Ph129 / CPS615
Lecture Notes
Communication in the Banded Algorithm

- To update the elements in the computational window we need to be able to communicate,
  \[ L_{k+i} = A_{k+i,k} \ (i = 0, 1, \ldots, \hat{m} - 1) \]  to the other processors in the same row of the template.
  \[ U_{k+j} = A_{k,k+j} \ (j = 0, 1, \ldots, \hat{m} - 1) \]  to the other processors in the same column of the template.
- This communication can be performed by a pipe broadcast using the \texttt{vread/vwrite} communication routines.

For example, for rows, if row\_pos is 0, 1 or 2 depending on whether a processor is in the first row, a middle row, or the last row of the current window:

```c
if ( row_pos == 0 )
    vwrite(Abuf, down, fsize, offset, mhat);
else if ( row_pos == 1 )
    vread(U, up, down, fsize, fsize, mmax);
else if ( row_pos == 2 )
    vread(U, up, 0, fsize, fsize, mmax);
```
Banded Matrix Decomposition

Scattered decomposition of a $20 \times 20$ matrix with band-width $b = 11$ for a 16 processor hypercube. A similar decomposition can be used for meshed-connected topologies.
Some References


Banded LU Decomposition

If the matrix, $A$, is banded with bandwidth, $b$, and half-width $m$ given by $b = 2m - 1$, then:

- The sequential algorithm is similar to the full matrix case, except at each stage only those elements within a computational "window" of $m$ rows and $m$ columns are updated.

- Partial pivoting can cause the number of columns in the computational window to be greater than $m$. This necessitates some extra bookkeeping in both the sequential and parallel algorithms.

- The parallel banded and full algorithms are similar, but use a different decomposition. To get better load balance a scattered decomposition over both rows and columns is used in the banded algorithm. In the full case a scattered decomposition over just rows was used.
(4) If the pivot row is in the same processor as row $k$ then columns $k$ to $M - 1$ of the pivot row are overwritten by the corresponding entries in row $k$. If the pivot row and row $k$ are not in the same processor columns $k$ to $M - 1$ of row $k$ are sent (by the shortest possible pipe) to the processor which had the pivot row, and are used to overwrite the corresponding pivot row entries.

(5) In the processor containing row $k$, columns $k$ to $M - 1$ of row $k$ are overwritten by the entries in the array pivot.
Parallel Pivoting

At step $k$ pivot selection is performed in parallel as follows:

1. Each processor checks its rows and chooses a pivot candidate.

2. Each candidate passes the absolute value of its pivot candidate, and the corresponding row number, to the CrOS III routine $combine$. This gives the pivot row number.

3. The entries in the pivot row from column $k$ to column $M - 1$ are piped (or broadcast) to all processors, and is stored in the array pivot.

(continued...)
int select_pivot ( pdata1, pdata2, size )
struct { float pval; int prow; } *pdata1, *pdata2;
int size;
{
  if ( pdata2->pval > pdata1->pval ){
    pdata1->pval = pdata2->pval;
    pdata1->prow = pdata2->prow;
  }
  return 0;
}

INTEGER FUNCTION SELPIV ( PDATA1, PDATA2, ISIZE )
REAL PDATA1(2), PDATA2(2)
INTEGER ISIZE

IF ( PDATA2(1) .GT. PDATA1(1) ) THEN
  PDATA1(1) = PDATA2(1)
  PDATA1(2) = PDATA2(2)
ENDIF

SELPIV = 0
RETURN
END
Communication in the Parallel LU Decomposition Algorithm

- We can perform the broadcast of the pivot row by means of the pipe algorithm, as used in the matrix multiplication algorithm.
- If pivoting is necessary at step $k$ we can send row $k$ to the appropriate processor using the shortest available pipe.
- The pivot row can be selected by using the CrOS III combine routine with the combining function shown on the next page.
- We decompose over rows, rather than columns, since this is more convenient if we subsequently want to do forward reduction and back substitution.
Scattered Row Decomposition

Work is approximately load balanced as computational window moves down diagonal.
Parallel Pseudocode

for_begin ( each step, $k = 0, 1, \ldots, M - 1$ )
    select pivot row, $r$
    broadcast columns $k$ to $M - 1$ of pivot row
to other processors
    replace columns $k$ to $M - 1$ of row $r$ with
    those of row $k$
    for_begin ( each row, $i = 1, 2, \ldots, M - 1 - k$ )
        $A_{k+i,k} = A_{k+i,k}/A_{k,k}$
    for_end
    for_begin ( each column, $j = 1, \ldots, M - 1 - k$ )
        for_begin ( each row, $i = 1, \ldots, M - 1 - k$ )
            $A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} \cdot A_{k,k+j}$
        for_end
    for_end
for_end
Block Row Decomposition

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Not load balanced. When computational window is as shown shaded above processor 0 is idle for the rest of the algorithm.
Decomposition

We must choose a decomposition which is load balanced throughout the algorithm, and which minimizes communication.

- *Contiguous blocks of rows or columns?* — Won’t work since not load balanced. Once processing of a block of rows or columns is completed the corresponding processor will have nothing to do.

- *Scattered (or wrap) row decomposition?* — Each processor gets a set of non-contiguous rows. We use the *gridmap* routines to map the processors onto a line. If processor $p$ is at position $B(p)$ on the line, then it handles rows,

\[ B(p), B(p) + N, B(p) + 2N, \ldots \]
Sequential Pseudocode

\textbf{for_begin} ( each step, $k = 0, 1, \ldots, M - 1$ )

select pivot row
exchange columns $k$ to $M - 1$ of row $k$ with
those of pivot row

\textbf{for_begin} ( each row, $i = 1, 2, \ldots, M - 1 - k$ )

\[ A_{k+i,k} = A_{k+i,k} / A_{k,k} \]

\textbf{for_end}

\textbf{for_begin} ( each column, $j = 1, \ldots, M - 1 - k$ )

\textbf{for_begin} ( each row, $i = 1, \ldots, M - 1 - k$ )

\[ A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} \times A_{k,k+j} \]

\textbf{for_end}

\textbf{for_end}

\textbf{for_end}
Pivot Selection at Step $k$

- Pivot is selected from shaded entries
- Exchange columns $k$ to $M-1$

Factorization After $k$ Steps

- Already factored into rows of $U$
- Only entries in shaded computational window are updated at step $k$
How does one solve
\[ A \mathbf{x} = \mathbf{b} \]

Gaussian elimination is the essential idea.

\[ a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_1 \]  
\[ a_{21} x_1 + a_{22} x_2 + a_{23} x_3 = b_2 \]  
\[ a_{31} x_1 + a_{32} x_2 + a_{33} x_3 = b_3 \]

Eliminate \( x_1 \) from equations \( 2 \) and \( 3 \)

\[ \begin{align*}
2 & \Rightarrow 2' = 2 - \frac{a_{21}}{a_{11}} - \frac{1}{1} \\
3 & \Rightarrow 3' = 3 - \frac{a_{31}}{a_{11}} - \frac{1}{1}
\end{align*} \]
\[ a_{33} x_3 = b_3 \quad (3') \]
\[ a_{22} x_2 + a_{23} x_3 = b_2 \quad (2') \]
\[ a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_3 \quad (1') \]

Solve \((3')\) for \(x_3\)

Use this value of \(x_3\) and

Solve \((2')\) for \(x_2\).

