Parallel Systems Course: Chapter VI

Dense Matrix Algorithms

KUMAR Chapter 8

Jan Lemeire
Parallel Systems lab
November 2017

Vrije Universiteit Brussel

Overview

1. Matrix-vector Multiplication
2. Matrix-matrix Multiplication
3. Shared-memory

Utility of Matrix Algorithms

Applied in several numerical and non-numerical contexts:

- 3D image calculations
- Solving (linear) equations
- Simulations of physical systems

E.g.: makes the basis of LINPACK, a software library for performing numerical linear algebra, by using the BLAS (Basic Linear Algebra Subprograms) libraries for performing basic vector and matrix operations

Dense versus Sparse Matrices

Dense matrices: have no or few known zero entries

Sparse matrices: are populated primarily with zeros

- often appear in science or engineering when solving partial differential equations
- easily compressed
- very large sparse matrices are impossible to manipulate with the standard algorithms, due to memory limitations
- special versions of the algorithms are necessary and are more efficient

Matrix – Vector Multiplication

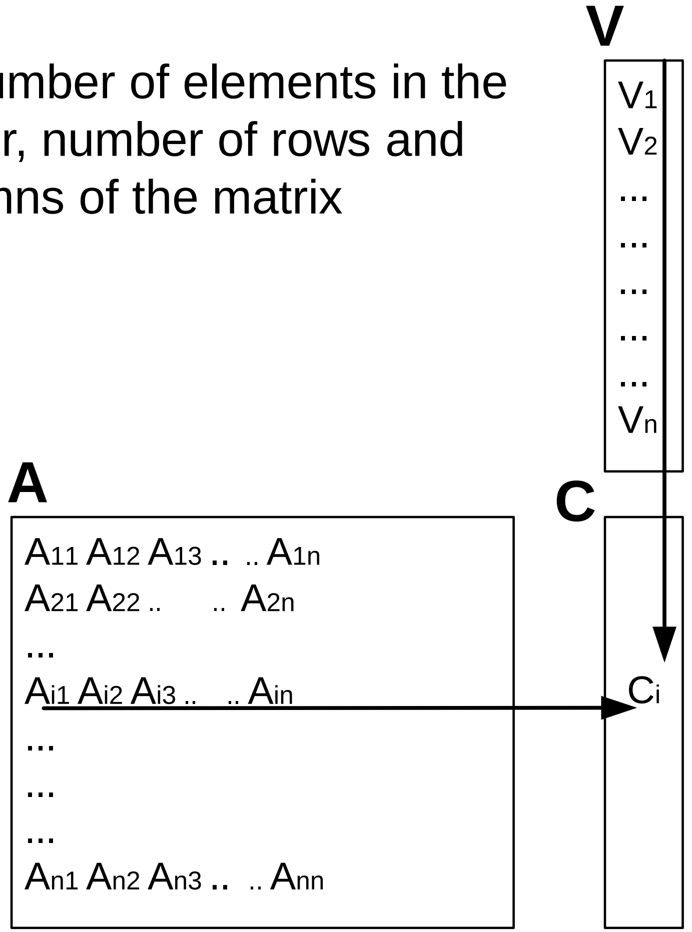
$$C = A \times V$$

$$C_i = \sum_{k=1}^{n} A_{ik} . V_k \quad (i : 1..n)$$

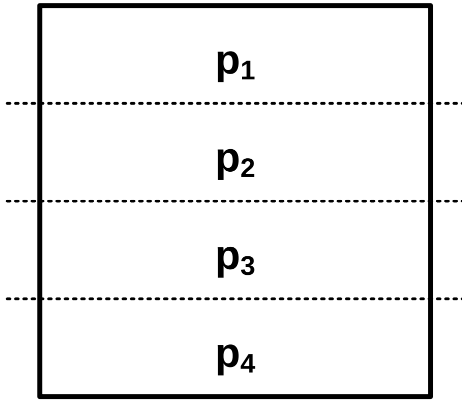
$$T_{seq} = \delta_{mm} . n^2$$

n : number of elements in the vector, number of rows and columns of the matrix

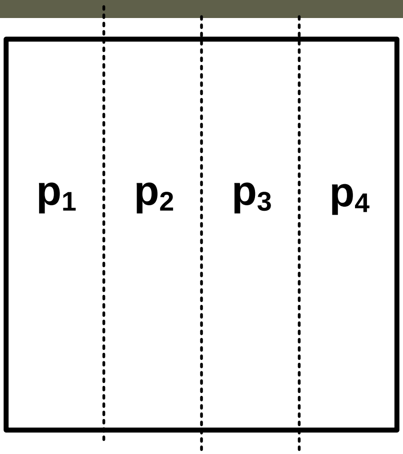
```
for (i=0; i<n; i++){
   C[i]=0;
   for (k=0; k<n; k++){
      C[i]+=A[i, k]*V[k];
   }
}
```



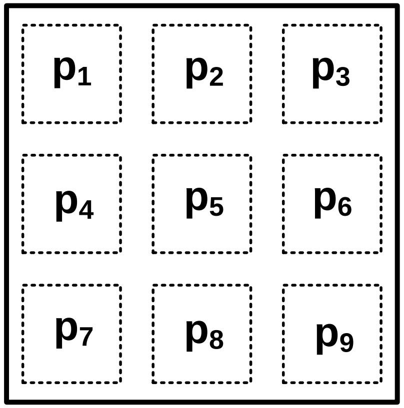
Matrix/Vector Partitioning?



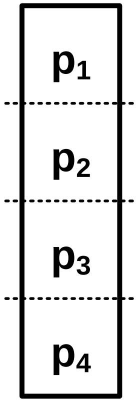
Row-wise block striping



Column-wise block striping



Checkerboard partitioning



Vector partitioning

Data & results distributed

Now: ***distributed memory solutions***

- In parallel applications, data remains distributed while operations (such as vector-matrix operations) are performed on them.

- In the following we assume that the data is already distributed among the processors

Parallel M x V Multi-plication Version 1

(n=p)

