
  \[ t_c n \approx E (t_c n + (t_s + t_w) p \log p) \]

  which holds if \( n = \Theta(p \log p) \), so isoefficiency function is \( \Theta(p \log p) \), since \( T_1 = \Theta(n) \)
Optimality for 1-D Mesh

To determine optimal number of processes for given $n$, take $p$ to be continuous variable and minimize $T_p$ with respect to $p$.

For 1-D mesh

$$T_p' = \frac{d}{dp} \left[ t_c \frac{n}{p} + (t_s + t_w)(p - 1) \right]$$

$$= -t_c \frac{n}{p^2} + (t_s + t_w) = 0$$

implies that optimal number of processes is

$$p \approx \sqrt{\frac{t_c n}{t_s + t_w}}$$
Optimality for 1-D Mesh

- If $n < (t_s + t_w)/t_c$, then only one process should be used.
- Substituting optimal $p$ into formula for $T_p$ shows that optimal time to compute inner product grows as $\sqrt{n}$ with increasing $n$ on 1-D mesh.
For hypercube

\[ T'_p = \frac{d}{dp} \left[ t_c n/p + (t_s + t_w) \log p \right] \]

\[ = -t_c n/p^2 + (t_s + t_w)/p = 0 \]

implies that optimal number of processes is

\[ p \approx \frac{t_c n}{t_s + t_w} \]

and optimal time grows as \( \log n \) with increasing \( n \)
Outer Product

- Outer product of two \( n \)-vectors \( x \) and \( y \) is \( n \times n \) matrix 
  \[ Z = xy^T \]  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, so model serial time as
  \[ T_1 = t_c n^2 \]

where \( t_c \) is time for one scalar multiplication.
Parallel Algorithm

**Partition**

- For $i, j = 1, \ldots, n$, fine-grain task $(i, j)$ computes and stores $z_{ij} = x_i y_j$, 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 $(i, j)$ stores $x_i$ and task $(j, i)$ stores $y_i$, or
  - task $(i, i)$ stores both $x_i$ and $y_i$, $i = 1, \ldots, n$

**Communicate**

- For $i = 1, \ldots, n$, task that stores $x_i$ broadcasts it to all other tasks in $i$th task row.
- For $j = 1, \ldots, n$, task that stores $y_j$ broadcasts it to all other tasks in $j$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 \}
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

- 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
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

\[ \begin{aligned}
&x_1 y_1 \quad x_1 y_2 \quad x_1 y_3 \quad x_1 y_4 \quad x_1 y_5 \quad x_1 y_6 \\
&x_2 y_1 \quad x_2 y_2 \quad x_2 y_3 \quad x_2 y_4 \quad x_2 y_5 \quad x_2 y_6 \\
&x_3 y_1 \quad x_3 y_2 \quad x_3 y_3 \quad x_3 y_4 \quad x_3 y_5 \quad x_3 y_6 \\
&x_4 y_1 \quad x_4 y_2 \quad x_4 y_3 \quad x_4 y_4 \quad x_4 y_5 \quad x_4 y_6 \\
&x_5 y_1 \quad x_5 y_2 \quad x_5 y_3 \quad x_5 y_4 \quad x_5 y_5 \quad x_5 y_6 \\
&x_6 y_1 \quad x_6 y_2 \quad x_6 y_3 \quad x_6 y_4 \quad x_6 y_5 \quad x_6 y_6 
\end{aligned} \]
1-D Row Agglomeration
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