Use these values of \(x_2\) and \(x_3\)

to solve \((1')\) for \(x_1\).
\[ a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_1 \]  
\[ a'_{22} x_2 + a'_{23} x_3 = b'_{2} \]  
\[ a'_{32} x_2 + a'_{33} x_3 = b'_{3} \]

Elminate \( x_2 \) from equation (3)

\[ (3') \rightarrow (3'') = (3') - \frac{a'_{32}}{a'_{22}} (2') \]

This scheme is "forward reduction"
Why is matrix multiplication unusually good on any machine?

work/ addressing (memory access/ communicating) decreased

is large

Compare with finite difference
Sequential LU Algorithm

Algorithm proceeds in $M$ steps.

- At the start of step $k$ we identify the row, $r$, containing the largest value of $|A_{i,k}|$ for $k \leq i \leq M-1$. If $r \neq k$ then rows $r$ and $k$ are exchanged. This is called partial pivoting, and is done to improve the numerical stability. After the exchange the element that is now $A_{k,k}$ is called the pivot.

- At each step $k$ column number $k$ of $L$ and row number $k$ of $U$ are found:

\[
L_{k,k} = 1
\]
\[
L_{k+i,k} = \frac{A_{k+i,k}}{A_{k,k}} \quad \text{for} \quad i = 1, \ldots, M-1-k
\]
\[
U_{k,k+j} = A_{k,k+j} \quad \text{for} \quad j = 0, 1, \ldots, M-1-k
\]

Then the rows and columns $> k$ are modified as follows:

\[
A_{k+i,k+j} = A_{k+i,k+j} - L_{k+i,k}U_{k,k+j}
\]

for $i = 1, \ldots, M-1-k$ and $j = 1, \ldots, M-1-k$. 
- After step $k$ the first $k$ rows and columns of $A$ are not used again. We can therefore overwrite $A$ with the columns of $L$ and the rows of $U$ as we find them. The diagonal of $L$ does not have to be explicitly stored since it is all 1's.
Some References

The following papers deal with parallel algorithms for the LU decomposition of full matrices, and contain useful references to other work:


Full LU Decomposition

We wish to decompose the matrix \( A \) into the product \( LU \), where \( L \) is a lower triangular matrix with 1's on the main diagonal, and \( U \) is an upper triangular matrix.

- We assume \( A \) is a full \( M \) by \( M \) matrix.
- In general pivoting is necessary to ensure numerical stability.
- \( LU \) decomposition is often used in the solution of systems of linear equations, \( Ax = b \). The equations can be written as two triangular systems,

\[
Ly = b, \quad \text{and} \quad Ux = y
\]

The first equation is solved for \( y \) by forward reduction, and the solution \( x \) is then obtained from the second equation by back substitution.
This well known scheme can be formalized as LU decomposition.

\[ A = LU \]

\[ L \text{ is matrix of multipliers} \]

\[
\begin{bmatrix}
1 & \frac{a_{21}}{a_{11}} & \frac{a_{31}}{a_{11}} & \frac{a_{32}}{a_{22}} & 1 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

\[ U \text{ is resultant matrix} \]

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
0 & a_{22} & a_{23} \\
0 & 0 & a_{33}
\end{bmatrix}
\]

\[ L^{-1} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
-\frac{a_{21}}{a_{11}} & 1 & 0 & 0 \\
? & -\frac{a_{32}}{a_{22}} & 1 & 0 \\
? & ? & ? & ?
\end{bmatrix} \]

\[ L^{-1}A = U \]
\[ a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_1 \]  
\[ a_{22} x_2 + a_{23} x_3 = b_2' \]  
\[ a_{32} x_2 + a_{33} x_3 = b_3' \]

ELIMINATE \( x_2 \) from equation \( 3' \)

\[ 3' \Rightarrow 3'' = 3' - a_{32} x_3 \]

This scheme is "forward reduction!"
Performance Analysis

Time to pipe $A = (m^2 + (\sqrt{N} - 2))t_{\text{comm}}$

Time to roll $B = m^2t_{\text{comm}}$

Time to do $C = C + TB = 2m^3t_{\text{calc}}$

Total time, $T_N(m) = \sqrt{N}[2m^3t_{\text{calc}} + (2m^2 + \sqrt{N} - 2)t_{\text{comm}}]$

The efficiency is given by,

$$\epsilon = \frac{T_1(M)}{T_N(m)} = \frac{2(m\sqrt{N})^3t_{\text{calc}}}{N^{3/2}[2m^3t_{\text{calc}} + (2m^2 + \sqrt{N} - 2)t_{\text{comm}}]}$$

The overhead is therefore,

$$f = \frac{1}{\epsilon} - 1 = \left(\frac{1}{m} + \frac{\sqrt{N} - 2}{2m^3}\right)\tau$$

where $\tau = t_{\text{comm}}/t_{\text{calc}}$. If $g = m^2$ is the grain size, then

$$f \approx \frac{\tau}{\sqrt{g}}$$
Comparison of Pipe and Broadcast

Time for naive broadcast = \( \frac{m^2}{2}(\sqrt{N} - 1)t_{\text{comm}} \)

Time for log broadcast = \( \frac{m^2 d}{2}t_{\text{comm}} \)

Time for pipe broadcast = \( m^2 t_{\text{comm}} + (\sqrt{N} - 2)t_{\text{comm}} \)

where,

- \( t_{\text{comm}} \) = Time to exchange a floating-point number
- \( m \) = Order of square sub-block matrix
- \( d \) = Dimension of hypercube
- \( N \) = Number of processors = \( 2^d \)

Note:

For sufficiently large grain-size the pipe broadcast is better than the logarithmic broadcast,

\[
\text{If } m^2 > 2 \left( \frac{\sqrt{N} - 2}{d - 2} \right) \text{ pipe wins}
\]
Pseudocode for Pipe Broadcast

\texttt{proc\_begin pipe\_A ( pipe A sub-block )}
\texttt{\hspace{1cm} determine source processor for pipe}
\texttt{\hspace{1cm} determine last processor in the pipe}
\texttt{if\_begin ( this processor is source ) then}
\texttt{\hspace{1cm} copy A to T}
\texttt{\hspace{1cm} send T to processor on right}
\texttt{else\_if ( this processor is not end of pipe ) then}
\texttt{\hspace{1cm} receive T from processor on left}
\texttt{\hspace{1cm} send T to processor on right}
\texttt{else}
\texttt{\hspace{1cm} receive T from processor on left}
\texttt{if\_end}
\texttt{proc\_end}
Schematic representation of a split pipe.
Schematic representation of a simple linear pipe.
The Use of broadcast

```c
int broadcast ( buffer, origin, bmask, nbytes )
char *buffer; /* data to be broadcast */
int origin;   /* node number of source */
int bmask;    /* specifies subcube */
int nbytes;   /* number of bytes to send */
```

- In this case buffer points to the storage for $T$.
- If processor is in row $i$, then origin is the processor at position $(i, j)$, where $j = (i + n) \mod \sqrt{N}$.
- bmask is $\sqrt{N} - 1$.
- nbytes is just the size of a sub-block in bytes.
Pseudocode for bcast_A

proc_begin bcast_A ( broadcast A sub-block )
determine source processor for broadcast
if_begin ( this processor is source ) then
    copy A to T
    broadcast T to row
else_if ( this processor is not source ) then
    receive sub-block and store in T
if_end
proc_end
Pseudocode for Matrix Multiplication

\texttt{proc\_begin mat\_mult ( find } C = AB \texttt{ )}
\begin{itemize}
  \item initialize sub-block matrix \( C \) to zero
  \item loop \texttt{for\_begin} ( \texttt{n = 0} to \( \sqrt{N} - 1 \) )
    \begin{itemize}
      \item \texttt{proc\_call bcast\_A} ( send appropriate \( A \)
        sub-block along rows, store in \( T \) )
      \item \( C \leftarrow C + TB \)
      \item \texttt{proc\_call roll\_B} ( roll \( B \) upwards)
    \end{itemize}
  \item \texttt{for\_end}
\end{itemize}
\texttt{proc\_end}
A Look At What Happens

Consider the case where $N = 16$, and look at what happens in a particular processor. We choose the one at position $(2, 1)$.