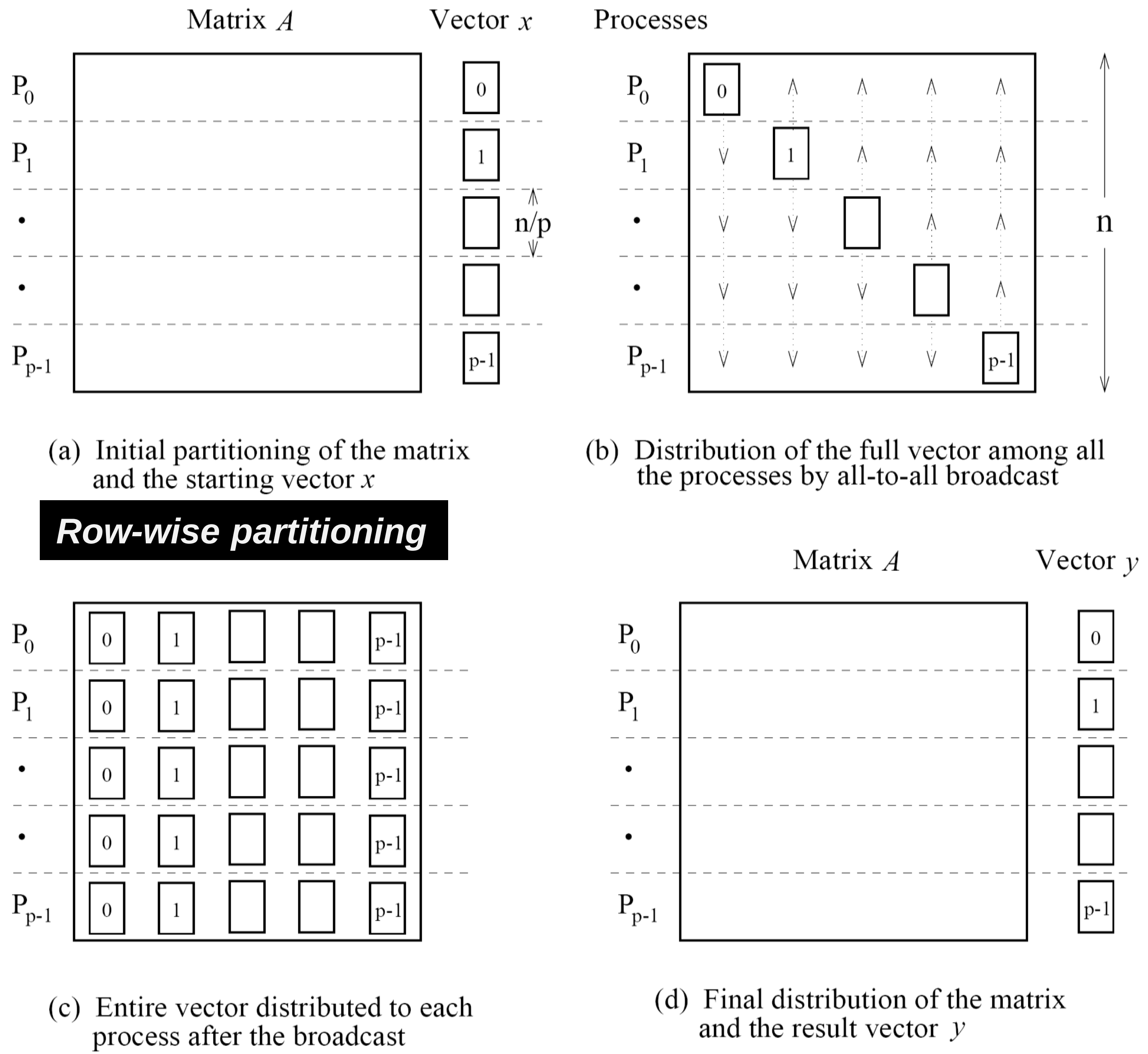


Figure 8.1 Multiplication of an $n \times n$ matrix with an $n \times 1$ vector using rowwise block 1-D partitioning. For the one-row-per-process case, $p = n$.

Parallel M x V Multiplication Version 1 $p < n$

n/p rows of matrix and n/p elements of vector per processor

Parallel M x V Multi-plication Version 2

($n^2=p$)

checkerboard partitioning

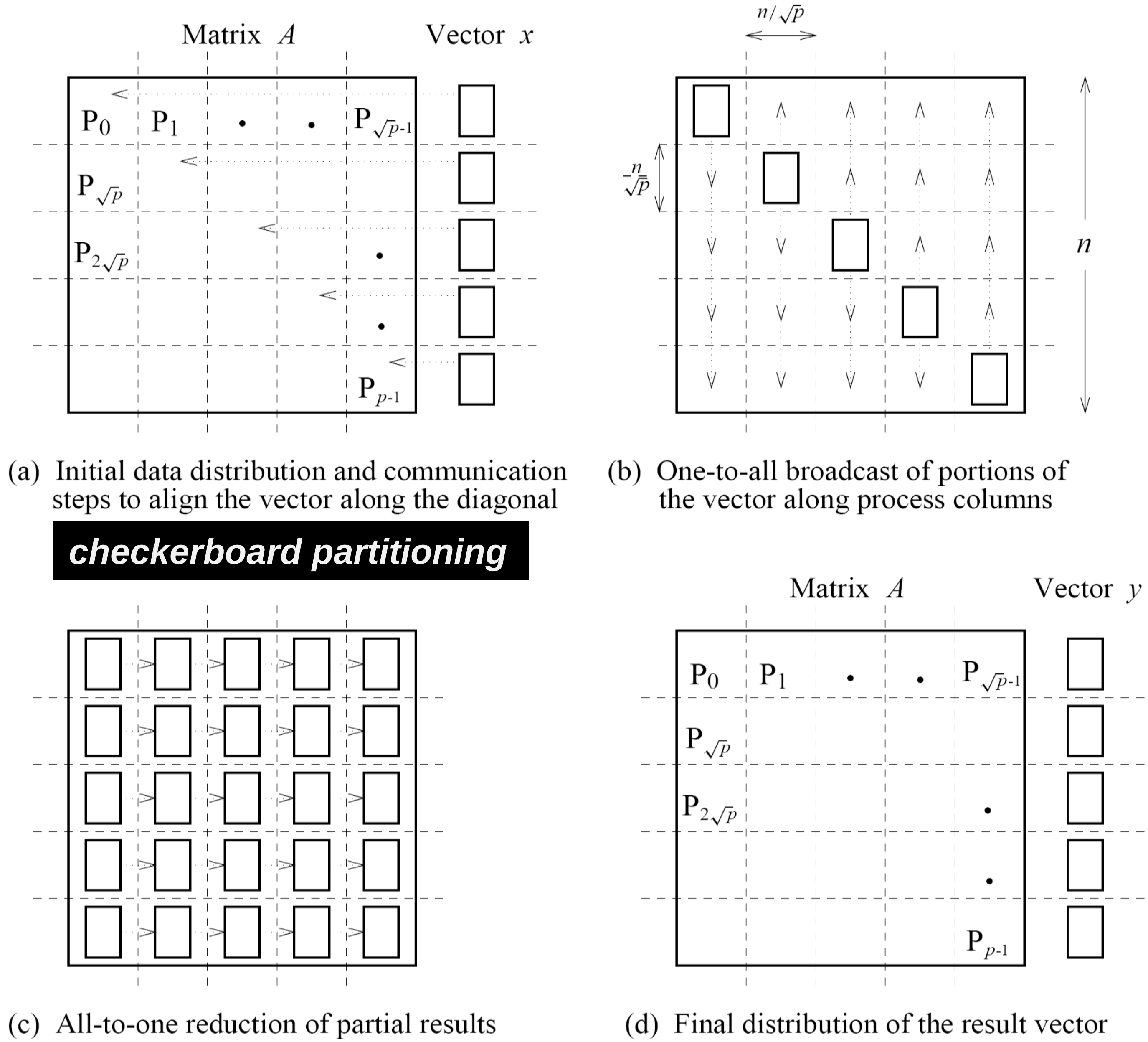


Figure 8.2 Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, $p = n^2$ if the matrix size is $n \times n$.

