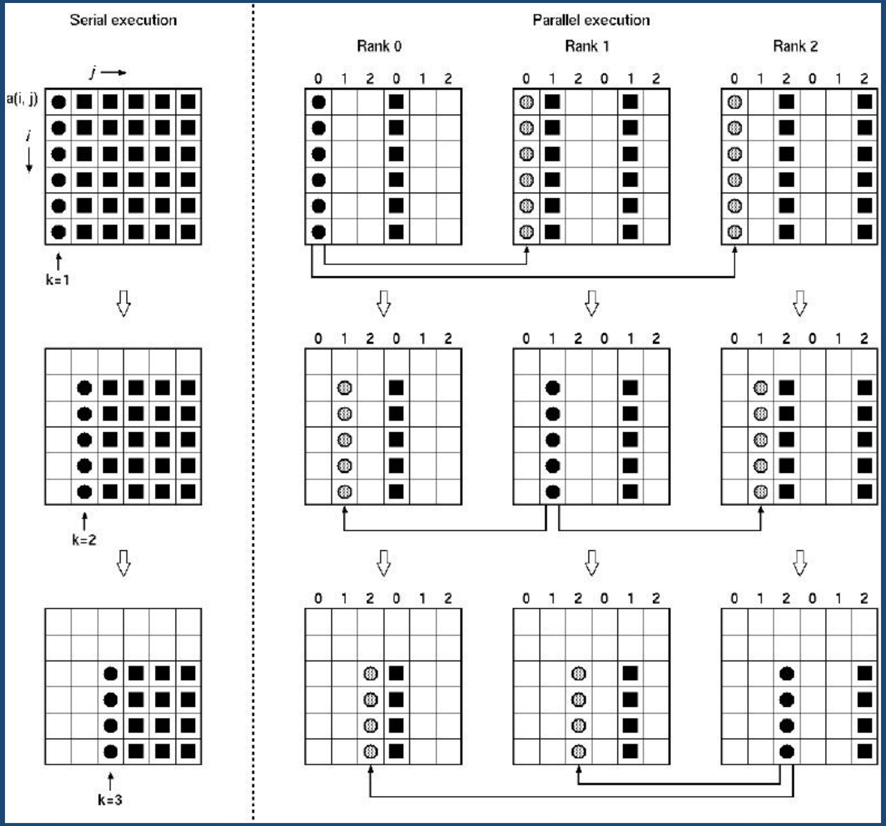


Parallel Linear Algebra

Some references

- J. Demmel, K. Yelick, Berkeley
- H. Casanova, University of Manoa, Hawaï
- **Parallel Algorithms,** H. Casanova, A. Legrand, Y. Robert
- **Parallel Programming – For Multicore and Cluster System,** T. Rauber, G. Rünger

A quick history of dense linear algebra libraries

- Libraries like EISPACK (for eigenvalues problems)
- Then the BLAS (1) came up (1973-1977)
- Standard library with 15 operations (mainly) on vectors
  - "AXPY" ($y = \alpha \cdot x + y$), dot product, scale ($x = \alpha \cdot x$), …
  - Up to 4 versions for each (S/D/C/Z), 46 routines, 3300 LOC

- **Goals**
  - Common pattern to simplify programming, readability, documentation
  - Robustness, thanks to a clean coding (avoid over/underflow)
  - Portability + efficiency thanks to specific implementations
- Why BLAS 1 ? These routines compute $O(n^1)$ operations over $O(n^1)$ data elements
- Used in libraries such as LINPACK (for linear systems)

A quick history of dense linear algebra libraries, contd.

But BLAS-1 were not sufficient

- Ex: AXPY operation ($y = \alpha \cdot x + y$): 2n flops for 3n reads/writes
- Computational intensity = (2n)/(3n) = 2/3
- Too small to reach performance close to peak performance (reads/writes dominate)
- Tough to vectorize ("SIMD'ize") over supercomputers of the 1980's

The BLAS-2 were invented (1984-1986)

- Standard library with 25 operations (mainly) on matrix/vector pairs
- "GEMV": $y = \alpha \cdot A \cdot x + \beta \cdot x$, "GER": $A = A + \alpha \cdot x \cdot yT$, $x = T\text{-}1 \cdot x$
- Up to 4 versions for each (S/D/C/Z), 66 routines, 18K LOC

• Why BLAS 2 ? They perform $O(n^2)$ operations over $O(n^2)$ data elements

• Computational intensity is $\sim(2n^2)/(n^2) = 2$

- Fine for vector machines, not for machines with caches

A quick history of dense linear algebra libraries, contd.

Next step: BLAS-3 (1987-1988)

- Standard library with 9 operations (mainly) on matrix/matrix pairs
- "GEMM": $C = \alpha \cdot A \cdot B + \beta \cdot C$, $C = \alpha \cdot A \cdot AT + \beta \cdot C$, $B = T\text{-}1 \cdot B$
- Up to 4 version of each (S/D/C/Z), 30 routines, 10K LOC

- Why BLAS 3 ? They perform $O(n^3)$ operations over $O(n^2)$ data elements
- This leads to a computational intensity equal to $(2n^3)/(4n^2) = n/2$!
  - Good to machines with caches, other memory hierarchies

Code volume of BLAS1/2/3 (available on www.netlib.org/blas)

- Source: 142 routines, 31K LOC,   Testing: 28K LOC
  - Reference implementation (non optimized)
  - Ex: 3 nested loops for the GEMM routine
- Much code optimized elsewhere
  - Motivates the "automatic tuning" of BLAS
- Part of standard mathematical libraries (AMD AMCL, Intel MKL)

A quick history of dense linear algebra libraries, contd.

LAPACK – "Linear Algebra PACKage" – uses BLAS-3 (1989 – today)

- **LAPACK content (summary)**
  - Algorithms we can transform in (almost) 100% BLAS 3
    - Linear systems: solve Ax = b for x
    - Least Squares: choose x to minimize $||Ax-b||^2$
  - Algorithms that are ≈ 50% BLAS 3
    - Eigenvalue problems: Find l and x where Ax = l x
    - Singular Value Decomposition, SVD
  - Generalized problems (Ax = l Bx)
  - Error bounds on all routines
  - Many variants according to the structure of A (band, $A = A^T$, …)
- **How much code?** (Release 3.7.0, 2016) (www.netlib.org/lapack)
  - Source: 1586 routines, 500K LOC,
  - Testing: 363K LOC

A quick history of dense linear algebra libraries, contd.

Is LAPACK parallel?

- Only if the BLAS are parallel (possible in shared memory)

ScaLAPACK – “Scalable LAPACK” (1995 – 2005)

- For distributed memory - uses MPI
- More complex data structures / algorithms than LAPACK
- Only a (small) subset of LAPACK features are available
- **Available** at www.netlib.org/scalapack

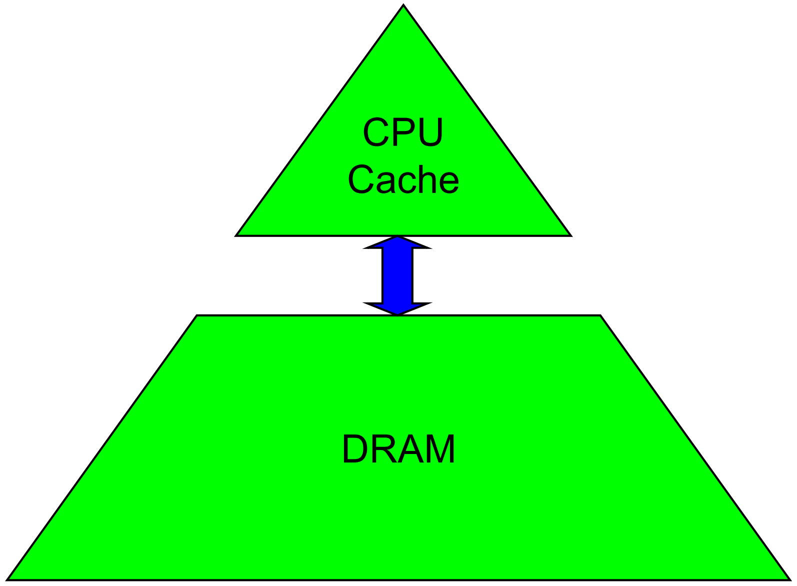
Plasma (2005-today)

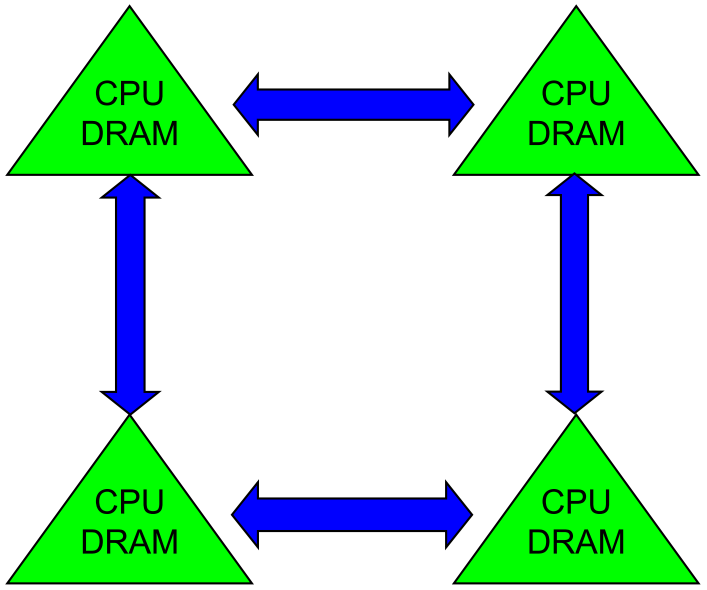
- Parallel Linear Algebra Software for Multicore Architectures
- Multicore friendly
- Using a scheduler for DAGs
- Reduction of granularity of calculations
- Decrease in the number of dependencies

Why should we avoid communications?

Algorithms have two costs

1. Arithmetic (flops)
2. Communications: moving data between
   - Memory hierarchy levels (sequential case)
   - Processors through a network or through a shared memory (parallel case)





Why should we avoid communications

The exécution of an algorithm is the sum of three terms

- # flops * time_per_flop
- # words moved / bandwidth
- # messages * latency

$\left.\begin{array}{l} \text{\# words moved / bandwidth} \\ \text{\# messages * latency} \end{array}\right\}$ communication

- Time_per_flop << 1/ bandwitdth << latency
  - Differences increase exponentially with the time

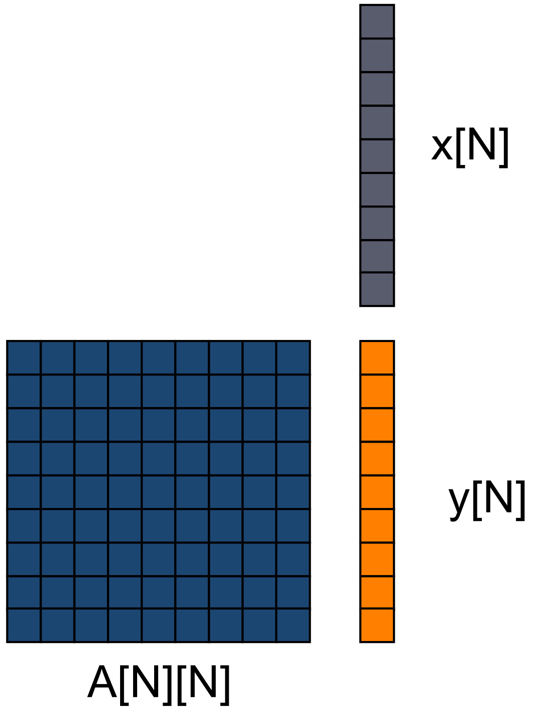
Annual improvements			
Time_per_flop		Bandwidth	Latency
59%	Network	26%	15%
	DRAM	23%	5%

- **Goal**: design algorithms that avoid communications
  - Between all the levels of memory hierarchies
    - L1 $\longleftrightarrow$ L2 $\longleftrightarrow$ DRAM $\longleftrightarrow$ network, etc
  - Not only hiding communications (overlap with arithmetic operations)
    - acceleration $\leq$ 2x
  - Any acceleration possible

FIRST ALGORITHMS ON A RING OF PROCESSORS

Matrix-Vector product

```
y = A x
Let N be the size of the matrix
Int A[N][N];
int x[N];

for i = 0 to n-1 {
   y[i] = 0;
   for j = 0 to n-1
       y[i] = y[i] + A[i,j] * x[j];
}
```



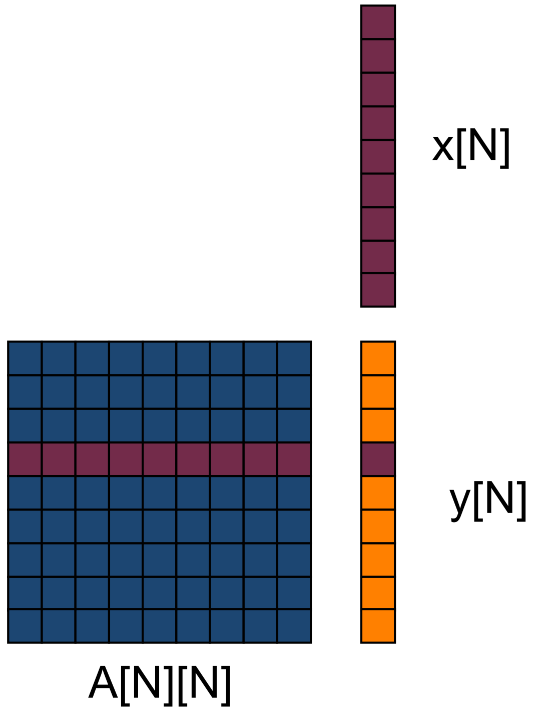
- How to parallelize this operation ?

Matrix-Vector product, contd.

- **Parallelism**
  - Computations on vector elements are independent
  - Each computation needs one line of `A` and vector `x`

- **In shared memory**

```
#pragma omp parallel for private(i,j)
for i = 0 to n-1 {
   y[i] = 0;
   for j = 0 to n-1
      y[i] = y[i] + A[i,j] * x[j];
}
```



Matrix-Vector product, contd.

- In distributed memory, one possibility consists in giving each process a copy of matrix `A` (needed rows) and vector `x`
- Each process declares a vector `y` of size `N/n`
  - We suppose that `n` divides `N`
- Then, the code can be

```
load(a); load(x)
n = NUM_PROCS();
r = MY_RANK();
for (i=r*N/n; i<(r+1)*N/n; i++) {
        for (j=0;j<N;j++)
                y[i-r*N/n] = a[i][j] * x[j];
}
```

- It is “*embarrassingly parallel*”. What’s about the result of the computation?

What's about the result of the computation?

- After completion of the computation, each process owns one part of the result
- One might want to store the result in a file.
  - Needs a synchronization such as to have the I/O performed in order

With the following code

```
if (r != 0)
        recv(&token,1);
open(file, “append”);
for (j=0; j<N/n ; j++)
        write(file, y[j]);
send(&token,1);
close(file)
barrier();   // optional
```

- One can also use a gather to get the whole vector on process 0
  - Vector `y` can stay in the memory of one node

What happens if the matrix is too large?

- The matrix may not fit in memory
  - This is one of the motivations for using a distributed version of this operation!