$n = 0$: \[ T = \hat{A}^{22}, \quad B = \hat{B}^{21}, \]
\[ C = \hat{A}^{22} \hat{B}^{21} \]

$n = 1$: \[ T = \hat{A}^{23}, \quad B = \hat{B}^{31}, \]
\[ C = \hat{A}^{22} \hat{B}^{21} + \hat{A}^{23} \hat{B}^{31} \]

$n = 2$: \[ T = \hat{A}^{20}, \quad B = \hat{B}^{01}, \]
\[ C = \hat{A}^{22} \hat{B}^{21} + \hat{A}^{23} \hat{B}^{31} + \hat{A}^{20} \hat{B}^{01} \]

$n = 3$: \[ T = \hat{A}^{21}, \quad B = \hat{B}^{11}, \]
\[ C = \hat{A}^{22} \hat{B}^{21} + \hat{A}^{23} \hat{B}^{31} + \hat{A}^{20} \hat{B}^{01} + \hat{A}^{21} \hat{B}^{11} \]
\[ C = T \]

\[
\begin{array}{cccc}
\hat{A}^{00} & \hat{B}^{00} & \hat{A}^{01} & \hat{B}^{01} \\
\hat{A}^{00} & \hat{B}^{01} & \hat{A}^{01} & \hat{B}^{01} \\
\hat{A}^{11} & \hat{B}^{11} & \hat{A}^{12} & \hat{B}^{12} \\
\hat{A}^{11} & \hat{B}^{11} & \hat{A}^{12} & \hat{B}^{12} \\
\hat{A}^{22} & \hat{B}^{22} & \hat{A}^{23} & \hat{B}^{23} \\
\hat{A}^{22} & \hat{B}^{22} & \hat{A}^{23} & \hat{B}^{23} \\
\hat{A}^{33} & \hat{B}^{33} & \hat{A}^{34} & \hat{B}^{34} \\
\hat{A}^{33} & \hat{B}^{33} & \hat{A}^{34} & \hat{B}^{34} \\
\end{array}
\]

\[
\begin{array}{cccc}
\hat{A}^{01} & \hat{A}^{01} & \hat{A}^{01} & \hat{A}^{01} \\
\hat{A}^{12} & \hat{A}^{12} & \hat{A}^{12} & \hat{A}^{12} \\
\hat{A}^{23} & \hat{A}^{23} & \hat{A}^{23} & \hat{A}^{23} \\
\hat{A}^{30} & \hat{A}^{30} & \hat{A}^{30} & \hat{A}^{30} \\
\end{array}
\]

\[
\begin{array}{cccc}
\hat{B}^{10} & \hat{B}^{11} & \hat{B}^{12} & \hat{B}^{13} \\
\hat{B}^{20} & \hat{B}^{21} & \hat{B}^{22} & \hat{B}^{23} \\
\hat{B}^{30} & \hat{B}^{31} & \hat{B}^{32} & \hat{B}^{33} \\
\hat{B}^{00} & \hat{B}^{01} & \hat{B}^{02} & \hat{B}^{03} \\
\end{array}
\]
Note each $\hat{c}_k$ needs $\hat{A}$ values stored in same row of processors and $\hat{B}$ values stored in same column of processors.

In stages - at each stage, each processor chooses a suitable value of $n$ and uses this to update $\hat{c}$. 
\[
\begin{array}{cccc}
\hat{A}^{00} & \hat{A}^{01} & \hat{A}^{02} & \hat{A}^{03} \\
\hat{A}^{10} & \hat{A}^{11} & \hat{A}^{12} & \hat{A}^{13} \\
\hat{A}^{20} & \hat{A}^{21} & \hat{A}^{22} & \hat{A}^{23} \\
\hat{A}^{30} & \hat{A}^{31} & \hat{A}^{32} & \hat{A}^{33}
\end{array}
\]
The Algorithm

If $\hat{C}^{lk}$ is the sub-block at position $(l, k)$ then the problem can be stated in block matrix form:

$$\hat{C}^{lk} = \sum_{n=0}^{\sqrt{N}-1} \hat{A}^{ln} \hat{B}^{nk}$$

(1) Initialize $C = 0$, $n = 0$.

(2) In each row, $i$, of processors broadcast the sub-block $\hat{A}^{ij}$ to the other processors in the row, where $j = (i + n) \mod \sqrt{N}$. Each processor stores the broadcast sub-block in $T$.

(2) Multiply $T$ in each processor by the current $B$ sub-block, and add result to $C$.

(3) Each processor sends its current $B$ sub-block to the processor above. At the same time it receives a sub-block from the processor below and makes this the new current $B$ sub-block. Processors in the top row communicate with those in the bottom row.

(4) Set $n = n + 1$. If $n < \sqrt{N}$ then go to (2), else quit.
Matrix Multiplication

Suppose we want to multiply the matrices $A$ and $B$ together to form the matrix $C$:

$$C = AB$$

- We will assume all matrices are square – the algorithm can be generalized to deal with rectangular matrices.
- The input matrices, $A$ and $B$, are decomposed into rectangular sub-blocks. If we have $N$ processors we have $\sqrt{N}$ rows and columns of sub-blocks. This means $N$ must be a perfect square, i.e., that the hypercube dimension is even. The algorithm can easily be generalized for hypercubes of odd dimension.
- One sub-block is assigned to each processor by means of the gridmap decomposition routines.
- The algorithm ensures that the output matrix $C$ has the same decomposition as $A$ and $B$. 
KEY COMMENTS ON EQUATION SOLUTION
which we will return to

1) In solving \( A \cdot x = b \)
   formally \( x = A^{-1} \cdot b \)
   but this is \textbf{NOT} formally
   \textbf{best numerical method}

2) If \( A \) is \textbf{Sparse} both
   \( A^{-1} \) and the better "LU
   decomposition" are \textbf{NOT} \textbf{Sparse}.
Note

1. **Matrix multiplication** very rarely used in scientific computing for large $N$.
   Yet favorite Computer Science algorithm!

2. **Equation solvers** (full matrix)
   \[ A \times x = b \]
   Sometimes used but not very common. Of course incredibly important for sparse matrices.

   Why? If matrix large
   a) "Physics" will make sparse
   b) Insoluble unless sparse
In chemistry, one needs for full equations:

- Eigenvalues/vectors - to find "bound states" ("equilibrium states")

\[ A \mathbf{x}_m = \lambda_m \mathbf{x}_m \]

*E.g.* MOPAC

- Equation solutions - for reasons similar to those just discussed

\[ A \mathbf{x} = \mathbf{b} \]

- Multiplication to "change basis"

\[ |f\rangle = \sum_{n=1}^{N} a_n |f_n\rangle \]

\[ = \sum_{n=1}^{M} b_n |f'_n\rangle \]

\[ |f''\rangle = \sum_{m} f'_{nm} |f_m\rangle \]

\[ b_m = \sum_{m} f'_{nm} a_m \]

\[ |f'''\rangle = \sum_{k=1}^{K} c_k |f'_k\rangle \]

\[ f'_k = \sum_{n} f''_{kn} |f_n\rangle \]

\[ c_k = \frac{\sum f''_{kn} f'_{nm} a_m}{\text{multipliers}} \]
Read the literature (e.g., Computer Physics Communications, Nov 1990)
for choices of fn, um. Clearly we will take fn as functions for which L fn can be easily calculated.

