triangular system solving

matrix multiplication

programming paradigm:

computational primitives using a distributed memory

We next consider two simple linear algebra
containing all 0's except a 1 in position i.

We have \( \mathbf{n} \) inner products of length \( n \). e.g. a vector

\[
\mathbf{B} = \sum_{i} \mathbf{q} \quad \text{and} \quad \mathbf{A}_L^s \mathbf{C} = \sum_{i} \mathbf{q}
\]

\[
\sum_{i} \mathbf{q}_L^s \mathbf{C} + \mathbf{q}_s \mathbf{c} \rightarrow \mathbf{c}
\]

**Inner product form:**

**Computations:** inner, middle, and outer product forms

There are three basic generic approaches to performing these

Matrix Multiplication \( \mathbf{C} \rightarrow \mathbf{C} \)

\( \mathbf{A} \mathbf{B} \)
products of size \( n_1 \times n_2 \). Independent primitives

We have a matrix-vector product:

\[
C^p \rightarrow C^p + Aq^p
\]
It is also possible to define other forms which organize the computation by diagonals. Stassen developed a version such that the total number of operations is less than \( 2n \) operations.

The result is a sum of the results of the primitives. Since there are dependences between the rank-1 updates we have \( n^2 \) rank-1 updates of size \( n \times n \). Note however:

\[
B_1^{\ell_2} = \mathcal{I}_q^2 \quad \text{and} \quad \mathcal{A} = \mathcal{I}_q^2 \quad \text{where} \quad \mathcal{I}_q^{2n} \bigoplus_{\mathcal{C}_n} \mathcal{C} \to \mathcal{C}
\]
In our matrix-vector product we used a simple one-dimensional partitioning. Here we consider an allowable communication pattern of a 2-D toroidal mesh (a 2-D mesh with wraparound connections on all columns and rows). This may or may not be the actual underlying network of the machine.

\[ C = AB \]
discussed by Box et al.

We consider the broadcast-multiplying roll algorithm.

is on processor \((i,j)\) in the mesh.

submatrices \(A_{i,j}, B_{i,j}\), and \(C_{i,j}\) where the \((i,j)\) submatrix

Assume a simple partitioning of all matrices into \(m \times m\)

\(A, B, \) and \(C\) are \(n \times u \) matrices with \(u \neq \tau m\).

clusters.

Assume an \(\tau \times \tau\) 2-D toroidal mesh of processors or

\(\ldots\).
\((\mathcal{B}_1 \mathcal{B}_2 \mathcal{B}_3) \mathcal{V} = \begin{pmatrix} B_3 \mathcal{B}_2 \mathcal{B}_1 \mathcal{V} \\ \mathcal{V}_3 \mathcal{V}_2 \mathcal{V}_1 \end{pmatrix} \mathcal{P}_3 \mathcal{P}_2 \mathcal{P}_1 = \mathcal{P}\) 

The block outer product has the form \(\mathcal{C} \in \mathcal{D}^3 + \mathcal{D}^2 + \mathcal{D}^1 = \mathcal{C}\) where \(\mathcal{D}\) is our example.

Standard block outer product approach. We assume \(p = 3\) for

This algorithm can be considered an implementation of the
which it is originally placed.

A row moves as a submatrix within the row of the mesh in

row of the mesh.

moves as a block row to each

rows 3 2 1

$B$ $B$ $B$

= $(B)$

does not move.

By the data motion of the three arrays:

The implementation the block outer product is characterized.
of the processors in the row.

of a block of a row is necessary to transmit the single

that is currently resident in its memories. The motion of

computes a portion of the that involves the block row

implementation. During the major step, each row of the mesh

The data motion of
Mesh row 3: $A_{33} \begin{bmatrix} B_{31} & B_{32} & B_{33} \end{bmatrix}$

Mesh row 2: $A_{22} \begin{bmatrix} B_{21} & B_{22} \end{bmatrix}$

Mesh row 1: $A_{11} \begin{bmatrix} B_{11} \end{bmatrix}$

Step 1.

