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 *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:

```
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 × 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	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10

Some References

G. C. Fox, 1984, "LU Decomposition for Banded Matrices," Caltech Report C^3P-99.

S. Lennart Johnsson, 1985, "Solving Narrow Banded Systems on Ensemble Architectures," *ACM Trans. on Math. Software*, **11**:271.

Y. Saad and M. H. Schultz, 1985, "Parallel Direct Methods for Solving Banded Linear Systems," Yale Research Report YALEU/DCS/RR-387.

J. J. Dongarra and S. Lennart Johnsson, 1987, "Solving Banded Systems on a Parallel Processor," *Parallel Computing*, **5**:219.

D. W. Walker, T. Aldcroft, A. Cisneros, G. C. Fox, and W. Furmanski, 1988, "LU Decomposition of Banded Matrices and the Solution of Linear Systems on Hypercubes," in *Proceedings of the Third Conference on Hypercube Concurrent Processors and Applications*, published by ACM Press, New York.

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

0
1
3
2
0
1
3
2
0
1
3
2
0
1
3
2

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, $i = 1, \ldots, M-1-k$)

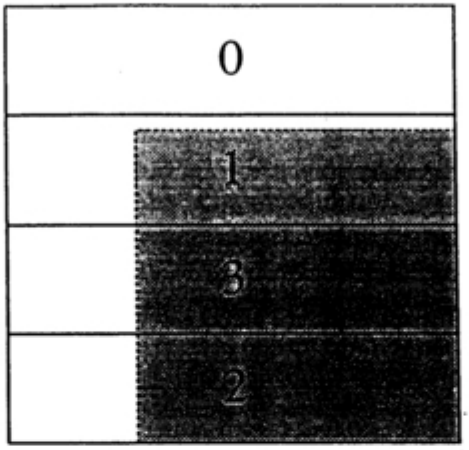
$A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} * A_{k,k+j}$

for_end

for_end

for_end

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

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

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, $i = 1, \ldots, M-1-k$)

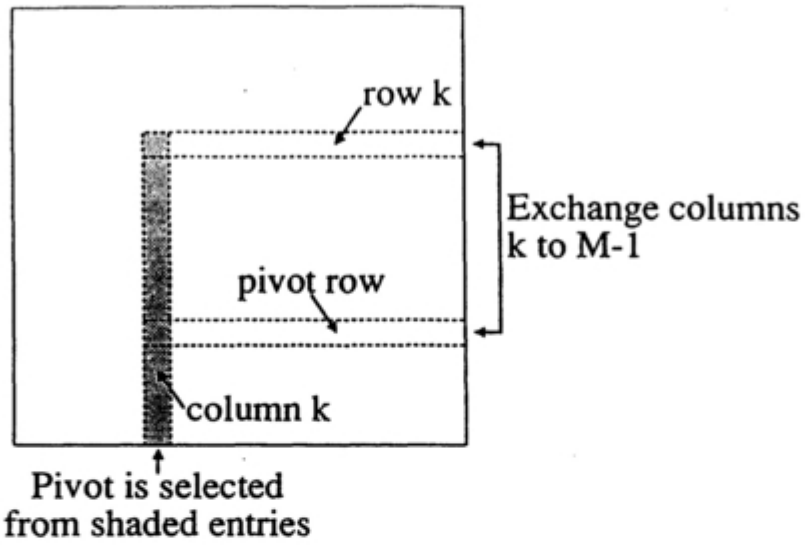
$A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} * A_{k,k+j}$

for_end

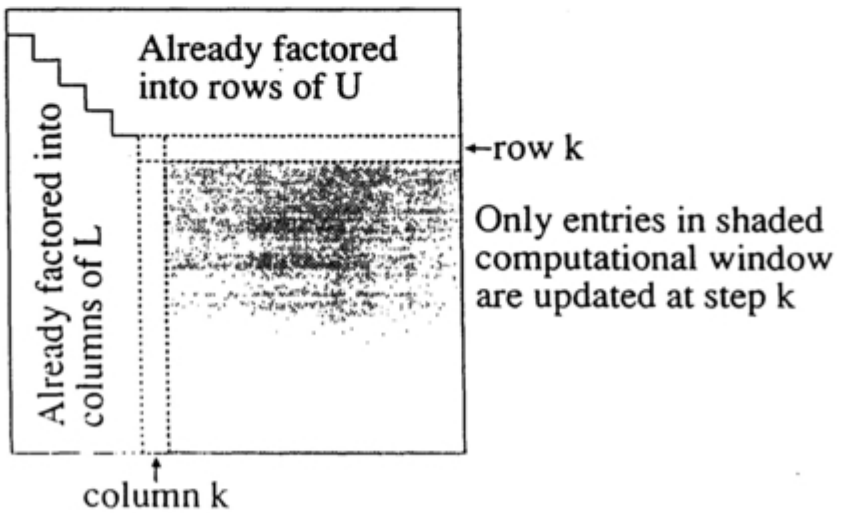
for_end

for_end

Pivot Selection at Step k



Factorization After k Steps



How does one solve

$$A\underline{x} = \underline{b} \quad ?$$

Gaussian Elimination is essential idea

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \quad (1)$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \quad (2)$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \quad (3)$$

ELIMINATE x_1 from equations (2) and (3)

$$(2) \Rightarrow (2') = (2) - \frac{a_{21}}{a_{11}}(1)$$

$$(3) \Rightarrow (3') = (3) - \frac{a_{31}}{a_{11}}(1)$$

$$a''_{33}\, x_3 = b''_3 \qquad (3'')$$

$$a'_{22}\, x_2 + a'_{23}\, x_3 = b'_2 \qquad (2')$$

$$a_{11}\, x_1 + a_{12}\, x_2 + a_{13}\, x_3 = b_3 \qquad (1)$$

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.

$$a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_1 \quad (1)$$

$$a'_{22} x_2 + a'_{23} x_3 = b'_2 \quad (2')$$

$$a'_{32} x_2 + a'_{33} x_3 = b'_3 \quad (3')$$

'ELIMINATE 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 / communication) overhead 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:

$$L_{k,k} = 1$$

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

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

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:

G. A. Geist and M. T. Heath, 1986, "Matrix Factorization on a Hypercube Multiprocessor," in *Hypercube Multiprocessors 1986*, published by SIAM Press, Philadelphia.

E. Chu and A. George, 1987, "Gaussian Elimination With Partial Pivoting and Load Balancing on a Multiprocessor," *Parallel Computing*, 5:65.

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,

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

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 schema can be formalized as LU decomposition

$$A = L\ U$$

L is matrix of multipliers

$$\begin{bmatrix} 1 & 0 & 0 \\ +a_{21}/a_{11} & 1 & 0 \\ +a_{31}/a_{11} & +\frac{a'_{32}}{a'_{22}} & 1 \end{bmatrix}$$

U is resultant matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix}$$

$$L^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -a_{21}/a_{11} & 1 & 0 \\ ? & -a'_{32}/a'_{22} & 1 \end{bmatrix}$$

$$L^{-1} A = U$$

$$a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_1 \quad (1)$$

$$a'_{22} x_2 + a'_{23} x_3 = b'_2 \quad (2')$$

$$a'_{32} x_2 + a'_{33} x_3 = b'_3 \quad (3')$$

ELIMINATE x_2 from equation (3')

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

This scheme is "forward reduction"

Performance Analysis

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

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

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

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

The efficiency is given by,

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

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

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

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

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

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

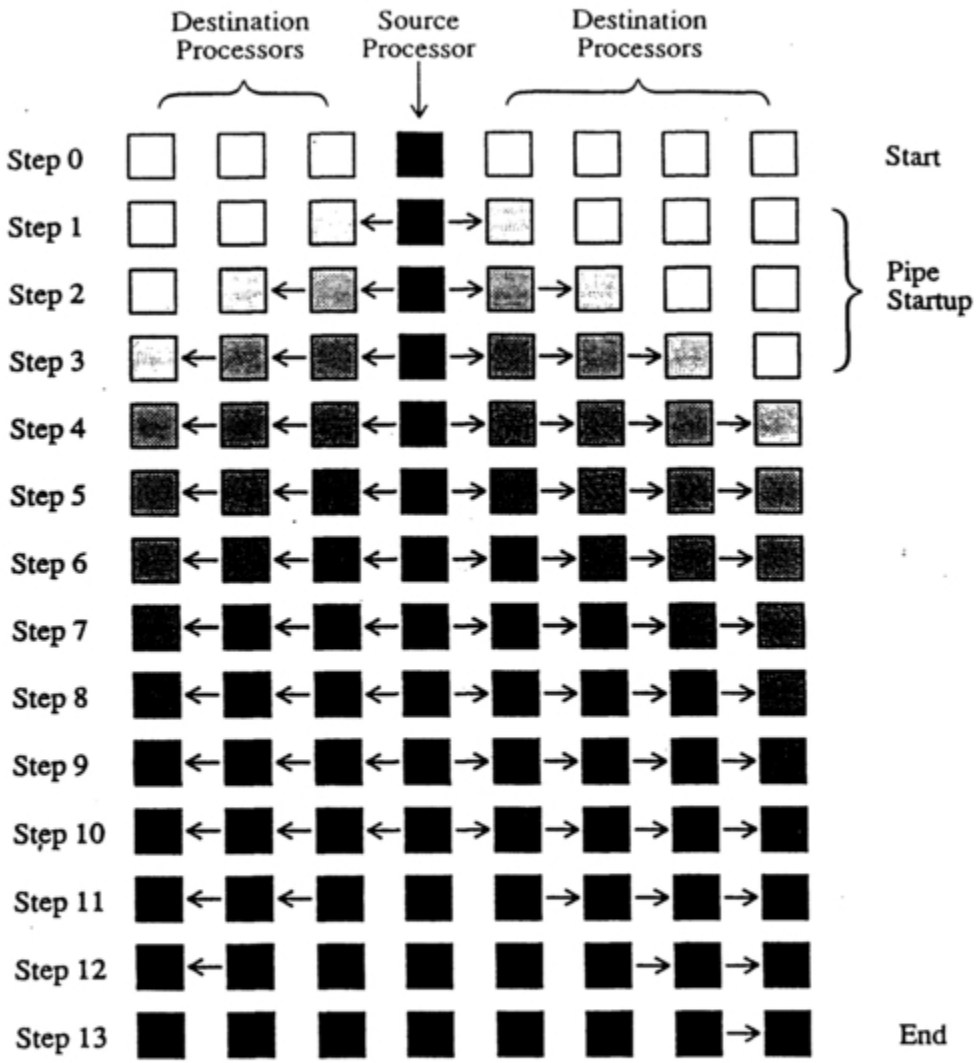
Note:

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

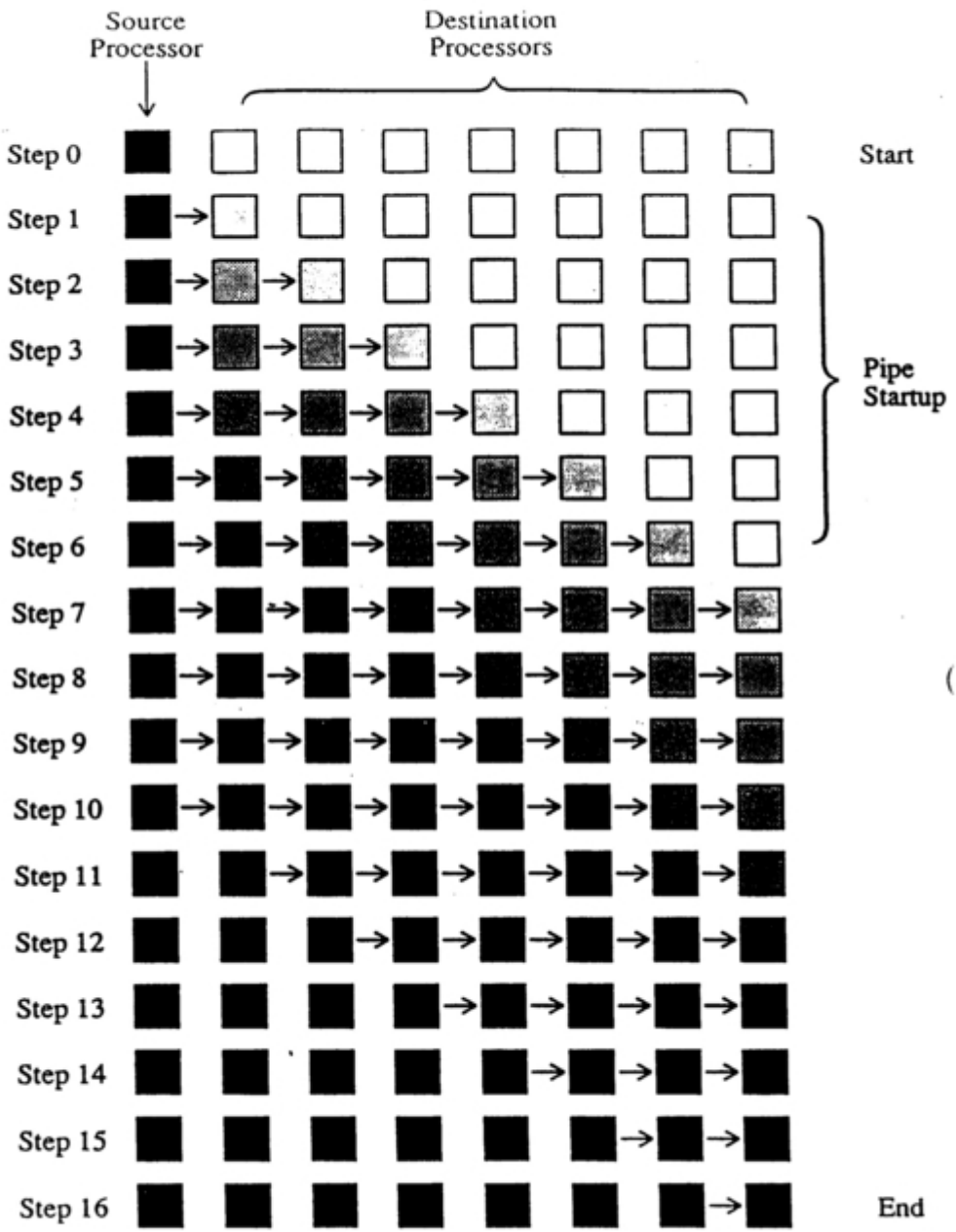
$$\text{If} \quad m^2 > 2\left(\frac{\sqrt{N}-2}{d-2}\right) \quad \text{pipe wins}$$

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