- In this case, each processor can only store part of the matrix `A`
- For the matrix-vector product, each processor can only store `N/n` rows of the matrix
  - Conceptually `A[N][N]`
  - But the program declares `A[N/n][N]`

- This raises the problem of global indexes and local indexes

Global and local indexes

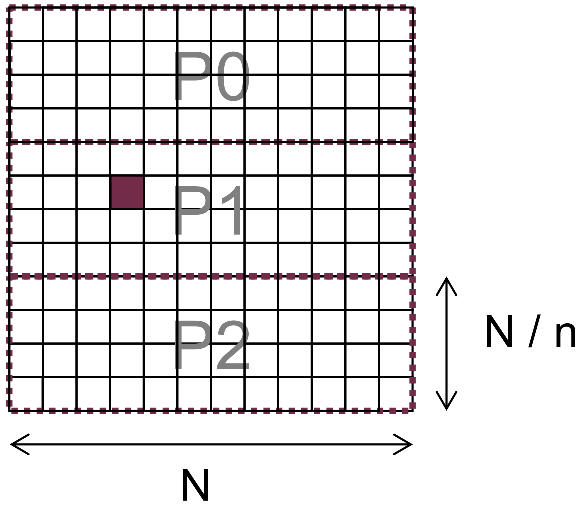
When a table is split between processes

- Global indexes (I, J) that reference an element of the matrix
- Local indexes (i, j) that reference an element of the local array that stores a part of the array

It is necessary to have a translation between the global and local indices

- The algorithm must be thought in terms of global indexes
- And implement it in terms of local indexes

```
Global:         A[5][3]
Local:          a[1][3] on process P1

a[i,j] = A[(N/n)*rank + i][j]
```



Global indexes computation

Parallel codes implement translation functions

- `GlobalToLocal()`
- `LocalToGlobal()`

- **Make a habit of implementing such functions**
  - Easy for ring algorithms with block distributions
  - More complicated for other topologies and other data distributions

Array distributions

We have the following distributions

- 2-D array `A` distributed
- 1-D array `y` distributed
- 1-D array `x` duplicated

Having distributed arrays allows to partition work between processes

- But this makes the code more complex because of translations of global and local indices
- This may require synchronization to load / save the elements of an array in a file

Distribute every vector ?

- Up to now we have the vector `x` duplicated
- Generally, one tries to have all the tables involved in a same computation distributed in the same way
  - This simplifies the reading of the code without constantly keeping in mind what is distributed and what is not

    For example, the local indices for the array are different from the global indices, but the local indices in array `x` are the same as the global indices. This leads to bugs!

We would like each process to have

- `N/n` rows of the matrix `A` in an array `A[N/n][N]`
- `N/n` pieces of vector `x` in an array `x[N/n]`
- `N/n` pieces of vector `y` in an array `y[N/n]`

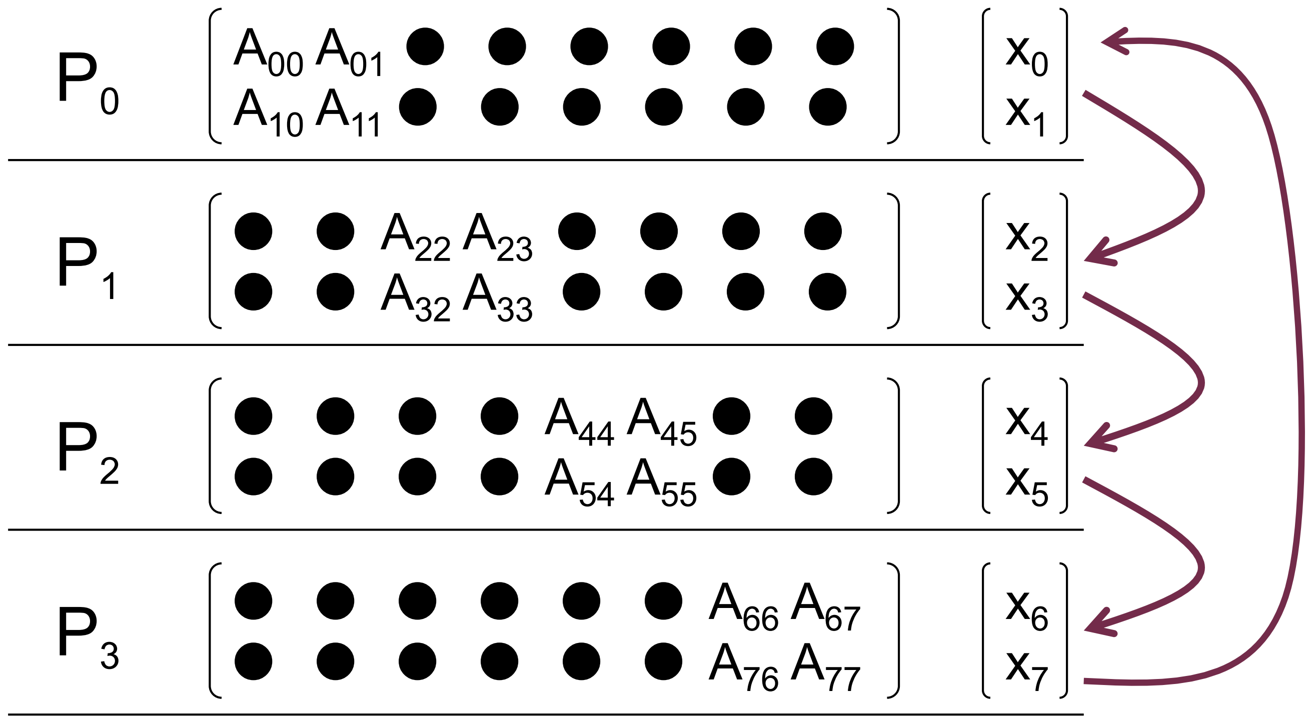
Principle of the algorithm

$$\begin{array}{lll}
P_0 & \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\ \hline
P_1 & \begin{pmatrix} A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\ \hline
P_2 & \begin{pmatrix} A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\ \hline
P_3 & \begin{pmatrix} A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \\ \hline
\end{array}$$

Initial data distribution for:

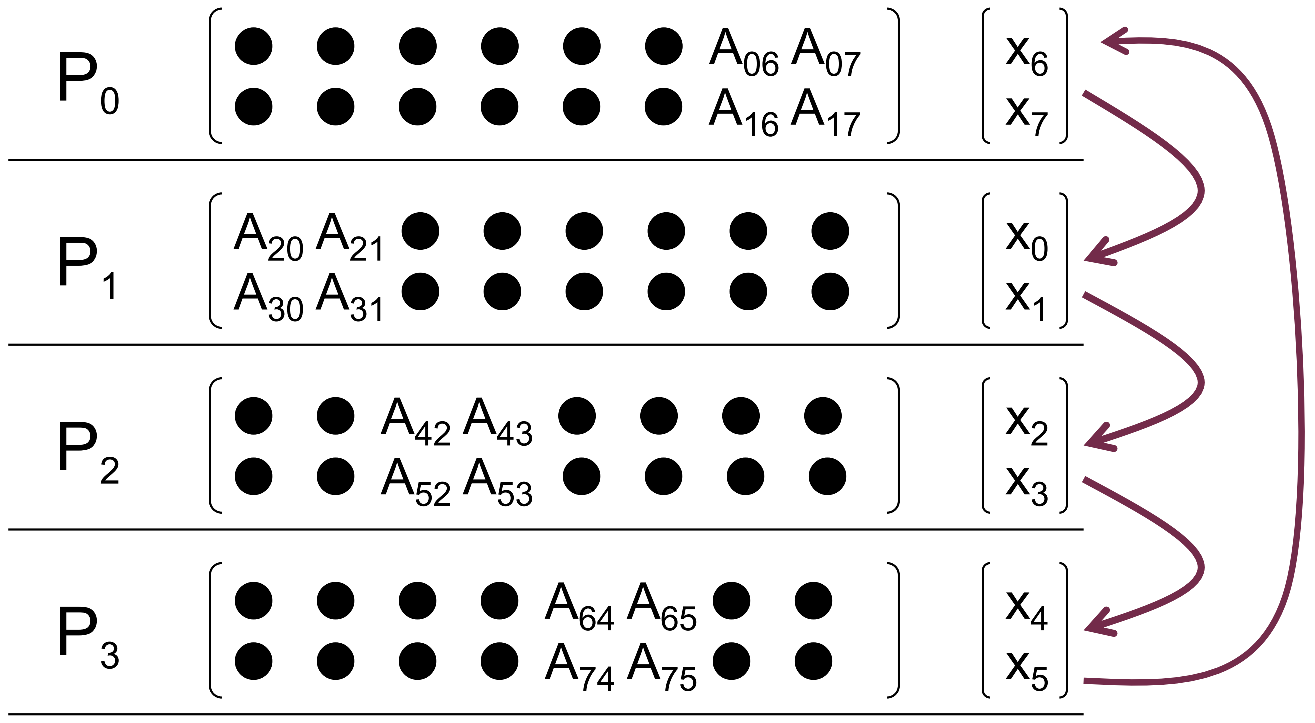
$N = 8$
$n = 4$
$N/n = 2$

Principle of the algorithm, contd.



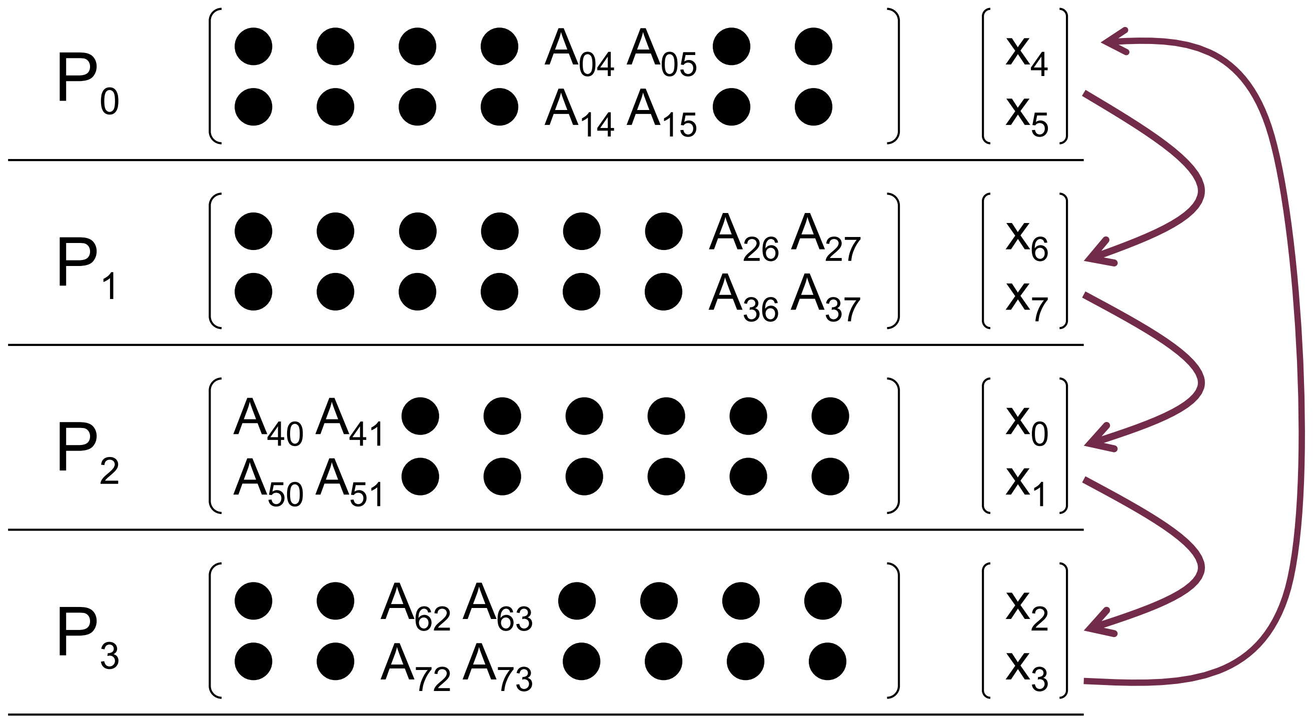
Step 0

Principle of the algorithm, contd.



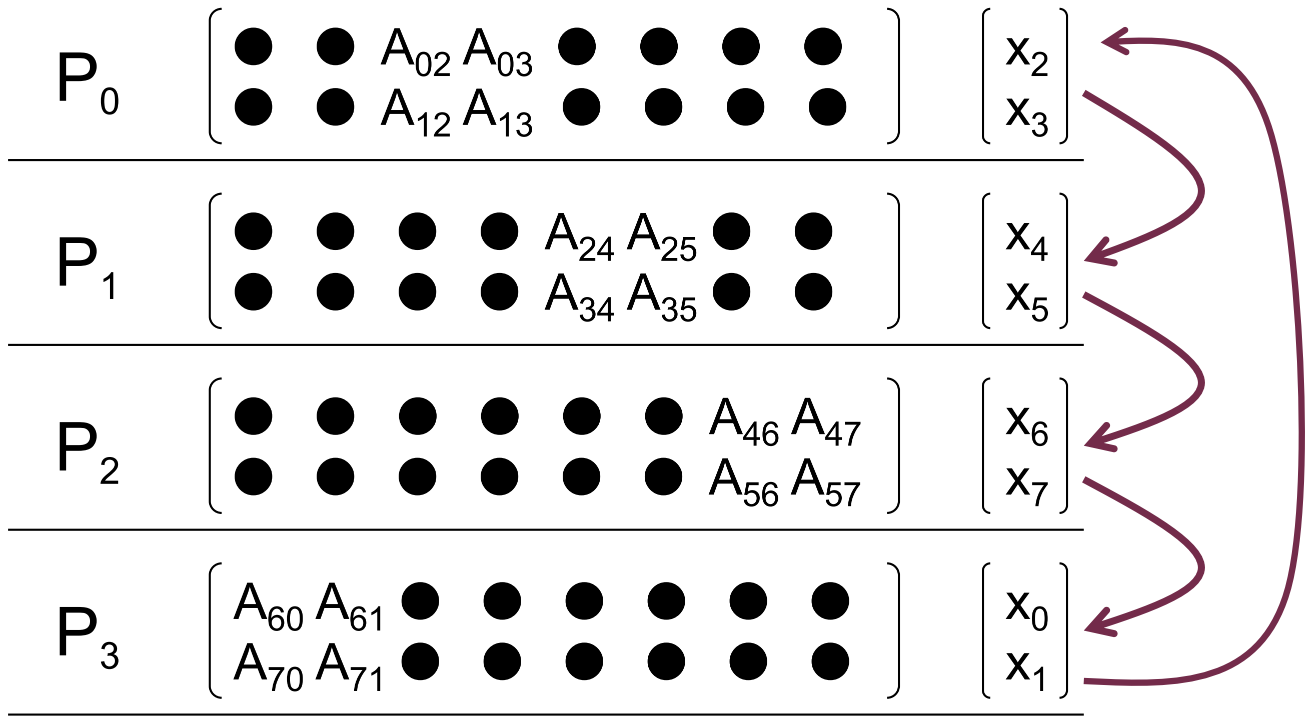
Step 1

Principle of the algorithm, contd.



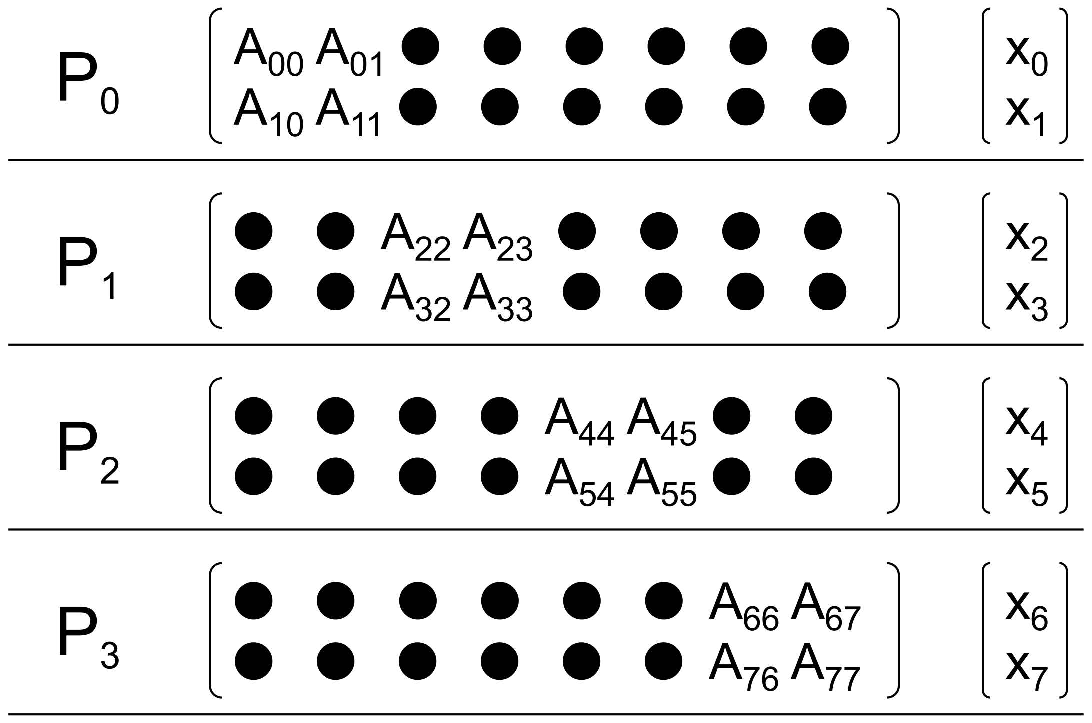
Step 2

Principle of the algorithm, contd.