Parallel M x V Multiplication Version 2 $p < n^2$

$(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks of matrix and $n/\sqrt{p}$ elements of vector per processor

Overview

1. Matrix-vector Multiplication
2. Matrix-matrix Multiplication
3. Shared-memory

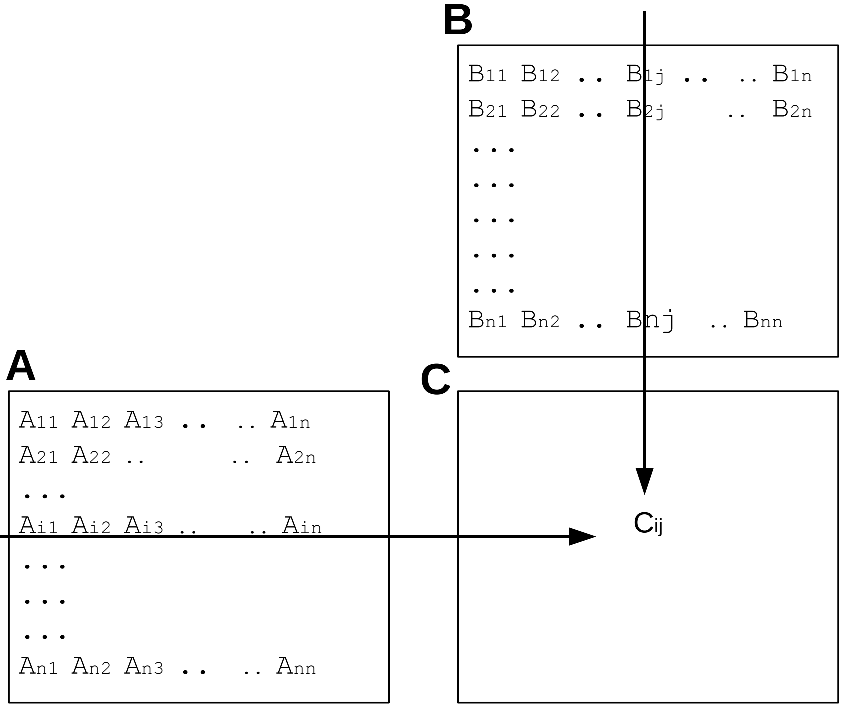
Matrix Multiplication

$$C = A \times B$$

$$C_{ij} = \sum_{k=1}^{n} A_{ik} . B_{kj} \quad (i, j : 1..n)$$

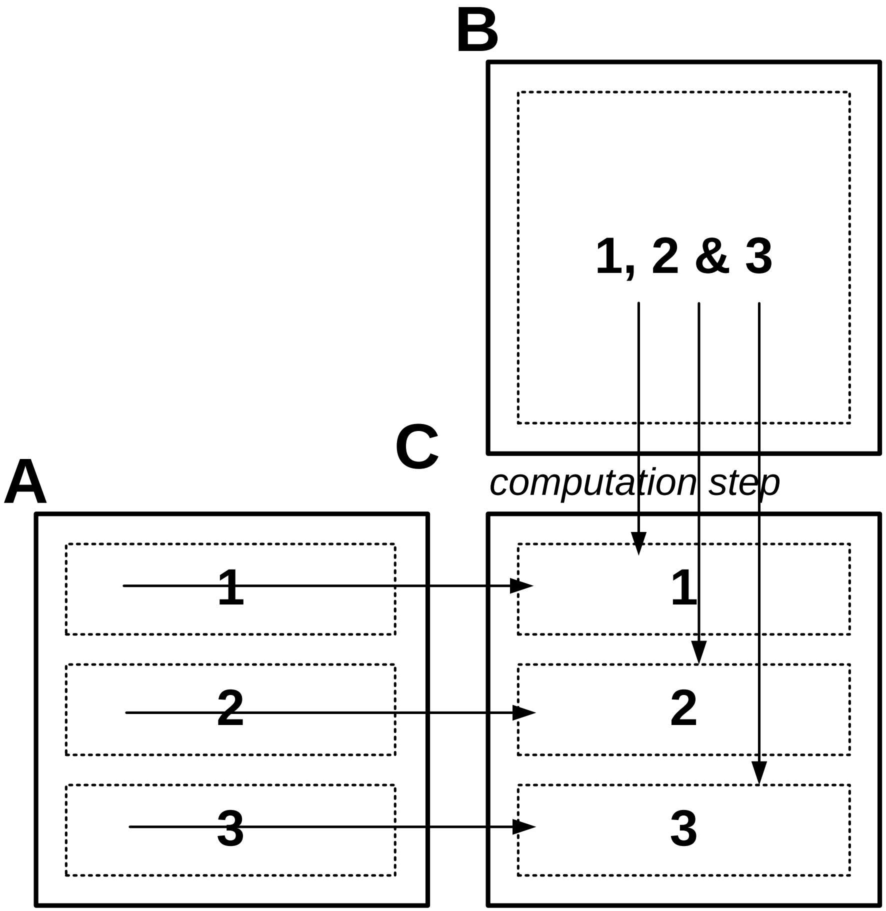
$$T_s = \delta_{mm} . n^3$$

```
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]+=A[i, k]*B[k,j];
        }
    }
}
```

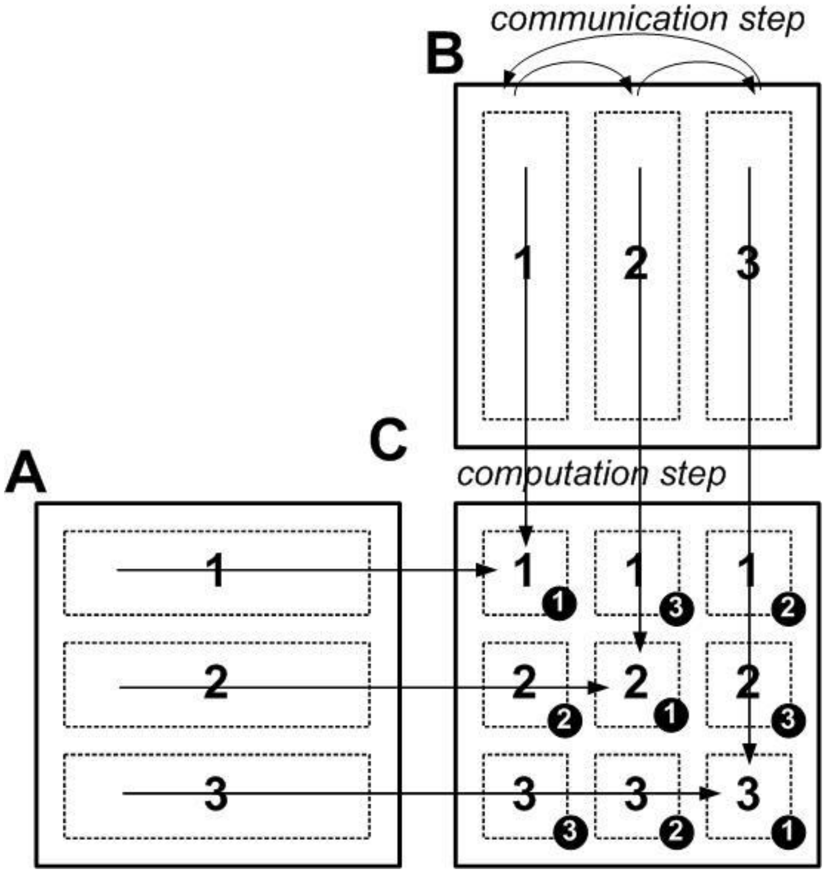


MxM: one-step version

- B is sent to all processors
- Computation in 1 step
- Same amount of communication