The algorithm has three major steps for our example. Each row computes the following matrix products on each step:
Mesh row 3: $A_{31}$

Mesh row 2: $A_{23}$

Mesh row 1: $A_{12}$

Step 2.
Mesh row 3: \( A_{32} B_{32} B_{33} \) part of \( P_2 \)

Mesh row 2: \( A_{21} B_{11} B_{12} B_{13} \) part of \( P_1 \)

Mesh row 1: \( A_{13} B_{31} B_{32} B_{33} \) part of \( P_1 \)

Step 3.
**STEP 1**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

**Multiply**

Matrix multiply (not a global matrix multiply)

**Local blocks**

Broadcast

**Starting Positions**

Each row

A within blocks of local blocks

Multiply blocks of A within each row

Multiple

Broadcast

Starting Positions
Multiply matrix (not a global matrix)
local blocks

Multiply each row within
blocks of A

Broadcast

Wrap around
row within
one mesh

Roll block

**STEP 2**
Multiply
matrix
local blocks
(local
blocks
not a
global
multiply)

Multiply

Broadcast each row
within
blocks of

Wrap around row within
one mesh rows of

Roll block

One more roll restores B to its initial location.

C is complete and in place after this step.

STEP 3
and local memory can be used efficiently. Therefore any local memory hierarchy of registers, cache, Note that the local operations are matrix multiplications

Hierarchical memory must be performed. A block size analysis similar to that required for matrices.
The algorithm can be adapted to rectangular meshes and
algorithm directly.

the data motion computation specification of the
performance and algorithm design issues then deriving

Often results in a clearer understanding of the

Perform appropriate block size analyses.

motion/computation specification of the algorithm.

Identify the correspondence between the phase of the

given data structure, determine the data motion that

Given data structure, determine the data motion that

Choose the algebraic form of the algorithm to be used.

The general approach used here:

algorithm directly.

the data motion computation specification of the
performance and algorithm design issues then deriving

Often results in a clearer understanding of the

Perform appropriate block size analyses.

motion/computation specification of the algorithm.

Identify the correspondence between the phase of the

given data structure, determine the data motion that

Given data structure, determine the data motion that

Choose the algebraic form of the algorithm to be used.
code into a doubly nested loop

since $F$ is linear we can expand this

where $F()$ is a linear function and arguments with negative

\[
((u-v)^{i} \cdot \cdot \cdot , (u-v)^{i}F + (v)^{i} = (v)^{i}
\]

de $u$, $v = 1, 1$

executes

A typical construct assumes $\gamma (u,v)^{1}$ has been initialized and

fact related to reductions like the dot product and sum.

they are in

problem paradigms that occur in many forms.

They are in

also in computer codes.

Recurrence relations arise naturally in many applications and

Recurrence Relations
coefficients that define the linear function \( f \).

where \( \text{COEF}(t \in n \cup t' \in f) \) is an array which contains the

\[
\begin{align*}
\text{end do} \\
\text{end do}
\end{align*}
\]

\[
(\text{COEF}) (t \in f) + (\text{COEF}) (t' \in f) \end{align*}
\]

\[
\text{do } t' = f \text{ do } u', t = t' \text{ do }
\]

\[
\text{end do}
\]

\[
\text{end do}
\]
unit lower triangular system of equations.

For $i = 1, \ldots, n$, this can be written as $L^T x = f$, where $L^T$ is a

\[ L^T x = f \]

Then we have

\[ \sum_{i=1}^{n} \gamma_{i,j} = 0, \quad \text{for all other } j \]
\[
\begin{pmatrix}
\forall \phi \\
\exists \phi \\
\forall \phi \\
\exists \phi \\
\end{pmatrix} =
\begin{pmatrix}
\forall \gamma \\
\exists \gamma \\
\forall \gamma \\
\end{pmatrix}
\begin{pmatrix}
\forall \gamma & \exists \gamma & \forall \gamma & \forall \gamma \\
\exists \gamma & \exists \gamma & \exists \gamma & \exists \gamma \\
\forall \gamma & \forall \gamma & \forall \gamma & \forall \gamma \\
\end{pmatrix}
\]

Example: \( \forall = u \), \( \exists \in [\exists] = x \)

\( u \in \exists \in [\exists] = x \), \( f \in [\exists] \), \( f \in [\exists] \) \( 0 = f \). \( \forall \in \exists \), \( u \times u \in \exists \in [\exists] = T \) where \( f = xT \)

Consider solving the lower triangular system

**Triangular Systems**
This defines the following identities:
endo

\[ \nabla \chi / \phi = \nabla Z \]
endo

\[ \nabla \chi - \nabla \phi = \nabla \phi \]

1 - \nabla I = \nabla I - \nabla I

\[ \nabla Z = \nabla \phi \]

\[ \nabla \chi / \phi = \nabla Z \]

Row-oriented columns.

the fact that one is oriented towards rows, and the other
systems that result from the identities above. They differ in
There are two standard sequential methods for solving these
\[ u \nu \chi / u \phi = u \zeta \]

\[ \text{end do} \]

\[ \text{end do} \]

\[ \zeta \zeta \nu \chi - ? \phi = ? \phi \]

\[ u', I + J = ? \text{ do} \]

\[ \zeta \zeta \nu \phi / ? \phi = J \zeta \]

\[ I - u', I = J \text{ do} \]

: COLUMN-ORIENTED
consider more specialized solvers for recurrences later.

