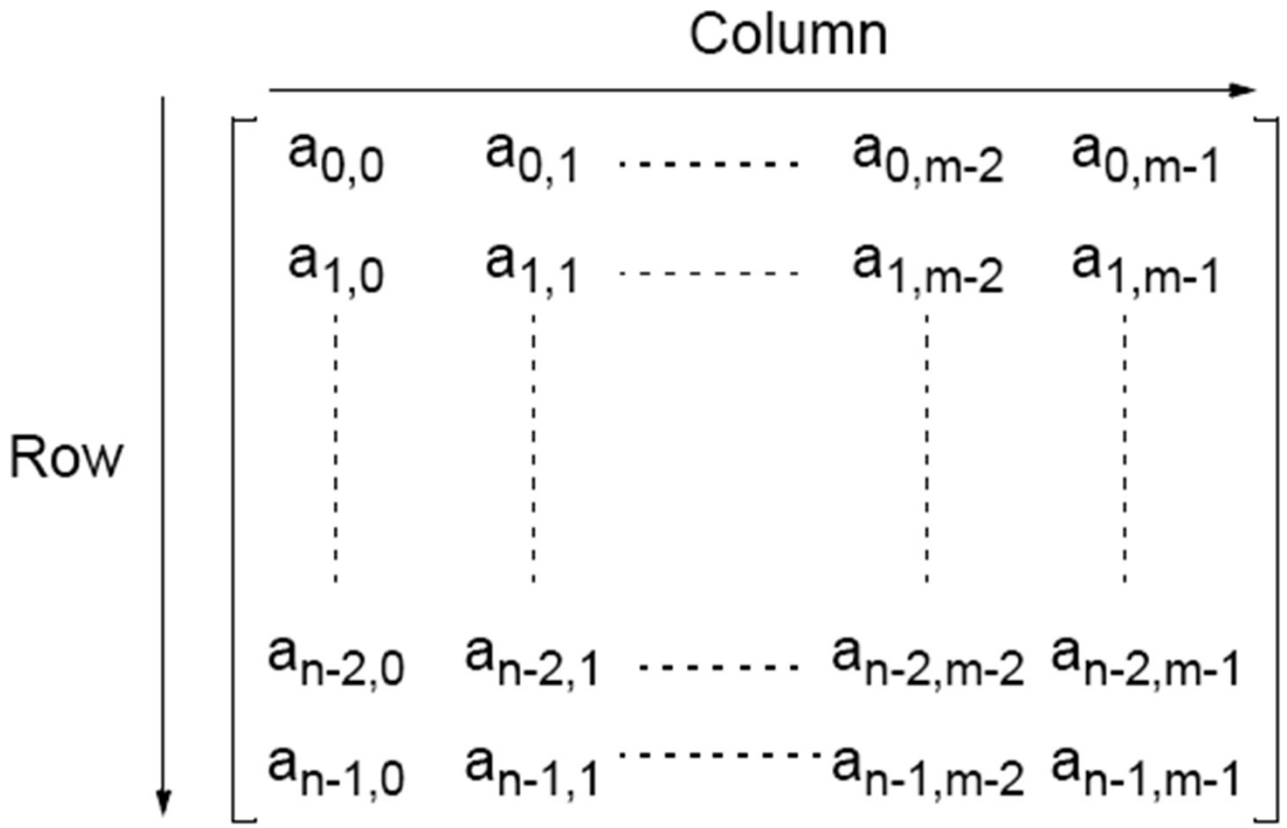
Numerical Algorithms

Courtesy to B. Wilkinson & M. Allen

- **Matrix multiplication**
- **Solving a system of linear equations**

Matrices — A Review

An *n x m* matrix



Matrix Addition

Involves adding corresponding elements of each matrix to form the result matrix.

Given the elements of **A** as *ai*,*j* and the elements of **B** as *bi*,*j*, each element of **C** is computed as

$$c_{i,j} = a_{i,j} + b_{i,j}$$

$$(0 \leq i < n, 0 \leq j < m)$$

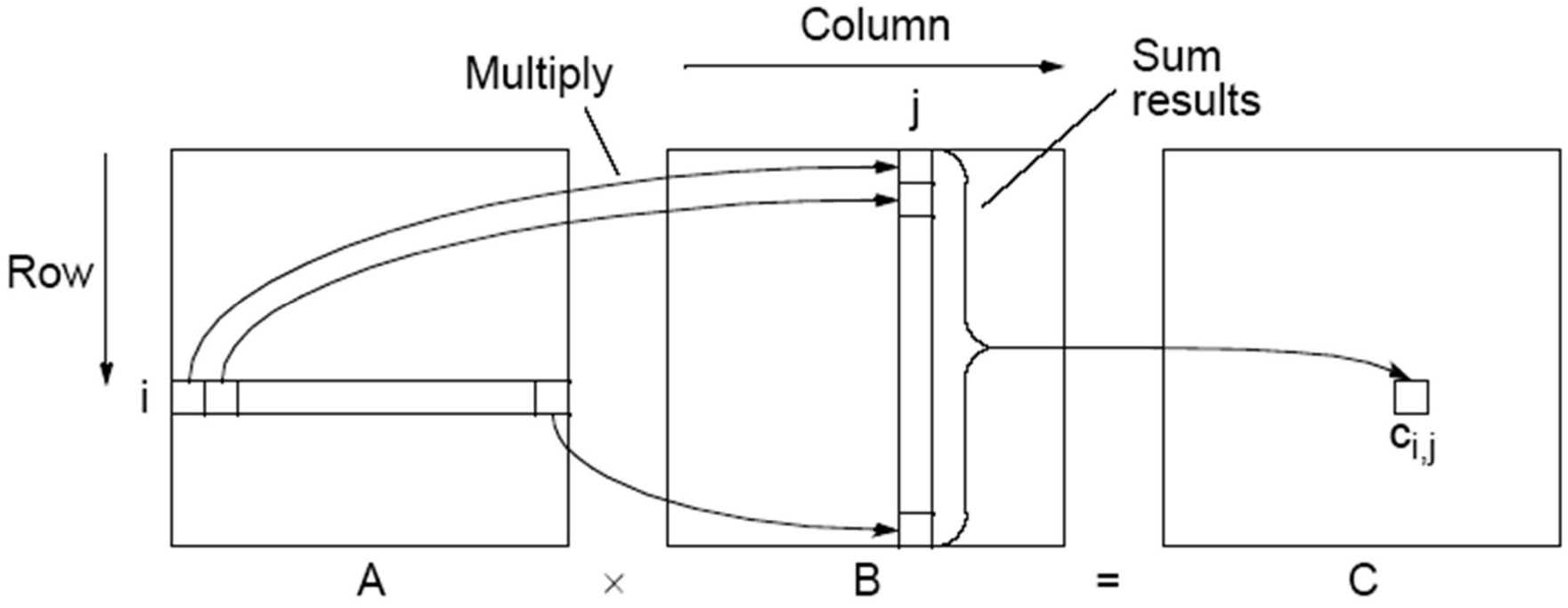
Matrix Multiplication

Multiplication of two matrices, **A** and **B**, produces the matrix **C** whose elements, *ci,j* (0 <= *i* < *n*, 0 <= *j* < *m*), are computed as follows:

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

where **A** is an *n* x *l* matrix and **B** is an *l* x *m* matrix.

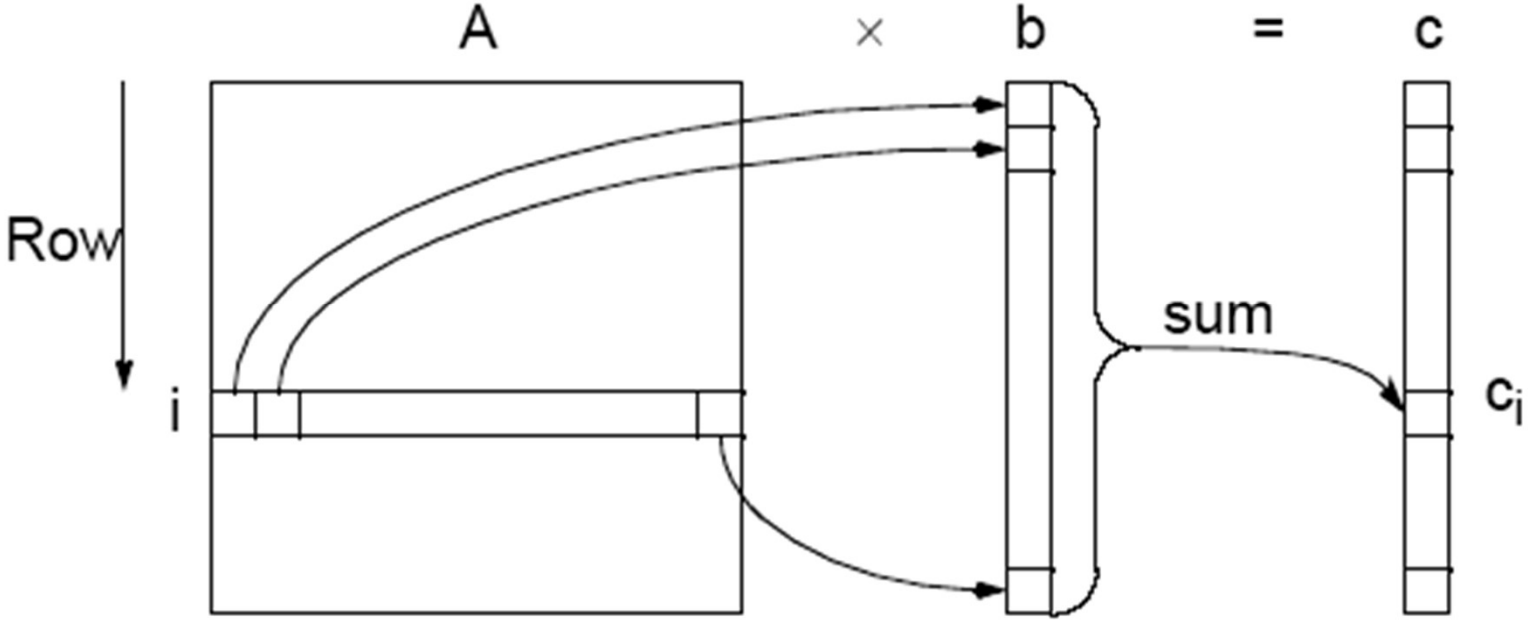
Matrix multiplication, C = A x B



Matrix-Vector Multiplication

c = A x b

Matrix-vector multiplication follows directly from the definition of matrix-matrix multiplication by making **B** an *l* x1 matrix (vector). Result an *n* x 1 matrix (vector).