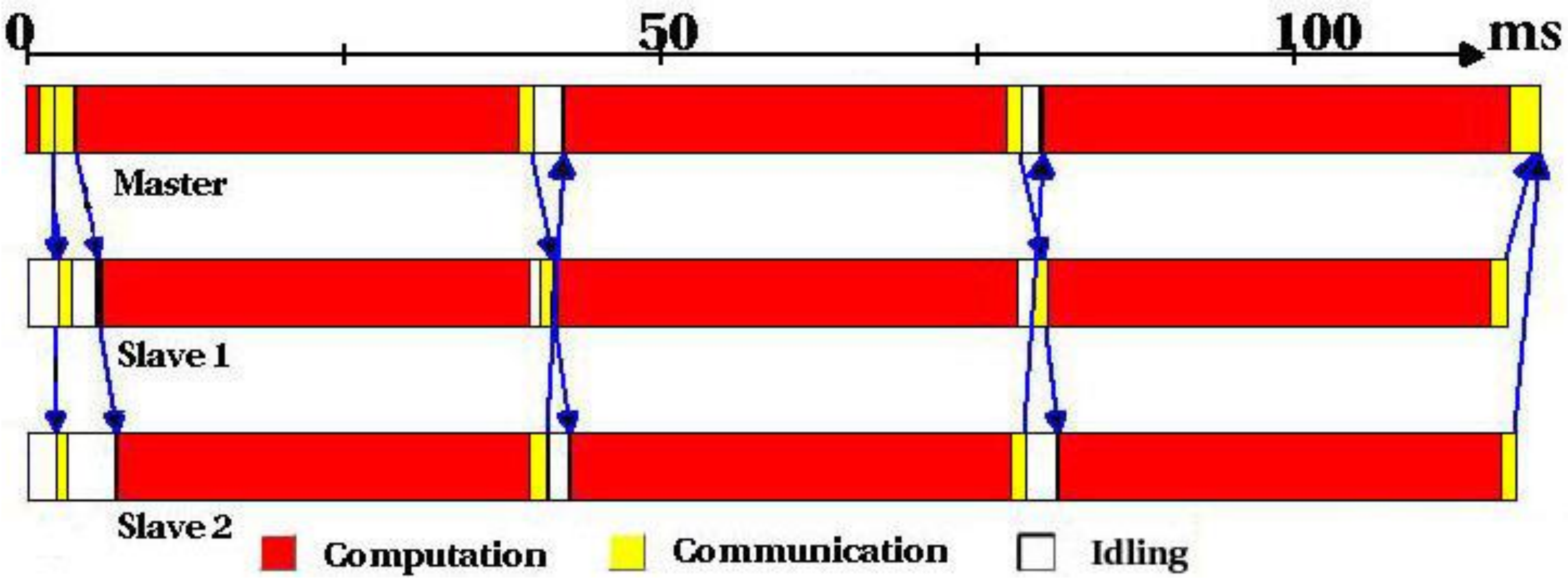


Alternate shift-compute version



- Algorithm alternates p computation and communication steps
- Computation step: each processor multiplies its A submatrix with its B submatrix, resulting in a submatrix of C. The black circles indicate the step in which each submatrix is computed.
- After multiplication: processor sends it B submatrix to next processor and receives one from the preceding processor. The communication forms a **circular shift operation**.

Parallel MxM: Execution Profile



Speedup=2.55 Efficiency = 85%

Note that the communication after the first step can be hidden behind the computation (Mehdi Moghaddmfar 2017).
MPI-2 supports non-blocking collective communication operations.

Theoretical Analysis of Matrix Multiplication

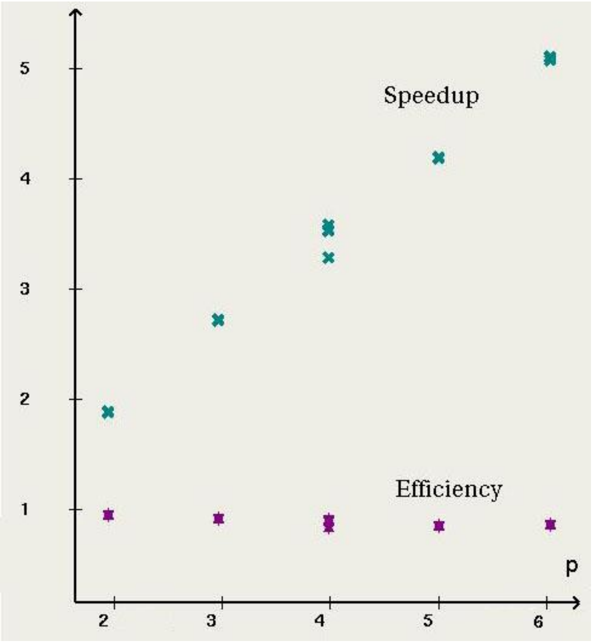
- Computation time (slave) $T^{i}_{work} = \frac{n^3}{p}.\delta_{mm}$
- Communication time (slave) – including data distribution (initially 2x1/p, each round 1/p [p-1 rounds] + 1/p result)

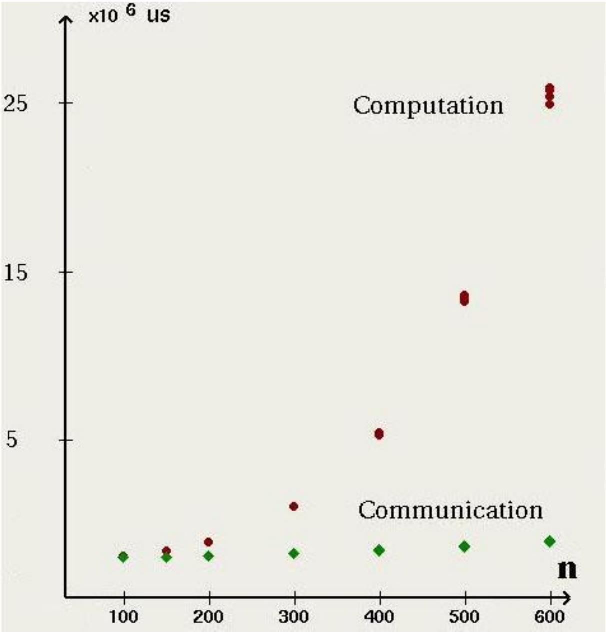
$$T^{i}_{comm} = \cancel{(p+1).t_s} + (1 + \frac{2}{p}).n^2.t_w$$

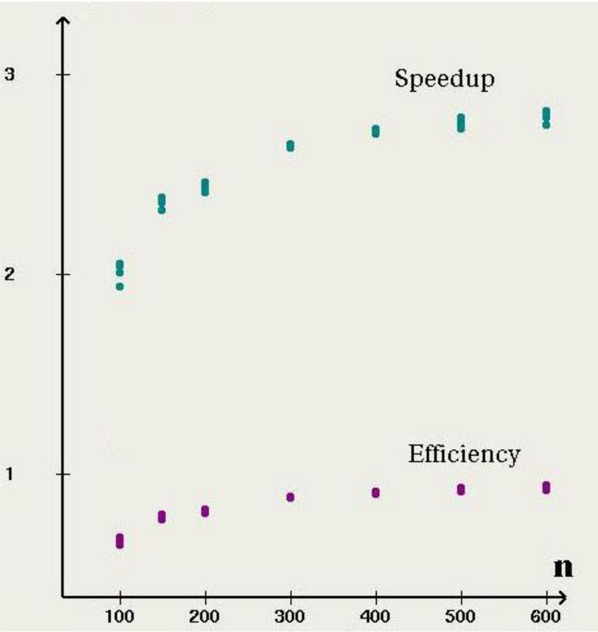
➡ Overhead ratio=overhead/computation = O(n^2)/O(n^3/p) = O(p/n)