Step 3

Principle of the algorithm, contd.



The final exchange of the vector $\mathbf{x}$ is not necessary for the calculation

It makes it possible to find the vector $\mathbf{x}$ distributed as at the start of the algorithm

Final step

Algorithm

- Uses two buffers
  - `tempS` to send and `tempR` to receive

```
float A[N/p][N], x[N/p], y[N/p];
r ← N/p
tempS ← x    /* My part of the vector (N/n elements) */
for (step=0; step<p; step++) {  /* p steps */
  SEND(tempS,r)
  RECV(tempR,r)
  for (i=0; i<N/p; i++)
    for (j=0; j <N/p; j++)
      y[i] ← y[i] + a[i,(rank - step mod p) * N/p + j] * tempS[j]
  tempS ↔ tempR
}
```

- In the example, the process of rank 2 in step 3 will work with a 2x2 block of the matrix that starts at column

$$((2 - 3) \bmod 4)*8/4 = 3 * 8 / 4 = 6;$$

Some general principles

- Large data must be distributed on processes (which run on different processors in a cluster)
  - Requires arithmetic operations to compute indexes
  - We write functions `local_to_global ()` and `global_to_local ()`
- Data must be loaded / written before / after calculations
  - Requires synchronization between processes
- Trying to have distributed data structures in the same way to avoid confusion between local and global indexes
- In the last algorithm, all the indexes are local

- Code is more complex than an OpenMP implementation
  - more freedom in optimizations

Performance analysis

- They are p identical steps
- At each step, each process performs three activities
  - Compute, send and receive

- **Compute**

  r^2w (w: time to perform a += * operation)
- **Receive**

  $L + r\,b$
- **Send**

  $L + r\,b$

- Thus a total

$$T(p) = p\,(r^2w + 2L + 2rb)$$

Asymptotic performance

$$T(p) = p(r^2w + 2L + 2mb)$$

- $\text{Speedup}(p) = T_s/T_p = n^2w \;/\; p\,(r^2w + 2L + 2mb)$
  $= n^2w \;/\; (n^2w/p + 2pL + 2pmb)$
- $\text{Efficiency}(p) = T_s/pT_p = n^2w \;/\; (n^2w + 2p^2L + 2p^2mb)$
- For p fixed, when n is big, then $\text{Efficiency}(p) \sim 1$

Conclusion

The algorithm is asymptotically optimal

Performance analysis, contd.

Remark that a naïve algorithm that will broadcast the whole vector to all the processes to allow them to compute in a independent way will execute in the following time

$$T(p) = (p-1)(L + nb) + pr^2 w$$

- It could use a pipelined broadcast with
  - The same asymptotic performance
  - More simple
  - Only waste a small fraction of memory (the vector)
  - Less “elegant”

We have to evaluate the different solutions to find out which the most performant one given the size of the target matrices, the parameters of the target platform, …

Back to the algorithm

```
float A[N/p][N], x[N/p], y[N/p];
r ← N/p
tempS ← x   /* My part of the vector (N/n elements) */
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r)
    RECV(tempR,r)
    for (i=0; i<N/p; i++)
       for (j=0; j <N/p; j++)
           y[i] ← y[i] + a[i,(rank - step mod p) * N/p + j] * tempS[j]
     tempS ↔ tempR
}
```

- In this code, at each iteration, SENDs, RECVs and the computation can be performed in parallel
- We can overlap communications and computations using non-clocking SEND and RECV
    - In MPI: MPI_ISend() et MPI_IRecv()

Algorithm with overlaps

```
float A[N/p][N], x[N/p], y[N/p];
tempS ← x                    /* My part of the vector (N/n elements) */
r ← N/p
for (step=0; step<p; step++) {                         /* p steps */
     SEND(tempS,r) || RECV(tempR,r) ||
     for (i=0; i<N/p; i++) {
       for (j=0; j <N/p; j++) {
         y[i] ← y[i]+a[i,(rank-step mod p)*N/p+j]*tempS[j]
       }
     }
     tempS ↔ tempR
}
```

Best performance

- There are p identical steps
- At each step, each processor performs the three activities (compute, send, receive) in parallel
  - **Compute**: r^2w
  - **Receive**: $L + rb$
  - **Send**: $L + rb$
- Thus a total time of

$$T(p) = p \max(r^2w , L + rb)$$

- Same asymptotic performance like before but better performance for small values of n

Hybrid parallelism

- Multicore architectures are standard now
- How to exploit multiple cores
- **Option 1:** Execute several process per node
  - Can lead to additional overheads and more communications
  - In fact, we will have network communications between processes inside a node!
  - MPI will not know that the processes are allocated to the same node
- **Option 2: Execute a single multithreaded process per node**
  - Less overhead, fast communications inside a node
  - Made by combining MPI with OpenMP
  - Write an MPI program
  - Add OpenMP pragmas around loops

Hybrid parallelism

```
float A[N/n][N], x[N/n], y[N/n];
tempS ← x          /* My part of the vector (N/n elements) */
for (step=0; step<p; step++) {                        /* n steps */
          SEND(tempS,r)
      || RECV(tempR,r)
      || #pragma omp parallel for private(i,j)
          for (i=0; i<N/p; i++)
            for (j=0; j <N/p; j++)
              y[i] ← y[i] + a[i,(rank - step mod p)*N/p+j]*
                                   tempS[j]
    tempS ↔ tempR
}
```

- We call this **hybrid parallelism**
- Communication through the network between nodes
- Communication through shared memory within nodes

Matrix multiplication on a ring

We can perform the matrix product with an algorithm close to the matrix-vector product

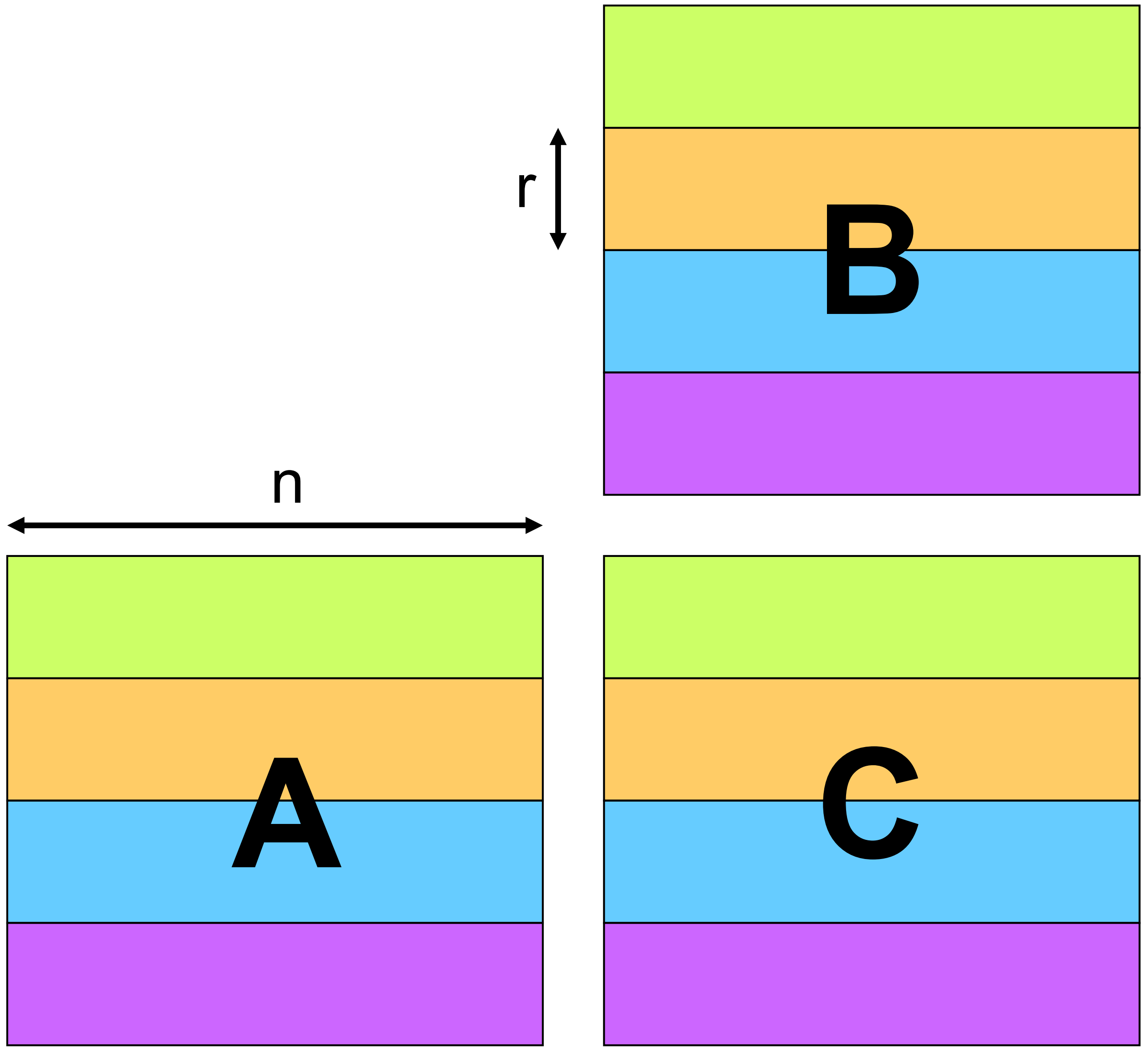
- A matrix product is just the computation of n^2 scalar products (not only n)

We have three matrices	A, B, and C
We want to compute	C = A*B

The matrices are distributed so that each processor contains a block of rows of each matrix

- Easy to do if matrices are stored in C because all matrix elements will be stored contiguously in memory