Relationship of Matrices to Linear Equations

A system of linear equations can be written in matrix form:

$$\mathbf{Ax} = \mathbf{b}$$

Matrix **A** holds *a* constants

x is a vector of the unknowns

b is a vector of the *b* constants.

Implementing Matrix Multiplication Sequential Code

Assume throughout that the matrices are square (n x n matrices). The sequential code to compute **A** x **B** could simply be

```
for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
                c[i][j] = 0;
                for (k = 0; k < n; k++)
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

This algorithm requires n^3 multiplications and n^3 additions, leading to a sequential time complexity of $O(n^3)$.

Very easy to parallelize.

Parallel Code

With n processors (and n x n matrices), can obtain:

- Time complexity of $O(n^2)$ with n processors
  Each instance of inner loop independent and can be done by a separate processor

- Time complexity of $O(n)$ with n^2 processors
  One element of A and B assigned to each processor.
  Cost optimal since $O(n^3) = n \times O(n^2) = n^2 \times O(n)$].

- Time complexity of $O(\log n)$ with n^3 processors
  By parallelizing the inner loop. Not cost-optimal since $O(n^3) \neq n^3 \times O(\log n)$).

$O(\log n)$ lower bound for parallel matrix multiplication.

Partitioning into Submatrices

Suppose matrix divided into s^2 submatrices. Each submatrix has n/s x n/s elements. Using notation Ap,q as submatrix in submatrix row p and submatrix column q:

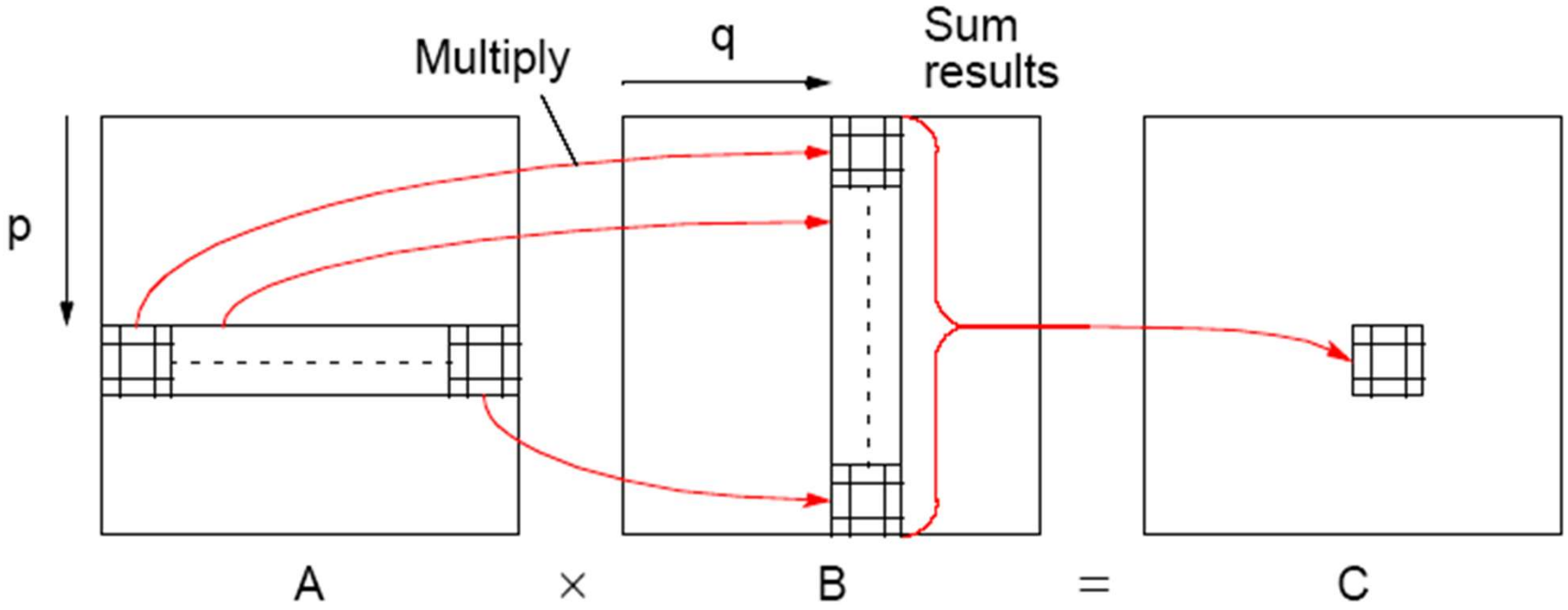
```
for (p = 0; p < s; p++)
  for (q = 0; q < s; q++) {
    Cp,q = 0;                          /* clear elements of submatrix */
    for (r = 0; r < m; r++)            /* submatrix multiplication &*/