```
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 $j = (i + n) \bmod \sqrt{N}$.
- `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_mult* (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

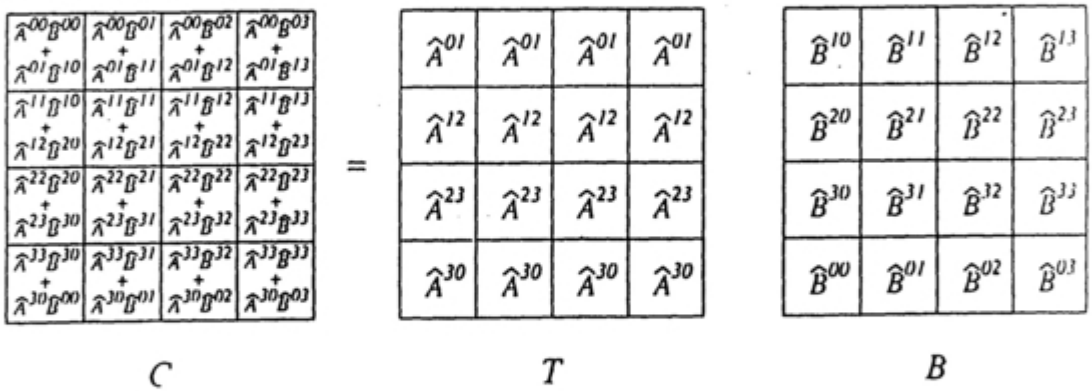
Consider the case where $N = 16$, and look at what happens in a particular processor. We choose the one at position $(2, 1)$.

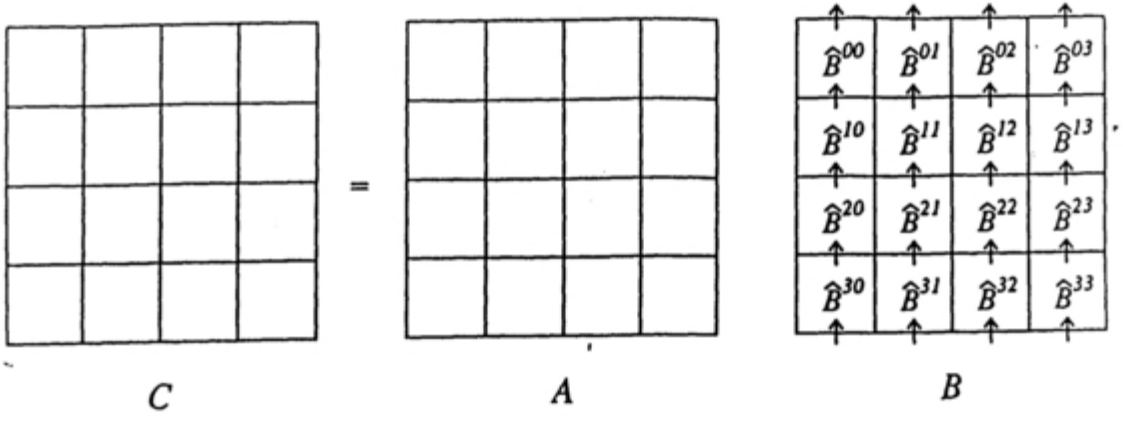
$n = 0$: $T = \hat{A}^{22}$, $B = \hat{B}^{21}$,
$C = \hat{A}^{22}\hat{B}^{21}$

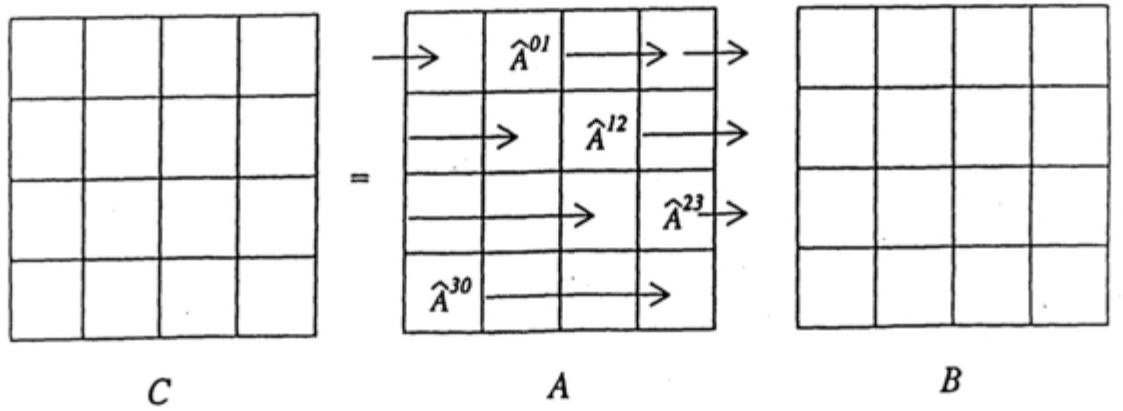
$n = 1$: $T = \hat{A}^{23}$, $B = \hat{B}^{31}$,
$C = \hat{A}^{22}\hat{B}^{21} + \hat{A}^{23}\hat{B}^{31}$