Commals.

N expansion functions fn
work ∝ N³

If I have N grid points
best methods, work ∝ N
worst "", work ∝ N².

However, wave equations have "oscillatory" solutions. These could be very hard to represent numerically.
\[ \sum_{l=1}^{N} a_n (L f_n) = g \]

\[ u(x) = \left[ \sum_{l=1}^{N} a_n (L f_n) - g \right] \]

Need \( u(x) = 0 \).

Choose "suitable" set of weight functions \( w_m \)

\[ \int_{\text{volume}} w_m^*(x) u(x) d^3x = 0 \]

\[ L \frac{\partial}{\partial x} = g \]

vector \( \vec{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \)

is matrix with matrix elements

\[ \int w_m^*(x) (L f_n(x)) d^3x \]

\( g \) is vector with coefficients

\[ \int w_m^*(x) g(x) d^3x \]
This is a very important method although you can't find eigenvectors often.

\[ \phi(x) = e^{i \omega t - k \cdot x} \]

for any \( k \) is an eigenfunction.

\[ \lambda = -[\omega^2 - \kappa |k|^2] \]

So we look at problem where \( f_n \) is not an eigenfunction.
According to survey, dominant use of "large" (N > 10,000) matrix inversion on supercomputers is the Method of Moments for Computational Electromagnetic

Invented by Harrington at Syracuse University (~1967?)

\[ Lf = g \]

\[ f = \sum_{n=1}^{N} a_n f_n \quad \text{where} \quad f_n \quad \text{are} \quad \text{suitable expansion functions for} \quad \text{which} \quad Lf_n \quad \text{can be calculated} \]

\[ \sum_{n=1}^{N} a_n (Lf_n) = g \]

Easiest would be use eigenfunctions

\[ Lf_n = \lambda_n f_n \]

\[ g = \sum_n g_n f_n \]
Often

You often want to find eigenstates

\[ H | \psi_i \rangle = \lambda_i | \psi_i \rangle \]

But \( | \psi_i \rangle \) \( H \) is diagonal

However this is usually impossible

Often one knows that

\[ H = H_A + H_B \]

\[ \uparrow \quad \uparrow \text{ Perturbation} \]

"basic"

E.g., Compound is \( \text{H}_2\text{O} \)

\( H_A \) is "free" Hamiltonian for isolated \( \text{H} \) \( \text{H}_2\text{O} \)

\( H_B \) is interaction (forces between atoms)

Simple states if diagonalize \( H_A \)

But \[ \langle \phi_j | H | \psi_i \rangle \] will be nonzero for "most" \( i,j \)
Full matrices come from "operators" basis which link several (or all) states used in expressing general state

Examples:

Chemistry:

Operator is Hamiltonian $H$

$\langle \alpha | H | \beta \rangle$

States $\alpha \rangle$ can be labelled by many quantities

Positions of Electrons

# Electrons

Orbits of Electrons

Vibrational modes

Chemical compound

"Channel"
Note
1) remove hypercube (granular)
2) "pipe" gives an optimal comm alg/rl
3) include "best" (local) MM alg/rl
FULL MATRICES
4) improve COLVE/BUDS

We have studied partial differential equation
\[ \nabla^2 \varphi = -4\pi g(x) \]
and shown how they translate
into matrix equations
\[ A \times x = b \]
\[ \nabla^2 \]
corresponds to \( \varphi \)
corresponds to \( \nabla \)
corresponds to \( \Delta \)
corresponds to \( \nabla \)

These matrices were "Sparse"

The operator \( A \) (\( \nabla^2 \)) only linked a few states (x values in \( \nabla \varphi(x) \), components of \( x \))

"Full" matrices are those with "essentially all" elements nonzero - more precisely, it is not worth exploring the zero's \( a_{i,0} = a \to a_{i,0} = 0 \) view as an "ordinary number"
This well known scheme can be formalized as LU decomposition

\[ A = L \cdot U \]

\( L \) is matrix of multipliers

\[
\begin{bmatrix}
1 & 0 & 0 \\
+\frac{a_{21}}{a_{11}} & 1 & 0 \\
+\frac{a_{31}}{a_{11}} + \frac{a_{32}}{a_{22}} & 1 \\
\end{bmatrix}
\]

\( U \) is resultant matrix

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
0 & a_{22} & a_{23} \\
0 & 0 & a_{33} \\
\end{bmatrix}
\]

\[ L^{-1} = 
\begin{bmatrix}
1 & 0 & 0 \\
-a_{21}/a_{11} & 1 & 0 \\
? & -a_{32}/a_{22} & 1 \\
\end{bmatrix}
\]

\[ L^{-1} A = U \]
\[ a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_1 \]  
\[ a'_{22} x_2 + a'_{23} x_3 = b'_2 \]  
\[ a'_{32} x_2 + a'_{33} x_3 = b'_3 \]  
\[ \text{Eliminate } x_2 \text{ from equation } (3') \]  
\[ (3') \rightarrow (3'') = (3') - \frac{a'_{32}}{a'_{22}} (2') \]  

This scheme is "forward reduction."
\[ a_{33} x_3 = b_3 \] (3"

\[ a'_{22} x_2 + a'_{23} x_3 = b'_2 \] (2'

\[ a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_3 \] (1)

Solve (3") for \( x_3 \)

Use this value of \( x_3 \) and

Solve (2') for \( x_2 \)

Use these values of \( x_2 \) and \( x_3 \)

to solve (1) for \( x_1 \).
How does one solve \( A \mathbf{x} = \mathbf{b} \) ?

Gaussian elimination is essential idea

\[
\begin{align*}
  a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \quad (1) \\
  a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \quad (2) \\
  a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \quad (3)
\end{align*}
\]

Eliminate \( x_1 \) from equations (2) and (3)

\[
\begin{align*}
  (2) &\Rightarrow (2') = (2) - \frac{a_{21}}{a_{11}} (1) \\
  (3) &\Rightarrow (3') = (3) - \frac{a_{31}}{a_{11}} (1)
\end{align*}
\]
(a) A High Edge/Area Ratio In The Time Direction

Time

↑

Space

Boundary of a Complex System

(b) A Better Edge/Area Ratio With Modest Communication

Time

↑

Space

(c) A More Practical Decomposition With More Communication

Time

↑

Space

Fig. 10. Three decompositions in space and time of the one dimensional wave equation discussed in the text of Sec. IVB.
The BLAS concept

- vector-vector
  \[ \mathbf{V} = \sum_{i=1}^{m} a_i \mathbf{b}_i \]

2m reads 1 write 2m-1 floating point operations
floats / memory access \( \sim 1 \)

- matrix-vector
  \[ \mathbf{V}_i = \sum_{j=1}^{m} a_{ij} \mathbf{b}_j \]

\( m^2 + m \) reads \( m \) writes \( m(m-1) + m^2 \)
floating point
floats / memory access \( \sim 2 \)

- matrix-matrix
  \[ \mathbf{V}_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \]

2m^2 reads m^2 writes m^3 + m^2(m-1)
floating point
floats / memory access \( \sim \frac{2m}{3} \)
why is matrix multiplication unusually good on any machine

work/addressing (memory access/communication) overhead is large.

compare with finite difference
Note each Elk needs $\hat{A}$ values stored in same row of processors and $\hat{B}$ values stored in same column of processors.