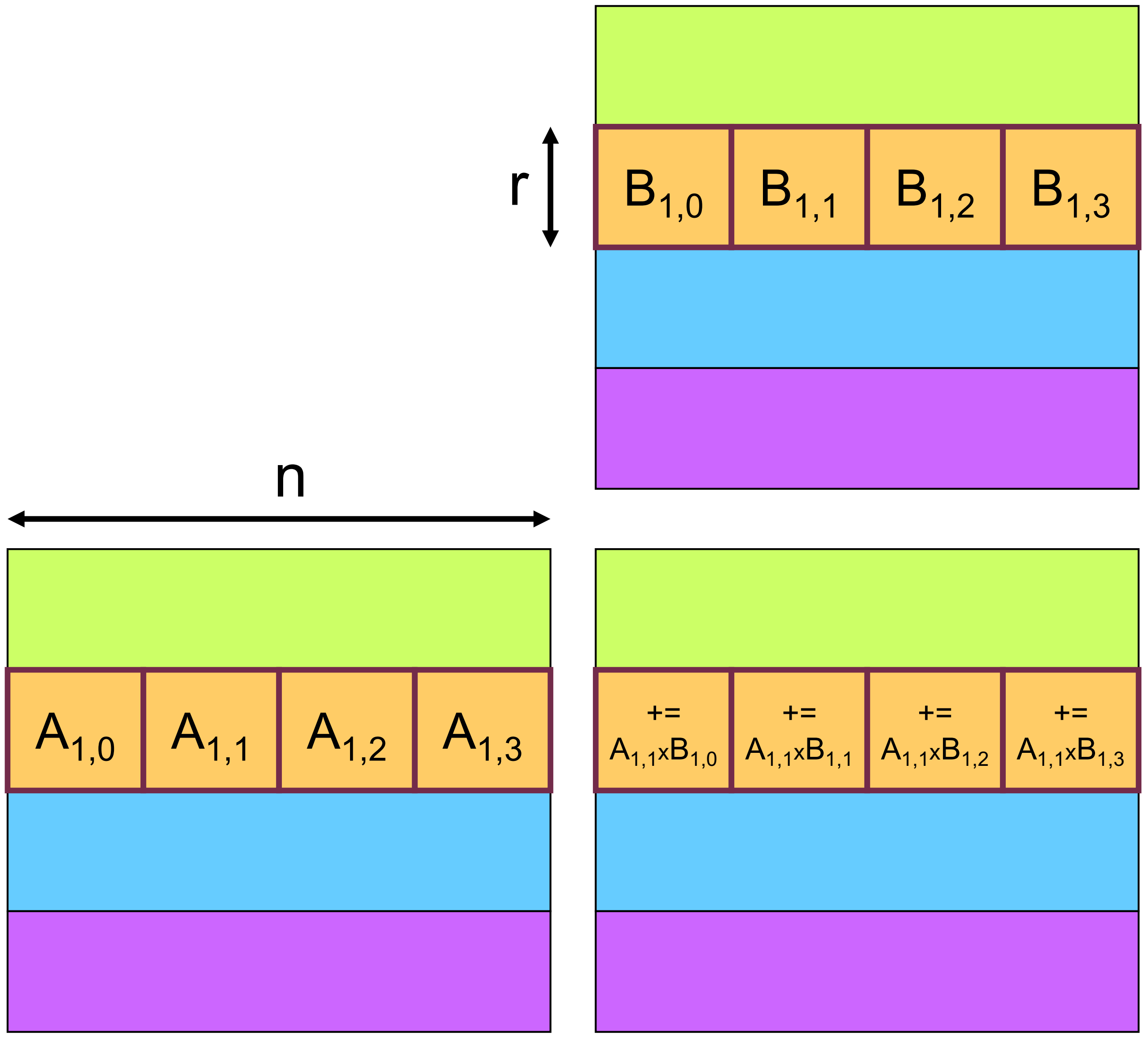
Data distribution



First step

p=4

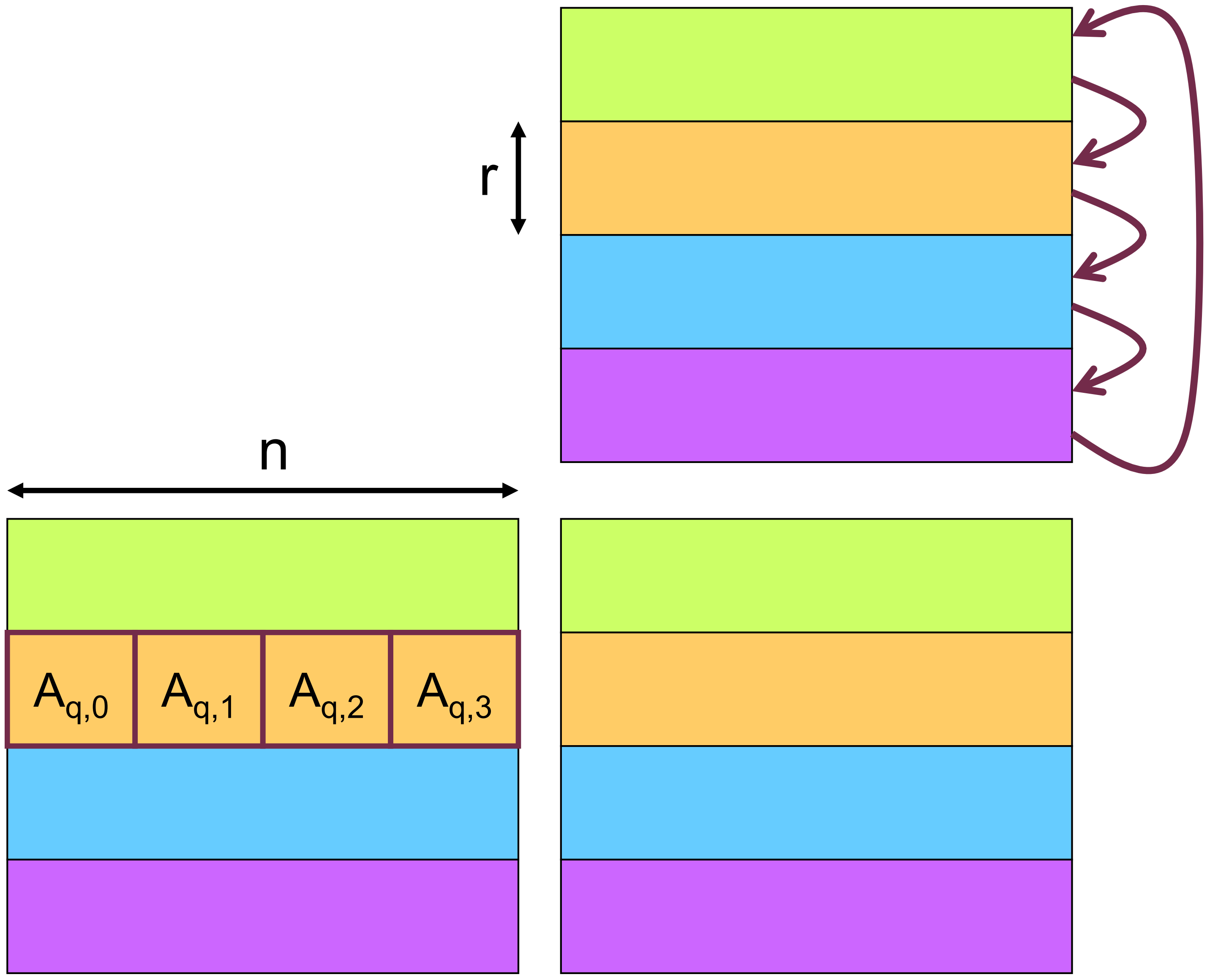
Processor P_1



Rotation of blocks of rows of B

p=4

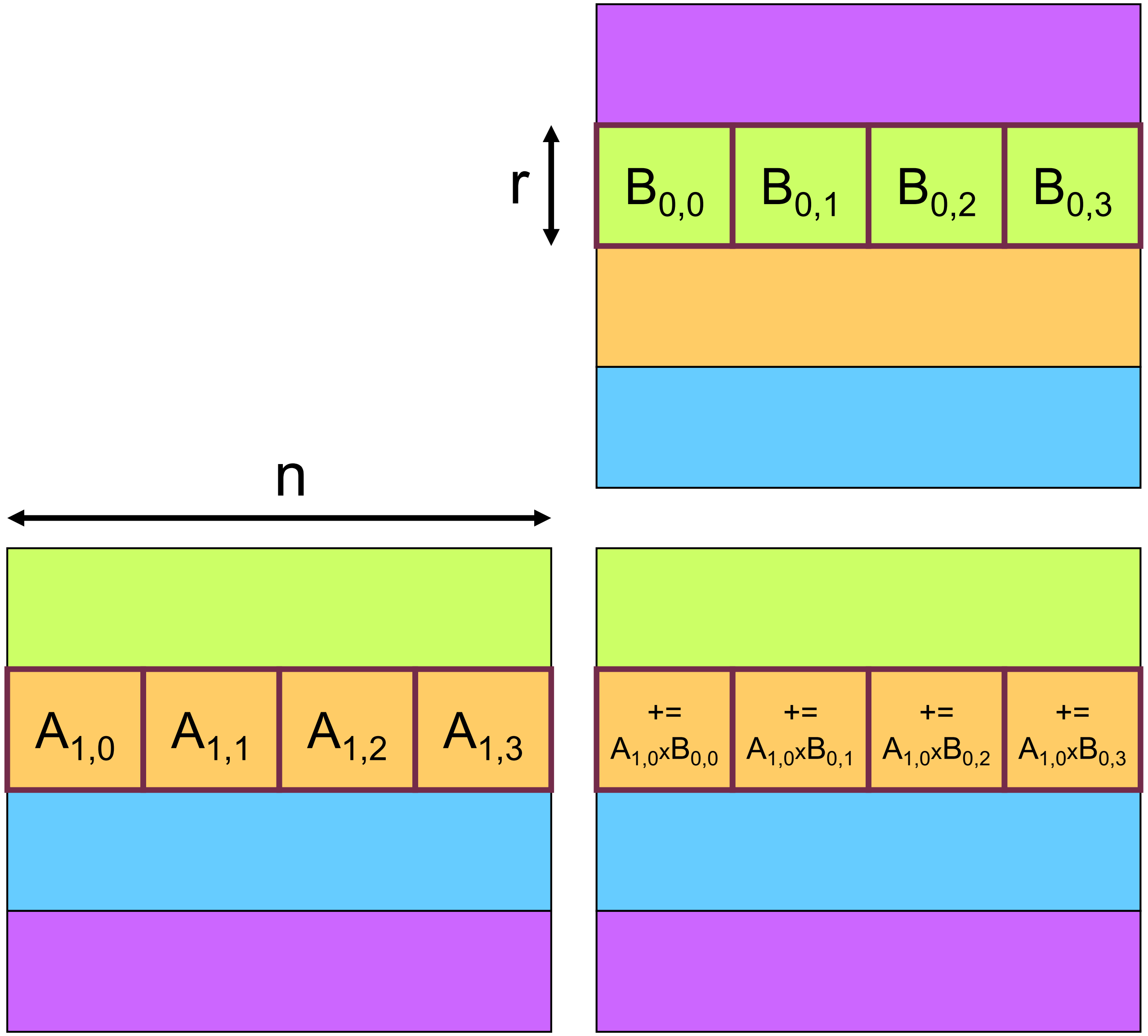
Processor P_q



Second step

p=4

Processor P_1



Algorithm

A the end of the computation, each block $C_{i,j}$ has a correct value:

$$A_{i,0} B_{0,j} + A_{i,1} B_{1,j} + \dots$$

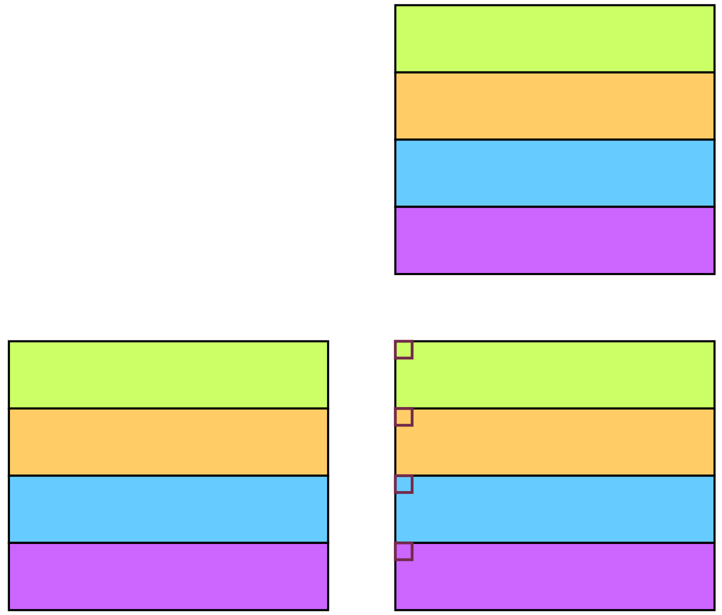
Basically, it is the same algorithm as the matrix-vector product, replacing the partial scalar products with sub-matrix products (complex with loops and indices)

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```

Algorithm

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```

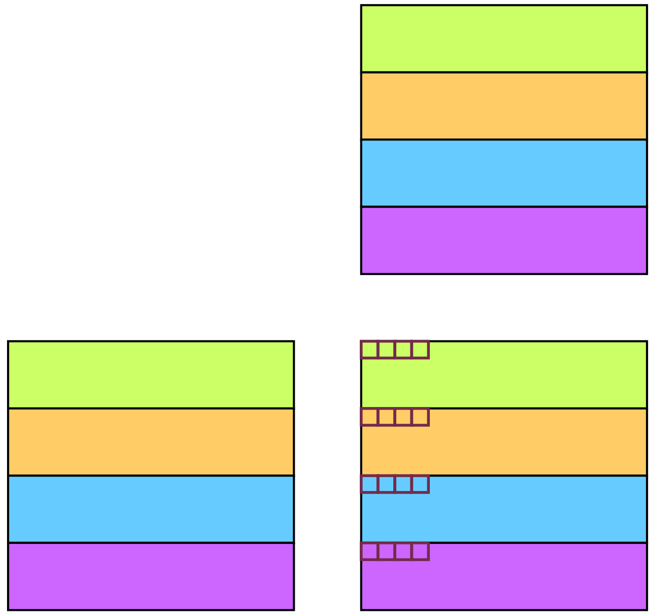
Step 0
l=0
i=0
j=0



Algorithm

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```

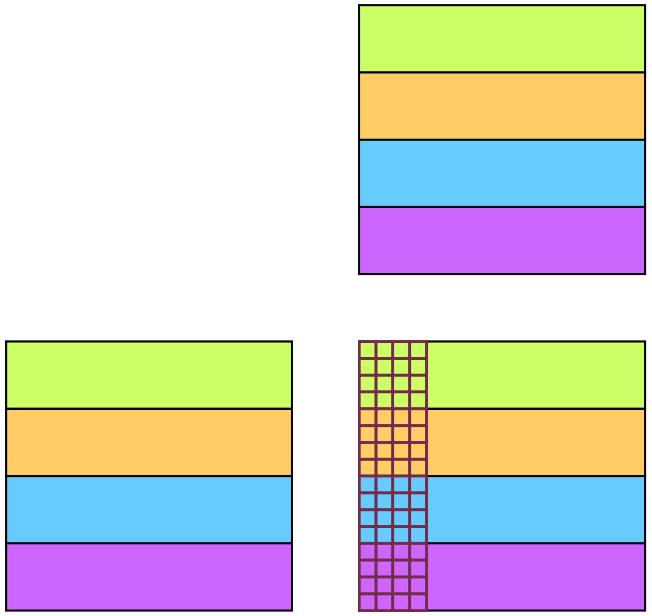
Step 0
l=0
i=0
j=*



Algorithm

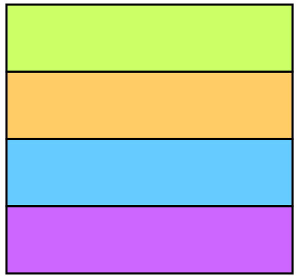
```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```

Step 0
l=0
i=*
j=*

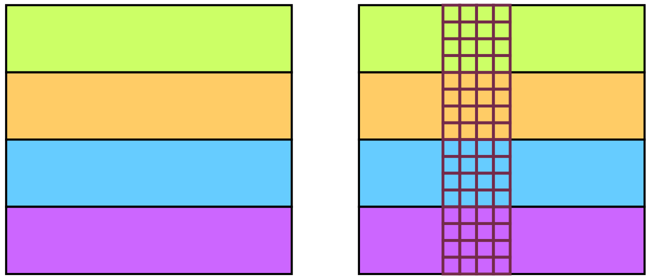


Algorithm

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```



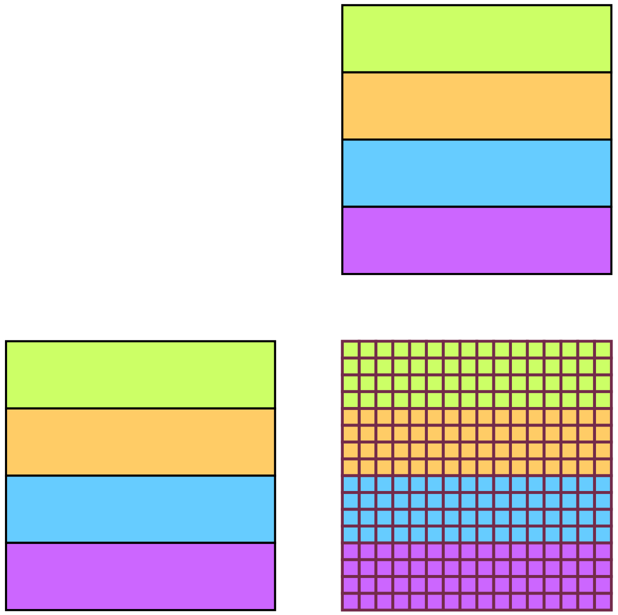
Step 0
l=1
i=*
j=*



Algorithm

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```

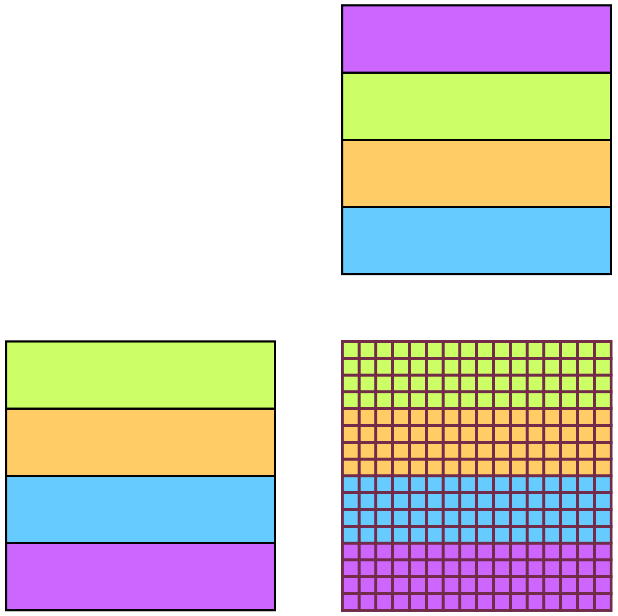
Step 0
l=*
i=*
j=*



Algorithm

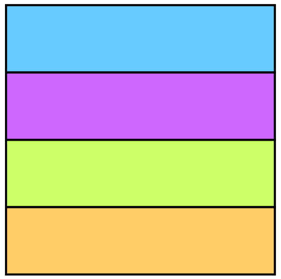
```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```