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

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

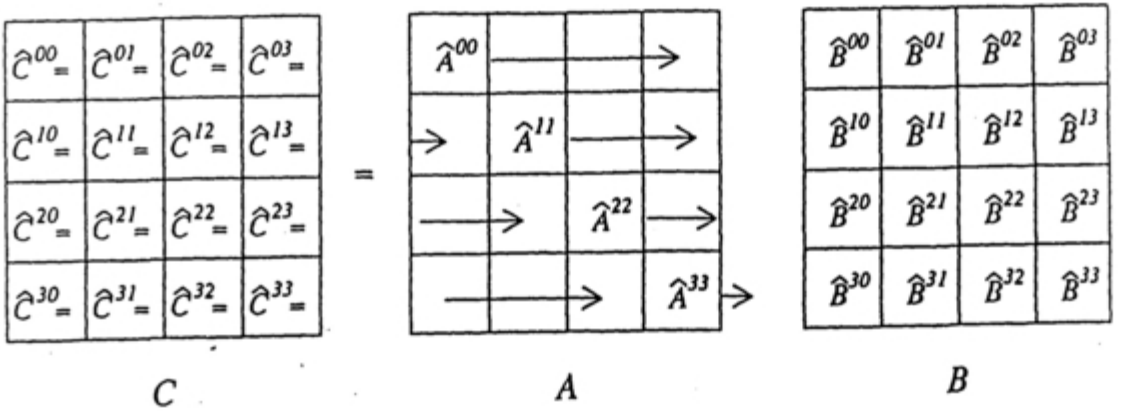

$\widehat{A}^{00}\widehat{B}^{00} + \widehat{A}^{01}\widehat{B}^{10}$
$\widehat{A}^{00}\widehat{B}^{01} + \widehat{A}^{01}\widehat{B}^{11}$
$\widehat{A}^{00}\widehat{B}^{02} + \widehat{A}^{01}\widehat{B}^{12}$
$\widehat{A}^{00}\widehat{B}^{03} + \widehat{A}^{01}\widehat{B}^{13}$
$\widehat{A}^{11}\widehat{B}^{10} + \widehat{A}^{12}\widehat{B}^{20}$
$\widehat{A}^{11}\widehat{B}^{11} + \widehat{A}^{12}\widehat{B}^{21}$
$\widehat{A}^{11}\widehat{B}^{12} + \widehat{A}^{12}\widehat{B}^{22}$
$\widehat{A}^{11}\widehat{B}^{13} + \widehat{A}^{12}\widehat{B}^{23}$
$\widehat{A}^{22}\widehat{B}^{20} + \widehat{A}^{23}\widehat{B}^{30}$
$\widehat{A}^{22}\widehat{B}^{21} + \widehat{A}^{23}\widehat{B}^{31}$
$\widehat{A}^{22}\widehat{B}^{22} + \widehat{A}^{23}\widehat{B}^{32}$
$\widehat{A}^{22}\widehat{B}^{23} + \widehat{A}^{23}\widehat{B}^{33}$
$\widehat{A}^{33}\widehat{B}^{30} + \widehat{A}^{30}\widehat{B}^{00}$
$\widehat{A}^{33}\widehat{B}^{31} + \widehat{A}^{30}\widehat{B}^{01}$
$\widehat{A}^{33}\widehat{B}^{32} + \widehat{A}^{30}\widehat{B}^{02}$
$\widehat{A}^{33}\widehat{B}^{33} + \widehat{A}^{30}\widehat{B}^{03}$
C
=
$\widehat{A}^{01}$ $\widehat{A}^{01}$ $\widehat{A}^{01}$ $\widehat{A}^{01}$
$\widehat{A}^{12}$ $\widehat{A}^{12}$ $\widehat{A}^{12}$ $\widehat{A}^{12}$
$\widehat{A}^{23}$ $\widehat{A}^{23}$ $\widehat{A}^{23}$ $\widehat{A}^{23}$
$\widehat{A}^{30}$ $\widehat{A}^{30}$ $\widehat{A}^{30}$ $\widehat{A}^{30}$
T
$\widehat{B}^{10}$ $\widehat{B}^{11}$ $\widehat{B}^{12}$ $\widehat{B}^{13}$
$\widehat{B}^{20}$ $\widehat{B}^{21}$ $\widehat{B}^{22}$ $\widehat{B}^{23}$
$\widehat{B}^{30}$ $\widehat{B}^{31}$ $\widehat{B}^{32}$ $\widehat{B}^{33}$
$\widehat{B}^{00}$ $\widehat{B}^{01}$ $\widehat{B}^{02}$ $\widehat{B}^{03}$
B

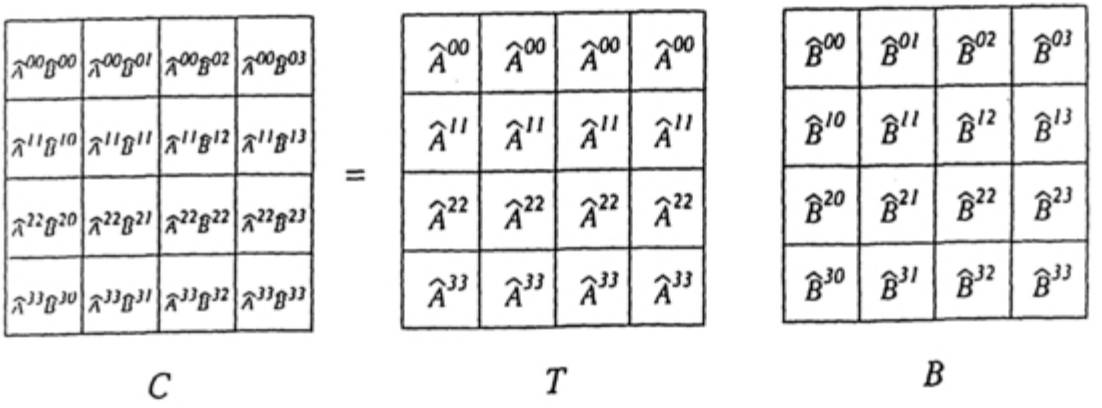

=
$\hat{B}^{00}$ $\hat{B}^{01}$ $\hat{B}^{02}$ $\hat{B}^{03}$
$\hat{B}^{10}$ $\hat{B}^{11}$ $\hat{B}^{12}$ $\hat{B}^{13}$
$\hat{B}^{20}$ $\hat{B}^{21}$ $\hat{B}^{22}$ $\hat{B}^{23}$
$\hat{B}^{30}$ $\hat{B}^{31}$ $\hat{B}^{32}$ $\hat{B}^{33}$
C
A
B


=
$\hat{A}^{01}$
$\hat{A}^{12}$
$\hat{A}^{23}$
$\hat{A}^{30}$
C
A
B

Note each $\hat{C}^{lk}$ 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 row chooses a suitable value of n and uses this to update $\hat{C}$.


$\hat{C}^{00}=$ $\hat{C}^{01}=$ $\hat{C}^{02}=$ $\hat{C}^{03}=$
$\hat{C}^{10}=$ $\hat{C}^{11}=$ $\hat{C}^{12}=$ $\hat{C}^{13}=$
$\hat{C}^{20}=$ $\hat{C}^{21}=$ $\hat{C}^{22}=$ $\hat{C}^{23}=$
$\hat{C}^{30}=$ $\hat{C}^{31}=$ $\hat{C}^{32}=$ $\hat{C}^{33}=$
=
$\hat{A}^{00}$
$\hat{A}^{11}$
$\hat{A}^{22}$
$\hat{A}^{33}$
$\hat{B}^{00}$ $\hat{B}^{01}$ $\hat{B}^{02}$ $\hat{B}^{03}$
$\hat{B}^{10}$ $\hat{B}^{11}$ $\hat{B}^{12}$ $\hat{B}^{13}$
$\hat{B}^{20}$ $\hat{B}^{21}$ $\hat{B}^{22}$ $\hat{B}^{23}$
$\hat{B}^{30}$ $\hat{B}^{31}$ $\hat{B}^{32}$ $\hat{B}^{33}$
C
A
B


$\hat{A}^{00}\hat{B}^{00}$ $\hat{A}^{00}\hat{B}^{01}$ $\hat{A}^{00}\hat{B}^{02}$ $\hat{A}^{00}\hat{B}^{03}$
$\hat{A}^{11}\hat{B}^{10}$ $\hat{A}^{11}\hat{B}^{11}$ $\hat{A}^{11}\hat{B}^{12}$ $\hat{A}^{11}\hat{B}^{13}$
$\hat{A}^{22}\hat{B}^{20}$ $\hat{A}^{22}\hat{B}^{21}$ $\hat{A}^{22}\hat{B}^{22}$ $\hat{A}^{22}\hat{B}^{23}$
$\hat{A}^{33}\hat{B}^{30}$ $\hat{A}^{33}\hat{B}^{31}$ $\hat{A}^{33}\hat{B}^{32}$ $\hat{A}^{33}\hat{B}^{33}$
=
$\hat{A}^{00}$ $\hat{A}^{00}$ $\hat{A}^{00}$ $\hat{A}^{00}$
$\hat{A}^{11}$ $\hat{A}^{11}$ $\hat{A}^{11}$ $\hat{A}^{11}$
$\hat{A}^{22}$ $\hat{A}^{22}$ $\hat{A}^{22}$ $\hat{A}^{22}$
$\hat{A}^{33}$ $\hat{A}^{33}$ $\hat{A}^{33}$ $\hat{A}^{33}$
$\hat{B}^{00}$ $\hat{B}^{01}$ $\hat{B}^{02}$ $\hat{B}^{03}$
$\hat{B}^{10}$ $\hat{B}^{11}$ $\hat{B}^{12}$ $\hat{B}^{13}$
$\hat{B}^{20}$ $\hat{B}^{21}$ $\hat{B}^{22}$ $\hat{B}^{23}$
$\hat{B}^{30}$ $\hat{B}^{31}$ $\hat{B}^{32}$ $\hat{B}^{33}$
C
T
B

$\widehat{A}^{00}$	$\widehat{A}^{01}$	$\widehat{A}^{02}$	$\widehat{A}^{03}$
$\widehat{A}^{10}$	$\widehat{A}^{11}$	$\widehat{A}^{12}$	$\widehat{A}^{13}$
$\widehat{A}^{20}$	$\widehat{A}^{21}$	$\widehat{A}^{22}$	$\widehat{A}^{23}$
$\widehat{A}^{30}$	$\widehat{A}^{31}$	$\widehat{A}^{32}$	$\widehat{A}^{33}$

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 $j = (i + n) \bmod \sqrt{N}$. 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 $\underline{\underline{A}}\,\underline{x} = \underline{b}$

formally $\underline{x} = \underline{\underline{A}}^{-1}\underline{b}$

but this is NOT normally best numerical method

2) If $\underline{\underline{A}}$ Sparse both $\underline{\underline{A}}^{-1}$ and the better "L U 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)

$$\underline{\underline{A}}\ \underline{x} = \underline{b}$$

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 "ground states" ("equilibrium states")

$$A \underline{x}_m = \lambda_m \underline{x}_m$$

e.g. MOPAC

- Equation solution – for reasons similar to those just discussed

$$A \underline{x} = b$$

- Multiplication to "change basis"

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

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

$$|f'_n\rangle = \sum_m f'_{nm} |f_m\rangle$$

$$b_n = \sum_m f'_{nm} a_m$$

$$|f''_\ell\rangle = \sum_{\ell=1}^{M} c_\ell |f'_\ell\rangle$$

$$f''_\ell = \sum_\ell f''_{\ell n} |f'_n\rangle$$

$$c_\ell = \sum_m \left(\sum_n f''_{\ell n} f'_{nm} \right) a_m$$

$\underline{\underline{F''}}\,\underline{\underline{F'}}$ matrix multiplication

Read the literature (e.g. Computer Physics Communications, Nov 1991) for choices of f_n, w_m. Clearly one will take f_n as functions for which $\mathcal{L} f_n$ can be easily calculated.

Comments.

N expansion functions f_n 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 ....

$$\sum_{n=1}^{N} a_n (L f_n) = g$$

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

Need $u(\underline{x}) = 0$.