➡ **Scalable** since efficiency remains constant if we increase both *p* and *n*

Parameter Dependence of Matrix Multiplication



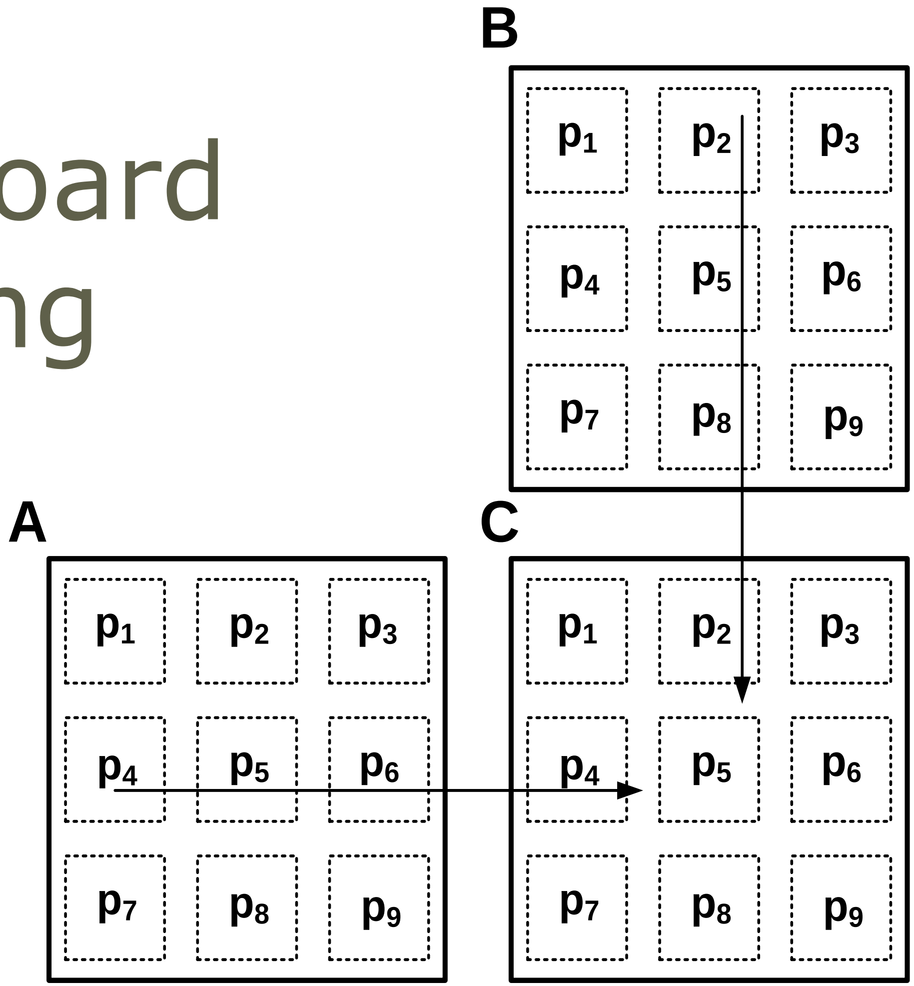




n: work size, here: matrix size

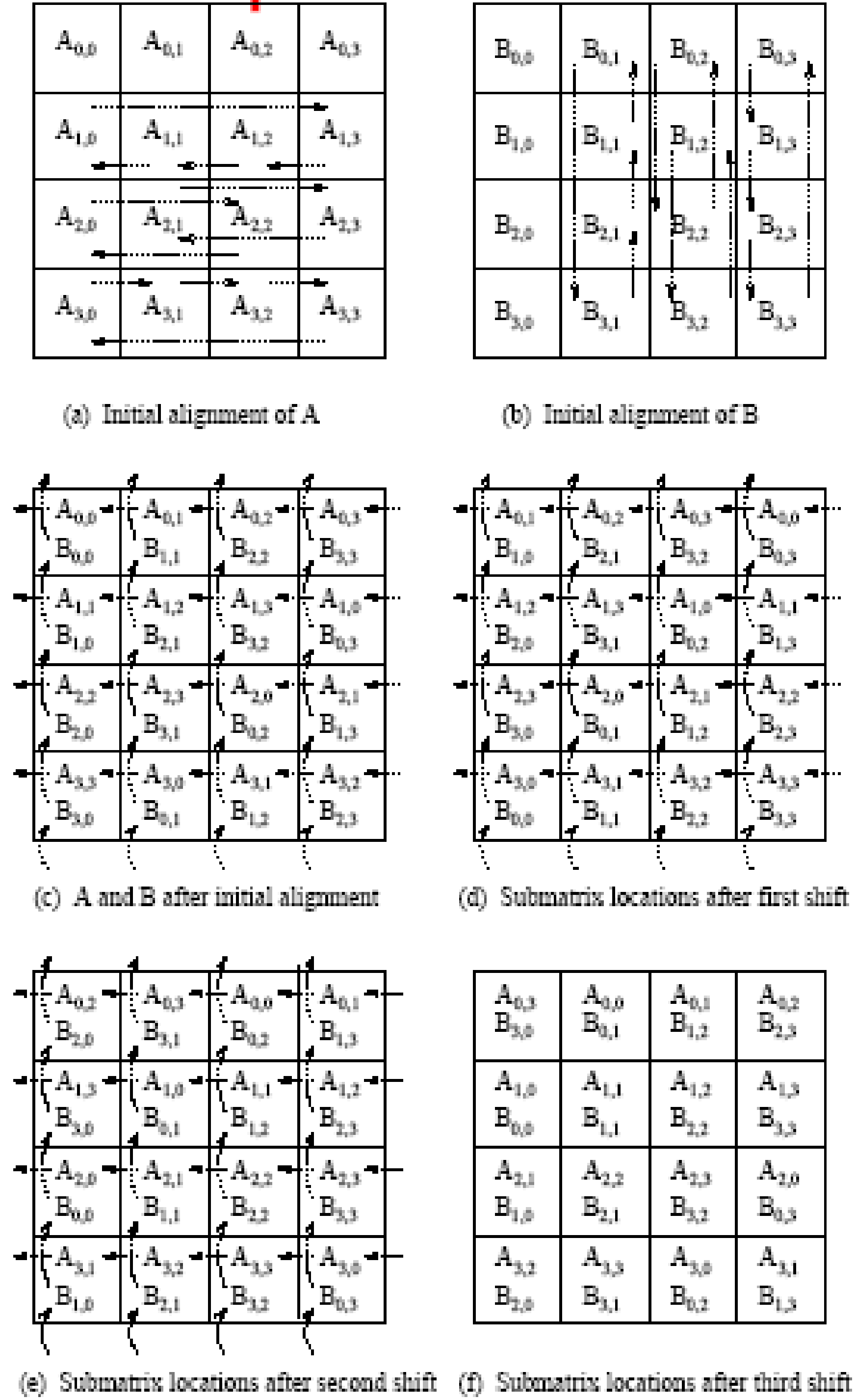
V3: Cannon’s algorithm

Checkerboard partitioning



Cannon's parallel MxM

Communication steps on 16 processes



(a) Initial alignment of A

(b) Initial alignment of B

(c) A and B after initial alignment

(d) Submatrix locations after first shift

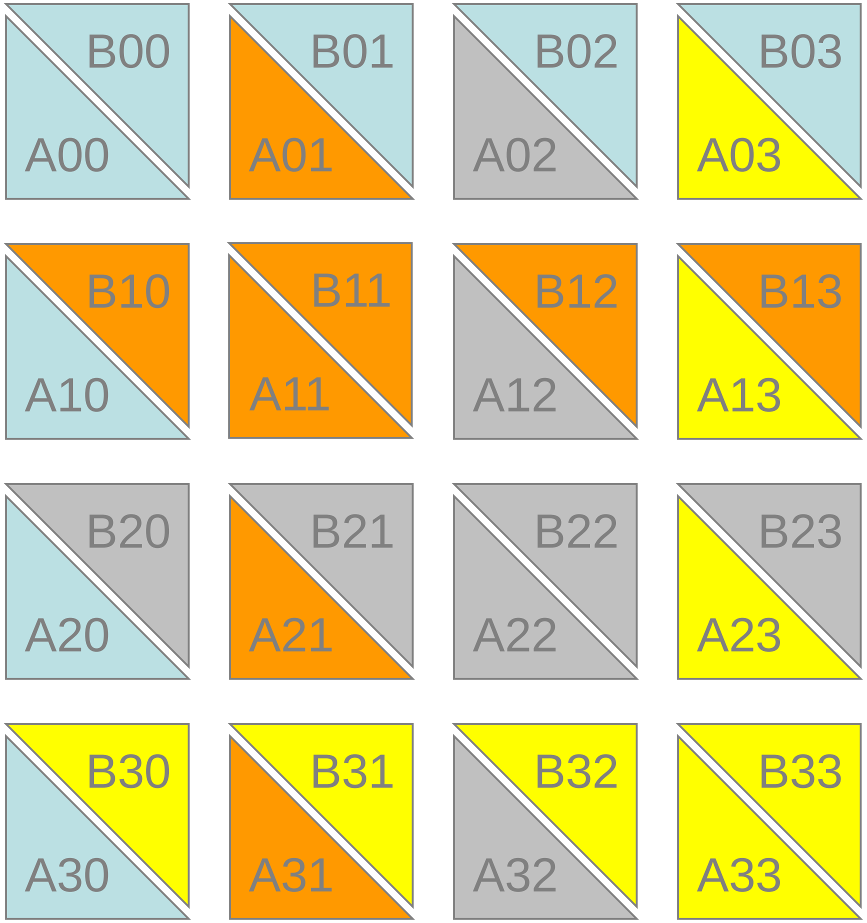
(e) Submatrix locations after second shift