      Cp,q = Cp,q + Ap,r * Br,q;       /*add to accum. submatrix*/
  }
```

The line

$$C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q};$$

means multiply submatrix Ap,r and Br,q using matrix multiplication and add to submatrix Cp,q using matrix addition. Known as *block matrix multiplication*.

Block Matrix Multiplication



Submatrix multiplication

(a) Matrices

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

(b) Multiplying $A_{0,0} \times B_{0,0}$ to obtain $C_{0,0}$

$$\overset{A_{0,0}}{\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}} \times \overset{B_{0,0}}{\begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix}} + \overset{A_{0,1}}{\begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix}} \times \overset{B_{1,0}}{\begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}}$$

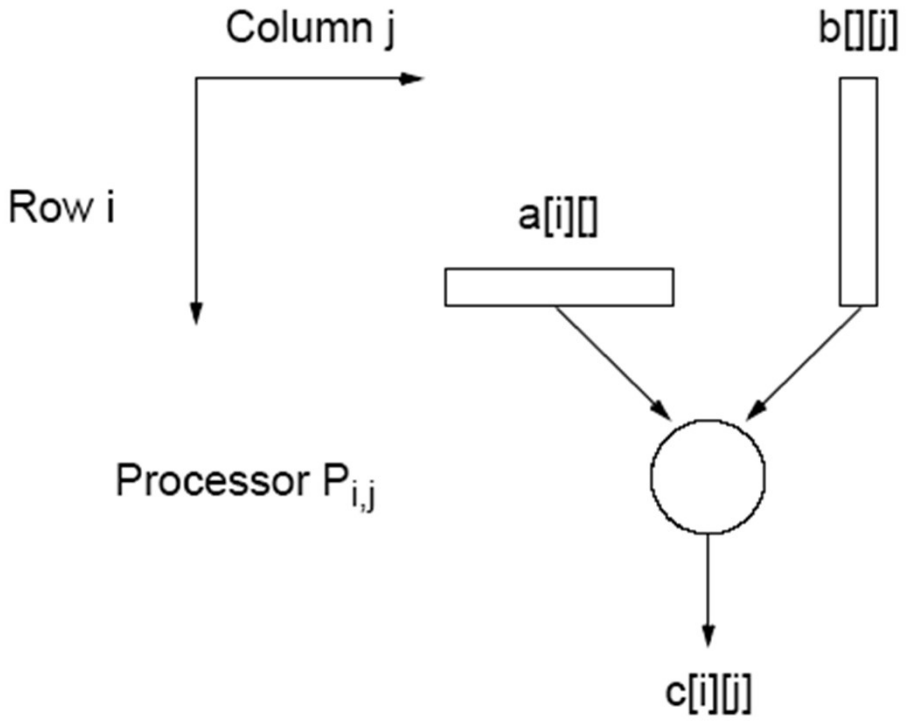
$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

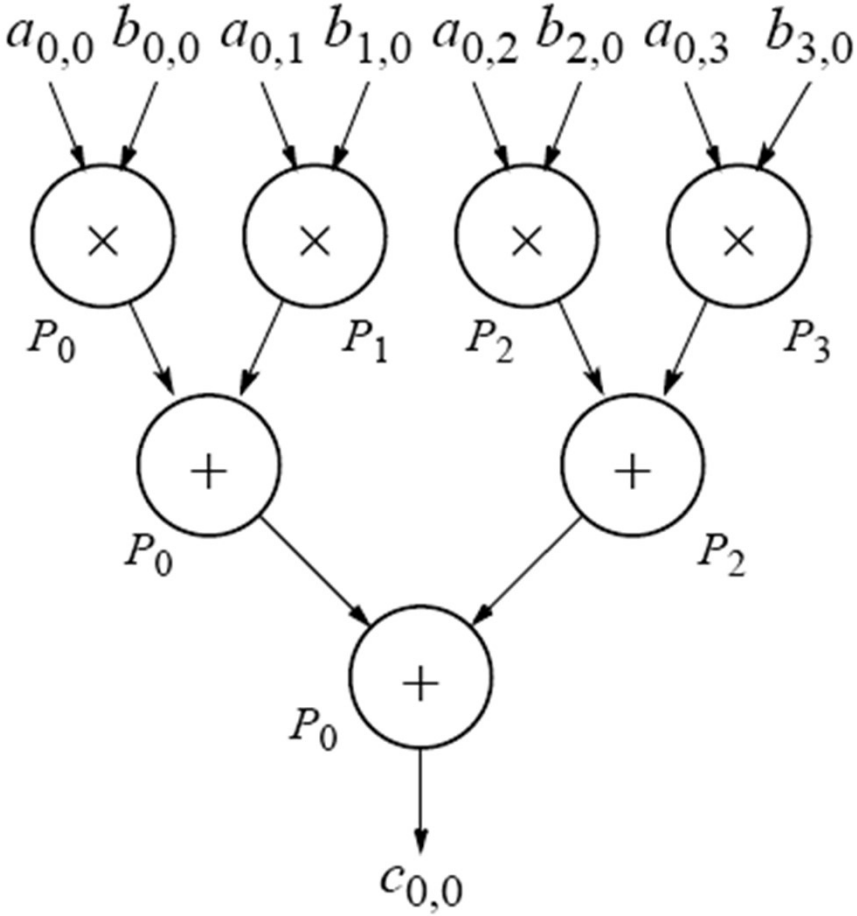
Direct Implementation

One processor to compute each element of **C** – n^2 processors would be needed. One row of elements of **A** and one column of elements of **B** needed. Some of same elements sent to more than one processor. Can use submatrices.



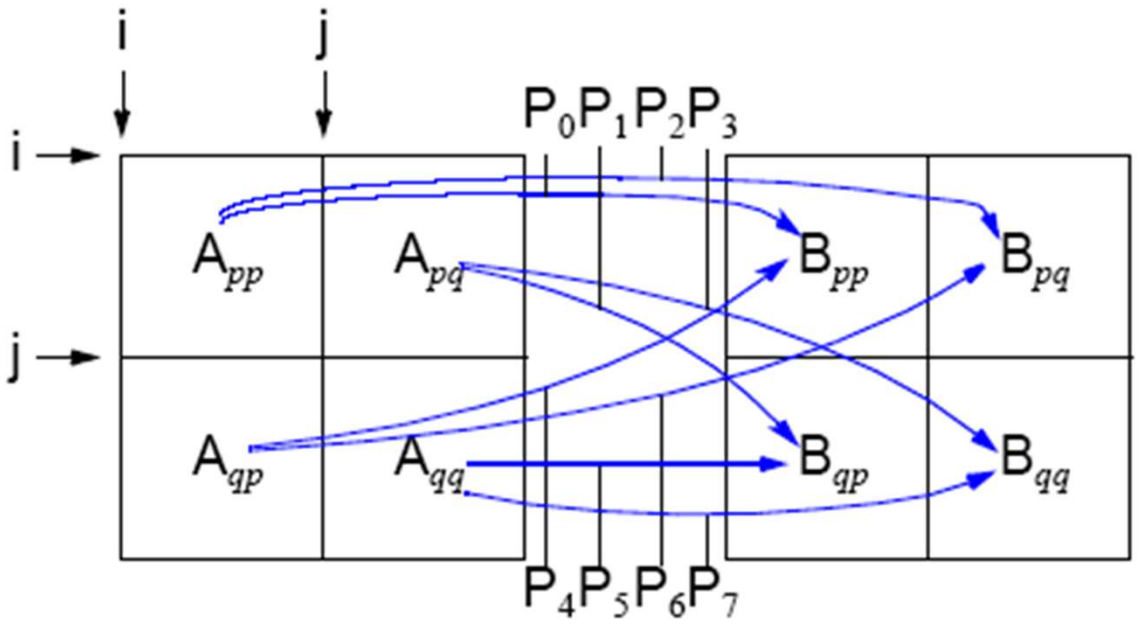
Performance Improvement

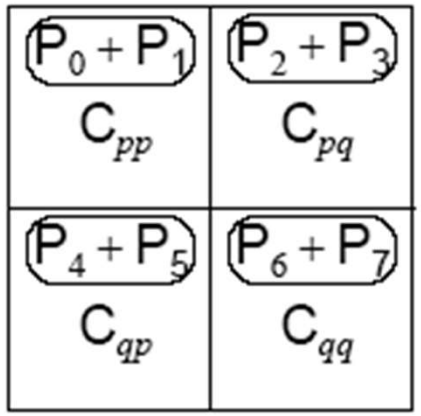
Using tree construction n numbers can be added in log n steps using n processors:



Computational time complexity of O(log n) using n^3 processors.

Recursive Implementation





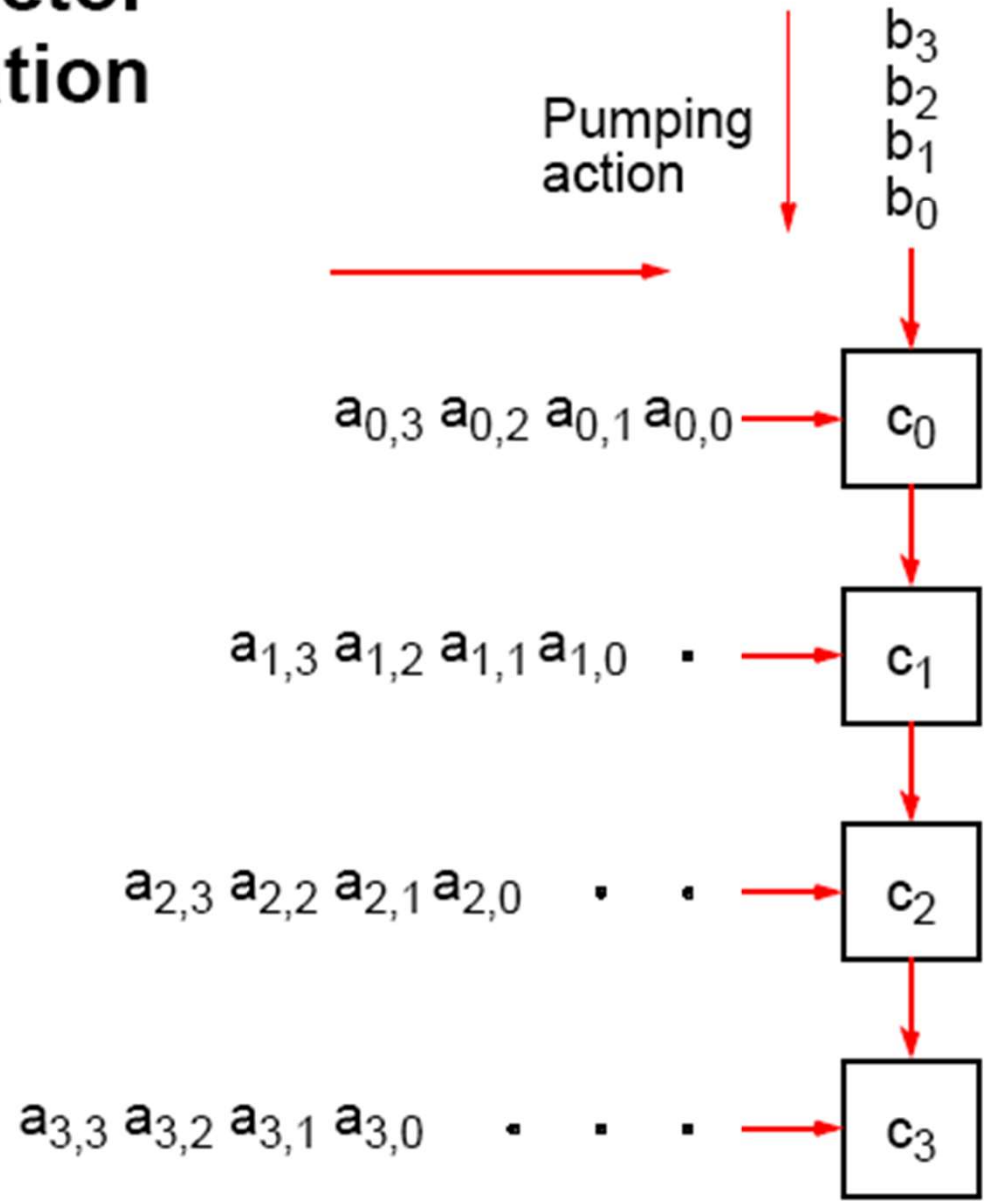
Apply same algorithm on each submatrix recursivly.

Excellent algorithm for a shared memory systems because of locality of operations.

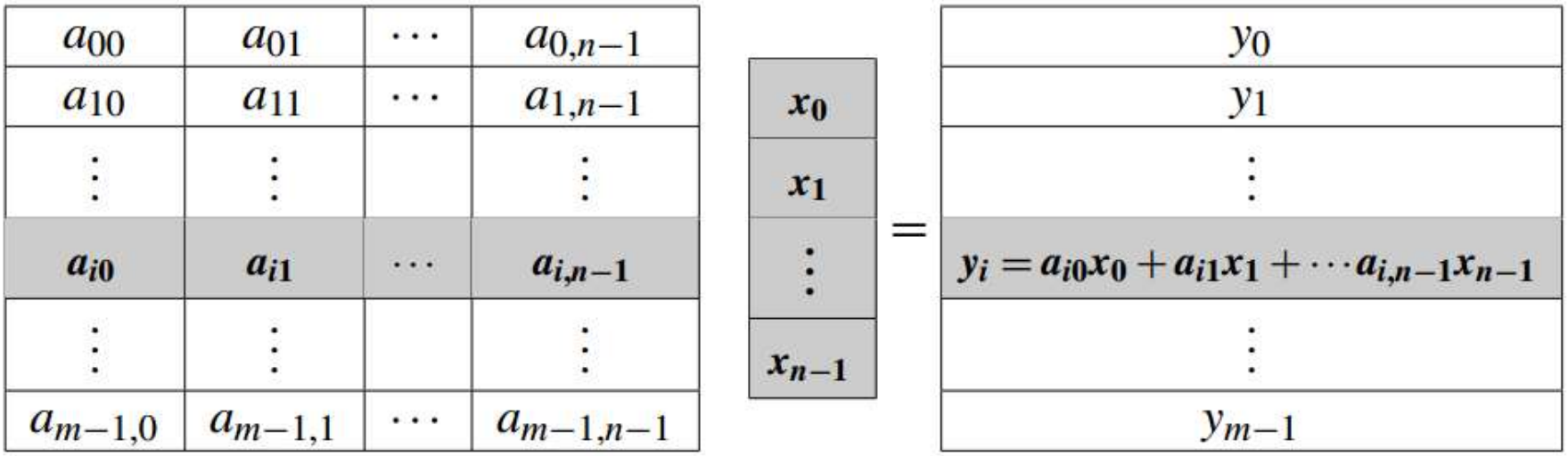
Recursive Algorithm