Choose "suitable" set of weight functions w_m

$$\int_{\text{volume}} w_m^*(\underline{x})\, u(\underline{x})\, d^3\underline{x} = 0$$

$$\underline{\underline{L}}\, \underline{a} = \underline{g}$$

vector $\underline{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix}$

$\underline{\underline{L}}$ is matrix with matrix elements

$$\int w_m^*(\underline{x})\, (L f_n(\underline{x}))\, d^3\underline{x}$$

$\underline{g}$ is vector with coefficients

$$\int w_n^*(\underline{x})\, g(\underline{x})\, d^3\underline{x}$$

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

$$\text{e.g.} \quad L\phi = \frac{\partial^2 \phi}{\partial t^2} - \kappa \nabla^2 \phi$$

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

for any $\underline{k}$ is an eigenfunction

$$\lambda = -\left[\omega^2 - \kappa |\underline{k}|^2\right].$$

So we look at problem where fn is <u>not</u> an eigenfunction

According to Survey, dominant use of "large" ($N \geq 10{,}000$) matrix inversion in Supercomputers is the method of Moments for Computational ~~Electromagnetics~~

Invented by Harrington at Syracuse University (~ 1967?)

$$L f = g$$

← operator

$f = \sum_{n=1}^{N} a_n f_n$ where f_n are Suitable expansion functions for which $L f_n$ can be calculated.

$$\sum_{n=1}^{N} a_n (L f_n) = g$$

Easiest would be use eigenfunctions

$$L f_n = \lambda_n f_n$$

$$g = \sum_n g_n f_n$$

Often—

You often want to find eigenstates

$$H|\varphi_i\rangle = \lambda_i|\varphi_i\rangle$$

wrt $|\varphi_i\rangle$ H is DIAGONAL

However this is usually impossible

Often one knows ~~that~~

$$H = H_A + H_B$$

↑ "basic" ↑ Perturbation

e.g., Compound is H_2O

H_A is "free" Hamiltonian for isolated H H O

H_B is interaction (forces between atoms)

Simple States $|\varphi_i\rangle$ diagonalize H_A

but $\langle\varphi_j|H|\varphi_i\rangle$ will be nonzero for "most" i,j.

Full matrices come from "operators" which link several (~ all) basis 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 (gridmesh)
2) "Pyre" goes to optimal comm algorithm
3) interface "test" (fold) MM algorithm
4) improve "FULL MATRICES" CORTE/BUDS.

We have studied partial differential equation

e.g. $\nabla^2 \varphi = -4\pi \rho(\underline{x})$

and shown how they translate into matrix equations

$$\underline{\underline{A}}\ \underline{x} = \underline{b}$$

$\underline{b}$ corresponds to ρ and boundary conditions

$\underline{\underline{A}}$ corresponds to ∇^2

$\underline{x}$ corresponds to φ

These matrices were "Sparse"

The operator $\underline{\underline{A}}$ (∇^2) only linked a few states ($\underline{x}$ values of $\varphi(\underline{x})$, components of $\underline{x}$)

"Full" matrices are those with "essentially all" elements nonzero — more precisely, it is not worth exploiting the zero's i.e. view as an "ordinary number" $a*0 = a$ / $a*0 = 0$

This well known scheme can be formalized as LU decomposition

$$A = L\ U$$

L is matrix of multipliers

$$\begin{bmatrix} 1 & 0 & 0 \\ +a_{21}/a_{11} & 1 & 0 \\ +a_{31}/a_{11} & +\frac{a'_{32}}{a'_{22}} & 1 \end{bmatrix}$$

U is resultant matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix}$$

$$L^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -a_{21}/a_{11} & 1 & 0 \\ ? & -a'_{32}/a'_{22} & 1 \end{bmatrix}$$

$$L^{-1} A = U$$

$$a_{11} x_1 + a_{12} x_2 + a_{13} x_3 = b_1 \quad (1)$$

$$a'_{22} x_2 + a'_{23} x_3 = b'_2 \quad (2')$$

$$a'_{32} x_2 + a'_{33} x_3 = b'_3 \quad (3')$$

ELIMINATE x_2 from equation (3')

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

This scheme is "forward reduction"

$$a''_{3,3}\, x_3 = b''_3 \qquad (3'')$$

$$a'_{2,2}\, x_2 + a'_{2,3}\, x_3 = b'_2 \qquad (2')$$

$$a_{1,1}\, x_1 + a_{1,2}\, x_2 + a_{1,3}\, x_3 = b_3 \qquad (1)$$

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

$$A\underline{x} = \underline{b} \quad ?$$

Gaussian Elimination is essential idea

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \quad (1)$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \quad (2)$$

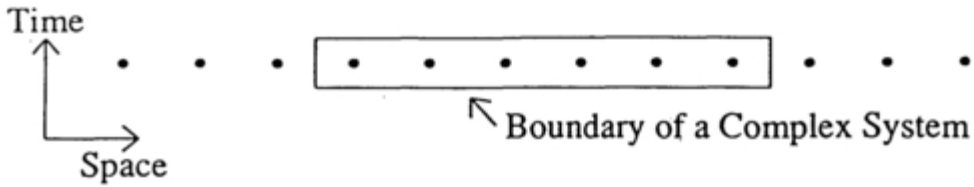
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \quad (3)$$

ELIMINATE x_1 from equations (2) and (3)

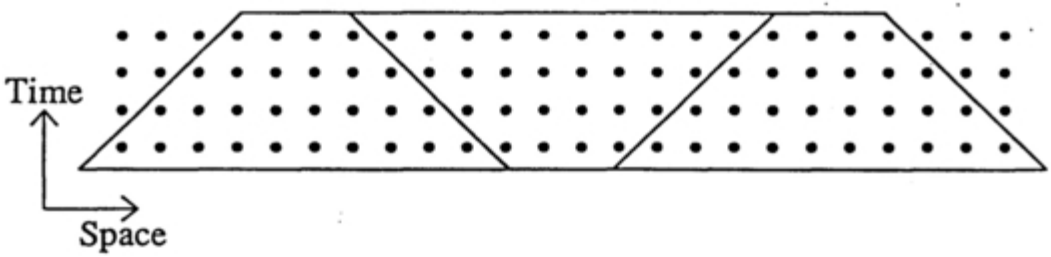
$$(2) \Rightarrow (2') = (2) - \frac{a_{21}}{a_{11}}(1)$$

$$(3) \Rightarrow (3') = (3) - \frac{a_{31}}{a_{11}}(1)$$

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



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



(c) A More Practical Decomposition With More Communication

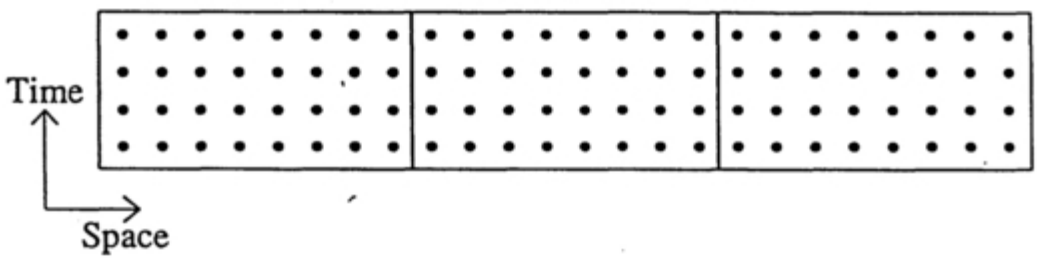


Fig. 10. Three decompositions in space and time of the one dimensional wave equation discussed in the text of Sec. IVB.