Step 1
l=*
i=*
j=*

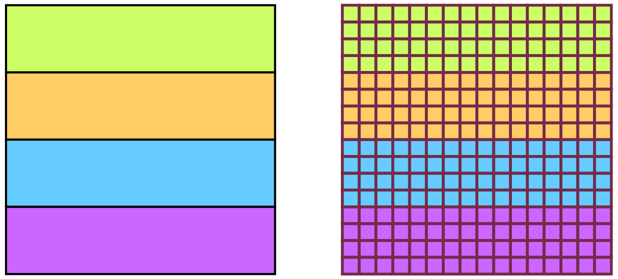


Algorithm

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```

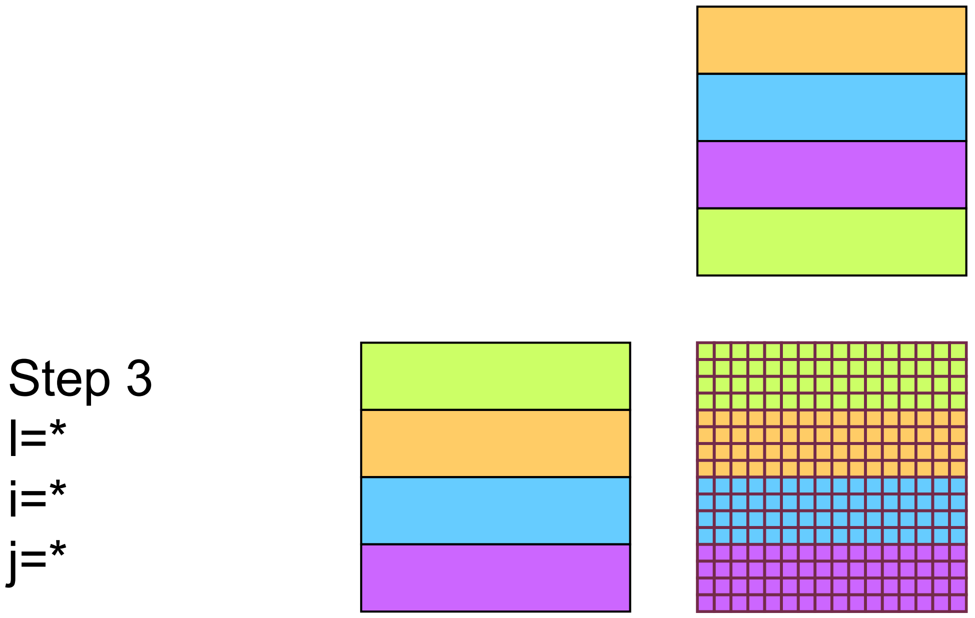


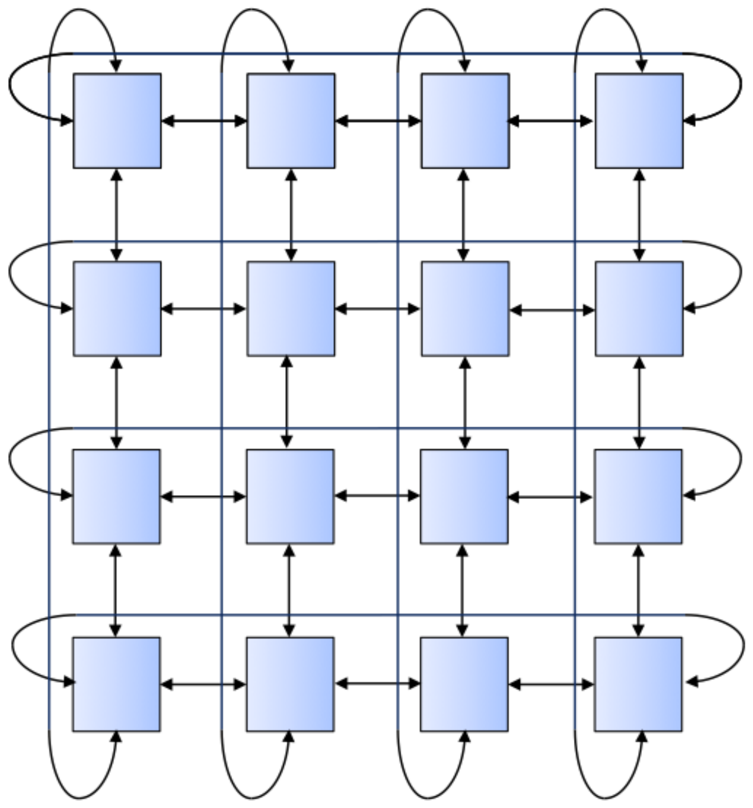
Step 2
l=*
i=*
j=*



Algorithm

```
float A[N/p][N], B[N/p][N], C[N/p][N];
r ← N/p
tempS ← B
q ← MY_RANK()
for (step=0; step<p; step++) {  /* p steps */
    SEND(tempS,r*N) || RECV(tempR,r*N)
    || for (l=0; l<p; l++)
     for (i=0; i<N/p; i++)
        for (j=0; j<N/p; j++)
           for (k=0; k<N/p; k++)
              C[i,l*r+j] ← C[i,l*r+j] + A[i,r((q - step)%p)+k] * tempS[k,l*r+j]
     tempS ↔ tempR
}
```





ALGORITHMS ON A GRID

Bi-dimensional grid of processors

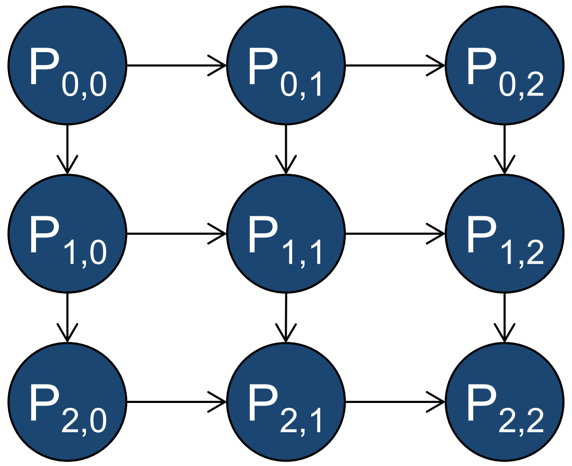
Let $p = q^2$ processors

They can be seen as being arranged in the form of a square grid

- One can also have a rectangular grid

Each processor is identified by two indexes

- i: its row
- j: its column



Bi-dimensional torus (2D torus)

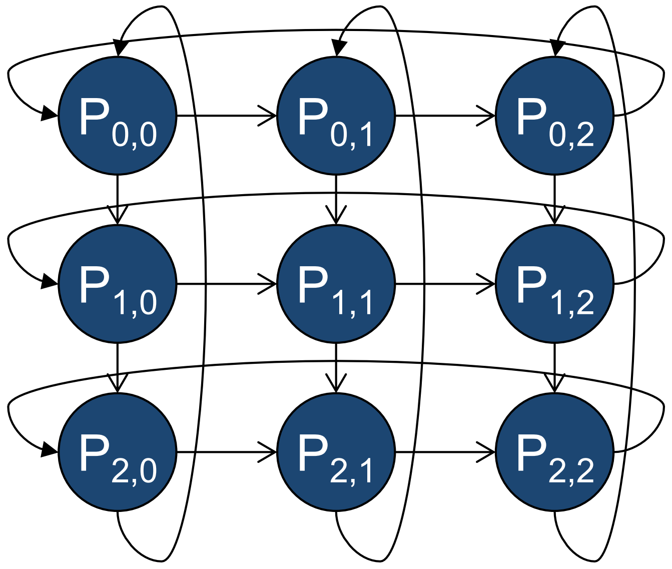
We have links which connect each side of the grid

Each processor belongs to two different rings

- Possibility to use algorithms designed for ring topologies

Mono-directional or bi-directional links

- Depends on what we need for our algorithm and/or physical resources

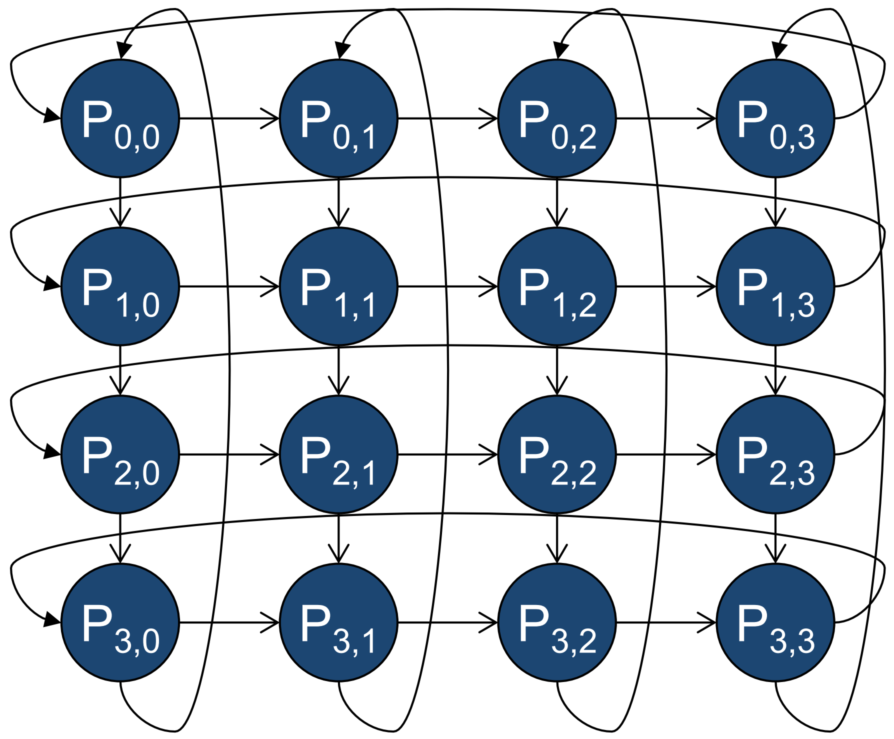


Matrix product on a grid

Algorithm very studied!

- Operations used in many applications and other calculation kernels
- Implements several communication and computation problems in grids and torus topologies

Bi-dimensional matrix distribution



- Let $a_{i,j}$ be a element of the matrix
- We denote by $A_{i,j}$ (or A_{ij}) the block of matrix A assigned to $P_{i,j}$

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

How are the matrices distributed ?

Develop a general routine used for other functions

Two options

- **Centralized version**
  - When a function is called (eg: matrix product)

    Input data is available on a single "master" machine (file?)

    Input data must be distributed over other processes

    The output data must be retrieved and returned to the master machine (file)
  - Easier for the user
  - Allows the library to make data distribution decisions seamlessly to the user
  - Costly if you carry out sequences of operations!

How are the matrices distributed ? Contd.

Distributed

- At the call of a function
- It is assumed that the data is already distributed
- Leaves data in already distributed outputs
- Can lead to having to redistribute data between calls for distributions match, which is more complicated for the user and sometimes more expensive
- For example, it may be desired to change the block size between calls, or to switch from a cyclic distribution to a non-cyclic distribution

Most software and libraries adopt the distributed approach

- More work for the user
- More flexibility and control

In the following, it is always assumed that the data are already distributed

Four parallel matrix product algorithms

- Scalar products (*Outer-Product*)
- Cannon
- Fox
- Snyder

Algorithm using scalar products

Matrix product algorithm

```
for i=0 to n-1
   for j=0 to n-1
        for k=0 to n-1
            $c_{i,j}$ += $a_{i,k}$ * $b_{k,j}$
```

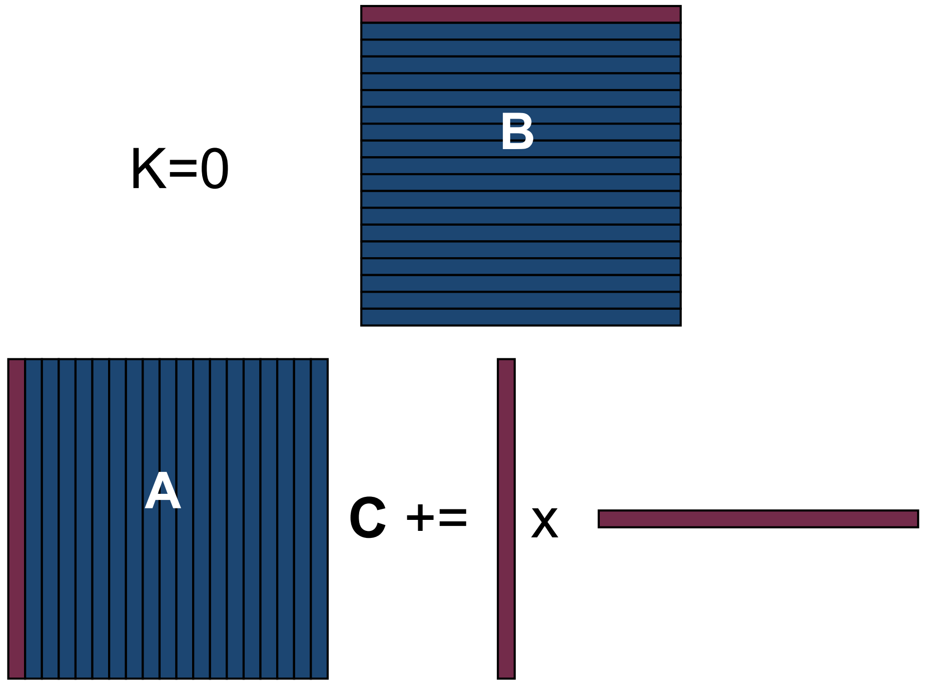
Loops can be reversed !

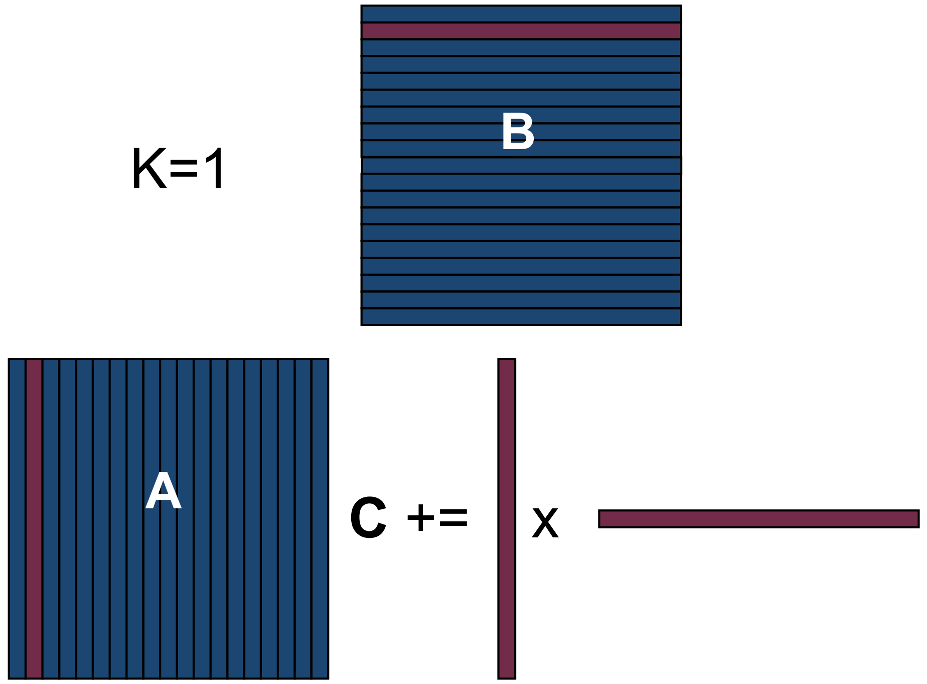
```
for k=0 to n-1
    for i=0 to n-1
         for j=0 to n-1
            $c_{i,j}$ += $a_{i,k}$ * $b_{k,j}$
```

- Sequence of scalar products !

Algorithm using scalar products, contd.

```
for k=0 to n-1
    for i=0 to n-1
        for j=0 to n-1
            $c_{i,j}$ += $a_{i,k}$ * $b_{k,j}$
```





Algorithm using scalar products, contd.

Thanks to the loop exchange, we can design a very simple parallel algorithm working on a grid of processors

First step: see the algorithm in terms of blocks assigned to processors (qxq grid)

```
for k=0 to q-1
    for i=0 to q-1
        for j=0 to q-1
            $C_{i,j}$ += $A_{i,k}$ * $B_{k,j}$
```

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Algorithm using scalar products, contd.

At step k, process $P_{i,j}$ needs $A_{i,k}$ and $B_{k,j}$

- If k = j, then the process already holds the needed block of A
  Otherwise it needs to ask it to $P_{i,k}$
- If k = i, then the process already holds the needed block of B
  Otherwise, it needs to ask it to $P_{k,j}$

```
for k=0 to q-1
    for i=0 to q-1
        for j=0 to q-1
            $C_{i,j}$ += $A_{i,k}$ * $B_{k,j}$
```

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Algorithm using scalar products, contd.