We will look at the row and column version now and

code containing the recurrence we started with.

Note that the row oriented solver is essentially the
Suppose the matrix $T$ is partitioned into $k(\ell + 1)$ columns using block row storage.

**Distributed Memory**
enddo.

enddo

\[ \int x \int - \int f = \int f \]

if \( i \) is one of my block row indices then

do \( i = 1 + 1 \) \( k \)

communicate(broadcast, ran-out) \( x \int \) to each processor

solve

if \( j \) is one of my block row indices then

do \( j = 1 + 1 \) \( k \)

Column-Version with Row-Storage:

Performing the column version:

Each processor then executes the following code when
on local index values must be such that each process can determine the root based on the broadcast, so that has to be sent or the partitioning each broadcast. Note however, for MPI the processors must know the root for the broadcast.

The basic performance determining factor is the performance of the processor. Keeping all of the fan-in dependences serialized on one particular row of the task graph to a single processor thereby Note that this has essentially assigned all of the work in a is done to simplify the code. This code uses simple cycle through all i and j index values
stored within processor $P(i)$ along with $x_i$ and $f_i$.

For this version we assume that the $i$-th block column of $T$ is

Row with block column storage:
\[(e - \int_f) = \int_x = \int_I \]
solve

if i is one of my block column indices then
\[(\int_d \int_m \int_t) = e \]
enddo

\[\int_x I + t = t \]
if i is one of my block column indices then
\[d o i = 1 \]
do i = 1 to k
\[0 = t \]
enddo

Row-Version with Column-Storeage:
Each processor performs the following code:
Note there is an initial fan-in of 0 from all processors to simplify code presentation. (This would not be done in MPI necessarily.)

This also uses simple cycling through all j index values to get local work rather than using a local list of work.
through the network toward processor $P_i(t)$.

accumulated as the $t$ vectors from each processor moves

and the fan-in routine itself allows partial sums to be

a local accumulation of all components of $t$ before sending

an update of the right-hand side vector in that it provides for

processor send a separate $t$ for each term it owns in the

vectors of length $m$) This is more efficient than letting each

in the processor containing block column $i$ (and $e$ are

products of accumulated in each processor, leaving the result $e$

fan-in) $f_i(t)$ collects and sums the partial block inner

Here, during stage $i$ of the algorithm, the pseudo-routing
wait for a particular block row computation for a processor to
occur with the row algorithm above. It is not necessary to
be performed all of the NY tasks from a given
Solve task depends have completed. This could be done
task from the next step once all of the NY tasks upon which
form above. But the processor could also perform an Solve
must be performed before doing anything else in the
to perform on a particular block iteration of the algorithm,
may have several NY tasks from a given block column
possible. For example 'in the column algorithm a processor
processor is strictly defined. This is not the only form
order in which the operations are performed on a given
Note that for both the column and row algorithms above the
used to define the parallel dependence graph. The resulting possibility of a block size larger than the value of \( \nu \) together to make up a single matrix vector product with the parallelism possible on a processor can be blocked that are ready to execute on a processor. A group of \( M \) tasks from a given row or column routine. A group of these tasks from a single row or column of the \( M \) tasks can be executed by a single processor. \( M \) tasks is called a compute-ahead and send-ahead scheme. It is also not necessary for a processor to treat each of the nodes of the given time have appeared in the literature and are sometimes algorithms that exploit this range of scheduling choice within any group of tasks that are available for execution at any appropriate processor. Variants of the column and row compute one of its \( M \) tasks and sends its contribution to the
Vectors were to be used to define multiple systems. Arrays which would be useful if multiple right-hand side create a ring implementation or a two-dimensional systolic dependencies of the original task graph. It can be used to synchronization points that do not violate any of the retiming version of the dependency graph addresses retiming in the systolic literature.

Retiming by adding extra dependencies. This is known as useful to eliminate these types of dependencies from the task broadcast efficiently, e.g., a ring network, it is sometimes for machines whose topology does not support fan-in and
• If $k = p$ there is limited parallelism and relatively large grain tasks.

• If $k$ is too large then there is too much communication and short messages are used.

• Note determining the blocksize and mapping the blocks to the processors is a key activity here.