The BLAS concept

- vector - vector

$$V = \sum_{i=1}^{m} a_i b_i$$

$2m$ reads 1 write $2m-1$ floating point operations

floats / memory access ~ 1

- matrix - vector

$$V_i = \sum_{j=1}^{m} a_{ij} b_j$$

$m^2 + m$ reads m writes $m(m-1) + m^2$ floating point

$\therefore$ floats / memory access ~ 2

- matrix - matrix

$$V_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

$2m^2$ reads m^2 writes $m^3 + m^2(m-1)$ floating point

$\therefore$ 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 $\hat{c}^{lk}$ 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 row 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 *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:

```
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 × 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	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10	8	9	11	10

Some References

G. C. Fox, 1984, "LU Decomposition for Banded Matrices," Caltech Report C^3P-99.

S. Lennart Johnsson, 1985, "Solving Narrow Banded Systems on Ensemble Architectures," *ACM Trans. on Math. Software*, **11**:271.

Y. Saad and M. H. Schultz, 1985, "Parallel Direct Methods for Solving Banded Linear Systems," Yale Research Report YALEU/DCS/RR-387.

J. J. Dongarra and S. Lennart Johnsson, 1987, "Solving Banded Systems on a Parallel Processor," *Parallel Computing*, **5**:219.

D. W. Walker, T. Aldcroft, A. Cisneros, G. C. Fox, and W. Furmanski, 1988, "LU Decomposition of Banded Matrices and the Solution of Linear Systems on Hypercubes," in *Proceedings of the Third Conference on Hypercube Concurrent Processors and Applications*, published by ACM Press, New York.

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.

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, $i = 1, \ldots, M-1-k$)

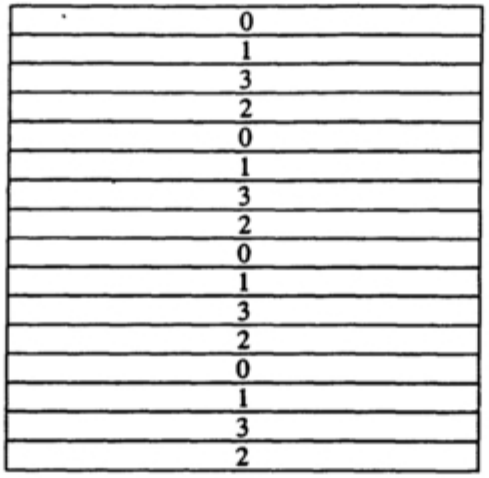
$A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} * A_{k,k+j}$

for_end

for_end

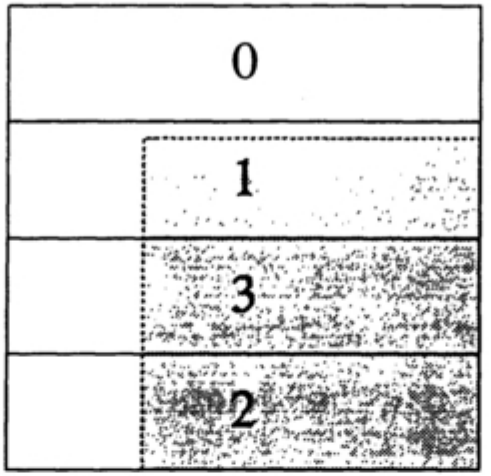
for_end

Scattered Row Decomposition



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

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

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, $i = 1, \ldots, M-1-k$)

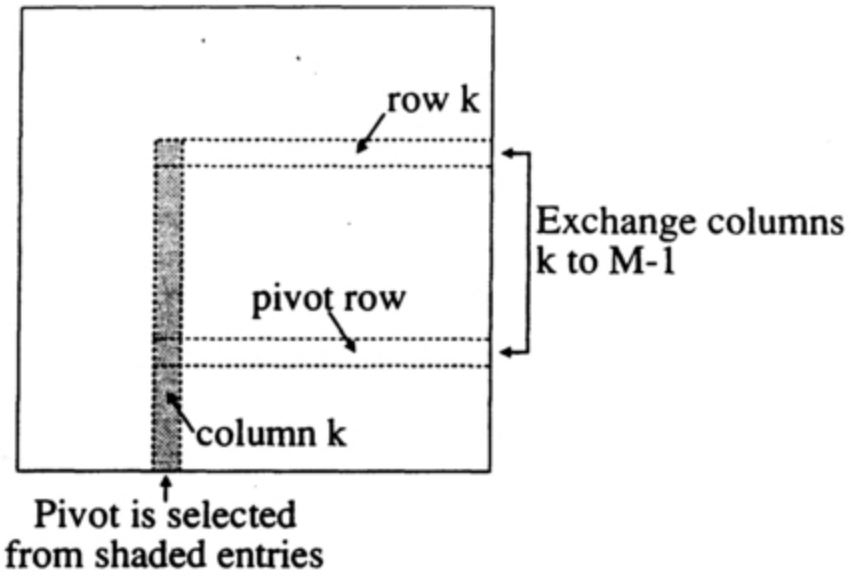
$A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} * A_{k,k+j}$

for_end

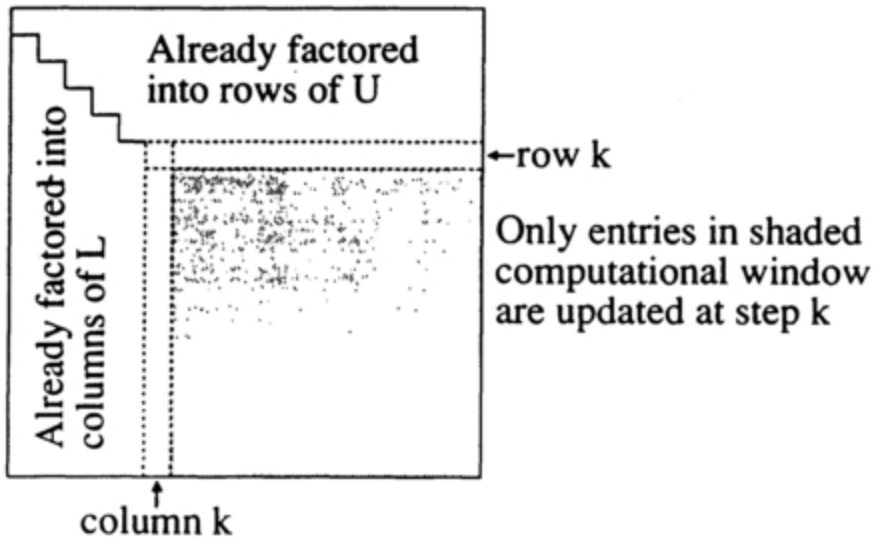
for_end

for_end

Pivot Selection at Step k



Factorization After k Steps



- 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:

$$L_{k,k} = 1$$

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

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

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:

G. A. Geist and M. T. Heath, 1986, "Matrix Factorization on a Hypercube Multiprocessor," in *Hypercube Multiprocessors 1986*, published by SIAM Press, Philadelphia.

E. Chu and A. George, 1987, "Gaussian Elimination With Partial Pivoting and Load Balancing on a Multiprocessor," *Parallel Computing*, **5**:65.

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,

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

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

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

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

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

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

The efficiency is given by,

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

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

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

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

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

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

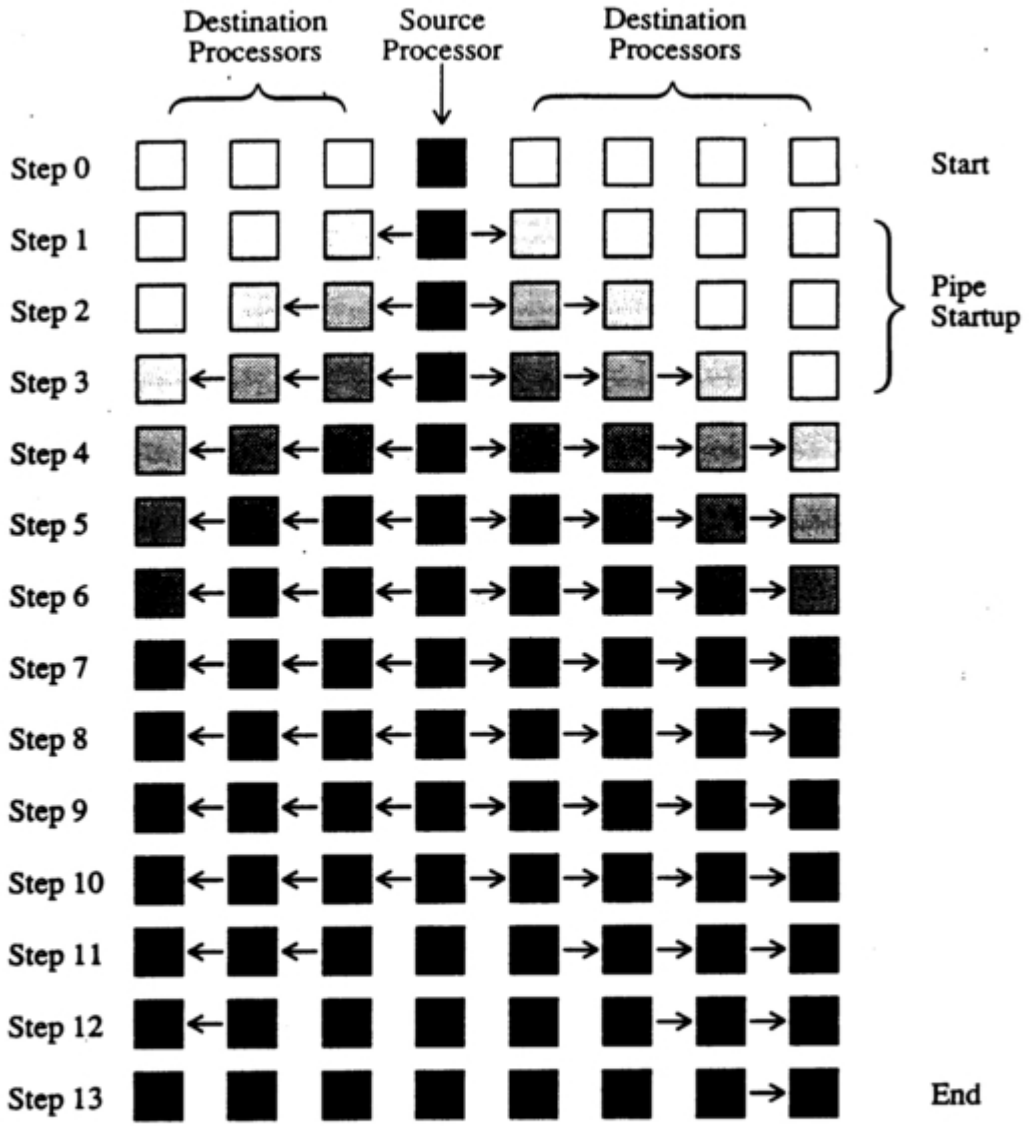
Note:

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

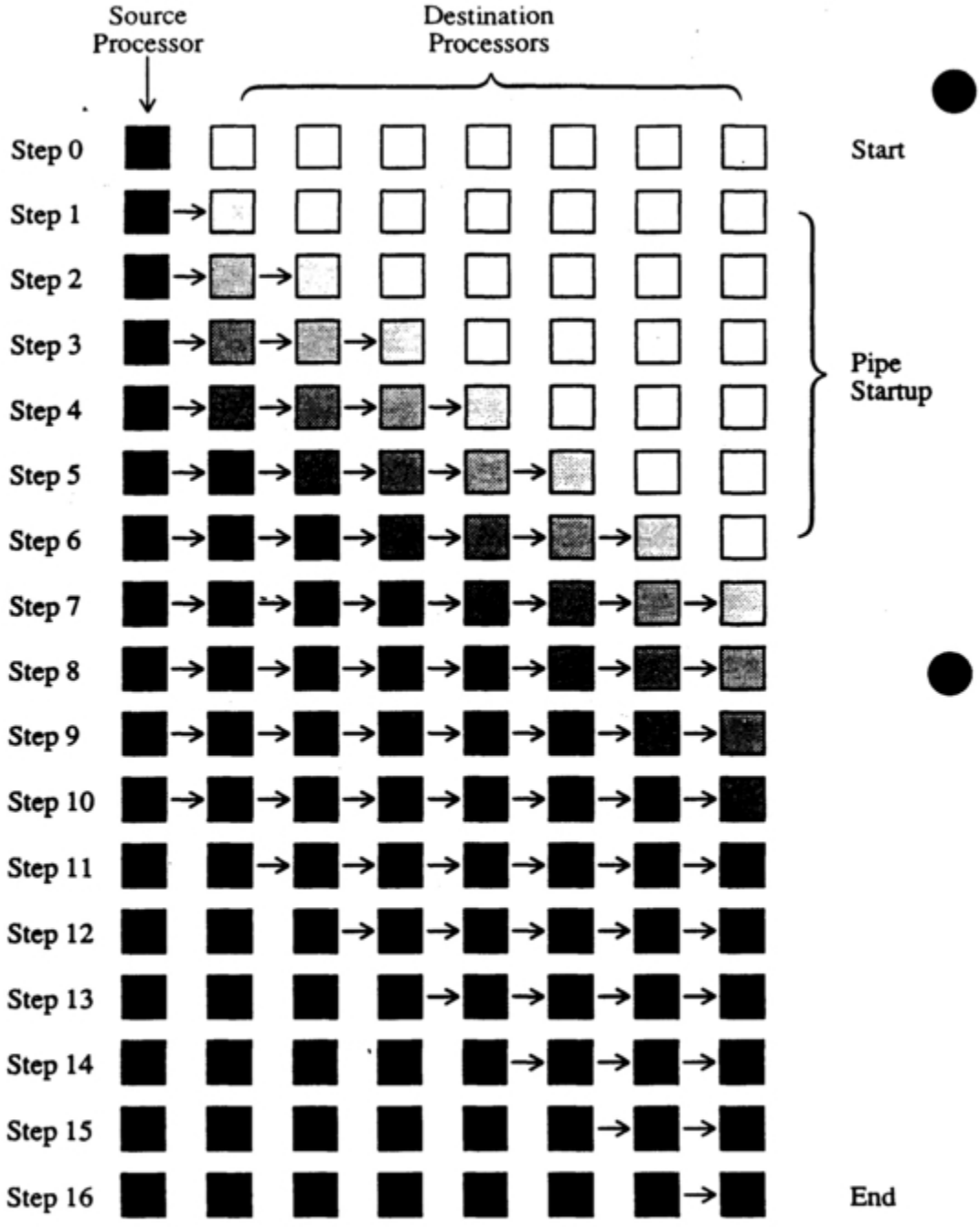
$$\text{If} \quad m^2 > 2\left(\frac{\sqrt{N} - 2}{d - 2}\right) \quad \text{pipe wins}$$

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