(f) Submatrix locations after third shift

Initially, blocks Need to Be Aligned

Each triangle represents a matrix block

Only same-color triangles should be multiplied

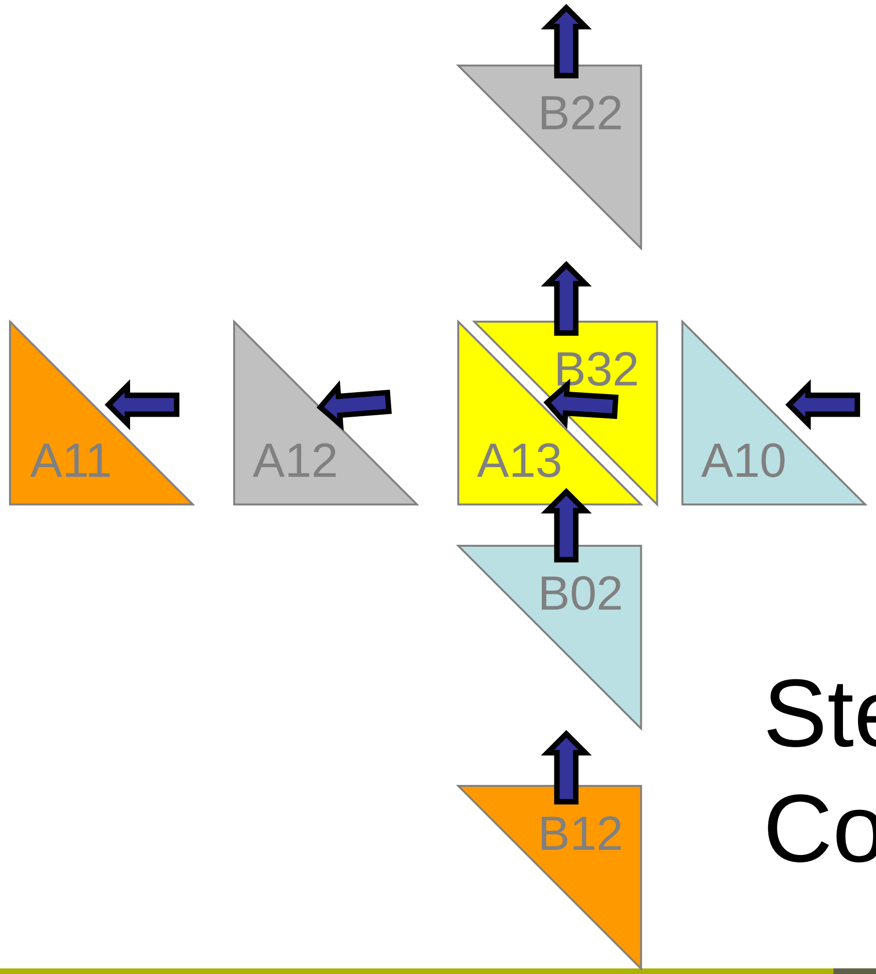


Rearrange Blocks

Block A_{ij} cycles left *i* positions

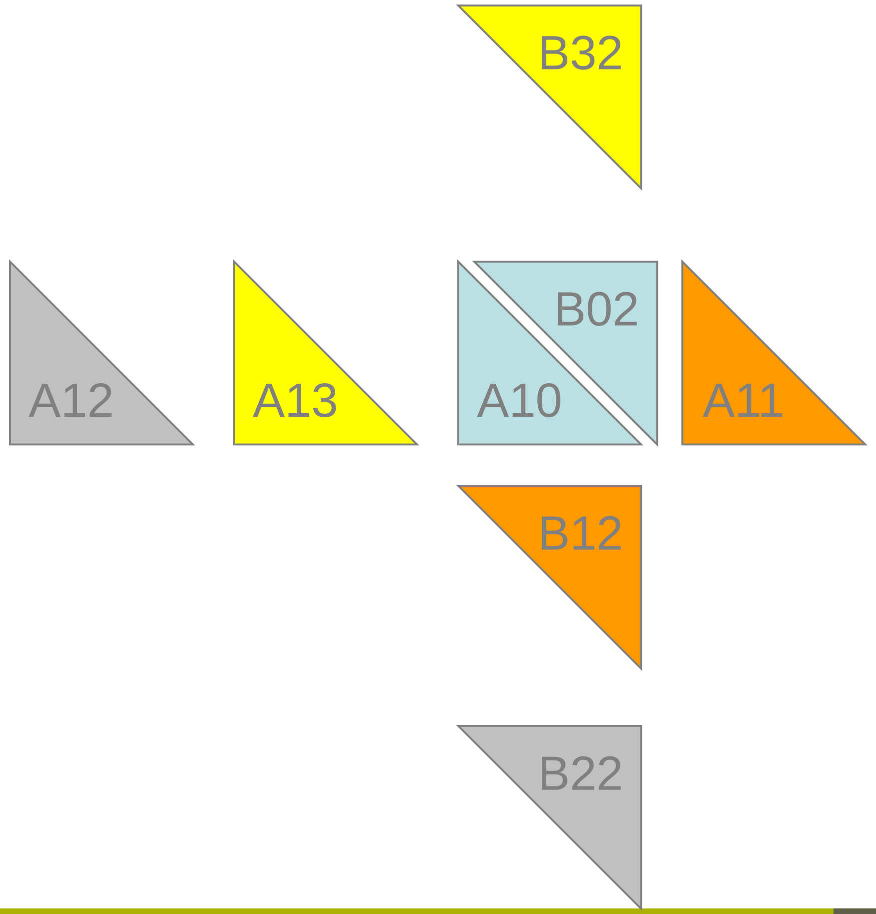
Block B*ij* cycles up *j* positions

Consider Process $P_{1,2}$



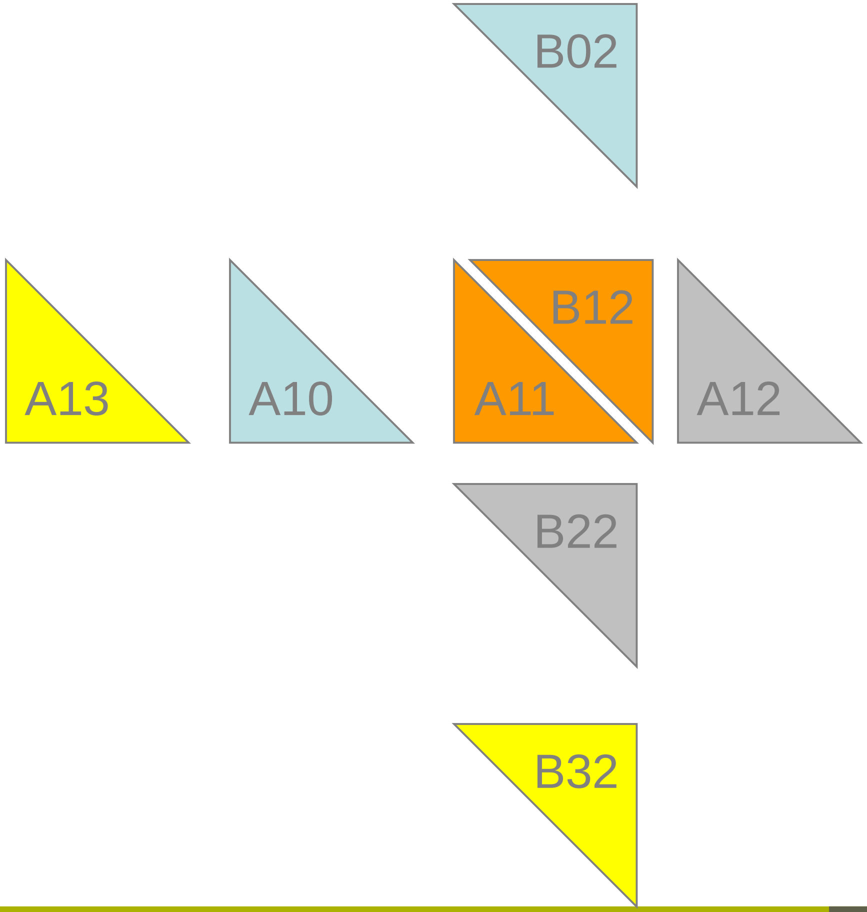
Step 1:
Computation & shift

Consider Process $P_{1,2}$



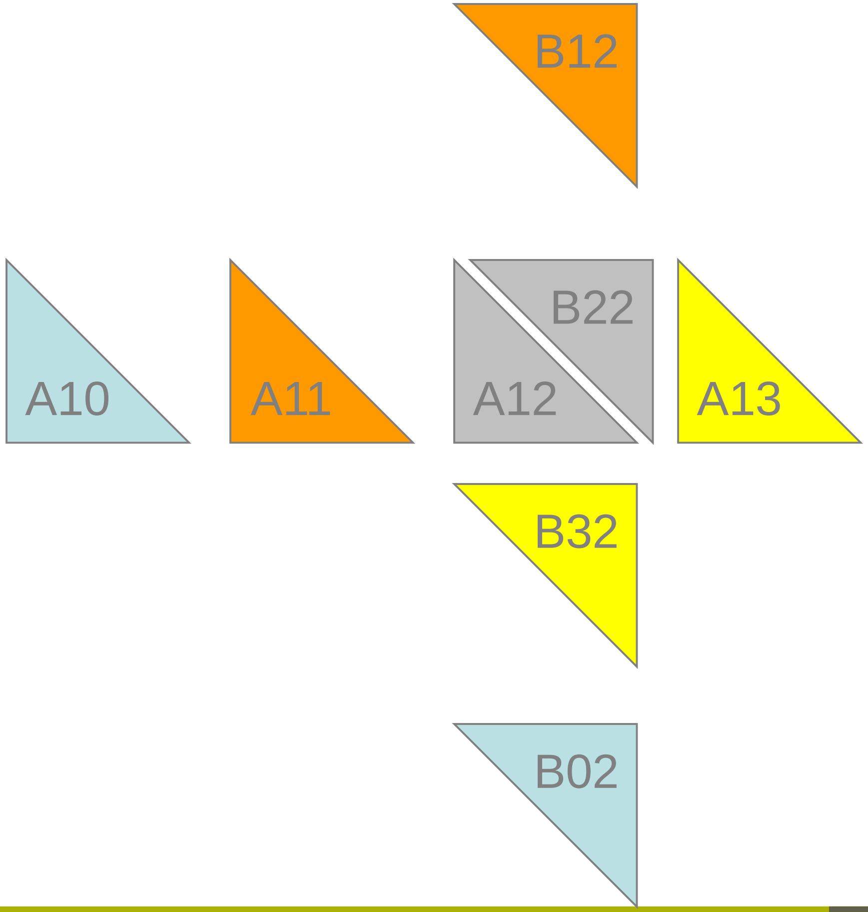
Step 2

Consider Process $P_{1,2}$



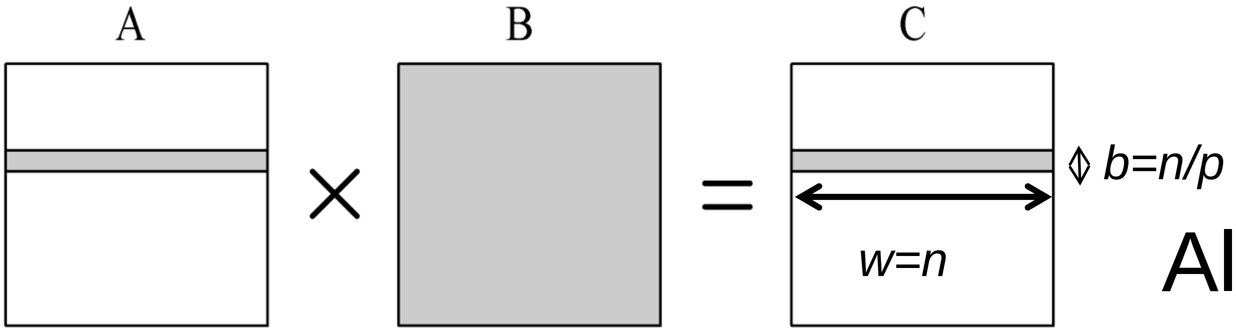
Step 3