$\sqrt{N}$ Stages – at each stage, each $\varphi$ neuron chooses a suitable value of $n$ and uses this to update $\hat{C}$. 

Communication in the Banded Algorithm

- To update the elements in the computational window we need to be able to communicate,

- \( L_{k+i} = A_{k+i,k} \ (i = 0, 1, \ldots, \hat{m} - 1) \) to the other processors in the same row of the template.

- \( U_{k+j} = A_{k,k+j} \ (j = 0, 1, \ldots, \hat{m} - 1) \) to the other processors in the same column of the template.

- This communication can be performed by a pipe broadcast using the \texttt{vread/vwrite} communication routines.

For example, for rows, if \texttt{row_pos} is 0, 1 or 2 depending on whether a processor is in the first row, a middle row, or the last row of the current window:

\[
\begin{align*}
\text{if ( row_pos == 0 )} & \quad \text{vwrite(Abuf,down,fsize,offset,mhat);} \\
\text{else if ( row_pos == 1 )} & \quad \text{vread(U,up,down,fsize,fsize,mmax);} \\
\text{else if ( row_pos == 2 )} & \quad \text{vread(U,up,0,fsize,fsize,mmax);} 
\end{align*}
\]
Banded Matrix Decomposition

Scattered decomposition of a $20 \times 20$ matrix with band-width $b = 11$ for a 16 processor hypercube. A similar decomposition can be used for meshed-connected topologies.

```
0 1 3 2 0 1 3 2 0 1 3 2 0 1 3 2
1 2 3 5 7 6 4 5 7 6 4 5 7 6 4 5
2 3 5 7 6 4 5 7 6 4 5 7 6 4 5 7
3 4 5 7 6 4 5 7 6 4 5 7 6 4 5 7
5 6 5 7 6 4 5 7 6 4 5 7 6 4 5 7
6 4 5 7 6 4 5 7 6 4 5 7 6 4 5 7
7 4 5 7 6 4 5 7 6 4 5 7 6 4 5 7
8 9 11 10 8 9 11 10 8 9 11 10 8 9 11 10
9 11 10 8 9 11 10 8 9 11 10 8 9 11 10
10 8 9 11 10 8 9 11 10 8 9 11 10 8 9 11 10
8 9 11 10 8 9 11 10 8 9 11 10 8 9 11 10
10 8 9 11 10 8 9 11 10 8 9 11 10 8 9 11 10
9 11 10 8 9 11 10 8 9 11 10 8 9 11 10
8 9 11 10 8 9 11 10 8 9 11 10 8 9 11 10
```

Some References


Banded LU Decomposition

If the matrix, $A$, is banded with bandwidth, $b$, and half-width $m$ given by $b = 2m - 1$, then:

- The sequential algorithm is similar to the full matrix case, except at each stage only those elements within a computational “window” of $m$ rows and $m$ columns are updated.

- Partial pivoting can cause the number of columns in the computational window to be greater than $m$. This necessitates some extra bookkeeping in both the sequential and parallel algorithms.

- The parallel banded and full algorithms are similar, but use a different decomposition. To get better load balance a scattered decomposition over both rows and columns is used in the banded algorithm. In the full case a scattered decomposition over just rows was used.
(4) If the pivot row is in the same processor as row $k$ then columns $k$ to $M - 1$ of the pivot row are overwritten by the corresponding entries in row $k$. If the pivot row and row $k$ are not in the same processor columns $k$ to $M - 1$ of row $k$ are sent (by the shortest possible pipe) to the processor which had the pivot row, and are used to overwrite the corresponding pivot row entries.

(5) In the processor containing row $k$, columns $k$ to $M - 1$ of row $k$ are overwritten by the entries in the array pivot.
Parallel Pivoting

At step \( k \) pivot selection is performed in parallel as follows:

1. Each processor checks its rows and chooses a pivot candidate.

2. Each candidate passes the absolute value of its pivot candidate, and the corresponding row number, to the CrOS III routine \textit{combine}. This gives the pivot row number.

3. The entries in the pivot row from column \( k \) to column \( M - 1 \) are piped (or broadcast) to all processors, and is stored in the array pivot.

(continued...)
int select_pivot ( pdata1, pdata2, size )
struct { float pval; int prow; } *pdata1, *pdata2;
int size;
{
    if ( pdata2->pval > pdata1->pval ){
        pdata1->pval = pdata2->pval;
        pdata1->prow = pdata2->prow;
    }
    return 0;
}

INTEGER FUNCTION SELPIV ( PDATA1, PDATA2, ISIZE )
REAL PDATA1(2), PDATA2(2)
INTEGER ISIZE

IF ( PDATA2(1) .GT. PDATA1(1) ) THEN
    PDATA1(1) = PDATA2(1)
    PDATA1(2) = PDATA2(2)
ENDIF

SELPIV = 0
RETURN
END
Communication in the Parallel LU Decomposition Algorithm

- We can perform the broadcast of the pivot row by means of the pipe algorithm, as used in the matrix multiplication algorithm.
- If pivoting is necessary at step $k$ we can send row $k$ to the appropriate processor using the shortest available pipe.
- The pivot row can be selected by using the CrOS III combine routine with the combining function shown on the next page.
- We decompose over rows, rather than columns, since this is more convenient if we subsequently want to do forward reduction and back substitution.
Parallel Pseudocode

\begin{verbatim}
for_begin ( each step, $k = 0, 1, \ldots, M - 1$ )
    select pivot row, $r$
    broadcast columns $k$ to $M - 1$ of pivot row
        to other processors
    replace columns $k$ to $M - 1$ of row $r$ with
        those of row $k$
    for_begin ( each row, $i = 1, 2, \ldots, M - 1 - k$ )
        $A_{k+i,k} = A_{k+i,k} / A_{k,k}$
    for_end
    for_begin ( each column, $j = 1, \ldots, M - 1 - k$ )
        for_begin ( each row, $i = 1, \ldots, M - 1 - k$ )
            $A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} * A_{k,k+j}$
        for_end
    for_end
for_end
\end{verbatim}
Scattered Row Decomposition

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>0</th>
<th></th>
<th>1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Work is approximately load balanced as computational window moves down diagonal.
Block Row Decomposition

Not load balanced. When computational window is as shown shaded above processor 0 is idle for the rest of the algorithm.
Decomposition

We must choose a decomposition which is load balanced throughout the algorithm, and which minimizes communication.

- **Contiguous blocks of rows or columns?** – Won't work since not load balanced. Once processing of a block of rows or columns is completed the corresponding processor will have nothing to do.