```
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 $j = (i + n) \bmod \sqrt{N}$.
- **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_mult* (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

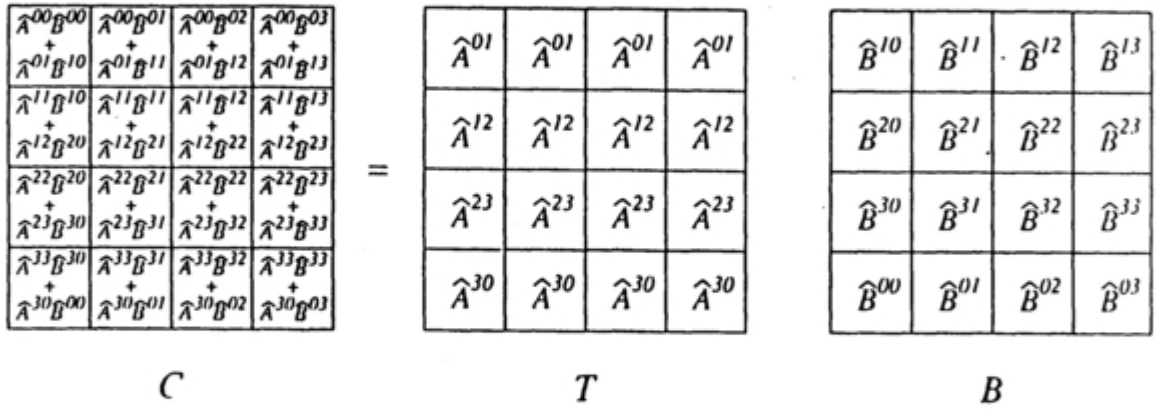
Consider the case where $N = 16$, and look at what happens in a particular processor. We choose the one at position $(2, 1)$.

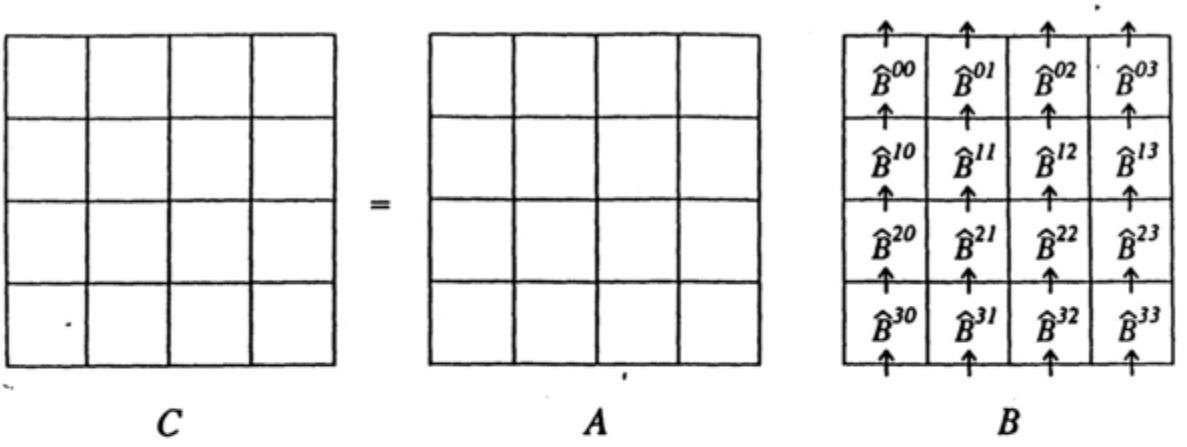
$n = 0$: $T = \hat{A}^{22}$, $B = \hat{B}^{21}$,
$C = \hat{A}^{22}\hat{B}^{21}$

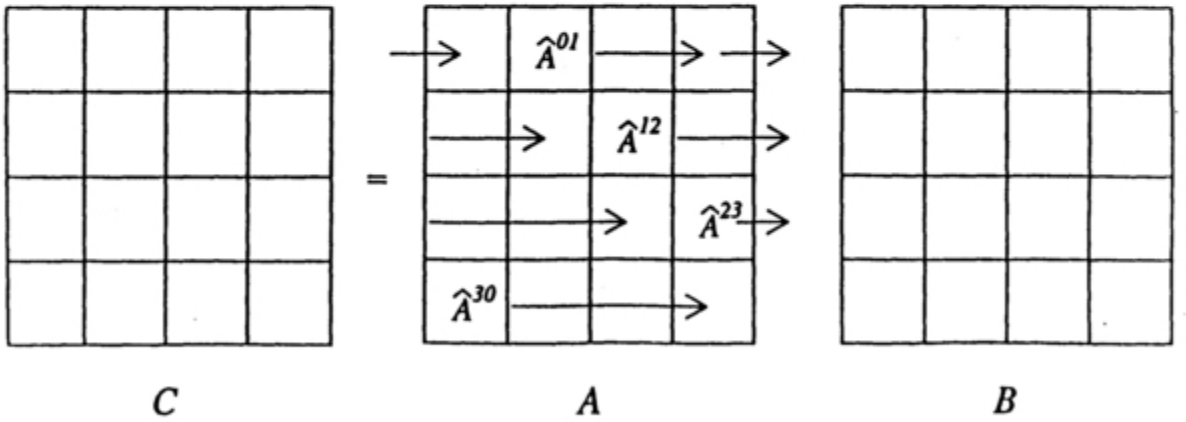
$n = 1$: $T = \hat{A}^{23}$, $B = \hat{B}^{31}$,
$C = \hat{A}^{22}\hat{B}^{21} + \hat{A}^{23}\hat{B}^{31}$

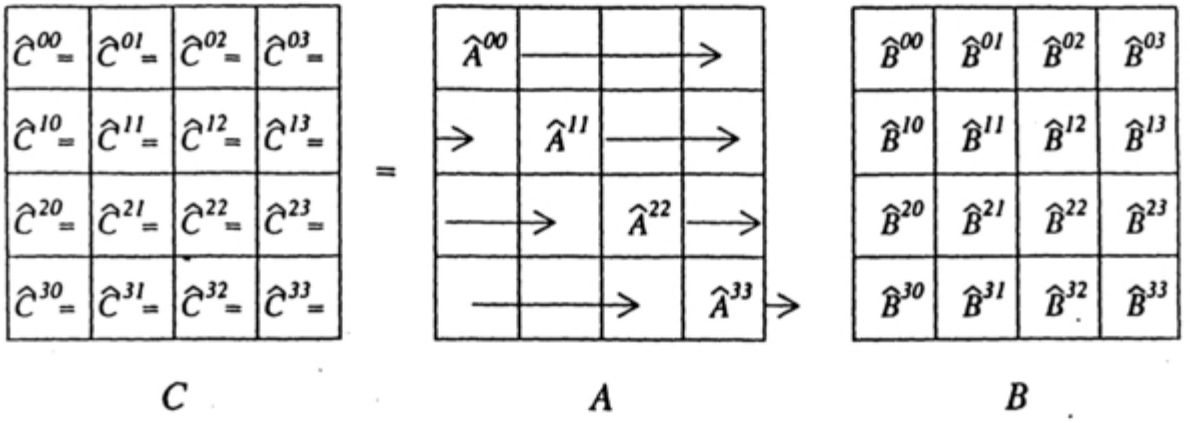
$n = 2$: $T = \hat{A}^{20}$, $B = \hat{B}^{01}$,
$C = \hat{A}^{22}\hat{B}^{21} + \hat{A}^{23}\hat{B}^{31} + \hat{A}^{20}\hat{B}^{01}$

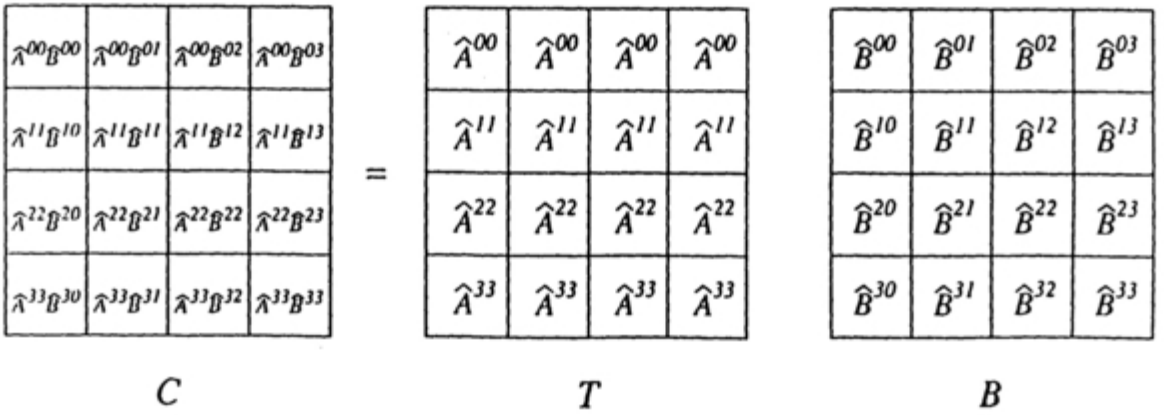
$n = 3$: $T = \hat{A}^{21}$, $B = \hat{B}^{11}$,
$C = \hat{A}^{22}\hat{B}^{21} + \hat{A}^{23}\hat{B}^{31} + \hat{A}^{20}\hat{B}^{01} + \hat{A}^{21}\hat{B}^{11}$