```
mat_mult(A_pp, B_pp, s)
{
if (s == 1)                 /* if submatrix has one element */
    C = A * B;              /* multiply elements */
else {                      /* continue to make recursive calls */
    s = s/2;                /* no of elements in each row/column */
    P0 = mat_mult(A_pp, B_pp, s);
    P1 = mat_mult(A_pq, B_qp, s);
    P2 = mat_mult(A_pp, B_pq, s);
    P3 = mat_mult(A_pq, B_qq, s);
    P4 = mat_mult(A_qp, B_pp, s);
    P5 = mat_mult(A_qq, B_qp, s);
    P6 = mat_mult(A_qp, B_pq, s);
    P7 = mat_mult(A_qq, B_qq, s);
    C_pp = P0 + P1;         /* add submatrix products to */
    C_pq = P2 + P3;         /* form submatrices of final matrix */
    C_qp = P4 + P5;
    C_qq = P6 + P7;
}
return (C);                 /* return final matrix */
}
```

Matrix-Vector Multiplication



- Matrix-vector multiplication



- Serial matrix-vector multiplication

```
1   void Mat_vect_mult(
2           double  A[]  /* in  */,
3           double  x[]  /* in  */,
4           double  y[]  /* out */,
5           int     m    /* in  */,
6           int     n    /* in  */) {
7       int i, j;
8
9       for (i = 0; i < m; i++) {
10          y[i] = 0.0;
11          for (j = 0; j < n; j++)
12              y[i] += A[i*n+j]*x[j];
13      }
14  }   /* Mat_vect_mult */
```

Parallel matrix-vector multiplication

```
int MPI_Allgather(
      void*          send_buf_p   /* in */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```

```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```

Solving a System of Linear Equations

$$
\begin{aligned}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad &\cdots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1} \\
&\vdots \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \quad &\cdots \quad + a_{2,n-1}x_{n-1} = b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \quad &\cdots \quad + a_{1,n-1}x_{n-1} = b_1 \\
a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \quad &\cdots \quad + a_{0,n-1}x_{n-1} = b_0
\end{aligned}
$$

which, in matrix form, is

$$\mathbf{Ax} = \mathbf{b}$$

Objective is to find values for the unknowns, $x0$, $x1$, …, xn-1, given values for $a0,0$, $a0,1$, …, an-1,n-1, and $b0$, …, bn .

Solving a System of Linear Equations

Dense matrices

Gaussian Elimination - parallel time complexity $O(n^2)$

Sparse matrices

By iteration - depends upon iteration method and number of iterations but typically O(log n)

- Jacobi iteration
- Gauss-Seidel relaxation (not good for parallelization)
- Red-Black ordering
- Multigrid

Gaussian Elimination

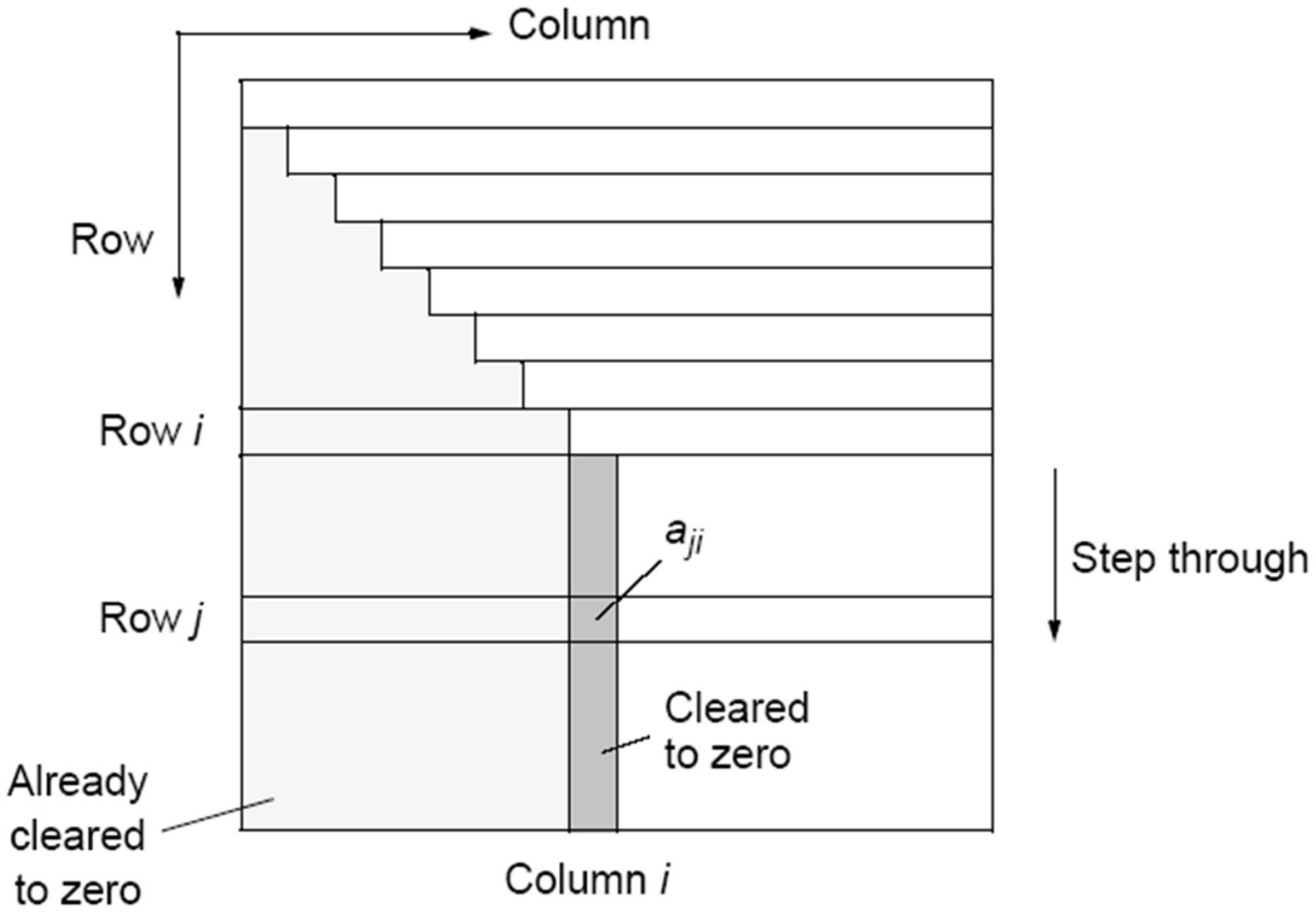
Convert general system of linear equations into triangular system of equations. Then be solved by Back Substitution.

Uses characteristic of linear equations that any row can be replaced by that row added to another row multiplied by a constant.

Starts at the first row and works toward the bottom row. At the *i*th row, each row *j* below the *i*th row is replaced by row *j* + (row *i*) ($-a_{j,i}/a_{i,i}$). The constant used for row *j* is $-a_{j,i}/a_{i,i}$. Has the effect of making all the elements in the *i* th column below the *i* th row zero because

$$a_{j,i} = a_{j,i} + a_{i,i}\left(\frac{-a_{j,i}}{a_{i,i}}\right) = 0$$

Gaussian elimination



Partial Pivoting

If $a_{i,i}$ is zero or close to zero, we will not be able to compute the quantity $-a_{j,i}/a_{i,i}$.

Procedure must be modified into so-called *partial pivoting* by swapping the *i* th row with the row below it that has the largest absolute element in the *i* th column of any of the rows below the *i* th row if there is one. (Reordering equations will not affect the system.)

In the following, we will not consider partial pivoting.

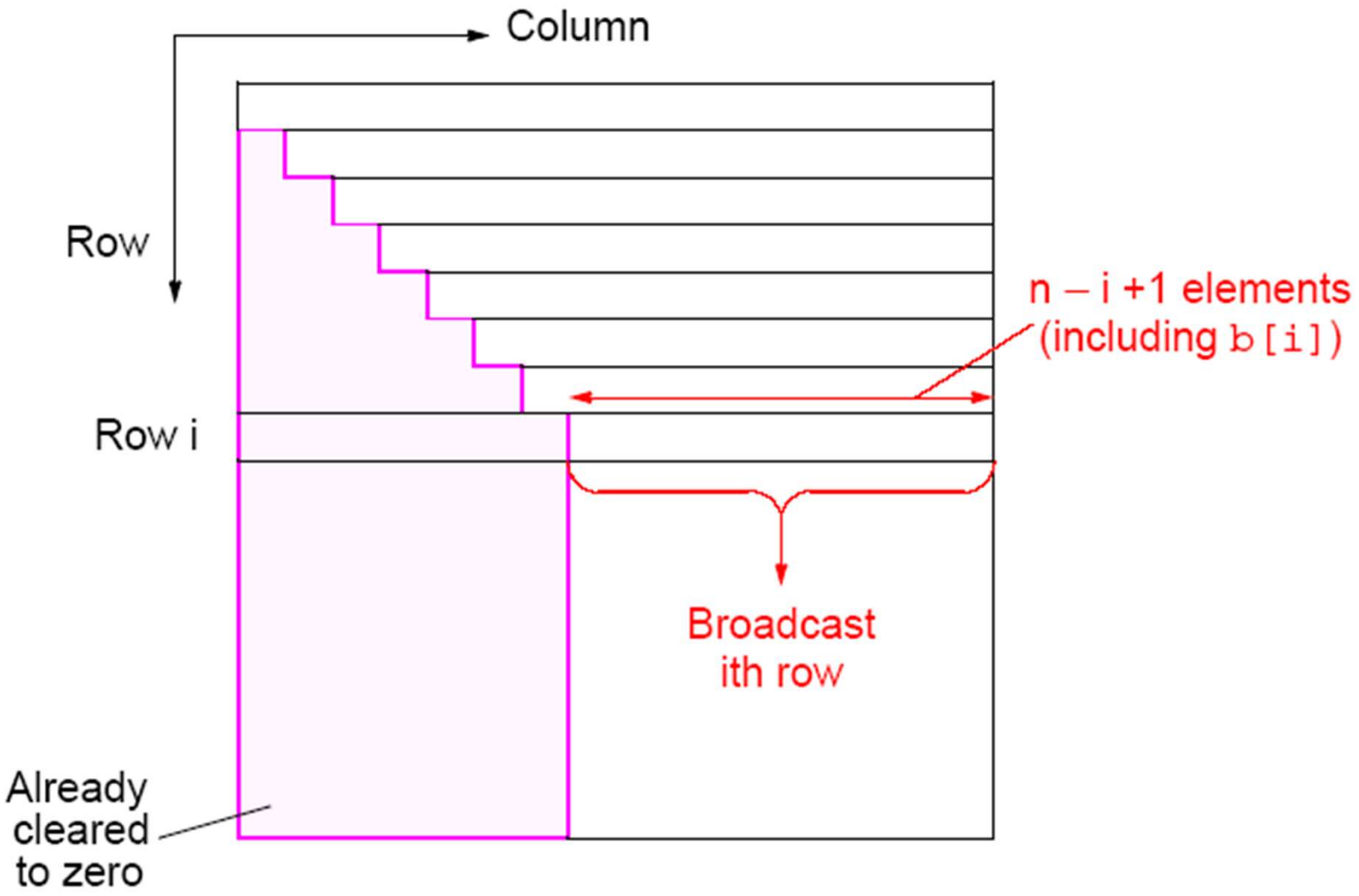
Sequential Code

Without partial pivoting:

