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.

Simplifying Assumptions

- 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

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

\[
\begin{align*}
    x_1 y_1 & \rightarrow x_2 y_2 \rightarrow x_3 y_3 \rightarrow \cdots \\
    & \rightarrow x_n y_n \\
\end{align*}
\]

**Fine-Grain Parallel Algorithm**

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

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

- Computation of inner product requires \( n \) multiplications and \( n - 1 \) additions
- For simplicity, model serial time as

\[
T_s = t_i n
\]

where \( t_i \) is time for one scalar multiply-add operation

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, sdot, snrm2</td>
<td>Scalar \times vector + vector Inner product Euclidean vector norm</td>
</tr>
<tr>
<td>2</td>
<td>( O(n^2) )</td>
<td>sgemv, strmv, sger</td>
<td>Matrix-vector product Triangular solution Rank-one update</td>
</tr>
<tr>
<td>3</td>
<td>( O(n^3) )</td>
<td>sgemm, strmm, ssyrk</td>
<td>Matrix-matrix product Multiple triang. solutions Rank-k update</td>
</tr>
</tbody>
</table>
Agglomeration and Mapping

Optimality for 1-D Mesh
- 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.

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
  $$T_p = \frac{t_c}{p} (n + (t_s + t_w) (p - 1))$$
  $$\frac{d}{dp} \left[ \frac{t_c}{p} (n + (t_s + t_w) (p - 1)) \right] = -\frac{t_c}{p^2} \left( n + (t_s + t_w) \right) = 0$$
  $$p \approx \frac{n}{t_s + t_w}$$

Time for computation phase is $T_{\text{comp}} = t_c n/p$ regardless of network.
- Depending on network, time for communication phase is
  - 1-D mesh: $T_{\text{comm}} = (t_c + t_w) (p - 1)$
  - 2-D mesh: $T_{\text{comm}} = (t_c + t_w) 2\sqrt{p} - 1$
  - hypercube: $T_{\text{comm}} = (t_c + t_w) \log_2 p$

For simplicity, ignore cost of additions in reduction, which is usually negligible.

Scalability for 2-D Mesh
- For 2-D mesh, total time is $T_p = t_c n/p + (t_s + t_w) (p - 1)$
- To determine isoefficiency function, set
  $$T_1 = E (p T_p)$$
  $$t_c n \approx E \left( t_c n + (t_s + t_w) p (p - 1) \right)$$
  which holds if $n = \Theta(p^2)$, so isoefficiency function is $\Theta(p^2)$, 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
  $$T_p = \frac{d}{dp} \left[ \frac{t_c}{p} (n + (t_s + t_w) (p - 1)) \right] = -\frac{t_c}{p^2} \left( n + (t_s + t_w) \right) = 0$$
  $$p \approx \frac{n}{t_s + t_w}$$

Coarse-Grain Parallel Algorithm
- $z = x[i]^T y[i] \quad \{$ local inner product $\}$
- reduce $z$ across all processes \{ sum reduction $\}$

$[x[i]]$ means subvector of $x$ assigned to process $i$ by mapping.

Scalability for 1-D Mesh
- For 1-D mesh, total time is $T_p = t_c n/p + (t_s + t_w) (p - 1)$
- To determine isoefficiency function, set
  $$T_1 = E (p T_p)$$
  $$t_c n \approx E \left( t_c n + (t_s + t_w) p (p - 1) \right)$$
  which holds if $n = \Theta(p^2)$, so isoefficiency function is $\Theta(p^2)$, since $T_1 = \Theta(n)$

Scalability for Hypercube
- For hypercube, total time is $T_p = t_c n/p + (t_s + t_w) \log p$
- 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
- 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.
Optimality for Hypercube

For hypercube

\[ T_p = \frac{d}{dp} \left[ t_c n/p + (t_s + t_w) \log p \right] \]

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 \( n \).

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 \( y_j \), 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 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 \}

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.

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,

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

- 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.

Fine-Grain Tasks and Communication

Agglomeration

- 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 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粗对 broadcast required to communicate needed values to other tasks

Matrix-Vector Product
Matrix-Matrix Product
Outer Product
Inner Product

1-D Column Agglomeration

- 2-D: Assign $(n/h)^2/p$ 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

Map

Coarse-Grain Parallel Algorithm

- Broadcast $x_{i,j}$ to $i$th process row
- Broadcast $y_{i,j}$ to $j$th process column
- $Z_{i,j} = Z_{i,j} M_{i,j}$

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

Scalability
Parallel Algorithm
Agglomeration Schemes
Scalability
Matrix-Vector Product

- Time for computation phase is
  \[ T_{\text{comp}} = t_c n^2 / p \]
  regardless of network or agglomeration scheme
- For 2-D agglomeration on 2-D mesh, communication time is at least
  \[ T_{\text{comm}} = (t_c + t_w n / \sqrt{p}) (\sqrt{p} - 1) \]
  assuming broadcasts can be overlapped

Scalability

- Total time for hypercube is at least
  \[ T_p = t_c n^2 / p + (t_c + t_w n / \sqrt{p}) (\log p)/2 \]
  \[ \approx t_c n^2 / p + t_p + t_w n \]
- To determine isoefficiency function, set
  \[ t_c n^2 \approx E (t_c n^2 / p + t_p + t_w n) \]
  which holds for large \( p \) if \( n = \Theta(\sqrt{p} \log p) \), so isoefficiency function is \( \Theta(p^2) \), since \( T_1 = \Theta(n^2) \)