\[ \begin{array}{cccccc}
\bar{x}_1 y_1 & x_1 y_2 & \bar{x}_2 y_2 & x_2 y_3 & x_2 y_4 & x_2 y_5 \\
\bar{x}_2 y_1 & \bar{x}_2 y_2 & \bar{x}_2 y_3 & \bar{x}_2 y_4 & \bar{x}_2 y_5 & \bar{x}_2 y_6 \\
\bar{x}_3 y_1 & \bar{x}_3 y_2 & \bar{x}_3 y_3 & \bar{x}_3 y_4 & \bar{x}_3 y_5 & \bar{x}_3 y_6 \\
\bar{x}_4 y_1 & \bar{x}_4 y_2 & \bar{x}_4 y_3 & \bar{x}_4 y_4 & \bar{x}_4 y_5 & \bar{x}_4 y_6 \\
\bar{x}_5 y_1 & \bar{x}_5 y_2 & \bar{x}_5 y_3 & \bar{x}_5 y_4 & \bar{x}_5 y_5 & \bar{x}_5 y_6 \\
\bar{x}_6 y_1 & \bar{x}_6 y_2 & \bar{x}_6 y_3 & \bar{x}_6 y_4 & \bar{x}_6 y_5 & \bar{x}_6 y_6 \\
\end{array} \]
1-D Column Agglomeration with Block Mapping
1-D Row Agglomeration with Block Mapping
Coarse-Grain Parallel Algorithm

broadcast $\mathbf{x}[i]$ to $i$th process row  \hspace{1cm} \{ horizontal broadcast \}

broadcast $\mathbf{y}[j]$ to $j$th process column \hspace{1cm} \{ vertical broadcast \}

$\mathbf{Z}[i][j] = \mathbf{x}[i] \mathbf{y}[j]^T$ \hspace{1cm} \{ local outer product \}

$[\mathbf{Z}[i][j]]$ means submatrix of $\mathbf{Z}$ assigned to process $(i, j)$ by mapping
Scalability

- Time for computation phase is
  \[ T_{\text{comp}} = t_c \frac{n^2}{p} \]
  regardless of network or agglomeration scheme

- For 2-D agglomeration on 2-D mesh, communication time is at least
  \[ T_{\text{comm}} = (t_s + t_w \frac{n}{\sqrt{p}}) (\sqrt{p} - 1) \]
  assuming broadcasts can be overlapped
Scalability for 2-D Mesh

- Total time for 2-D mesh is at least

\[ T_p = \frac{t_c n^2}{p} + (t_s + t_w n / \sqrt{p}) (\sqrt{p} - 1) \]

\[ \approx t_c n^2 / p + t_s \sqrt{p} + t_w n \]

- To determine isoefficiency function, set

\[ t_c n^2 \approx E \left( t_c n^2 + t_s p^{3/2} + t_w n p \right) \]

which holds for large \( p \) if \( n = \Theta(p) \), so isoefficiency function is \( \Theta(p^2) \), since \( T_1 = \Theta(n^2) \)
Scalability for Hypercube

- Total time for hypercube is at least
  \[ T_p = \frac{t_c n^2}{p} + \left( t_s + t_w \frac{n}{\sqrt{p}} \right) \left( \log p \right)/2 \]
  \[ = \frac{t_c n^2}{p} + t_s \left( \log p \right)/2 + t_w n \left( \log p \right)/(2\sqrt{p}) \]

- To determine isoefficiency function, set
  \[ t_c n^2 \approx E \left( t_c n^2 + t_s p \left( \log p \right)/2 + t_w n \sqrt{p} \left( \log p \right)/2 \right) \]
  which holds for large \( p \) if \( n = \Theta(\sqrt{p} \log p) \), so isoefficiency function is \( \Theta(p (\log p)^2) \), since \( T_1 = \Theta(n^2) \)
Scalability for 1-D mesh

- Depending on network, time for communication phase with 1-D agglomeration is at least
  - 1-D mesh: $T_{\text{comm}} = (t_s + t_w \frac{n}{p}) (p - 1)$
  - 2-D mesh: $T_{\text{comm}} = (t_s + t_w \frac{n}{p}) 2(\sqrt{p} - 1)$
  - hypercube: $T_{\text{comm}} = (t_s + t_w \frac{n}{p}) \log p$

assuming broadcasts can be overlapped
Scalability for 1-D Mesh

For 1-D mesh, total time is at least

\[ T_p = t_c \frac{n^2}{p} + (t_s + t_w \frac{n}{p}) (p - 1) \]

\[ \approx t_c \frac{n^2}{p} + t_s p + t_w n \]