```
    for (i = 0; i < n-1; i++)              /* for each row, except last */
        for (j = i+1; j < n; j++) {     /*step thro subsequent rows */
            m = a[j][i]/a[i][i];          /* Compute multiplier */
            for (k = i; k < n; k++)      /*last n-i-1 elements of row j*/
              a[j][k] = a[j][k] - a[i][k] * m;
            b[j] = b[j] - b[i] * m;          /* modify right side */
}
```

The time complexity is O(*n*3).

Parallel Implementation



Case Study-Linpack Benchmark

2021 ACM A.M. Turing Award Laureate

Prof. Jack J. Dongarra

the University of Tennessee and the Oak Ridge National Laboratory

- As a yardstick of performance we are using the `best' performance as measured by the LINPACK Benchmark. LINPACK was chosen because it is widely used and performance numbers are available for almost all relevant systems.
- The LINPACK Benchmark was introduced by Jack Dongarra.

ACM named Jack J. Dongarra recipient of the 2021 ACM A.M. Turing Award for pioneering contributions to numerical algorithms and libraries that enabled high performance computational software to keep pace with exponential hardware improvements for over four decades.

- A parallel implementation of the Linpack benchmark and instructions on how to run it can be found at http://www.netlib.org/benchmark/hpl/.

HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers

Innovative Computing Laboratory
UNIVERSITY OF TENNESSEE
COMPUTER SCIENCE DEPARTMENT

Version 2.3 | A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary | December 2, 2018

https://netlib.org/benchmark/hpl/

- **HPL** is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers. It can thus be regarded as a portable as well as freely available implementation of the High Performance Computing Linpack Benchmark.
- The **algorithm** used by HPL can be summarized by the following keywords:

- Two-dimensional block-cyclic data distribution
- Right-looking variant of the LU (Lower Upper) factorization with row partial pivoting featuring multiple look-ahead depths
- Recursive panel factorization with pivot search and column broadcast combined
- Various virtual panel broadcast topologies
- bandwidth reducing swap-broadcast algorithm
- backward substitution with look-ahead of depth 1

- The HPL package provides a testing and timing program to quantify the **accuracy** of the obtained solution as well as the time it took to compute it.
- The best **performance** achievable by this software on your system depends on a large variety of factors. Nonetheless, with some restrictive assumptions on the interconnection network, the algorithm and its attached implementation are **scalable** in the sense that their parallel efficiency is maintained constant with respect to the per processor memory usage.

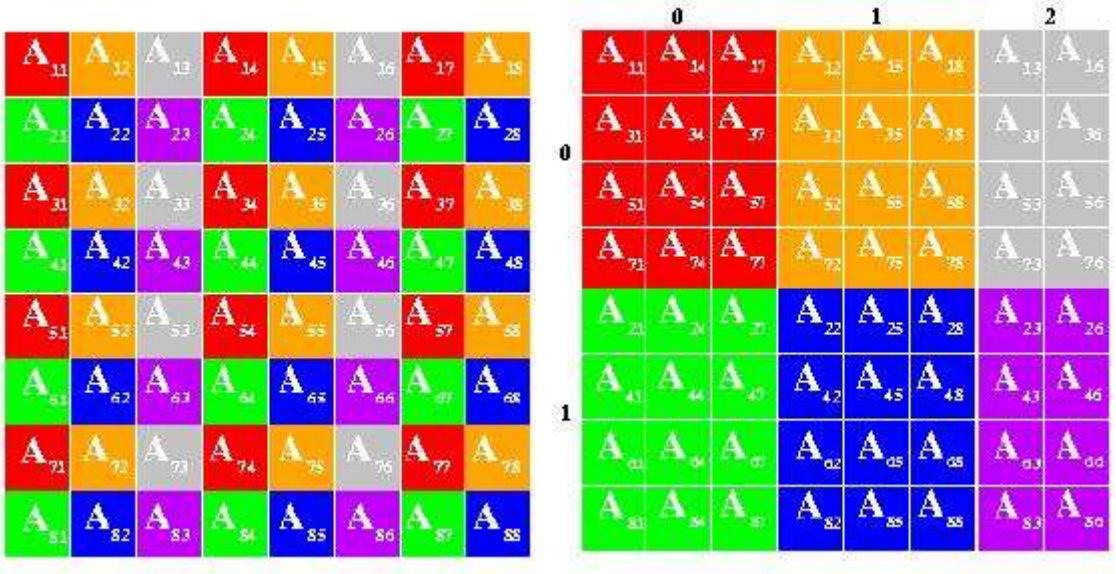
- The HPL software package **requires** the availability on your system of an implementation of the Message Passing Interface **MPI** (1.1 compliant). An implementation of **either** the Basic Linear Algebra Subprograms **BLAS or** the Vector Signal Image Processing Library **VSIPL** is also needed.

- Machine-specific as well as generic implementations of MPI, the BLAS and VSIPL are available for a large variety of systems.

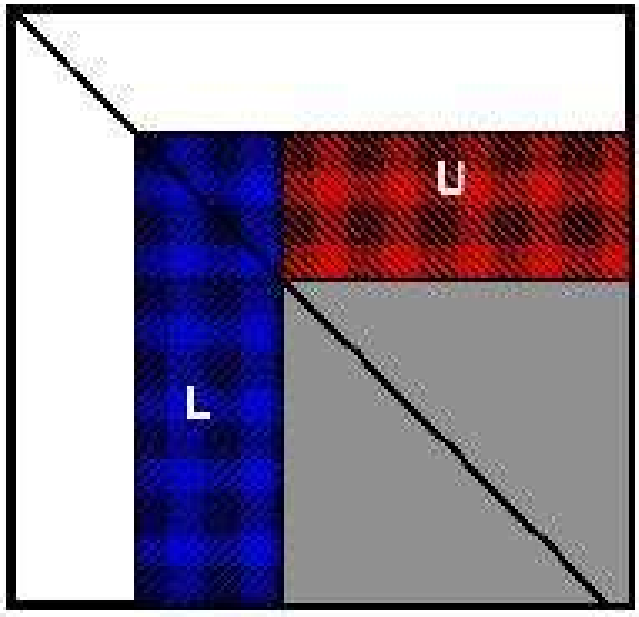
Main Algorithm of HPL

- This software package solves a linear system of order n:

  A x = b by first computing the LU factorization with row partial pivoting of the n-by-n+1 coefficient matrix [A b] = [[L,U] y]. Since the lower triangular factor L is applied to b as the factorization progresses, the solution x is obtained by solving the upper triangular system U x = y. The lower triangular matrix L is left unpivoted and the array of pivots is not returned.



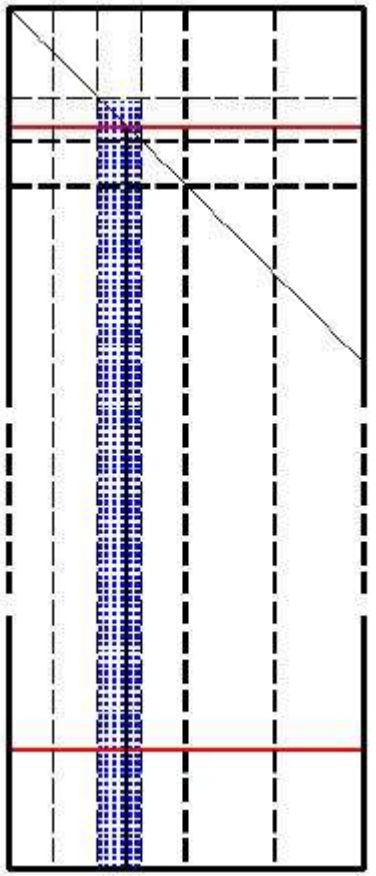
- The data is distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure "good" load balance as well as the scalability of the algorithm. The n-by-n+1 coefficient matrix is first logically partitioned into nb-by-nb blocks, that are cyclically "dealt" onto the P-by-Q process grid. This is done in both dimensions of the matrix.



- The right-looking variant has been chosen for the main loop of the LU factorization. This means that at each iteration of the loop a panel of nb columns is factorized, and the trailing submatrix is updated. Note that this computation is thus logically partitioned with the same block size nb that was used for the data distribution.

Panel Factorization

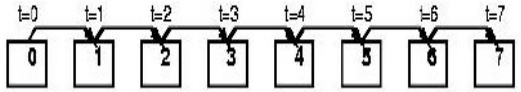
- At a given iteration of the main loop, and because of the cartesian property of the distribution scheme, each panel factorization occurs in one column of processes. This particular part of the computation lies on the critical path of the overall algorithm. The user is offered the choice of three (Crout, left- and right-looking) matrix-multiply based recursive variants.
- The software also allows the user to choose in how many sub-panels the current panel should be divided into during the recursion.