Consider Process $P_{1,2}$



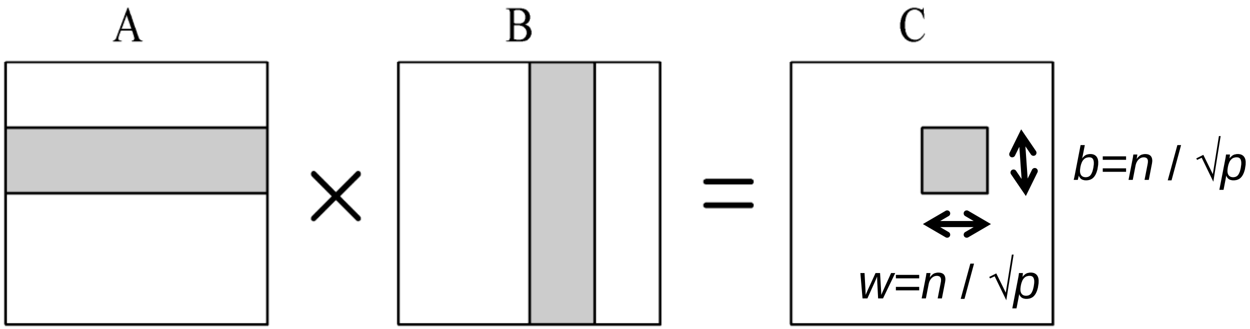
Step 4

Elements of A and B Needed to Compute the process's portion of C



(a)

Algorithm 1 & 2



(b)

Cannon's Algorithm

Why Cannon's requires less communication

- Amount of **computations** per process: n^3/p=breadth.width.n/p ~ *surface*
- Amount of **communication** per process: breadth.n+width.n=(breadth+width).n ~ *circumference*
  - Version 1 & 2: n^2/p (A-matrix) + n^2 (B-matrix) = (n+n/p).n
  - Cannon: $n^2/\sqrt{p}$ (A-matrix) + $n^2/\sqrt{p}$ (B-matrix) = $(2n/\sqrt{p})$.n
- Granularity = computations / communication