How the algorithm works

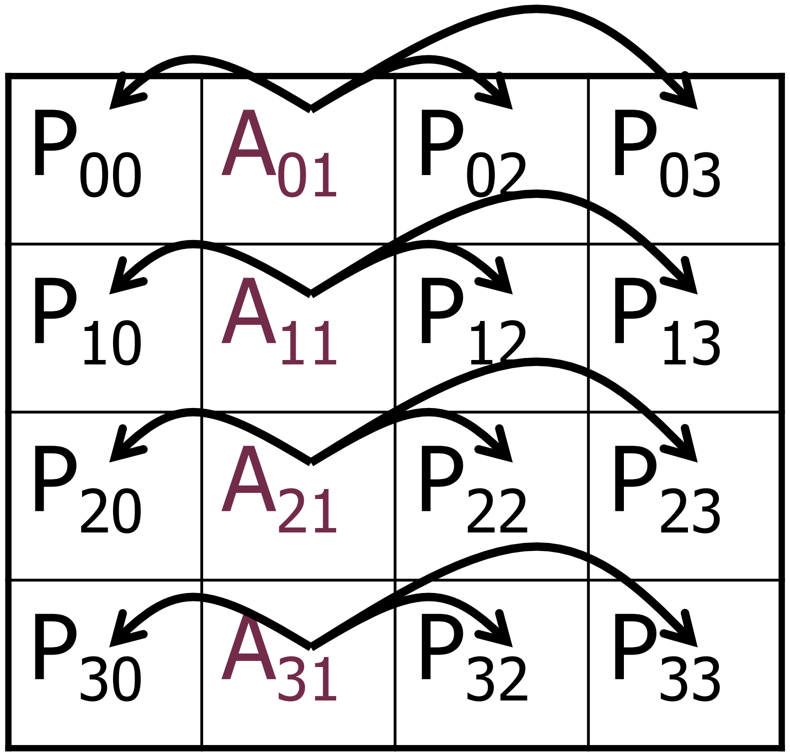
At step k

- Process $P_{i,k}$ broadcasts its block of matrix A to all the processes of line i
  True for every i
- Process $P_{k,j}$ broadcasts its block of matrix B to all the processes of column j
  True for every j

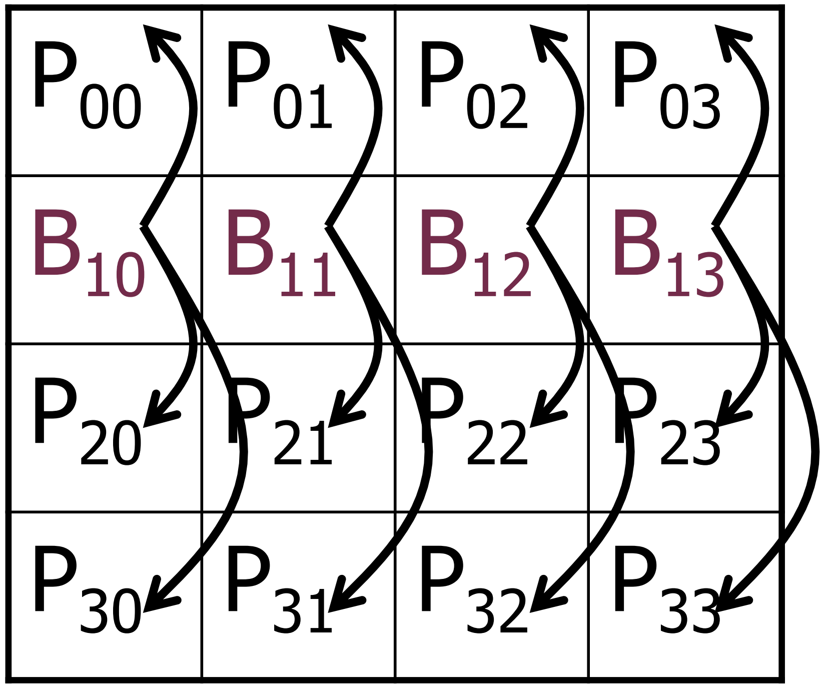
There are q-1 steps

Algorithm using scalar products, contd.

Step k=1 of the algorithm



P_{00} A_{01} P_{02} P_{03}
P_{10} A_{11} P_{12} P_{13}
P_{20} A_{21} P_{22} P_{23}
P_{30} A_{31} P_{32} P_{33}



P_{00} P_{01} P_{02} P_{03}
B_{10} B_{11} B_{12} B_{13}
P_{20} P_{21} P_{22} P_{23}
P_{30} P_{31} P_{32} P_{33}

Algorithm using scalar products, contd.

```
// m = n/q
var A, B, C: array[0..m-1, 0..m-1] of real
var bufferA, bufferB: array[0..m-1, 0..m-1] of real
var myrow, mycol

myrow = My_Proc_Row()
mycol = My_Proc_Col()

for k = 0 to q-1
   for i = 0 to m-1
         BroadcastRow(i,k,A,bufferA,m*m) // Broadcast of A over rows
   for j=0 to m-1
         BroadcastCol(k,j,B,bufferB,m*m) // Broadcast of B over columns
   // Matrix product over blocks
   if (myrow == k) and (mycol == k) MatrixMultiplyAdd(C,A,B,m)
   else if (myrow == k) MatrixMultiplyAdd(C,bufferA,B,m)
   else if (mycol == k) MatrixMultiplyAdd(C, A, bufferB, m)
   else MatrixMultiplyAdd(C, bufferA, bufferB, m)
```

Performance analysis

1-port model

- Matrix product at step k can be executed at the same time as broadcast of step k+1
- The two broadcasts execute in sequence

$$T(m,q) = 2\ T_{bcast} + (q-1) \max(2T_{bcast}, m^3w) + m^3w$$

w: elementary += * operation

T_{bcast}: execution time of a broadcast

Multi-port model

- Broadcasts can be executed in parallel

$$T(m,q) = T_{bcast} + (q-1) \max(T_{bcast}, m^3w) + m^3w$$

- Execution time for the pipelined broadcast

$$Tbcast = (\mathrm{sqrt}((q-2)L) + \mathrm{sqrt}(m^2\ b))^2$$

When n grows: $T(m,q) \sim q\ m^3 = n^3 / q^2$

Asymptotic efficiency is 1!

And then

- On a ring platform, we have already presented an asymptotically optimal algorithm
- Why bother to have another algorithm (a little more complicated) also asymptotically optimal?
- When n is big, it is not a problem
- But in reality, communication costs can not be neglected
    - When n has a "moderate" size,
    - When the ratio w / b is small
    - Grid topology is beneficial in reducing communications costs!

Ring versus grid

The ring algorithm gave a communication complexity of

$$n^2 b$$

- At each step, the algorithm sends n^2/p elements to neighboring processes and there are p steps

For the algorithm on a grid

- Each step requires two broadcast of n^2/p elements

  We assume a 1-port model so as not to give too much advantage to the grid
- With a pipelined diffusion, this operation can be carried out at the same time as sending n^2/p elements of matrices between the neighbors in each ring (unless n is really small)
- At each step, the algorithm performs twice as much communication as the ring but with sqrt (p) fewer steps

Conclusion

- The algorithm on a grid has sqrt (p) less time in communications than the algorithm on a ring

Ring versus grid, contd.

Why is the algorithm on the grid better?

Reason: More communication links can be used in parallel

- Point-to-point communications replaced by broadcasts
- More communication links used at each stage

• On the other hand, if the underlying network is not really a grid, then less advantage

• But the 2D distribution is always better than the 1D distribution, whatever the underlying platform

Ring versus grid, contd.

On a ring

- The algorithm communicates p blocks of matrices contain n^2/p elements, p times
- Total number of elements communicated: pn^2

On a grid

- At each step, 2sqrt (p) blocks of n^2/p elements are sent, each to sqrt(p)-1 process, sqrt(p) times
- Total number of elements communicated: 2sqrt(p)n^2

Conclusion

- The algorithm with a grid sends less data than on a ring
- The use of a 2D distribution is better than a 1D distribution, even if the underlying platform is a basic network
- A non-switched Ethernet network for example which is a network with a link that is closer to a ring (p communication links) than a grid (p^2 communication links)

Ring versus grid, conclusion

- Writing algorithms on a grid is a bit more complicated
- But generally gain more in terms of performance

Cannon matrix product algorithm

Old algorithm

- Designed for systolic architectures (SIMD)
- Adapted to a 2D grid

The algorithm starts with a redistribution of matrices A and B

- Called “*preskewing*”

Then matrices are multiplied together

At the end, the matrices are re-distributed to find their initial distribution

- Called “*postskewing*”

Cannon Preskewing

Matrix A

Each block of matrix A is shifted to the left until the process of the first process column contains a block of the diagonal of the matrix

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{11}	A_{12}	A_{13}	A_{10}
A_{22}	A_{23}	A_{20}	A_{21}
A_{33}	A_{30}	A_{31}	A_{32}

Cannon Preskewing, contd.

Matrix B

Each block of matrix B is shifted upward until process of the first process line contains a block of the diagonal of the matrix

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

B_{00}	B_{11}	B_{22}	B_{33}
B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}

Cannon algorithm

- The algorithm runs in q steps
- At each step, each processor executes a multiplication of its block of A and its block of B and adds it to its block of C
- Then the blocks of A are shifted to the left and the blocks of B are shifted upwards
- C blocks do not move

```
Participate to the preskewing of A
Participate to the preskewing of B
For k = 1 to q
   Local C = C + A*B
   Horizontal shift of A
   Vertical shift of B
Participate to the postskewing of A
Participate to the postskewing of B
```

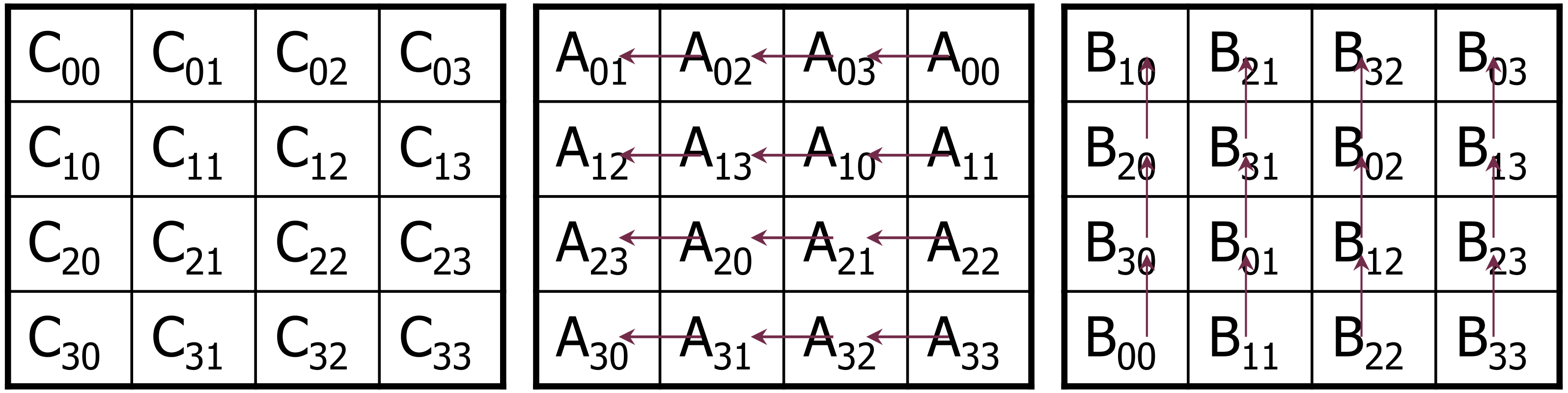
Steps of the Cannon algorithm

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{11}	A_{12}	A_{13}	A_{10}
A_{22}	A_{23}	A_{20}	A_{21}
A_{33}	A_{30}	A_{31}	A_{32}

B_{00}	B_{11}	B_{22}	B_{33}
B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}

Local computation on processor (0,0)



C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{01}	A_{02}	A_{03}	A_{00}
A_{12}	A_{13}	A_{10}	A_{11}
A_{23}	A_{20}	A_{21}	A_{22}
A_{30}	A_{31}	A_{32}	A_{33}

B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}
B_{00}	B_{11}	B_{22}	B_{33}

Shifts

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{01}	A_{02}	A_{03}	A_{00}
A_{12}	A_{13}	A_{10}	A_{11}
A_{23}	A_{20}	A_{21}	A_{22}
A_{30}	A_{31}	A_{32}	A_{33}

B_{10}	B_{21}	B_{32}	B_{03}
B_{20}	B_{31}	B_{02}	B_{13}
B_{30}	B_{01}	B_{12}	B_{23}
B_{00}	B_{11}	B_{22}	B_{33}

Local computation on processor (0,0)

Performance analysis

Performance analysis with the 4-port model

- 1-port model is usually simpler

Symbols

- n:	matrix size
- qxq:	process grid size
- m =	n / q
- L:	communication start size
- b:	time to communicate one element of a matrix
- w:	basic arithmetic operation cost (+= . * .)

$$T(m,q) = T_{preskew} + T_{compute} + T_{postskew}$$

Pre/Post-skewing execution time

Horizontal shift

- Each row has to be shifted so as the diagonal of the matrix ends up on the first column of processes

On a uni-directional ring

- The last line has to be shifted $(q\text{-}1)$ times
- All the rows can be shifted in parallel

Total time: $(q\text{-}1)(L + m^2b)$

On a bi-directional ring

- A row can be shifted on the right or on the left depending of the distance
- A row is shifted at most $\left\lfloor \frac{q}{2} \right\rfloor$ times
- All the rows can be shifted in parallel

Total time: $\left\lfloor \frac{q}{2} \right\rfloor \left(L + m^2 b\right)$

With the 4-port model, preskewing of A and B can occur in parallel (horizontal and vertical shifts do not interfer)

Tpreskew = Tpostskew = $\left\lfloor \frac{q}{2} \right\rfloor \left(L + m^2 b\right)$

Execution time for each step

At each step, each process executes one matrix multiplication mxm

- Computation time: m^3w

At each step, each process sends/receives a block of size mxm in its row of processes and in its column of processes

- All communications can occur simultaneously in the 4-port model

$$Time: L + m^2b$$

Then, the total time for the q steps is given by

$$T_{total} = q \ \ max\ (L + m^2b, m^3w)$$

Performance model for the Cannon algorithm

$$T(m,n) = 2 * \left\lfloor \frac{q}{2} \right\rfloor \left(L + m^2 b\right) + q \max(m^3 w, L + m^2 b)$$

This performance model can be easily adapted to other models

- If we suppose that we have mono-directional links, the $\left\lfloor \frac{q}{2} \right\rfloor$ becomes *(q-1)*
- If we suppose that we have a 1-port model, there is a factor 2 to add in front of communication terms
- If we suppose that there are no communication/computation overlap, then the maximum becomes a sum

Fox algorithm

This algorithm was originally developed to run on a hypercube topology

- But in fact it uses a grid, mapped on a hypercube

- It does not require any pre / post-skewing
- It is based on horizontal broadcast of the diagonals of matrix A and vertical shifts of matrix B

- Sometimes also called the **broadcast-multiply-roll** algorithm

Steps of the Fox algorithm

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Initial state

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{00}	A_{00}	A_{00}
A_{11}	A_{11}	A_{11}	A_{11}
A_{22}	A_{22}	A_{22}	A_{22}
A_{33}	A_{33}	A_{33}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Broadcast of the 1st diagonal of A (stored in a separate buffer)

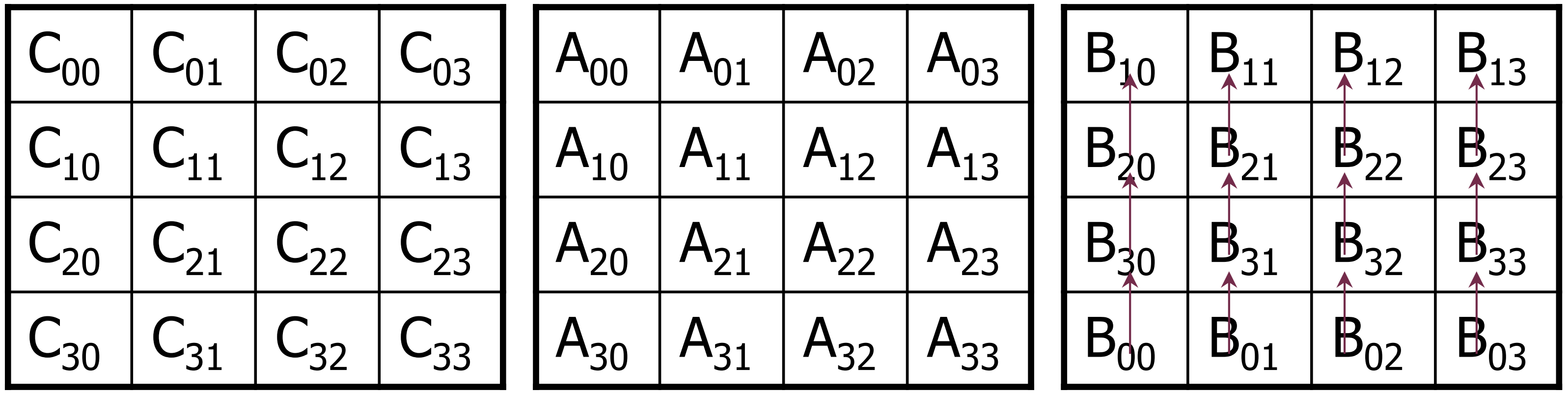
C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{00}	A_{00}	A_{00}
A_{11}	A_{11}	A_{11}	A_{11}
A_{22}	A_{22}	A_{22}	A_{22}
A_{33}	A_{33}	A_{33}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Local computations

Steps of the Fox algorithm, contd.