- **Scattered (or wrap) row decomposition?** – Each processor gets a set of non-contiguous rows. We use the *gridmap* routines to map the processors onto a line. If processor $p$ is at position $B(p)$ on the line, then it handles rows,

  $B(p), B(p) + N, B(p) + 2N, \ldots$
Sequential Pseudocode

\textbf{for_begin} ( each step, $k = 0, 1, \ldots, M - 1$ )

select pivot row

exchange columns $k$ to $M - 1$ of row $k$ with
those of pivot row

\textbf{for_begin} ( each row, $i = 1, 2, \ldots, M - 1 - k$ )

\[ A_{k+i,k} = A_{k+i,k}/A_{k,k} \]

\textbf{for_end}

\textbf{for_begin} ( each column, $j = 1, \ldots, M - 1 - k$ )

\textbf{for_begin} ( each row, $i = 1, \ldots, M - 1 - k$ )

\[ A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} * A_{k,k+j} \]

\textbf{for_end}

\textbf{for_end}

\textbf{for_end}
Pivot Selection at Step k

- Pivot is selected from shaded entries
- Exchange columns k to M-1

Factorization After k Steps

- Already factored into rows of U
- Already factored into columns of L
- Only entries in shaded computational window are updated at step k
After step $k$ the first $k$ rows and columns of $A$ are not used again. We can therefore overwrite $A$ with the columns of $L$ and the rows of $U$ as we find them. The diagonal of $L$ does not have to be explicitly stored since it is all 1's.
Sequential LU Algorithm

Algorithm proceeds in $M$ steps.

- At the start of step $k$ we identify the row, $r$, containing the largest value of $|A_{i,k}|$ for $k \leq i \leq M - 1$. If $r \neq k$ then rows $r$ and $k$ are exchanged. This is called partial pivoting, and is done to improve the numerical stability. After the exchange the element that is now $A_{k,k}$ is called the pivot.

- At each step $k$ column number $k$ of $L$ and row number $k$ of $U$ are found:

$$L_{k,k} = 1$$
$$L_{k+i,k} = A_{k+i,k}/A_{k,k} \quad \text{for} \quad i = 1, \ldots, M - 1 - k$$
$$U_{k,k+j} = A_{k,k+j} \quad \text{for} \quad j = 0, 1, \ldots, M - 1 - k$$

Then the rows and columns $> k$ are modified as follows:

$$A_{k+i,k+j} = A_{k+i,k+j} - L_{k+i,k}U_{k,k+j}$$

for $i = 1, \ldots, M - 1 - k$ and $j = 1, \ldots, M - 1 - k$. 
Some References

The following papers deal with parallel algorithms for the LU decomposition of full matrices, and contain useful references to other work:


Full LU Decomposition

We wish to decompose the matrix $A$ into the product $LU$, where $L$ is a lower triangular matrix with 1's on the main diagonal, and $U$ is an upper triangular matrix.

- We assume $A$ is a full $M$ by $M$ matrix.
- In general pivoting is necessary to ensure numerical stability.

- $LU$ decomposition is often used in the solution of systems of linear equations, $Ax = b$. The equations can be written as two triangular systems,

  $$Ly = b, \quad \text{and} \quad Ux = y$$

The first equation is solved for $y$ by forward reduction, and the solution $x$ is then obtained from the second equation by back substitution.
Performance Analysis

Time to pipe $A = (m^2 + (\sqrt{N} - 2))t_{comm}$

Time to roll $B = m^2t_{comm}$

Time to do $C = C + TB = 2m^3t_{calc}$

Total time, $T_N(m) = \sqrt{N}[2m^3t_{calc} + (2m^2 + \sqrt{N} - 2)t_{comm}]$

The efficiency is given by,

$$\epsilon = \frac{T_1(M)}{T_N(m)} = \frac{2(m\sqrt{N})^3t_{calc}}{N^{3/2}[2m^3t_{calc} + (2m^2 + \sqrt{N} - 2)t_{comm}]}$$

The overhead is therefore,

$$f = \frac{1}{\epsilon} - 1 = \left(\frac{1}{m} + \frac{\sqrt{N} - 2}{2m^3}\right)\tau$$

where $\tau = t_{comm}/t_{calc}$. If $g = m^2$ is the grain size, then

$$f \approx \frac{\tau}{\sqrt{g}}$$
Comparison of Pipe and Broadcast

Time for naive broadcast = \( \frac{m^2}{2} (\sqrt{N} - 1) t_{comm} \)

Time for log broadcast = \( \frac{m^2 d}{2} t_{comm} \)

Time for pipe broadcast = \( m^2 t_{comm} + (\sqrt{N} - 2) t_{comm} \)

where,

\( t_{comm} \) = Time to exchange a floating-point number

\( m \) = Order of square sub-block matrix

\( d \) = Dimension of hypercube

\( N \) = Number of processors = \( 2^d \)

Note:

For sufficiently large grain-size the pipe broadcast is better than the logarithmic broadcast,

If \( m^2 > 2 \left( \frac{\sqrt{N} - 2}{d - 2} \right) \) pipe wins
Pseudocode for Pipe Broadcast

proc_begin pipe_A ( pipe A sub-block )
    determine source processor for pipe
    determine last processor in the pipe
    if_begin ( this processor is source ) then
        copy A to T
        send T to processor on right
    else_if ( this processor is not end of pipe ) then
        receive T from processor on left
        send T to processor on right
    else
        receive T from processor on left
    if_end
proc_end
Schematic representation of a split pipe.
Schematic representation of a simple linear pipe.
The Use of broadcast

```c
int broadcast ( buffer, origin, bmask, nbytes )
char *buffer; /* data to be broadcast */
int origin;  /* node number of source */
int bmask;   /* specifies subcube */
int nbytes;  /* number of bytes to send */
```

- In this case buffer points to the storage for $T$.
- If processor is in row $i$, then origin is the processor at position $(i, j)$, where $j = (i + n) \mod \sqrt{N}$.
- bmask is $\sqrt{N} - 1$.
- nbytes is just the size of a sub-block in bytes.
Pseudocode for bcast_A

proc_begin bcast_A ( broadcast A sub-block )
determine source processor for broadcast
if_begin ( this processor is source ) then
copy A to T
broadcast T to row
else_if ( this processor is not source ) then
receive sub-block and store in T
if_end
proc_end
Pseudocode for Matrix Multiplication

proc_begin mat_mul ( find C = AB )
initialize sub-block matrix C to zero
for_begin ( n = 0 to \( \sqrt{N} - 1 \) )
    proc_call bcast_A ( send appropriate A sub-block along rows, store in T )
    \( C \leftarrow C + TB \)
    proc_call roll_B ( roll B upwards )
for_end
proc_end
A Look At What Happens

Consider the case where $N = 16$, and look at what happens in a particular processor. We choose the one at position $(2, 1)$.

$n = 0: \quad T = \hat{A}^{22}, \quad B = \hat{B}^{21},$
\[ C = \hat{A}^{22} \hat{B}^{21} \]

$n = 1: \quad T = \hat{A}^{23}, \quad B = \hat{B}^{31},$
\[ C = \hat{A}^{22} \hat{B}^{21} + \hat{A}^{23} \hat{B}^{31} \]

$n = 2: \quad T = \hat{A}^{20}, \quad B = \hat{B}^{01},$
\[ C = \hat{A}^{22} \hat{B}^{21} + \hat{A}^{23} \hat{B}^{31} + \hat{A}^{20} \hat{B}^{01} \]