To determine isoefficiency function, set

\[ t_c n^2 \approx E (t_c n^2 + t_s p^2 + t_w n p) \]

which holds if \( n = \Theta(p) \), so isoefficiency function is \( \Theta(p^2) \), since \( T_1 = \Theta(n^2) \)
Memory Requirements

- With either 1-D or 2-D algorithm, straightforward broadcasting of \( x \) or \( y \) could require as much total memory as replication of entire vector in all processes.

- Memory requirements can be reduced by circulating portions of \( x \) or \( y \) through processes in ring fashion, with each process using each portion as it passes through, so that no process need store entire vector at once.
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
  \[ y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \ldots, n \]

- Each of \( n \) components requires \( n \) multiply-add operations, so model serial time as
  \[ T_1 = t_c n^2 \]
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
broadcast $x_j$ to tasks $(k, j)$, $k = 1, \ldots, n$ \{ vertical broadcast \}

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

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

- 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

\[
\begin{align*}
\begin{array}{c}
    a_{11}x_1 \\
    y_1 \\
    a_{21}x_1 \\
    a_{22}x_2 \\
    y_2 \\
    a_{31}x_1 \\
    a_{32}x_2 \\
    a_{41}x_1 \\
    a_{42}x_2 \\
    a_{51}x_1 \\
    a_{52}x_2 \\
    a_{61}x_1 \\
    a_{62}x_2 \\
\end{array} & \quad & \begin{array}{c}
    a_{12}x_2 \\
    a_{13}x_3 \\
    a_{23}x_3 \\
    a_{33}x_3 \\
    a_{43}x_3 \\
    a_{53}x_3 \\
    a_{63}x_3 \\
\end{array} & \quad & \begin{array}{c}
    a_{14}x_4 \\
    a_{15}x_5 \\
    a_{25}x_5 \\
    a_{35}x_5 \\
    a_{45}x_5 \\
    a_{55}x_5 \\
\end{array} & \quad & \begin{array}{c}
    a_{16}x_6 \\
    a_{26}x_6 \\
    a_{36}x_6 \\
    a_{46}x_6 \\
    a_{56}x_6 \\
    a_{66}x_6 \\
\end{array}
\end{align*}
\]
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

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

Column and row algorithms are dual to each other

- Column algorithm begins with communication-free local \texttt{saxpy} computations followed by sum reduction
- Row algorithm begins with broadcast followed by communication-free local \texttt{sdot} 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

Diagram showing the process of 2-D agglomeration with block mapping, involving operations with indices for different elements of matrices and vectors.
1-D Column Agglomeration with Block Mapping

$\begin{align*}
a_{11}x_1 & \quad a_{12}x_2 \\
y_1 & \quad a_{13}x_3
\end{align*}$

$\begin{align*}
a_{21}x_1 & \quad a_{22}x_2 \\
y_2 & \quad a_{23}x_3
\end{align*}$

$\begin{align*}
a_{31}x_1 & \quad a_{32}x_2 \\
y_3 & \quad a_{33}x_3
\end{align*}$

$\begin{align*}
a_{41}x_1 & \quad a_{42}x_2 \\
y_4 & \quad a_{43}x_3
\end{align*}$

$\begin{align*}
a_{51}x_1 & \quad a_{52}x_2 \\
y_5 & \quad a_{53}x_3
\end{align*}$

$\begin{align*}
a_{61}x_1 & \quad a_{62}x_2 \\
y_6 & \quad a_{63}x_3
\end{align*}$
1-D Row Agglomeration with Block Mapping
Coarse-Grain Parallel Algorithm

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

$y[i] = A[i][j] \mathbf{x}[j]$  \{ local matrix-vector product \}

reduce $y[i]$ across $i$th process row  \{ horizontal sum reduction \}
Scalability