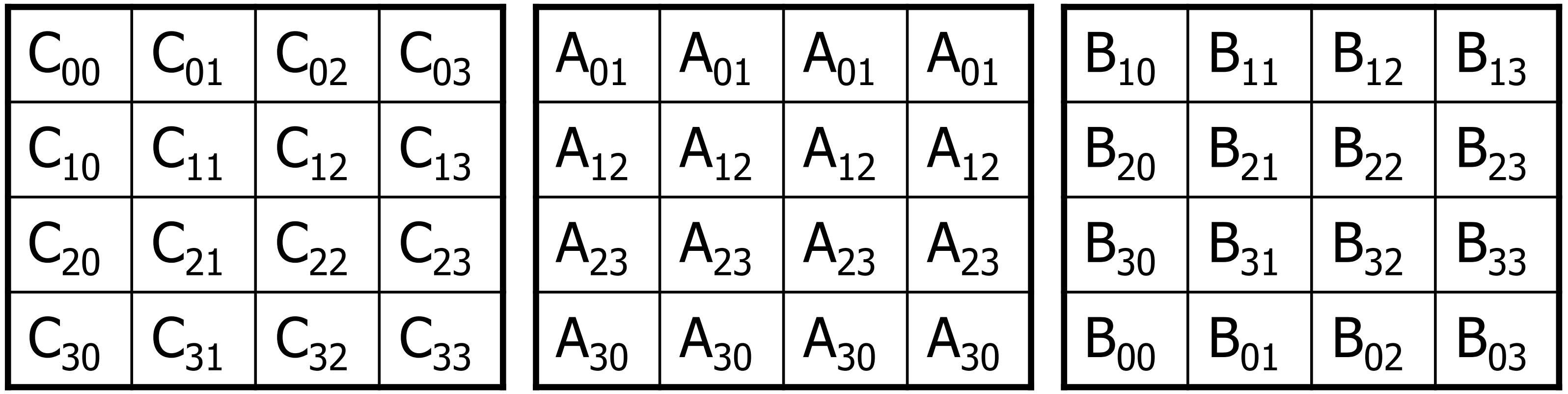


C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}
B_{00}	B_{01}	B_{02}	B_{03}

Shift of B

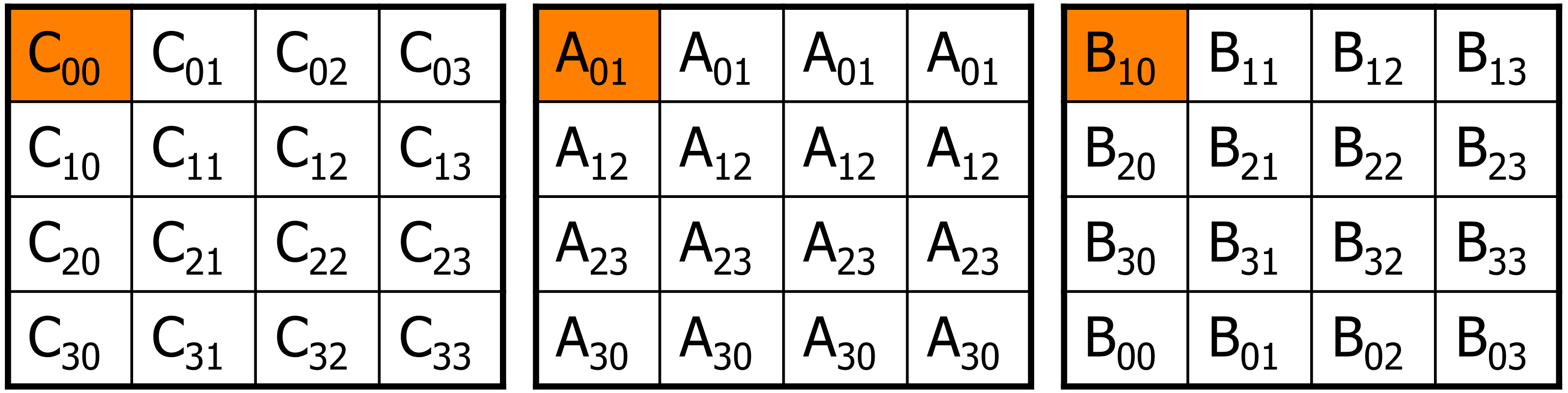


C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{01}	A_{01}	A_{01}	A_{01}
A_{12}	A_{12}	A_{12}	A_{12}
A_{23}	A_{23}	A_{23}	A_{23}
A_{30}	A_{30}	A_{30}	A_{30}

B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}
B_{00}	B_{01}	B_{02}	B_{03}

Broadcast of the 2nd diagonal of A (stored in a separate buffer)



C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{01}	A_{01}	A_{01}	A_{01}
A_{12}	A_{12}	A_{12}	A_{12}
A_{23}	A_{23}	A_{23}	A_{23}
A_{30}	A_{30}	A_{30}	A_{30}

B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}
B_{00}	B_{01}	B_{02}	B_{03}

Local computations

Fox algorithm

```
// No initial move
for k = 1 to q  in parallel
  Broadcast of the $k^{th}$ diagonal of A
  Local computation C = C + A*B
  Vertical shift of B
// No final move
```

- We need an additional array to store the diagonal blocks that are received on processes
- This is the array used for multiplication A * B

Snyder algorithm (1992)

- A bit more complicated that Fox et Cannon's algorithms
- Start by transposing matrix B
- Uses reductions (sums) over rows of matrix C
- And shifts for matrix B

Steps of the Snyder's algorithm

Initial state

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Transpose B

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{10}	B_{20}	B_{30}
B_{01}	B_{11}	B_{21}	B_{31}
B_{02}	B_{12}	B_{22}	B_{32}
B_{03}	B_{13}	B_{23}	B_{33}

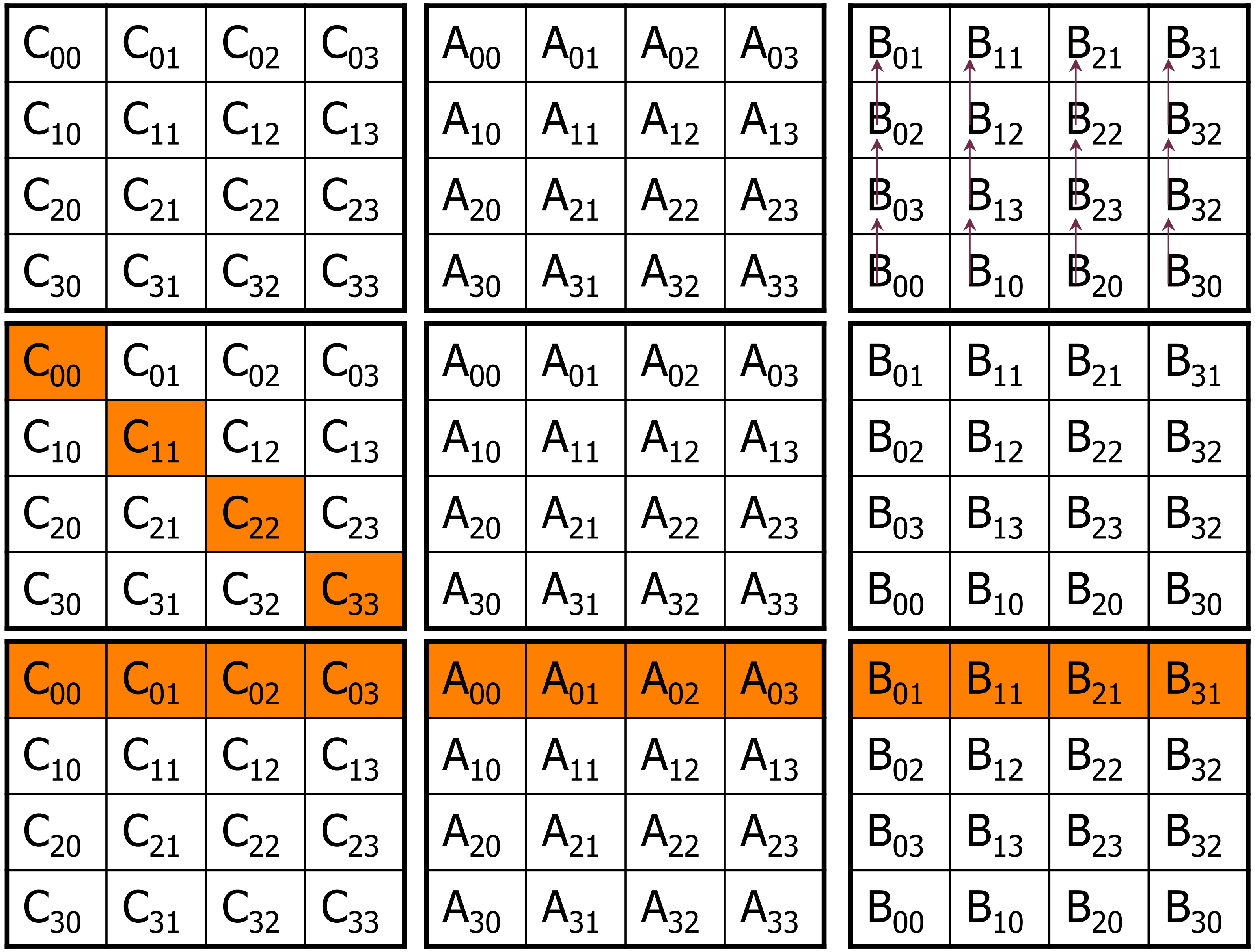
Local computations

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{10}	B_{20}	B_{30}
B_{01}	B_{11}	B_{21}	B_{31}
B_{02}	B_{12}	B_{22}	B_{32}
B_{03}	B_{13}	B_{23}	B_{33}

Steps of the Snyder's algorithm, contd.

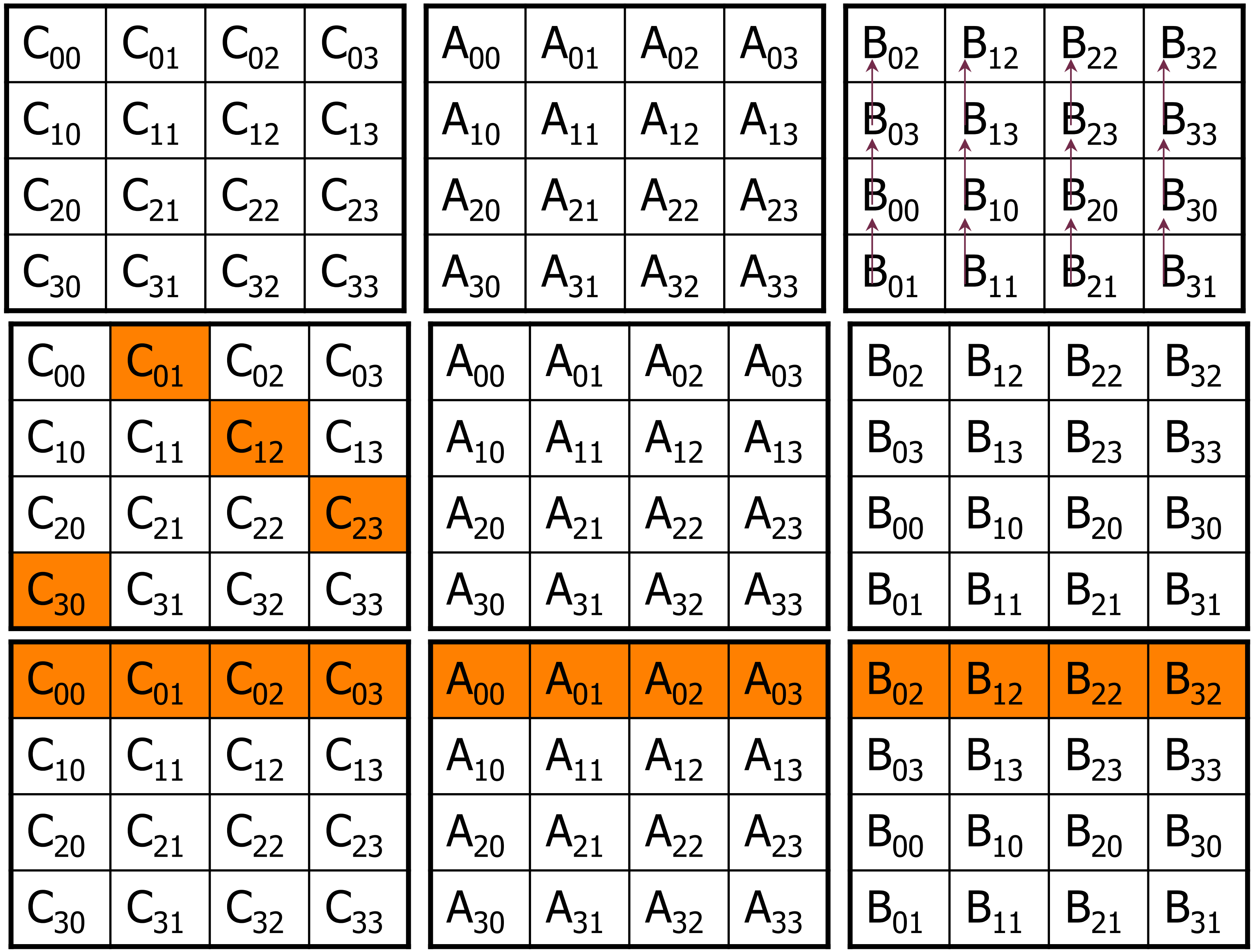


Shift of B

Global sum over rows of C

Local computations

Steps of the Snyder's algorithm, contd.



C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{02}	B_{12}	B_{22}	B_{32}
B_{03}	B_{13}	B_{23}	B_{33}
B_{00}	B_{10}	B_{20}	B_{30}
B_{01}	B_{11}	B_{21}	B_{31}

Shift of B

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{02}	B_{12}	B_{22}	B_{32}
B_{03}	B_{13}	B_{23}	B_{33}
B_{00}	B_{10}	B_{20}	B_{30}
B_{01}	B_{11}	B_{21}	B_{31}

Global sum over rows of C

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

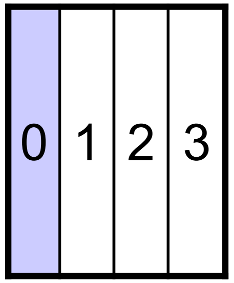
B_{02}	B_{12}	B_{22}	B_{32}
B_{03}	B_{13}	B_{23}	B_{33}
B_{00}	B_{10}	B_{20}	B_{30}
B_{01}	B_{11}	B_{21}	B_{31}

Local computations

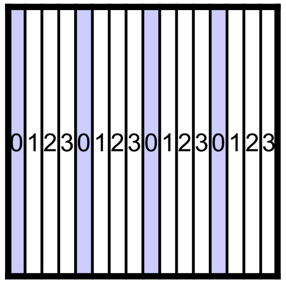
Snyder's algorithm

```
var A,B,C: array[0..m-1][0..m-1] of real
var bufferC: array[0..m-1][0..m-1] of real

Transpose B

MatrixMultiplyAdd(bufferC, A, B, m)
Vertical shift of B
For k = 1 to q-1
      Global sum of bufferC over rows of proc in $C_{i,(i+k-1)\%q}$
      MatrixMultiplyAdd(bufferC, A, B, m)
      Vertical shift of B
Global sum of bufferC over rows of proc in $C_{i,(i+k-1)\%q}$

Transpose B
```

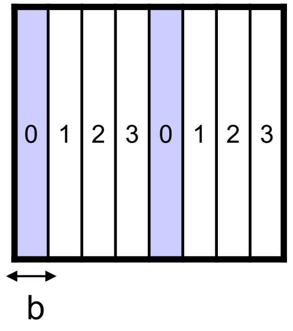
Some storage strategies for dense matrices



1) 1D Block columns

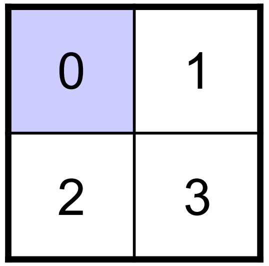


2) Columns 1D cyclic

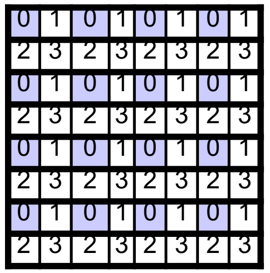


3) Block columns 1D cyclic

4) Row versions of previous storage strategies



5) Block 2D rows and columns



Generalization of the other ones

6) Block 2D rows and columns cyclic

Which data storage?