Scalability for 1-D Mesh

- For 1-D mesh, total time is at least
  \[ T_p = t_c n^2 / p + (t_c + t_w n / \sqrt{p}) (p - 1) \]
  \[ \approx t_c n^2 / p + t_p + t_w n \]
- To determine isoefficiency function, set
  \[ t_c n^2 \approx E (t_c n^2 / p + t_p + t_w n) \]
  which holds if \( n = \Theta(p) \), so isoefficiency function is \( \Theta(p^2) \), since \( T_1 = \Theta(n^2) \)

Matrix-Matrix 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 \((i, j)\) stores \( x_j \) 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 \( y_j \) broadcasts it to all other tasks in \( j \)th task column
    - For \( i = 1, \ldots, n \), sum reduction over \( i \)th task row gives \( y_i \)
Agglomeration

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 $y$

1-D 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 Column Agglomeration

- If $x$ stored in one task, then broadcast required to communicate needed values to all other tasks
- If $x$ distributed across tasks, then min/max 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 $y$
Column and row algorithms are dual to each other:

- Column algorithm begins with communication-free local saxpy computations followed by sum reduction.
- Row algorithm begins with broadcast followed by communication-free local adot computations.

For 2-D agglomeration on 2-D mesh, each of two communication phases requires time:

\[
T_\text{comp} = t_c n^2 / p
\]

regardless of network or agglomeration scheme.

For 2-D agglomeration on 2-D mesh, each of two communication phases requires time:

\[
(t_s + t_w n / \sqrt{p}) (\sqrt{p} - 1) \approx t_s \sqrt{p} + t_w n
\]

so total time is:

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

To determine isoefficiency function, set:

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

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

- Total time for hypercube is
  \[ T_p = t_c n^2/p + (t_s + t_w n/\sqrt{p}) \log p \]
  
- 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 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
  \[ T_p = t_c n^2/p + (t_s + t_w n/p)(p - 1) \approx t_c n^2/p + t_s p + t_w n \]

Matrix-Matrix Product

- Consider matrix-matrix product
  \[ C = AB \]
  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
  \[ T_1 = t_c n^3 \]

Parallel Algorithm

**Partition**
- For \( i, j, k = 1, \ldots, n \), fine-grain task \((i, j, k)\) computes product \( a_{ij} b_{kj} \), yielding 3-D array of \( n \times n \) fine-grain tasks
- Assuming no replication of data, at most \( 3n^2 \) fine-grain tasks store entries of \( A, B \), or \( C \), say task \((i, j, k)\) 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 \( i \)th row of \( B \) are stored in \( i \)th layer of tasks

**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

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

\[
\begin{align*}
A_{11} B_{12} + A_{22} B_{22} & = 1-D \text{ column} \\
A_{11} B_{21} + A_{22} B_{21} & = 1-D \text{ row}
\end{align*}
\]

eliminating horizontal broadcasts and sum reductions

\[
\begin{align*}
2-D: \quad & \text{Combine } A_{21} B_{12} + A_{22} B_{22} \\
& \quad + A_{21} B_{11} + A_{22} B_{21} \\
& \quad + A_{11} B_{12} + A_{12} B_{22}
\end{align*}
\]

eliminating sum reductions

\[
\begin{align*}
3-D: \quad & \text{Combine } A_{21} B_{12} + A_{22} B_{22} \\
& \quad + A_{21} B_{11} + A_{22} B_{21} \\
& \quad + A_{11} B_{12} + A_{12} B_{22}
\end{align*}
\]

This algorithm is due to Fox et al.

Coarse-Grain 3-D Parallel Algorithm

Map

Corresponding mapping strategies are:
- **3-D**: Assign \( (n/q)^3/p \) 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 \( (n/q)^2/p \) 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

1-D row: \( A_{11} B_{12} + A_{22} B_{22} \)
1-D col: \( A_{11} B_{11} + A_{12} B_{21} \)
2-D: \( A_{21} B_{12} + A_{22} B_{22} + A_{21} B_{11} + A_{22} B_{21} + A_{11} B_{12} + A_{12} B_{22} \)
3-D: \( A_{21} B_{12} + A_{22} B_{22} + A_{21} B_{11} + A_{22} B_{21} + A_{11} B_{12} + A_{12} B_{22} \)

Fox Algorithm

- 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.

Cannon Algorithm

- 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
Scalability for 2-D Agglomeration

- For 2-D agglomeration, computing each of \( p \) blocks \( C_{ij}^{[[j]]} \) requires \( \sqrt{p} \) matrix-matrix products of \( (n/\sqrt{p}) \times (n/\sqrt{p}) \) blocks, so
  \[
  T_{\text{comp}} = t_c (n/\sqrt{p})^3 = t_c n^3/p
  \]
- For 2-D mesh, each communication or reduction takes time
  \[
  (t_s + t_w (n/\sqrt{p})^2)(\sqrt{p} - 1) \approx t_s p^{1/3} + t_w n^2/p^{1/3}
  \]
- Total time is therefore
  \[
  T_p = t_c n^3/p + 3t_s p^{1/3} + 3t_w n^2/p^{1/3}
  \]

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

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

On 3-D mesh, each broadcast or reduction takes time
\[
(t_s + t_w (n/\sqrt[3]{p})^3)(\sqrt[3]{p} - 1) \approx t_s p^{1/3} + t_w n^2/p^{1/3}
\]

Total time is therefore
\[
T_p = t_c n^3/p + 3t_s p^{1/3} + 3t_w n^2/p^{1/3}
\]

To determine isoefficiency function, set
\[
t_c n^3 \approx E (t_c n^3 + 3t_s p^{1/3} + 3t_w n^2 p^{1/3})
\]
which holds for large \( p \) if \( n = \Theta(p^{2/3}) \), so isoefficiency function is \( \Theta(p^{2/3}) \), 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 \( \Theta(p(\log p)^2) \)

Scalability for 3-D Agglomeration

References

References