- Time for computation phase is
  \[ T_{\text{comp}} = \frac{t_c n^2}{p} \]
  regardless of network or agglomeration scheme

- For 2-D agglomeration on 2-D mesh, each of two communication phases requires time
  \[ (t_s + t_w \frac{n}{\sqrt{p}}) (\sqrt{p} - 1) \approx t_s \sqrt{p} + t_w n \]
  so total time is
  \[ T_p \approx t_c \frac{n^2}{p} + 2(t_s \sqrt{p} + t_w n) \]
Scalability for 2-D Mesh

To determine isoefficiency function, set

\[ t_c n^2 \approx E \left( t_c n^2 + 2 \left( t_s p^{3/2} + t_w n p \right) \right) \]

which holds if \( n = \Theta(p) \), so isoefficiency function is \( \Theta(p^2) \), since \( T_1 = \Theta(n^2) \)
Total time for hypercube is

\[ T_p = \frac{t_c n^2}{p} + (t_s + t_w n/\sqrt{p}) \log p \]

\[ = t_c n^2/p + t_s \log p + t_w n \frac{(\log p)}{\sqrt{p}} \]

To determine isoefficiency function, set

\[ t_c n^2 \approx E (t_c n^2 + t_s p \log p + t_w n \sqrt{p} \log p) \]

which holds for large \( p \) if \( n = \Theta(\sqrt{p} \log p) \), so isoefficiency function is \( \Theta(p(\log p)^2) \), since \( T_1 = \Theta(n^2) \)
Scalability for 1-D Mesh

Depending on network, time for communication phase with 1-D agglomeration is at least

- 1-D mesh: $T_{\text{comm}} = (t_s + t_w n/p) (p - 1)$
- 2-D mesh: $T_{\text{comm}} = (t_s + t_w n/p) 2(\sqrt{p} - 1)$
- hypercube: $T_{\text{comm}} = (t_s + t_w n/p) \log p$

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

$$T_p = t_c n^2 / p + (t_s + t_w n/p) (p - 1) \approx t_c n^2 / p + t_s p + t_w n$$
To determine isoefficiency function, set

\[ tc \, n^2 \approx E \left( tc \, n^2 + ts \, p^2 + tw \, np \right) \]

which holds if \( n = \Theta(p) \), so isoefficiency function is \( \Theta(p^2) \), since \( T_1 = \Theta(n^2) \).
Matrix-Matrix Product

- Consider matrix-matrix product
  \[ C = A B \]
  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
  \[ T_1 = t_c n^3 \]
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.

We will develop parallel algorithms for this problem directly, however.
Parallel Algorithm

Partition

- 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 $(i, q, k), \; q = 1, \ldots, n$ \{ horizontal broadcast \}

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

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

reduce $c_{ij}$ across tasks $(i, j, q), \; q = 1, \ldots, n$ \{ 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

Map

Corresponding mapping strategies are

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

- **2-D**: Assign \( \left( \frac{n}{q} \right)^2/p \) 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.
Agglomeration with Block Mapping

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

\[ A[i][k] \] to \( i \)th process row

\[ B[k][j] \] to \( j \)th process column

\[ C[i][j] = A[i][k] B[k][j] \]

reduce \( C[i][j] \) across process layers

{ horizontal broadcast }

{ vertical broadcast }

{ local matrix product }

{ lateral sum reduction }
Agglomeration with Block Mapping

2-D:

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

1-D column:

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

1-D row:

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

\[
\begin{align*}
\text{all-to-all bcast } & A_{[i][j]} \text{ in } i\text{th process row} & \{ \text{horizontal broadcast} \} \\
\text{all-to-all bcast } & B_{[i][j]} \text{ in } j\text{th process column} & \{ \text{vertical broadcast} \} \\
C_{[i][j]} &= O \\
\text{for } & k = 1, \ldots, \sqrt{p} & \{ \text{sum local products} \} \\
C_{[i][j]} &= C_{[i][j]} + A_{[i][k]} B_{[k][j]} \\
\text{end}
\end{align*}
\]
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 process rows, and
- circulate blocks of $B$ in ring fashion vertically along process 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.
Another approach, due to Cannon, 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 even less memory than Fox algorithm, but trickier to program because of shifts required.