We have seen

- Block distributions
- 1D distributions
- 2D distributions
- Cyclic distributions

• What is the best choice?

• 2D block cyclic distribution is the Swiss army knife of the dense matrix distribution!

2-D block-cyclic distribution

Goal:

- Trying to get at the same time the advantages of 1D horizontal and vertical block-cyclic distributions
- Works whatever the progresses of the computations are
  Left-to-right, top-to-bottom, wave, …

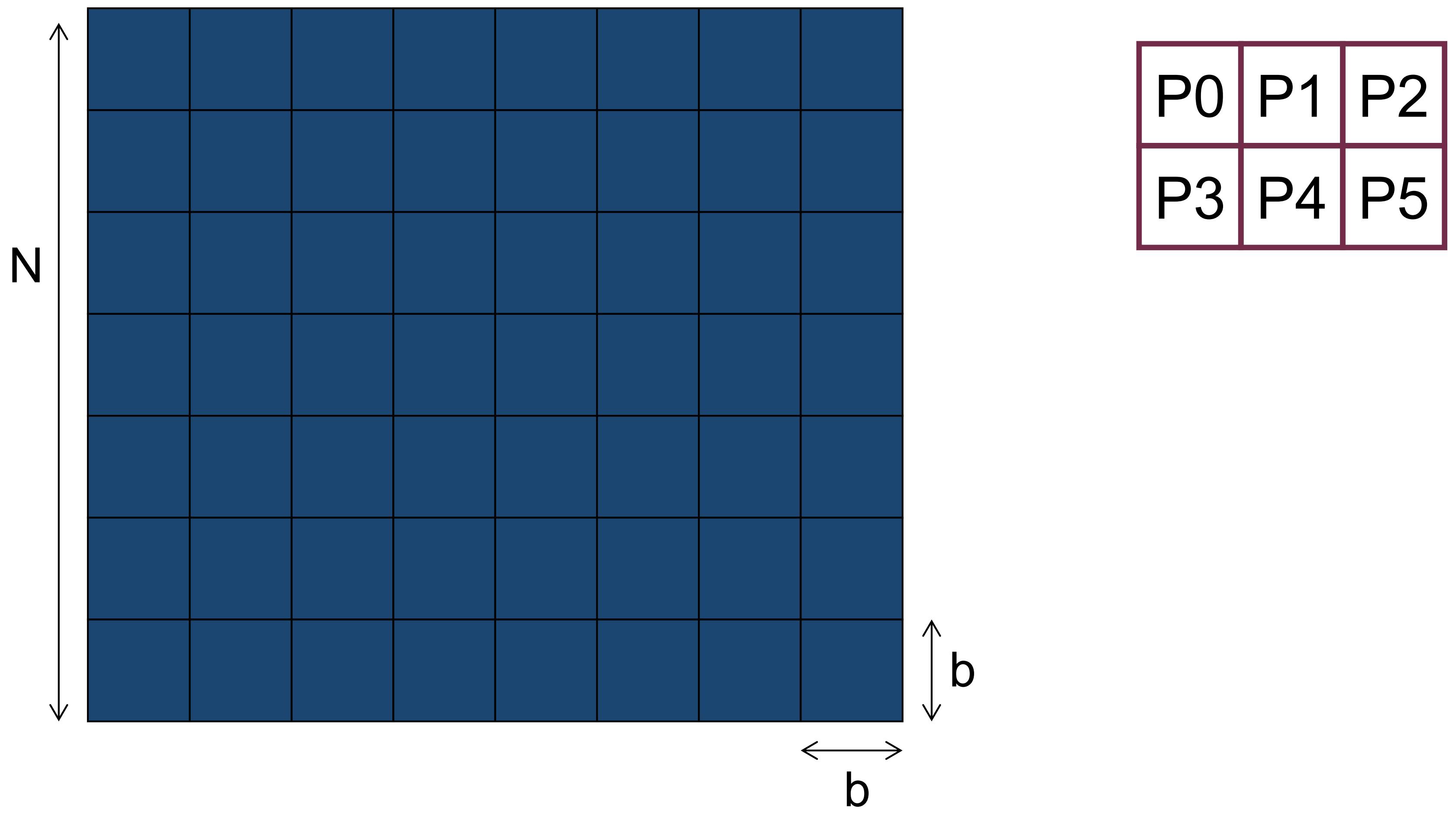
Consider the number of processes $p = r * c$

- Arranged like a $r * c$ matrix

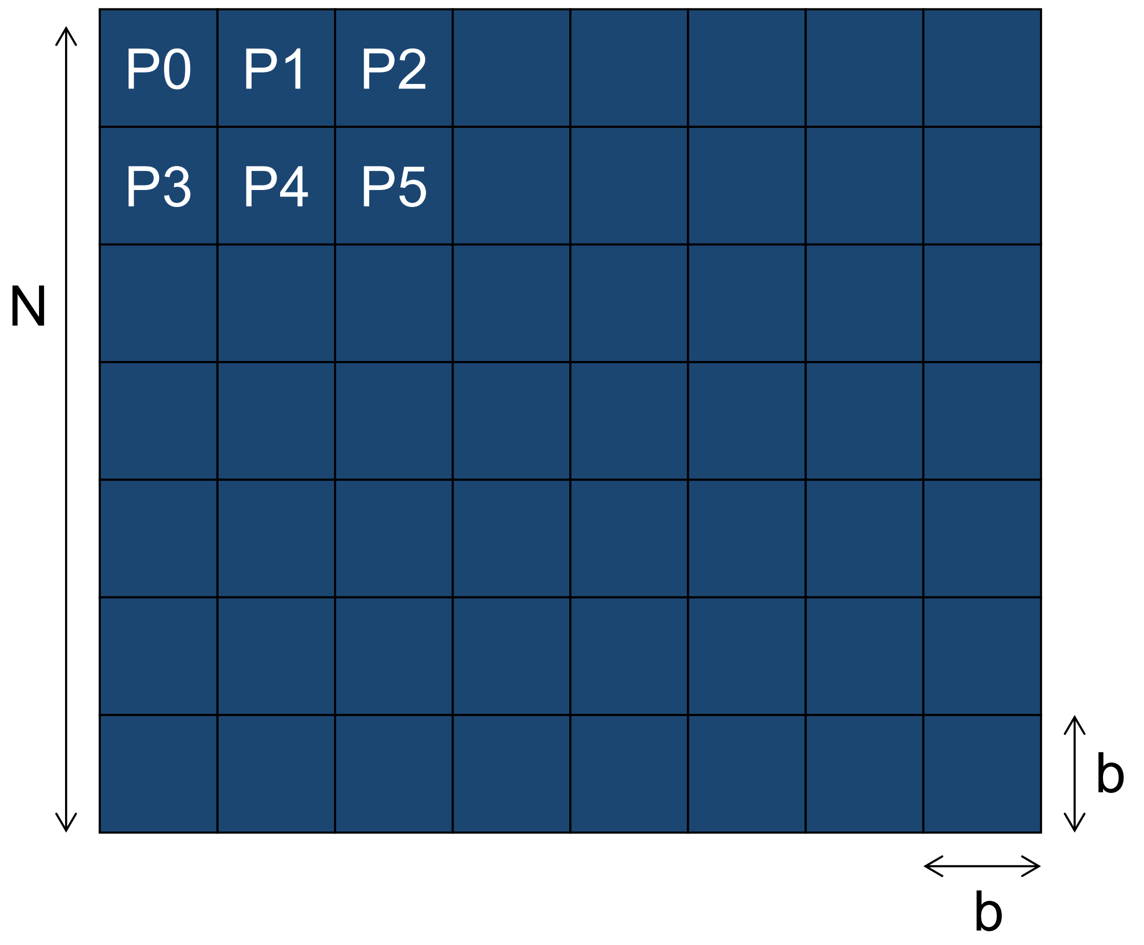
Consider a 2D matrix of size NxN

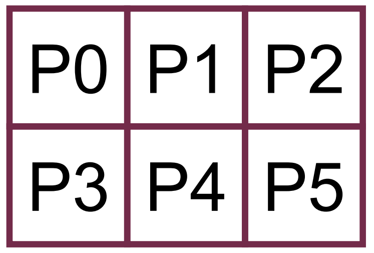
Consider a block size b (that divides N)

2-D block-cyclic distribution

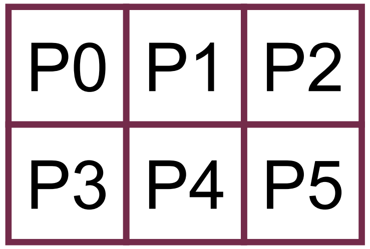


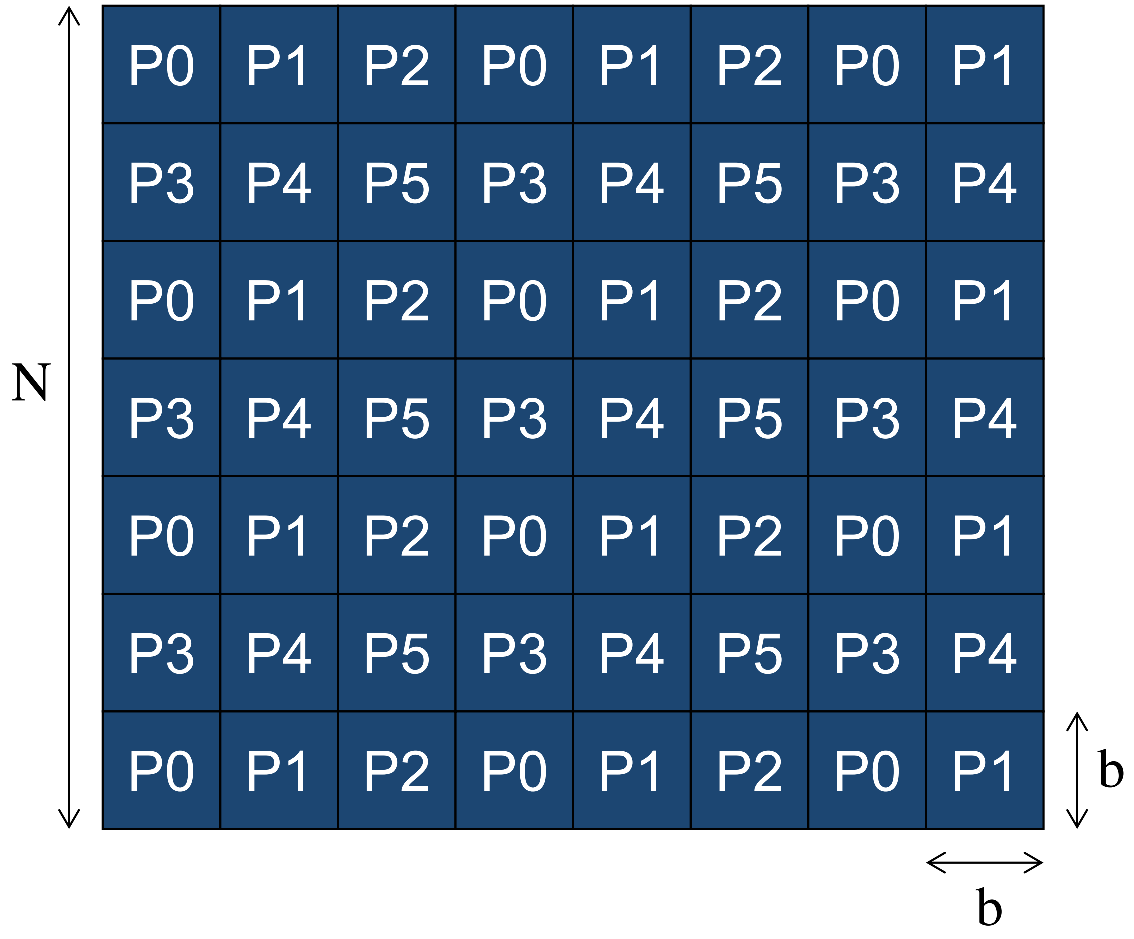
2-D block-cyclic distribution, contd.





2-D block-cyclic distribution, contd.



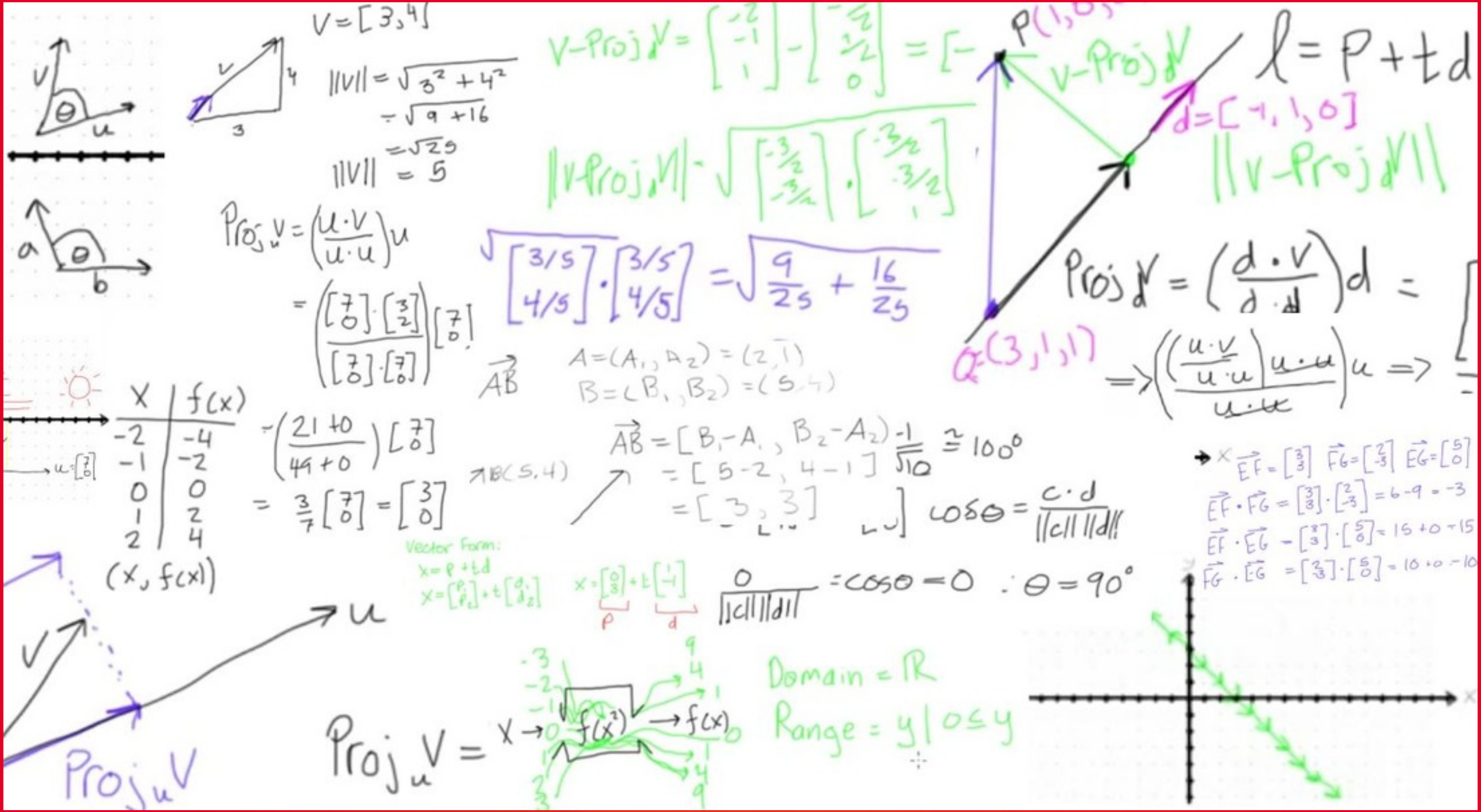


Slight disequilibrium

- Can be neglected with a large number of blocks

Indexed computations have to be implemented in other functions

+ functions that give the neighbors from a given position in the grid



INVENTEURS DU MONDE NUMÉRIQUE