- Furthermore, one can also select at run-time the recursion stopping criterium in terms of the number of columns left to factorize. When this threshold is reached, the sub-panel will then be factorized using one of the three Crout, left- or right-looking matrix-vector based variant.
- Finally, for each panel column the pivot search, the associated swap and broadcast operation of the pivot row are combined into one single communication step. A binary-exchange (leave-on-all) reduction performs these three operations at once.



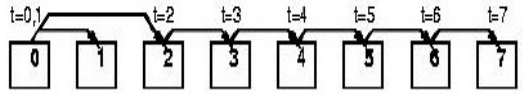
Panel Broadcast

- Once the panel factorization has been computed, this panel of columns is broadcast to the other process columns. There are many possible broadcast algorithms and the software currently offers 6 variants to choose from. These variants are described below assuming that process 0 is the source of the broadcast for convenience. "->" means "sends to".

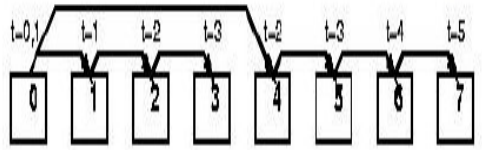
- **Increasing-ring**: 0 -> 1; 1 -> 2; 2 -> 3 and so on. This algorithm is the classic one; it has the caveat that process 1 has to send a message.



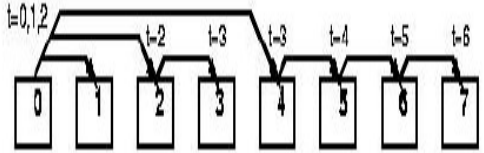
- **Increasing-ring (modified)**: 0 -> 1; 0 -> 2; 2 -> 3 and so on. Process 0 sends two messages and process 1 only receives one message. This algorithm is almost always better, if not the best.



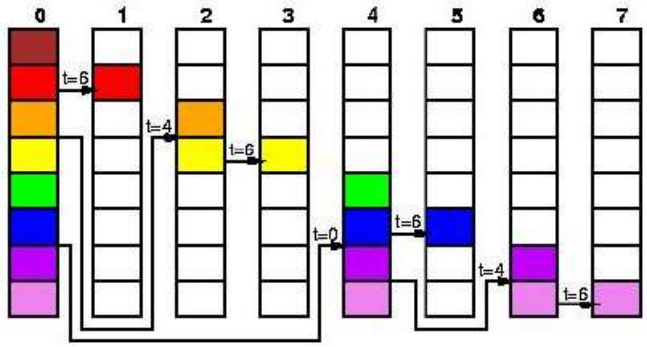
- **Increasing-2-ring**: The Q processes are divided into two parts: 0 -> 1 and 0 -> Q/2; Then processes 1 and Q/2 act as sources of two rings: 1 -> 2, Q/2 -> Q/2+1; 2 -> 3, Q/2+1 -> to Q/2+2 and so on. This algorithm has the advantage of reducing the time by which the last process will receive the panel at the cost of process 0 sending 2 messages.



- **Increasing-2-ring (modified)**: As one may expect, first 0 -> 1, then the Q-1 processes left are divided into two equal parts: 0 -> 2 and 0 -> Q/2; Processes 2 and Q/2 act then as sources of two rings: 2 -> 3, Q/2 -> Q/2+1; 3 -> 4, Q/2+1 -> to Q/2+2 and so on. This algorithm is probably the most serious competitor to the increasing ring modified variant.



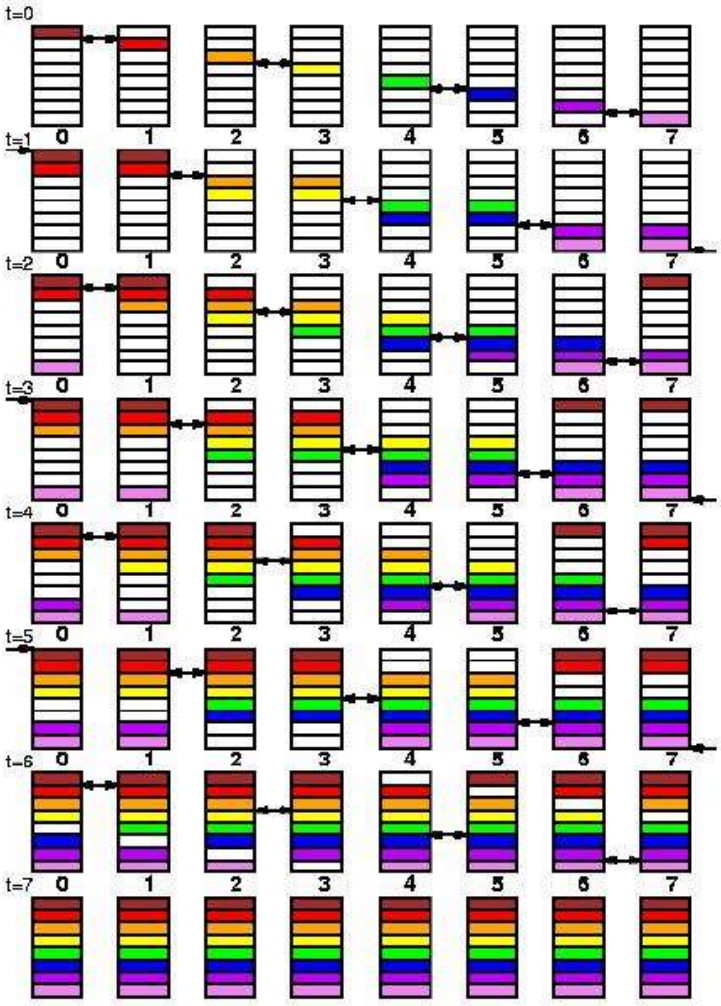
- **Long (bandwidth reducing)**: as opposed to the previous variants, this algorithm and its follower synchronize all processes involved in the operation. The message is chopped into Q equal pieces that are scattered across the Q processes.



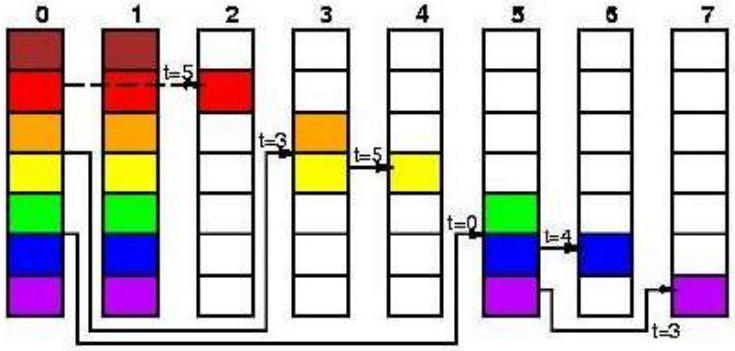
- The pieces are then rolled in Q-1 steps.

The scatter phase uses a binary tree and the rolling phase exclusively uses mutual message exchanges.

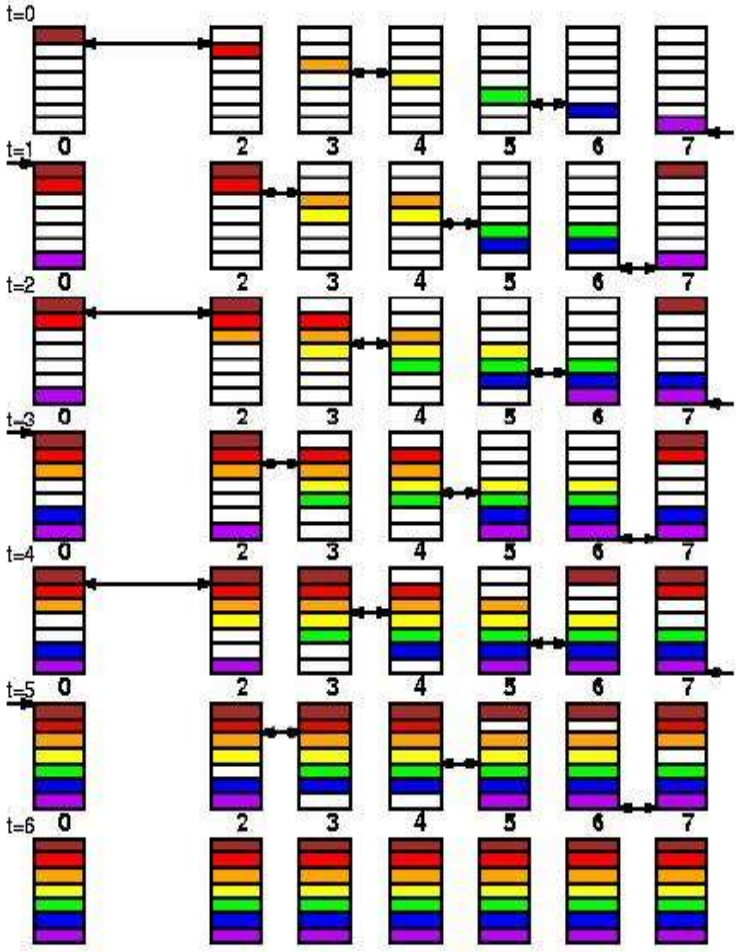
In odd steps 0 <-> 1, 2 <-> 3, 4 <-> 5 and so on; in even steps Q-1 <-> 0, 1 <-> 2, 3 <-> 4, 5 <-> 6 and so on.



- **Long (bandwidth reducing modified)**: same as above, except that 0 -> 1 first, and then the Long variant is used on processes 0,2,3,4 .. Q-1.



- The rings variants are distinguished by a probe mechanism that activates them. In other words, a process involved in the broadcast and different from the source asynchronously probes for the message to receive. When the message is available the broadcast proceeds, and otherwise the function returns. This allows to interleave the broadcast operation with the update phase. This contributes to reduce the idle time spent by those processes waiting for the factorized panel. This mechanism is necessary to accomodate for various computation/communication performance ratio.



Look-ahead

- Once the panel has been broadcast or say during this broadcast operation, the trailing submatrix is updated using the last panel in the look-ahead pipe: as mentioned before, the panel factorization lies on the critical path, which means that when the kth panel has been factorized and then broadcast, the next most urgent task to complete is the factorization and broadcast of the k+1 th panel.
- This technique is often refered to as "look-ahead" or "send-ahead" in the literature. This package allows to select various "depth" of look-ahead. By convention, a depth of zero corresponds to no lookahead, in which case the trailing submatrix is updated by the panel currently broadcast.
- Look-ahead consumes some extra memory to essentially keep all the panels of columns currently in the look-ahead pipe. A look-ahead of depth 1 (maybe 2) is likely to achieve the best performance gain.

Update

- The update of the trailing submatrix by the last panel in the look-ahead pipe is made of two phases.
  - First, the pivots must be applied to form the current row panel U. U should then be solved by the upper triangle of the column panel.
  - U finally needs to be broadcast to each process row so that the local rank-nb update can take place. We choose to combine the swapping and broadcast of U at the cost of replicating the solve.
- Two algorithms are available for this communication
  - Binary-exchange
  - Long

Backward Substitution

- The factorization has just now ended, the back-substitution remains to be done. For this, we choose a look-ahead of depth one variant. The right-hand-side is forwarded in process rows in a decreasing-ring fashion, so that we solve Q * nb entries at a time.