$n = 3: \quad T = \hat{A}^{21}, \quad B = \hat{B}^{11},$
\[ C = \hat{A}^{22} \hat{B}^{21} + \hat{A}^{23} \hat{B}^{31} + \hat{A}^{20} \hat{B}^{01} + \hat{A}^{21} \hat{B}^{11} \]
\[
\begin{align*}
C & = \begin{array}{cccc}
\hat{A}^{00}B^{00} & \hat{A}^{00}B^{01} & \hat{A}^{00}B^{02} & \hat{A}^{00}B^{03} \\
\hat{A}^{01}B^{10} & \hat{A}^{01}B^{11} & \hat{A}^{01}B^{12} & \hat{A}^{01}B^{13} \\
\hat{A}^{11}B^{10} & \hat{A}^{11}B^{11} & \hat{A}^{11}B^{12} & \hat{A}^{11}B^{13} \\
\hat{A}^{12}B^{20} & \hat{A}^{12}B^{21} & \hat{A}^{12}B^{22} & \hat{A}^{12}B^{23} \\
\hat{A}^{22}B^{20} & \hat{A}^{22}B^{21} & \hat{A}^{22}B^{22} & \hat{A}^{22}B^{23} \\
\hat{A}^{23}B^{30} & \hat{A}^{23}B^{31} & \hat{A}^{23}B^{32} & \hat{A}^{23}B^{33} \\
\hat{A}^{33}B^{30} & \hat{A}^{33}B^{31} & \hat{A}^{33}B^{32} & \hat{A}^{33}B^{33} \\
\hat{A}^{30}B^{00} & \hat{A}^{30}B^{01} & \hat{A}^{30}B^{02} & \hat{A}^{30}B^{03}
\end{array} \\
T & = \begin{array}{cccc}
\hat{A}^{01} & \hat{A}^{01} & \hat{A}^{01} & \hat{A}^{01} \\
\hat{A}^{12} & \hat{A}^{12} & \hat{A}^{12} & \hat{A}^{12} \\
\hat{A}^{23} & \hat{A}^{23} & \hat{A}^{23} & \hat{A}^{23} \\
\hat{A}^{30} & \hat{A}^{30} & \hat{A}^{30} & \hat{A}^{30}
\end{array} \\
B & = \begin{array}{cccc}
\hat{B}^{10} & \hat{B}^{11} & \hat{B}^{12} & \hat{B}^{13} \\
\hat{B}^{20} & \hat{B}^{21} & \hat{B}^{22} & \hat{B}^{23} \\
\hat{B}^{30} & \hat{B}^{31} & \hat{B}^{32} & \hat{B}^{33} \\
\hat{B}^{00} & \hat{B}^{01} & \hat{B}^{02} & \hat{B}^{03}
\end{array}
\end{align*}
\]
<table>
<thead>
<tr>
<th>$\hat{A}^{00}$</th>
<th>$\hat{A}^{01}$</th>
<th>$\hat{A}^{02}$</th>
<th>$\hat{A}^{03}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{A}^{10}$</td>
<td>$\hat{A}^{11}$</td>
<td>$\hat{A}^{12}$</td>
<td>$\hat{A}^{13}$</td>
</tr>
<tr>
<td>$\hat{A}^{20}$</td>
<td>$\hat{A}^{21}$</td>
<td>$\hat{A}^{22}$</td>
<td>$\hat{A}^{23}$</td>
</tr>
<tr>
<td>$\hat{A}^{30}$</td>
<td>$\hat{A}^{31}$</td>
<td>$\hat{A}^{32}$</td>
<td>$\hat{A}^{33}$</td>
</tr>
</tbody>
</table>
The Algorithm

If \( \hat{C}^{lk} \) is the sub-block at position \((l, k)\) then the problem can be stated in block matrix form:

\[
\hat{C}^{lk} = \sum_{n=0}^{\sqrt{N}-1} \hat{A}^{ln} \hat{B}^{nk}
\]

1. Initialize \( C = 0, \ n = 0. \)

2. In each row, \( i, \) of processors broadcast the sub-block \( \hat{A}^{ij} \) to the other processors in the row, where \( j = (i + n) \mod \sqrt{N} \). Each processor stores the broadcast sub-block in \( T. \)

3. Multiply \( T \) in each processor by the current \( B \) sub-block, and add result to \( C. \)

4. Each processor sends its current \( B \) sub-block to the processor above. At the same time it receives a sub-block from the processor below and makes this the new current \( B \) sub-block. Processors in the top row communicate with those in the bottom row.

5. Set \( n = n + 1. \) If \( n < \sqrt{N} \) then go to (2), else quit.
Some References

This is by no means a complete list:


Matrix Multiplication

Suppose we want to multiply the matrices $A$ and $B$ together to form the matrix $C$:

$$C = AB$$

- We will assume all matrices are square – the algorithm can be generalized to deal with rectangular matrices.

- The input matrices, $A$ and $B$, are decomposed into rectangular sub-blocks. If we have $N$ processors we have $\sqrt{N}$ rows and columns of sub-blocks. This means $N$ must be a perfect square, i.e., that the hypercube dimension is even. The algorithm can easily be generalized for hypercubes of odd dimension.

- One sub-block is assigned to each processor by means of the *gridmap* decomposition routines.

- The algorithm ensures that the output matrix $C$ has the same decomposition as $A$ and $B$. 
KEY COMMENTS ON EQUATION SOLUTION
which we will return to

1) In solving \( A x = b \)
   formally \( x = A^{-1} b \)
   but this is NOT normally
   best numerical method

2) If \( A \) Sparse both
   \( A^{-1} \) and the better "LU
decomposition" are NOT Sparse.
Note:

1. Matrix multiplication very rarely used in scientific computing for large N. Yet favorite computer science algorithm!

2. Equation Solver (Full Matrix)

\[ A \times x = b \]

Sometimes used but not very common, of course incredibly important for Sparse matrices.

Why? If matrix large:

a) "Physics" will make Sparse
b) Insoluble unless Sparse
In chemistry, one needs for full matrices:

- Eigenvalues/vectors - to find "bound states" ("equilibrium states")

\[ A \hat{x}_m = \lambda_m \hat{x}_m \]

E.g., MO PAC

- Equation solution - for reasons similar to those just discussed

\[ A \hat{x} = \hat{b} \]

- Multiplication to "change basis"

\[ |f\rangle = \sum_{n=1}^{N} a_n |f_n\rangle \]
\[ = \sum_{n=1}^{M} b_n |f'_n\rangle \]
\[ |f'_n\rangle = -\sum_{m} f'_{nm} |f_m\rangle \]

\[ b_n = \sum_{m} f'_{nm} a_m \]

\[ |f''\rangle = \sum_{k=1}^{m} c_k |f_k\rangle \]
\[ f_k'' = \sum_{k} f''_{kn} |f'_n\rangle \]

\[ c_k = \sum_{n} (\sum_{m} f''_{in} f_{nm}) a_m \]
Read the literature (e.g. Computer Physics Communications, Nov 1991) for choices of fn, wn. Clearly one will take fn as functions for which L fn can be easily calculated.

**Comnels**

- N expansion functions fn
- work $\propto N^3$

- if I have N grid points
  - best methods, work $\propto N$
  - worst "", work $\propto N^2$

However, wave equations have "oscillatory" solutions. These could be very hard to represent numerically...
\[ \sum_{i=1}^{N} a_i (4f_i) = g \]

\[ u(x) = \left[ \sum_{i=1}^{N} a_i (4f_i) - g \right] \]

Need \( u(x) = 0 \).

Choose "suitable" set of weight functions \( w_m \)

\[ \int_{\text{volume}} w_m^*(x) u(x) d^3x = 0 \]

\[ \frac{L}{a} = g \]

\[ \text{vector } \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \]

\( L \) is matrix with matrix elements

\[ \int w_m^*(x) (L f_n(x)) d^3x \]

\( g \) is vector with coefficients

\[ \int w_m^*(x) g(x) d^3x \]
This is a very important method although you can't find eigenvectors often.