Performance and scalability of Fox and Cannon algorithms are not significantly different from that of previous 2-D algorithm, but memory requirements are much less.
For 3-D agglomeration, computing each of $p$ blocks $C_{i,j}$ requires matrix-matrix product of two $(n/3\sqrt{p}) \times (n/3\sqrt{p})$ blocks, so

$$T_{\text{comp}} = t_c (n/3\sqrt{p})^3 = t_c n^3/p$$

On 3-D mesh, each broadcast or reduction takes time

$$(t_s + t_w (n/3\sqrt{p})^2) (3\sqrt{p} - 1) \approx t_s p^{1/3} + t_w n^2/p^{1/3}$$

Total time is therefore

$$T_p = t_c n^3/p + 3t_s p^{1/3} + 3t_w n^2/p^{1/3}$$
Scalability for 3-D Agglomeration

- To determine isoefficiency function, set

\[ t_c n^3 \approx E (t_c n^3 + 3t_s p^{4/3} + 3t_w n^2 p^{2/3}) \]

which holds for large \( p \) if \( n = \Theta(p^{2/3}) \), so isoefficiency function is \( \Theta(p^2) \), since \( T_1 = \Theta(n^3) \)

- For hypercube, total time becomes

\[ T_p = t_c n^3/p + t_s \log p + t_w n^2 (\log p)/p^{2/3} \]

which leads to isoefficiency function of \( \Theta(p (\log p)^3) \)
Scalability for 2-D Agglomeration

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

$$T_{comp} = t_c \sqrt{p} \left(\frac{n}{\sqrt{p}}\right)^3 = t_c \frac{n^3}{p}$$

- For 2-D mesh, communication time for broadcasts along rows and columns is

$$T_{comm} = (t_s + t_w \frac{n^2}{p})(\sqrt{p} - 1) \approx t_s \sqrt{p} + t_w \frac{n^2}{\sqrt{p}}$$

assuming horizontal and vertical broadcasts can overlap (multiply by two otherwise)
Scalability for 2-D Agglomeration

- Total time for 2-D mesh is
  \[ T_p \approx t_c \frac{n^3}{p} + t_s \sqrt{p} + t_w \frac{n^2}{\sqrt{p}} \]

- To determine isoefficiency function, set
  \[ t_c n^3 \approx E \left( t_c n^3 + t_s p^{3/2} + t_w n^2 \sqrt{p} \right) \]
  which holds for large \( p \) if \( n = \Theta(\sqrt{p}) \), so isoefficiency function is \( \Theta(p^{3/2}) \), since \( T_1 = \Theta(n^3) \).
Scalability for 1-D Agglomeration

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

\[ T_p = \frac{t_c n^3}{p} + (t_s + t_w \frac{n^2}{p})(p - 1) \]

\[ \approx \frac{t_c n^3}{p} + t_s p + t_w n^2 \]

To determine isoefficiency function, set

\[ t_c n^3 \approx E (t_c n^3 + t_s p^2 + t_w n^2 p) \]

which holds for large \( p \) if \( n = \Theta(p) \), so isoefficiency function is \( \Theta(p^3) \) since \( T_1 = \Theta(n^3) \)
Communication vs. Memory Tradeoff

- Communication volume 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.
- Block-recursive 3-D algorithm can reduce communication volume by factor of $p^{-1/6}$ while increasing memory usage by factor of $p^{1/3}$.
- Recently, “2.5-D” algorithms have been developed that interpolate between 2-D and 3-D algorithms, using partial storage replication to reduce communication volume to whatever extent available memory allows.
References


References


References