- At each step, this shrinking piece of the right-hand-side is updated. The process just above the one owning the current diagonal block of the matrix A updates first its last nb piece of x, forwards it to the previous process column, then broadcast it in the process column in a decreasing-ring fashion as well. The solution is then updated and sent to the previous process column. The solution of the linear system is left replicated in every process row.

Checking the Solution

- To verify the result obtained, the input matrix and right-hand side are regenerated. The normwise backward error (see formula below) is then computed. A solution is considered as "numerically correct" when this quantity is less than a threshold value of the order of 1.0. In the expression below, eps is the relative (distributed-memory) machine precision.

  - $\| Ax - b \|_{oo} / (eps * (\| A \|_{oo} * \| x \|_{oo} + \| b \|_{oo}) * n)$

HPL Performance Results

- **4 AMD Athlon K7 500 Mhz (256 Mb) - (2x) 100 Mbs Switched - 2 NICs per node (channel bonding)**

OS	Linux 6.2 RedHat (Kernel 2.2.14)
C compiler	gcc (egcs-2.91.66 egcs-1.1.2 release)
C flags	-fomit-frame-pointer -O3 -funroll-loops
MPI	MPIch 1.2.1
BLAS	ATLAS (Version 3.0 beta)
Comments	09 / 00

Performance (Gflops) w.r.t Problem size on 4 nodes.

GRID	**2000**	**5000**	**8000**	**10000**
1 x 4	1.28	1.73	1.89	1.95
2 x 2	1.17	1.68	1.88	1.93
4 x 1	0.81	1.43	1.70	1.80

8 Duals Intel PIII 550 Mhz (512 Mb) - Myrinet

OS	Linux 6.1 RedHat (Kernel 2.2.15)
C compiler	gcc (egcs-2.91.66 egcs-1.1.2 release)
C flags	-fomit-frame-pointer -O3 -funroll-loops
MPI	MPI GM (Version 1.2.3)
BLAS	ATLAS (Version 3.0 beta)
Comments	UTK / ICL - Torc cluster - 09 / 00

Performance (Gflops) w.r.t Problem size on 8- and 16-processors grids.

GRID	**2000**	**5000**	**8000**	**10000**	**15000**	**20000**
2 x 4	1.76	2.32	2.51	2.58	2.72	2.73
4 x 4	2.27	3.94	4.46	4.68	5.00	5.16

Compaq 64 nodes (4 ev67 667 Mhz processors per node) AlphaServer SC

OS	Tru64 Version 5
C compiler	cc Version 6.1
C flags	-arch host -tune host -std -O5
MPI	-lmpi -lelan
BLAS	CXML
Comments	ORNL / NCCS - falcon - 09 / 00

- In the table below, each row corresponds to a given number of cpus (or processors) and nodes. The first row for example is denoted by 1 / 1, i.e., 1 cpu / 1 node. Rmax is given in Gflops, and the value of Nmax in fact corresponds to 351 Mb per cpu for all machine configurations.

CPUS / NODES	GRID	N 1/2	Nmax	Rmax (Gflops)	Parallel Efficiency
1 / 1	**1 x 1**	150	6625	1.136	1.000
4 / 1	**2 x 2**	800	13250	4.360	0.960
16 / 4	**4 x 4**	2300	26500	17.00	0.935
64 / 16	**8 x 8**	5700	53000	67.50	0.928
256 / 64	**16 x 16**	14000	106000	263.6	0.906

Case Study-High Performance Conjugate Gradients **(HPCG) Benchmark**

https://hpcg-benchmark.org/software/index.html

overview

- The High Performance Conjugate Gradients (HPCG) Benchmark project is an effort to create a new metric for ranking HPC systems. HPCG is intended as a complement to the High Performance LINPACK (HPL) benchmark, currently used to rank the TOP500 computing systems. The computational and data access patterns of HPL are still representative of some important scalable applications, but not all.
- HPCG generates and uses sparse data structures that have a very low compute-to-data-movement ratio, especially compared to HPL.
- HPCG is designed to exercise computational and data access patterns that more closely match a different and broad set of important applications, and to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications.

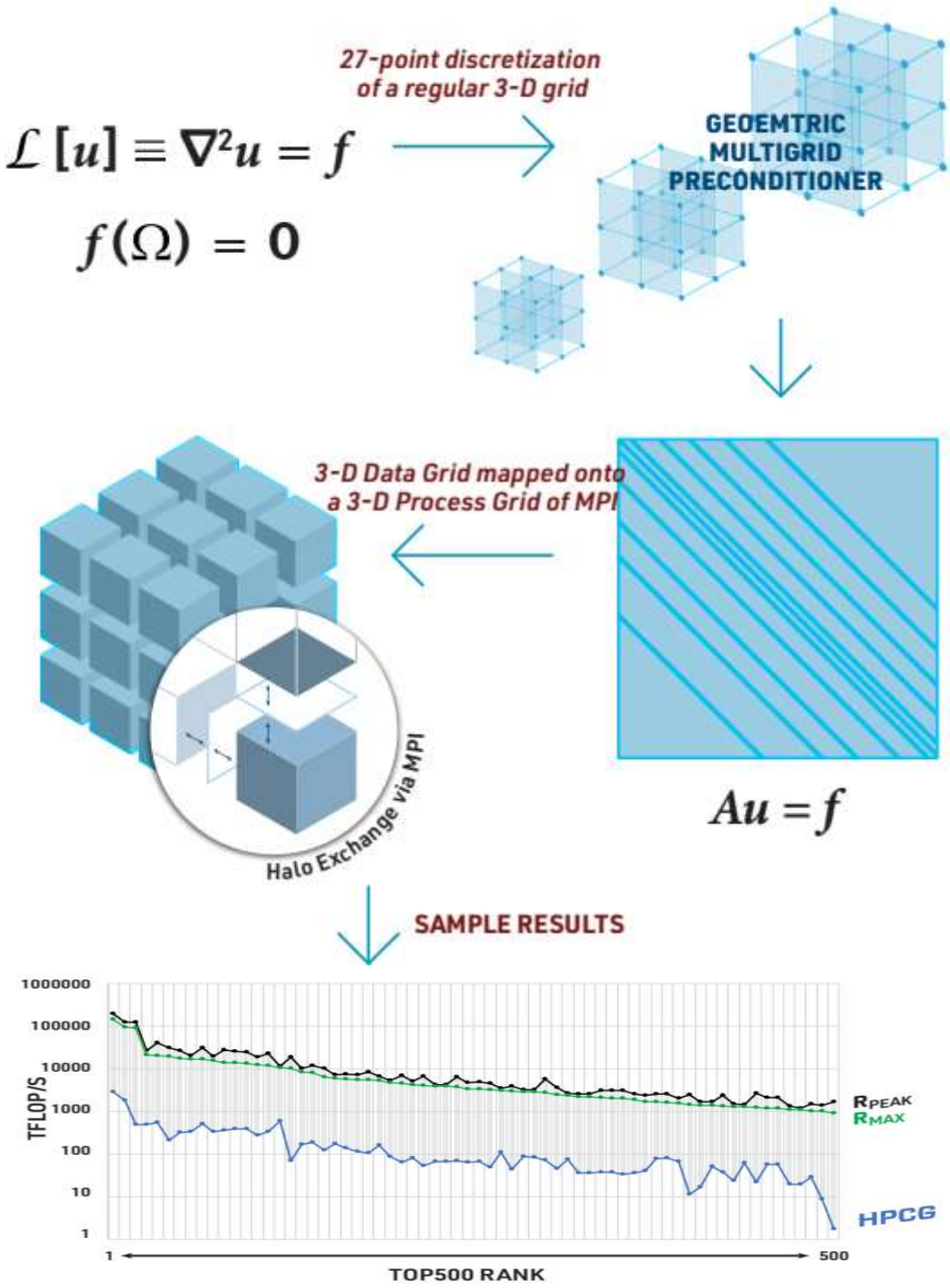
NOVEMBER 2019 HPCG TOP 5		SITE	COUNTRY	RMAX PFLOP/S	HPCG PFLOP/S
1	**Summit**	DOE/SC/ORNL	USA	***143.5***	***2.93***
2	**Sierra**	DOE/NNSA/LLNL	USA	***94.6***	***1.80***
3	**Trinity**	DOE/NNSA/LANL/SNL	USA	***20.2***	***0.54***
4	**ACBI**	AIST	Japan	***16.9***	***0.51***
5	**Piz Daint**	CSCS	Switzerland	***21.2***	***0.50***

HPCG- JUNE 2022

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	HPCG (TFlop/s)
1	2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, **Fujitsu** RIKEN Center for Computational Science Japan	7,630,848	442.01	16004.50
2	4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, **IBM** DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	2925.75
3	3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** EuroHPC/CSC Finland	1,110,144	151.90	1935.73
4	7	**Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, **HPE** DOE/SC/LBNL/NERSC United States	761,856	70.87	1905.44
5	5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, **IBM / NVIDIA / Mellanox** DOE/NNSA/LLNL United States	1,572,480	94.64	1795.67

- The HPC Conjugate Gradient (HPCG) benchmark uses a preconditioned conjugate gradient (PCG) algorithm to measure the performance of HPC platforms with respect to frequently observed, yet challenging, patterns of execution, memory access, and global communication