E.g. for "scattering problems" (which are usual in electromagnetics), eigenvalues are continuous.

\[ \frac{d^2 \phi}{dt^2} - \lambda \nabla^2 \phi = 0 \]

\[ \phi = e^{i(\omega t - k \cdot x)} \]

For any \( k \) is an eigenfunction

\[ \lambda = -\left(\omega^2 - k^2 \right) \]

So we look at problems where \( f_0 \) is not an eigenfunction.
According to Survey, dominant use of "large" (N > 10,000) matrix inversion on supercomputers is the Method of Moments for computational electromagnetics.

Invented by Hamington at Syracuse University (~ 1967?)

\[ Lf = g \]

\[ f = \sum_{n=1}^{N} a_n f_n \] where \( f_n \) are suitable expansion functions for which \( Lf_n \) can be calculated.

\[ \sum_{n=1}^{N} a_n (Lf_n) = g \]

Easiest would be use eigenfunctions

\[ Lf_n = \lambda_n f_n \]

\[ g = \sum_n g_n f_n \]

\[ a_n = \frac{g_n}{\lambda_n} \]
Often you often want to find eigenstates
\[ H |\psi_i\rangle = \lambda_i |\psi_i\rangle \]
with \( |\psi_i\rangle \) \( H \) is diagonal
However this is usually impossible

Often one knows that
\[ H = H_A + H_B \]
\[ \uparrow \quad \uparrow \text{Perturbation} \]
"basic"

E.g., compound is \( H_2O \)
\( H_A \) is "free" Hamiltonian
for isolated \( H \) \( H_2O \)
\( H_B \) is interaction (forces between atoms)

Simple states \( |\psi_i\rangle \) diagonalize \( H_A \)
but \( \langle \psi_j | H | \psi_i \rangle \) will be nonzero for "most" \( i \neq j \).
Block Row Decomposition

Not load balanced. When computational window is as shown shaded above processor 0 is idle for the rest of the algorithm.
Scattered Row Decomposition

Work is approximately load balanced as computational window moves down diagonal.

\[
\frac{M \times M}{m/2N_{procs}} \quad \text{mws per process}
\]

\[
\text{Load balance } \propto \frac{2N_{procs}}{m}
\]
Parallel Pseudocode

for_begin ( each step, $k = 0, 1, \ldots, M - 1$ )
    select pivot row, $r$
    broadcast columns $k$ to $M - 1$ of pivot row
to other processors
replace columns $k$ to $M - 1$ of row $r$ with
    those of row $k$
for_begin ( each row, $i = 1, 2, \ldots, M - 1 - k$ )
    $A_{k+i,k} = A_{k+i,k}/A_{k,k}$
for_end
for_begin ( each column, $j = 1, \ldots, M - 1 - k$ )
    for_begin ( each row, $i = 1, \ldots, M - 1 - k$ )
        $A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} \times A_{k,k+j}$
    for_end
for_end
for_end
Communication in the Parallel
LU Decomposition Algorithm

- We can perform the broadcast of the pivot row by means of the *pipe* algorithm, as used in the matrix multiplication algorithm.

- If pivoting is necessary at step $k$ we can send row $k$ to the appropriate processor using the shortest available pipe.

- The pivot row can be selected by using the CrOS III *combine* routine with the combining function shown on the next page.

- We decompose over rows, rather than columns, since this is more convenient if we subsequently want to do forward reduction and back substitution.
int select_pivot ( pdata1, pdata2, size )
struct { float pval; int prow; } *pdata1, *pdata2;
int size;
{
    if ( pdata2->pval > pdata1->pval ){
        pdata1->pval = pdata2->pval;
        pdata1->prow = pdata2->prow;
    }
    return 0;
}

INTEGER FUNCTION SELPIV ( PDATA1, PDATA2, ISIZE )
REAL PDATA1(2), PDATA2(2)
INTEGER ISIZE

IF ( PDATA2(1) .GT. PDATA1(1) ) THEN
    PDATA1(1) = PDATA2(1)
    PDATA1(2) = PDATA2(2)
ENDIF

SELPIV = 0
RETURN
END
work per processor \[ 2 \frac{m^2}{N_{proc}} \]

\[ A_{ij} = A_{ij} - L_i U_j \]

\( M \) elements
Parallel Pivoting

At step $k$ pivot selection is performed in parallel as follows:

1. Each processor checks its rows and chooses a pivot candidate.

2. Each candidate passes the absolute value of its pivot candidate, and the corresponding row number, to the CrOS III routine `combine`. This gives the pivot row number.

3. The entries in the pivot row from column $k$ to column $M - 1$ are piped (or broadcast) to all processors, and is stored in the array `pivot`.

(continued...)
(4) If the pivot row is in the same processor as row $k$ then columns $k$ to $M - 1$ of the pivot row are overwritten by the corresponding entries in row $k$. If the pivot row and row $k$ are not in the same processor columns $k$ to $M - 1$ of row $k$ are sent (by the shortest possible pipe) to the processor which had the pivot row, and are used to overwrite the corresponding pivot row entries. 

(5) In the processor containing row $k$, columns $k$ to $M - 1$ of row $k$ are overwritten by the entries in the array pivot.
"Block decomposition"
Banded LU Decomposition

If the matrix, $A$, is banded with bandwidth, $b$, and half-width $m$ given by $b = 2m - 1$, then:

- The sequential algorithm is similar to the full matrix case, except at each stage only those elements within a computational “window” of $m$ rows and $m$ columns are updated.

- Partial pivoting can cause the number of columns in the computational window to be greater than $m$. This necessitates some extra bookkeeping in both the sequential and parallel algorithms.

- The parallel banded and full algorithms are similar, but use a different decomposition. To get better load balance a scattered decomposition over both rows and columns is used in the banded algorithm. In the full case a scattered decomposition over just rows was used.
Some References


Banded Matrix Decomposition

Scattered decomposition of a $20 \times 20$ matrix with band-width $b = 11$ for a 16 processor hypercube. A similar decomposition can be used for meshed-connected topologies.
Communication in the Banded Algorithm

- To update the elements in the computational window we need to be able to communicate,
- \( L_{k+i} = A_{k+i,k} \) \( (i = 0, 1, \ldots, \hat{m} - 1) \) to the other processors in the same row of the template.
- \( U_{k+j} = A_{k,k+j} \) \( (j = 0, 1, \ldots, \hat{m} - 1) \) to the other processors in the same column of the template.
- This communication can be performed by a pipe broadcast using the \textit{vread/vwrite} communication routines.

For example, for rows, if \texttt{row_pos} is 0, 1 or 2 depending on whether a processor is in the first row, a middle row, or the last row of the current window:

```c
if ( row_pos == 0 )
    vwrite(Abuf,down,fsize,offset,mhat);
else if ( row_pos == 1 )
    vread(U,up,down,fsize,fsize,mmax);
else if ( row_pos == 2 )
    vread(U,up,0,fsize,fsize,mmax);
```
WOULD BE BETTER TO do in two dimensions so max mult.

(a) A High Edge/Area Ratio In The Time Direction

Time

Space

Boundary of a Complex System

(b) A Better Edge/Area Ratio With Modest Communication

Time

Space

(c) A More Practical Decomposition With More Communication

Time

Space

Fig. 10. Three decompositions in space and time of the one dimensional wave equation discussed in the text of Sec. IVB.