$\hat{A}^{00}\hat{B}^{00} + \hat{A}^{01}\hat{B}^{10}$
$\hat{A}^{00}\hat{B}^{01} + \hat{A}^{01}\hat{B}^{11}$
$\hat{A}^{00}\hat{B}^{02} + \hat{A}^{01}\hat{B}^{12}$
$\hat{A}^{00}\hat{B}^{03} + \hat{A}^{01}\hat{B}^{13}$
$\hat{A}^{11}\hat{B}^{10} + \hat{A}^{12}\hat{B}^{20}$
$\hat{A}^{11}\hat{B}^{11} + \hat{A}^{12}\hat{B}^{21}$
$\hat{A}^{11}\hat{B}^{12} + \hat{A}^{12}\hat{B}^{22}$
$\hat{A}^{11}\hat{B}^{13} + \hat{A}^{12}\hat{B}^{23}$
$\hat{A}^{22}\hat{B}^{20} + \hat{A}^{23}\hat{B}^{30}$
$\hat{A}^{22}\hat{B}^{21} + \hat{A}^{23}\hat{B}^{31}$
$\hat{A}^{22}\hat{B}^{22} + \hat{A}^{23}\hat{B}^{32}$
$\hat{A}^{22}\hat{B}^{23} + \hat{A}^{23}\hat{B}^{33}$
$\hat{A}^{33}\hat{B}^{30} + \hat{A}^{30}\hat{B}^{00}$
$\hat{A}^{33}\hat{B}^{31} + \hat{A}^{30}\hat{B}^{01}$
$\hat{A}^{33}\hat{B}^{32} + \hat{A}^{30}\hat{B}^{02}$
$\hat{A}^{33}\hat{B}^{33} + \hat{A}^{30}\hat{B}^{03}$
=
$\hat{A}^{01}$ $\hat{A}^{01}$ $\hat{A}^{01}$ $\hat{A}^{01}$
$\hat{A}^{12}$ $\hat{A}^{12}$ $\hat{A}^{12}$ $\hat{A}^{12}$
$\hat{A}^{23}$ $\hat{A}^{23}$ $\hat{A}^{23}$ $\hat{A}^{23}$
$\hat{A}^{30}$ $\hat{A}^{30}$ $\hat{A}^{30}$ $\hat{A}^{30}$
$\hat{B}^{10}$ $\hat{B}^{11}$ $\hat{B}^{12}$ $\hat{B}^{13}$
$\hat{B}^{20}$ $\hat{B}^{21}$ $\hat{B}^{22}$ $\hat{B}^{23}$
$\hat{B}^{30}$ $\hat{B}^{31}$ $\hat{B}^{32}$ $\hat{B}^{33}$
$\hat{B}^{00}$ $\hat{B}^{01}$ $\hat{B}^{02}$ $\hat{B}^{03}$
C
T
B


=
$\hat{B}^{00}$ $\hat{B}^{01}$ $\hat{B}^{02}$ $\hat{B}^{03}$
$\hat{B}^{10}$ $\hat{B}^{11}$ $\hat{B}^{12}$ $\hat{B}^{13}$
$\hat{B}^{20}$ $\hat{B}^{21}$ $\hat{B}^{22}$ $\hat{B}^{23}$
$\hat{B}^{30}$ $\hat{B}^{31}$ $\hat{B}^{32}$ $\hat{B}^{33}$
C
A
B


=
$\hat{A}^{01}$
$\hat{A}^{12}$
$\hat{A}^{23}$
$\hat{A}^{30}$
C
A
B


$\hat{C}^{00}=$ $\hat{C}^{01}=$ $\hat{C}^{02}=$ $\hat{C}^{03}=$
$\hat{C}^{10}=$ $\hat{C}^{11}=$ $\hat{C}^{12}=$ $\hat{C}^{13}=$
$\hat{C}^{20}=$ $\hat{C}^{21}=$ $\hat{C}^{22}=$ $\hat{C}^{23}=$
$\hat{C}^{30}=$ $\hat{C}^{31}=$ $\hat{C}^{32}=$ $\hat{C}^{33}=$
=
$\hat{A}^{00}$
$\hat{A}^{11}$
$\hat{A}^{22}$
$\hat{A}^{33}$
$\hat{B}^{00}$ $\hat{B}^{01}$ $\hat{B}^{02}$ $\hat{B}^{03}$
$\hat{B}^{10}$ $\hat{B}^{11}$ $\hat{B}^{12}$ $\hat{B}^{13}$
$\hat{B}^{20}$ $\hat{B}^{21}$ $\hat{B}^{22}$ $\hat{B}^{23}$
$\hat{B}^{30}$ $\hat{B}^{31}$ $\hat{B}^{32}$ $\hat{B}^{33}$
C
A
B


$\hat{A}^{00}\hat{B}^{00}$ $\hat{A}^{00}\hat{B}^{01}$ $\hat{A}^{00}\hat{B}^{02}$ $\hat{A}^{00}\hat{B}^{03}$
$\hat{A}^{11}\hat{B}^{10}$ $\hat{A}^{11}\hat{B}^{11}$ $\hat{A}^{11}\hat{B}^{12}$ $\hat{A}^{11}\hat{B}^{13}$
$\hat{A}^{22}\hat{B}^{20}$ $\hat{A}^{22}\hat{B}^{21}$ $\hat{A}^{22}\hat{B}^{22}$ $\hat{A}^{22}\hat{B}^{23}$
$\hat{A}^{33}\hat{B}^{30}$ $\hat{A}^{33}\hat{B}^{31}$ $\hat{A}^{33}\hat{B}^{32}$ $\hat{A}^{33}\hat{B}^{33}$
=
$\hat{A}^{00}$ $\hat{A}^{00}$ $\hat{A}^{00}$ $\hat{A}^{00}$
$\hat{A}^{11}$ $\hat{A}^{11}$ $\hat{A}^{11}$ $\hat{A}^{11}$
$\hat{A}^{22}$ $\hat{A}^{22}$ $\hat{A}^{22}$ $\hat{A}^{22}$
$\hat{A}^{33}$ $\hat{A}^{33}$ $\hat{A}^{33}$ $\hat{A}^{33}$
$\hat{B}^{00}$ $\hat{B}^{01}$ $\hat{B}^{02}$ $\hat{B}^{03}$
$\hat{B}^{10}$ $\hat{B}^{11}$ $\hat{B}^{12}$ $\hat{B}^{13}$
$\hat{B}^{20}$ $\hat{B}^{21}$ $\hat{B}^{22}$ $\hat{B}^{23}$
$\hat{B}^{30}$ $\hat{B}^{31}$ $\hat{B}^{32}$ $\hat{B}^{33}$
C
T
B

$\hat{A}^{00}$	$\hat{A}^{01}$	$\hat{A}^{02}$	$\hat{A}^{03}$
$\hat{A}^{10}$	$\hat{A}^{11}$	$\hat{A}^{12}$	$\hat{A}^{13}$
$\hat{A}^{20}$	$\hat{A}^{21}$	$\hat{A}^{22}$	$\hat{A}^{23}$
$\hat{A}^{30}$	$\hat{A}^{31}$	$\hat{A}^{32}$	$\hat{A}^{33}$

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 $j = (i + n) \bmod \sqrt{N}$. 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.

Some References

This is by no means a complete list:

S. Lennart Johnsson, 1987, "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures," *Journal of Parallel and Distributed Computing*, **4**:133.

G. C. Fox, S. W. Otto, and A. J. G. Hey, 1987, "Matrix Algorithms on a Hypercube, I: Matrix Multiplication," *Parallel Computing*, **4**:17.

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 $\underline{\underline{A}}\,\underline{x} = \underline{b}$

formally $\underline{x} = \underline{\underline{A}}^{-1}\underline{b}$

but this is NOT normally best numerical method

2) If $\underline{\underline{A}}$ 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 Solvers (FULL MATRIX)

$$\underline{\underline{A}}\,\underline{x} = \underline{b}$$

Sometimes used but not very common. ~~for~~ 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 "ground states" ("equilibrium states")

$$A \underline{x}_m = \lambda_m \underline{x}_m$$

e.g. MOPAC

- Equation solution – for reasons similar to those just discussed

$$A \underline{x} = b$$

- Multiplication to "change basis"

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

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

$$|f'_n\rangle = \sum_m f'_{nm} |f_m\rangle$$

$$b_n = \sum_m f'_{nm} a_m$$

$$|f''_\ell\rangle = \sum_{\ell=1}^{M} c_\ell |f'_\ell\rangle$$

$$f''_\ell = \sum_\ell f''_{\ell n} |f'_n\rangle$$

$$c_\ell = \sum_n \left(\sum_m f''_{\ell n} f'_{nm} \right) a_m$$

$\underline{\underline{F}}'' \underline{\underline{F}}'$ matrix multiplication

Read the literature (e.g. Computer Physics Communications, Nov 1991) for choices of f_n, w_n. Clearly one will take f_n as functions for which $\mathcal{L} f_n$ can be easily calculated.

Comments.

N expansion functions f_n 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 . . . .

$$\sum_{n=1}^{N} a_n (L f_n) = g$$

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

Need $u(\underline{x}) = 0$.

Choose "subset" set of weight functions w_m

$$\int_{\text{volume}} w_m^*(\underline{x})\, u(\underline{x})\, d^3\underline{x} = 0$$

$$\underline{\underline{L}}\, \underline{a} = \underline{g}$$

vector $\underline{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix}$

$\underline{\underline{L}}$ is matrix with matrix elements

$$\int w_m^*(\underline{x})\, (L f_n(\underline{x}))\, d^3\underline{x}$$

g is vector with coefficients

$$\int w_m^*(\underline{x})\, g(\underline{x})\, d^3\underline{x}$$

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

$$\text{e.g.} \quad L\phi = \frac{\partial^2 \phi}{\partial t^2} - \kappa \nabla^2 \phi$$

$$\phi = e^{i(\omega t - \underline{k}.\underline{x})}$$

for any $\underline{k}$ is an eigenfunction

$$\lambda = -\left[\omega^2 - \kappa |\underline{k}|^2\right].$$

So we look at problem where fn is <u>not</u> an eigenfunction

According to Survey, dominant use of "large" ($N \geq 10{,}000$) matrix inversion in Supercomputers is the method of Moments for Computational Electromagnetics

Invented by Harrington at Syracuse University (~ 1967?)