The PCG implementation uses a regular 27-point stencil discretization in 3 dimensions of an elliptic partial differential equation (PDE) with zero Dirichlet boundary condition. The 3-D domain is scaled to fill a 3-D virtual process grid of all available MPI process ranks. The CG iteration includes a local and symmetric Gauss-Seidel preconditioner, which computes a forward and a back solve with a triangular matrix. All of these features combined allow HPCG to deliver a more accurate performance metric for modern HPC hardware architectures.



PRECONDITIONED CONJUGATE GRADIENT SOLVER

$p_0 \leftarrow x_0, r_0 \leftarrow b - Ap_0$
for $i = 1, 2,$ to max_iterations **do**
  $z_i \leftarrow M^{-1} r_{i-1}$
  if $i = 1$ **then**
    $p_l \leftarrow z_i$
    $\alpha_i \leftarrow \text{dot\_prod}(r_{i-1}, z_i)$
  else
    $\alpha_i \leftarrow \text{dot\_prod}(r_{i-1}, z_i)$
    $\beta_i \leftarrow \alpha_i / \alpha_{i-1}$
    $p_i \leftarrow \beta_i p_{i-1} + z_i$
  end if
  $\alpha_l \leftarrow \text{dot\_prod}(r_{i-1}, z_i) / \text{dot\_prod}(p_i, Ap_i)$
  $x_{i+1} \leftarrow x_i + \alpha_i p_i$
  $r_i \leftarrow r_{i-1} - \alpha_i Ap_i$
  if $\|r_i\|_2 <$ tolerance **then**
    STOP
  end if
end for

- HPCG is a complete, stand-alone code that measures the performance of basic operations in a unified code:
  - Sparse matrix-vector multiplication.
  - Vector updates.
  - Global dot products.
  - Local symmetric Gauss-Seidel smoother.
  - Sparse triangular solve (as part of the Gauss-Seidel smoother).
  - Driven by multigrid preconditioned conjugate gradient algorithm that exercises the key kernels on a nested set of coarse grids.
  - Reference implementation is written in C++ with MPI and OpenMP support.

Sparse Matrix Vector Multiplication (SpMV):

- SpMV is a common and challenging kernel for many applications. It stresses the memory system by using both streaming memory and indexed reads of memory.
- The operation can take advantage of temporal locality (when computing y = Ax, each value of x is typically use about 25 times). It also requires a "neighborhood collective" where each processor communicates with on average 26 neighboring processors.
- Also, optimal implementation of the neighborhood collective will overlap this communication with local computation of the y values.
- Finally, this kernel can be vectorized, either with small vector lengths (order 25) in the reference sparse matrix format used in HPCG, or with long vector lengths (order N), if a different (e.g., ELLPACK) format is used.

Symmetric Gauss-Seidel smoother (SymGS)

- SymGS is a form of sparse triangular solve requiring fine-grain recursive execution that tests the latency optimization capabilities of a processor.
- Also, in the case of a multicore or manycore processor, it presents a challenging target for fine-grain cooperative threading.
- It also has the indexed read challenge of SpMV.

Global Dot Product

- Global collective operations such dot product require the scanning of a large distributed array of values to produce a single value that is available to every processor on the system.
- This operation is so ubiquitous and important that some supercomputers have dedicated special hardware to perform this specific type of operation.
- HPCG requires efficient execution of this operation across all processors on the computer system.

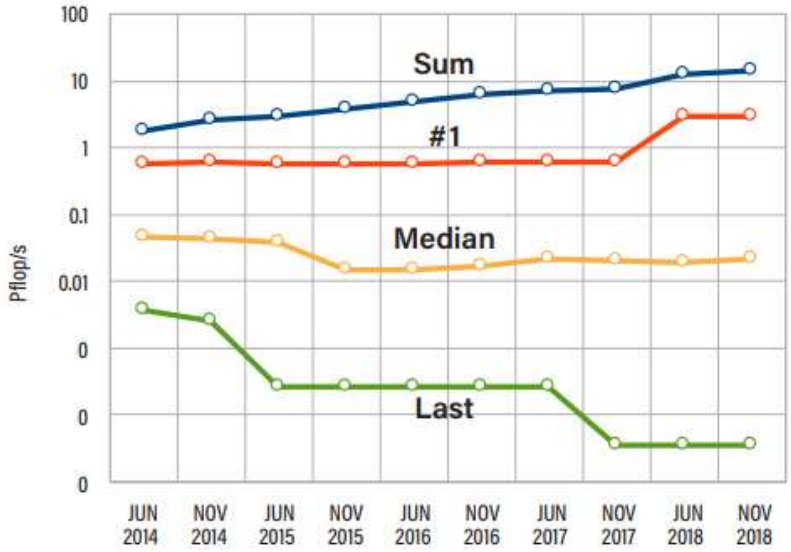
Vector Update

- This is the simplest kernel, which tests the raw streaming bandwidth of the processor. Although it is essentially a STREAMS operation
- it consumes a tiny fraction of the overall execution time in HPCG

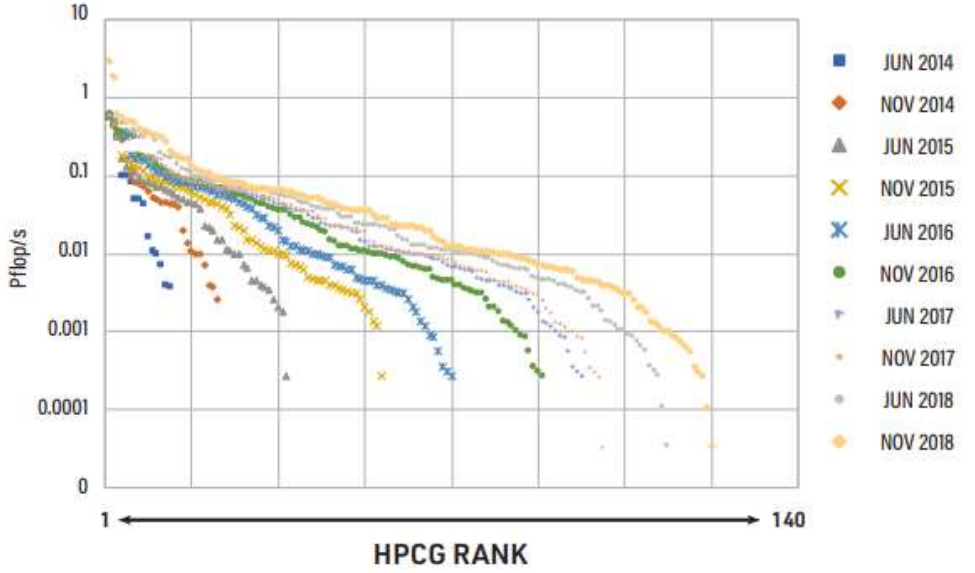
Multigrid preconditioner

- The multigrid preconditioner provides further challenge to the processor by presenting all of the above kernels at 4 different grid sizes that vary in size by a factor of 8 from level to level
- so that the coarsest grid is approximately 4000 times smaller than the original fine grid

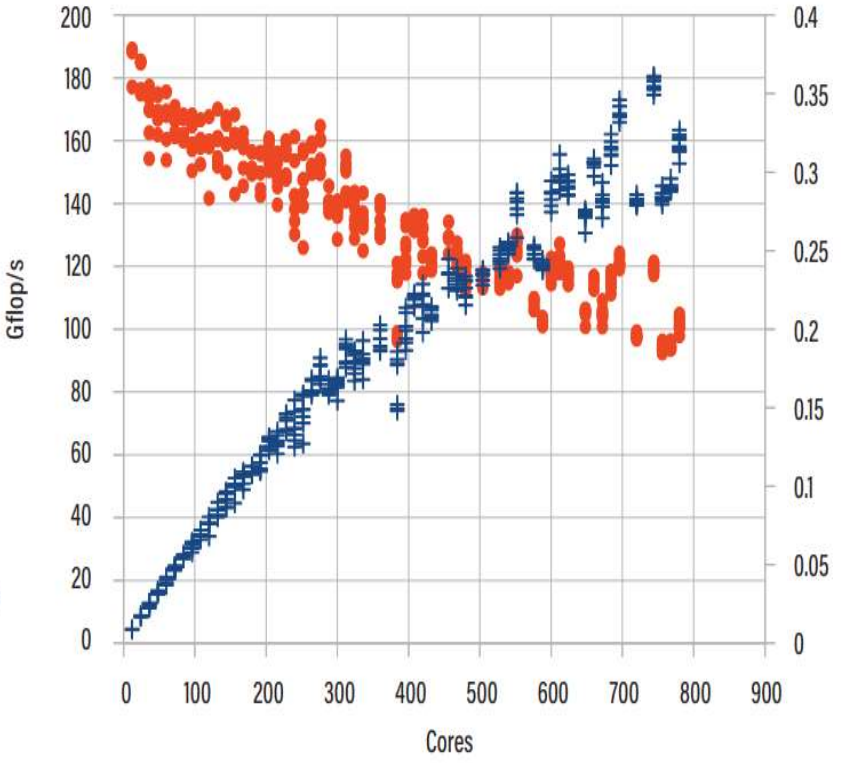
The chart to the right shows various performance statistics from the past HPCG result lists. The sum of performance indicates steady growth of results accumulated on the list. The median and smallest values decrease as HPCG list increases the number of systems on the list, thus expanding the range of recorded values. The #1 entry experienced a large increase in June 2018 with the introduction of Summit, nearly reaching 3 Pflop/s.



The chart to the right shows all performance results from the past HPCG result lists. Different lists are represented by different colors. The number of entries on the lists keeps increasing and reached 130 in 2018.



The chart to the right shows scaling performance results of the reference HPCG code on 780-core cluster with 180 InfiniBand interconnect. The scaling is 160 linear (blue marks) with constant per-core performance (red marks) for 140 small core counts (below about 200 cores). Beyond that point, the scaling 120 slowly deteriorates to reveal hardware's deficiency for codes that 100 HPCG was designed to represent.



INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Sandia National Laboratories

SPONSORED BY U.S. DEPARTMENT OF ENERGY NNSA National Nuclear Security Administration

Thank you!