$$Lf = g \qquad \leftarrow \text{operator}$$

$f = \sum_{n=1}^{N} a_n f_n$ 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

$$Lf_n = \lambda_n f_n$$

$$g = \sum_n g_n f_n$$

$$a_n = g_n / \lambda_n$$

Often

You often want to find eigenstates

$$H|\varphi_i\rangle = \lambda_i|\varphi_i\rangle$$

wrt $|\varphi_i\rangle$ H is DIAGONAL

However this is usually impossible

Often one knows ~~that~~

$$H = H_A + H_B$$

H_A: "basic"; H_B: Perturbation

e.g., Compound is H_2O

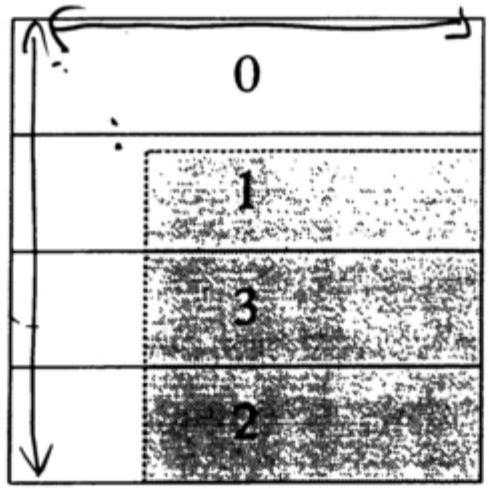
H_A is "free" Hamiltonian for isolated H H O

H_B is Interaction (forces between atoms)

Simple States $|\varphi_i\rangle$ diagonalize H_A

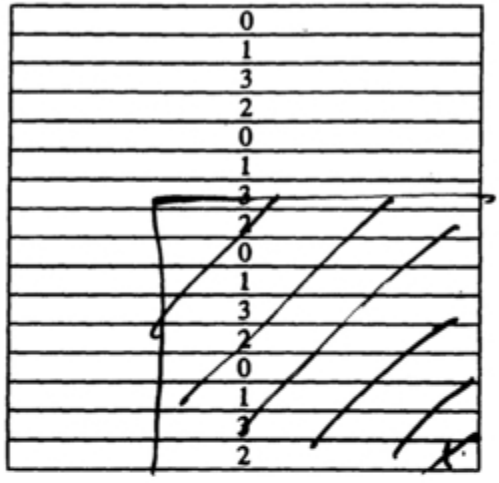
but $\langle\varphi_j|H|\varphi_i\rangle$ will be nonzero for "most" i,j.

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.

$m \times m$

$\frac{m}{2N_{proc}}$ rows per processor

Load balance $\propto \frac{2N_{proc}}{m}$

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, $i = 1, \ldots, M-1-k$)

$A_{k+i,k+j} = A_{k+i,k+j} - A_{k+i,k} * A_{k,k+j}$

for_end

for_end

for_end

Communication in the Parallel LU Decomposition Algorithm

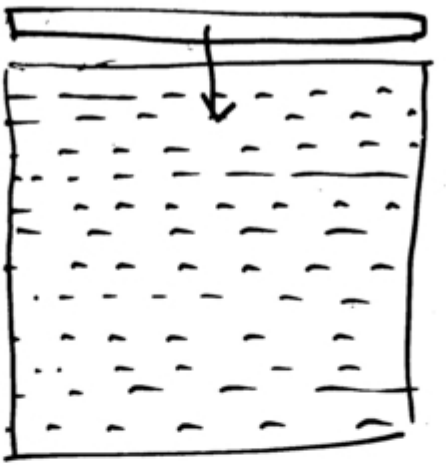
- We can perform the broadcast of the pivot row by means of the *pipe* algorithm, as used in the matrix multiplication algorithm.
- If pivoting is necessary at step k we can send row k to the appropriate processor using the shortest available pipe.
- The pivot row can be selected by using the CrOS III *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

$2\,m^2/N_{proc}$

$A_{ij} = A_{ij} - L \cdot U$

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.

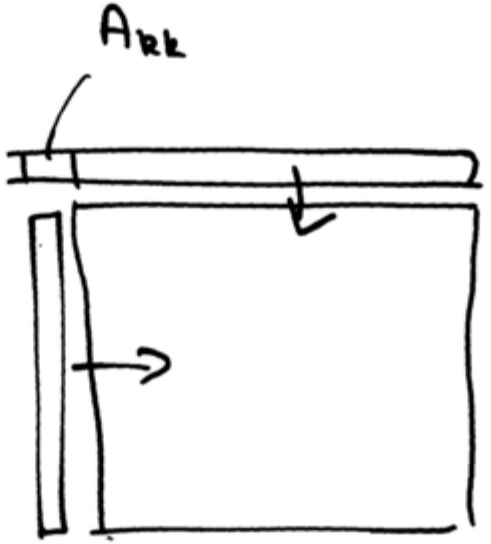
(2) Each candidate passes the absolute value of its pivot candidate, and the corresponding row number, to the CrOS III routine *combine*. This gives the pivot row number.

(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

G. C. Fox, 1984, "LU Decomposition for Banded Matrices," Caltech Report C^3P-99.

S. Lennart Johnsson, 1985, "Solving Narrow Banded Systems on Ensemble Architectures," *ACM Trans. on Math. Software*, **11**:271.

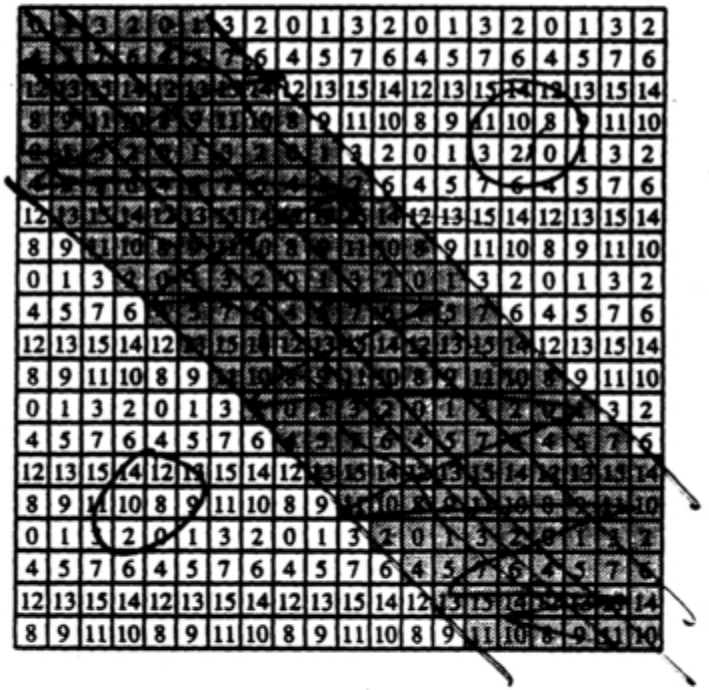
Y. Saad and M. H. Schultz, 1985, "Parallel Direct Methods for Solving Banded Linear Systems," Yale Research Report YALEU/DCS/RR-387.

J. J. Dongarra and S. Lennart Johnsson, 1987, "Solving Banded Systems on a Parallel Processor," *Parallel Computing*, **5**:219.

D. W. Walker, T. Aldcroft, A. Cisneros, G. C. Fox, and W. Furmanski, 1988, "LU Decomposition of Banded Matrices and the Solution of Linear Systems on Hypercubes," in *Proceedings of the Third Conference on Hypercube Concurrent Processors and Applications*, published by ACM Press, New York.

Banded Matrix Decomposition

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



$A x = b$

Communication in the Banded Algorithm

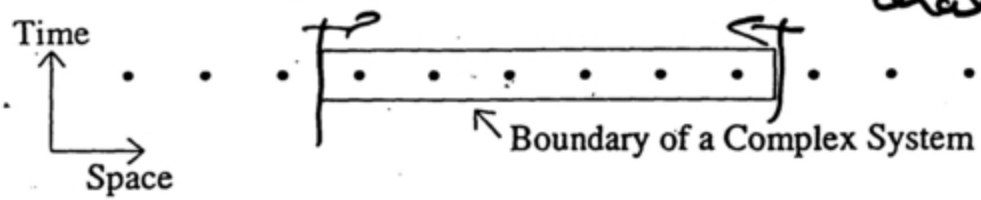
- To update the elements in the computational window we need to be able to communicate,
  - $L_{k+i} = A_{k+i,k}$ $(i = 0, 1, \ldots, \hat{m} - 1)$ to the other processors in the same row of the template.
  - $U_{k+j} = A_{k,k+j}$ $(j = 0, 1, \ldots, \hat{m} - 1)$ to the other processors in the same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

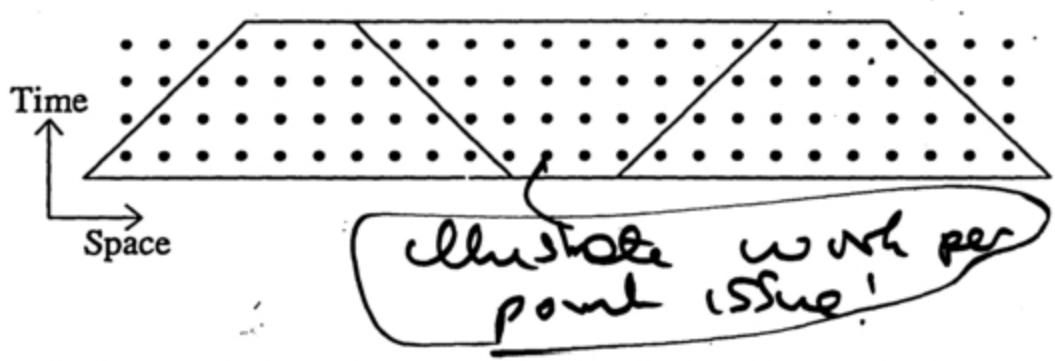
```
if ( row_pos == 0 )
  vwrite(Abuf,down,fsize,offset,mhat);
else if ( row_pos == 1 )
  vread(U,up,down,fsize,fsize,mmax);
else if ( row_pos == 2 )
  vread(U,up,0,fsize,fsize,mmax);
```

WOULD BE BETTER TO do in two dimensions so matrix multiply analogy case

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



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



(c) A More Practical Decomposition With More Communication

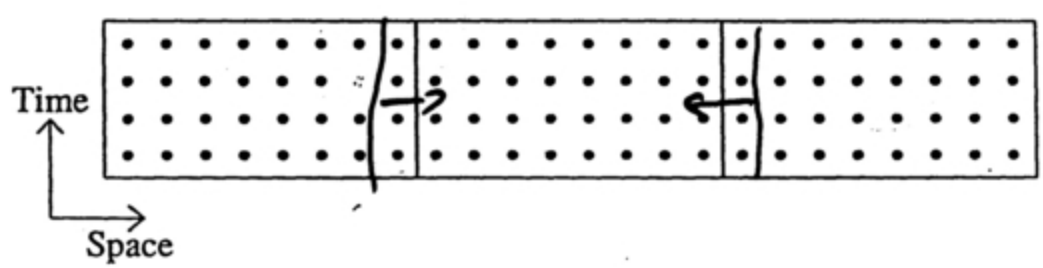


Fig. 10. Three decompositions in space and time of the one dimensional wave equation discussed in the text of Sec. IVB.