  = surface / circumference

  should be minimized

  for a given surface, circumference is minimal for a **square**

=> Cannon (square) is optimal and better than rectangle (v1&2)

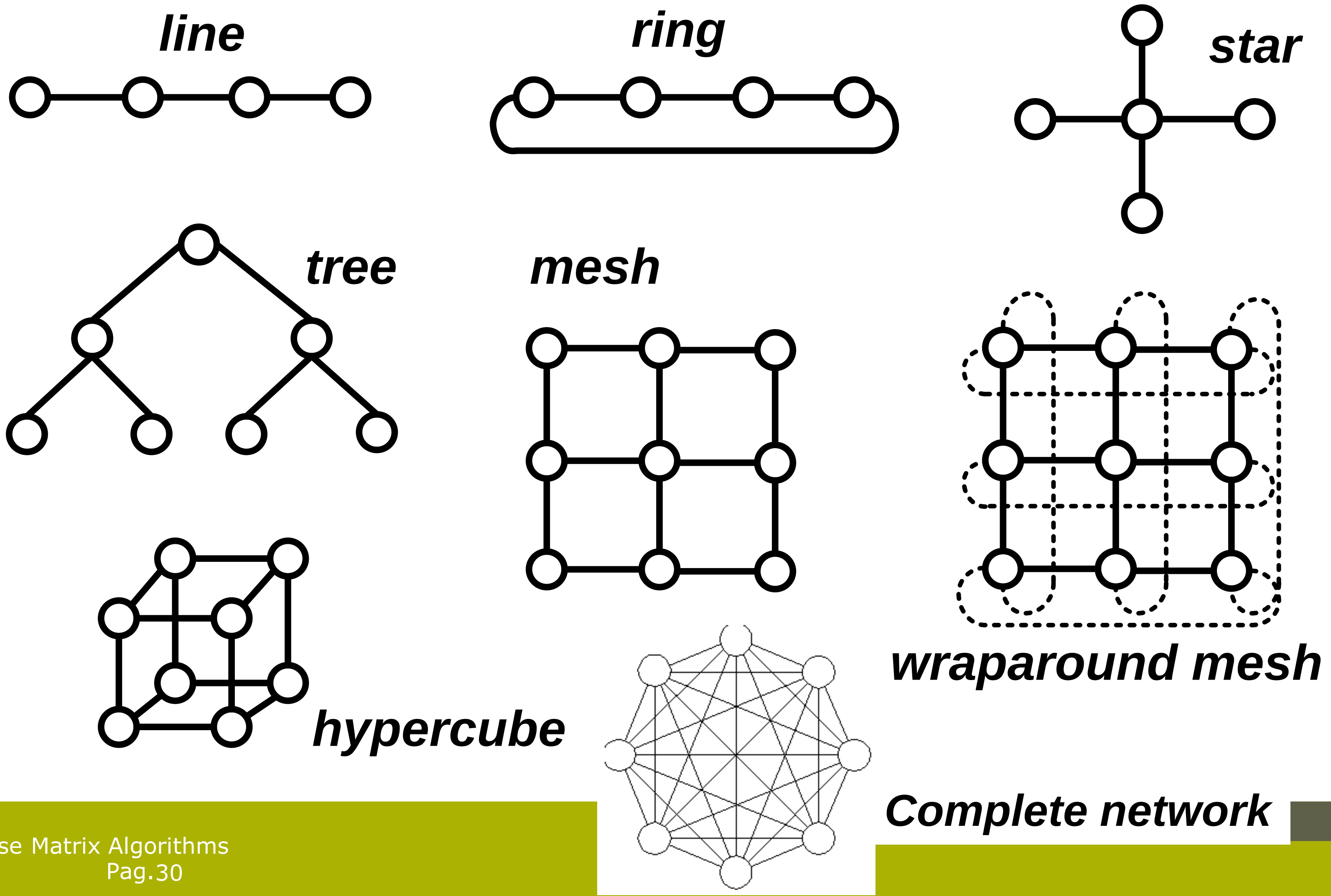
Complexity Analysis

- Algorithm has $\sqrt{p}$ iterations
- During each iteration process multiplies two $(n / \sqrt{p}) \times (n / \sqrt{p})$ matrices: $\Theta(n^3 / p^{3/2})$
- **Computational complexity**: $\Theta(n^3 / p)$ [the same]
- During each iteration process sends and receives two blocks of size $(n / \sqrt{p}) \times (n / \sqrt{p})$
- **Communication complexity**: $\Theta(n^2 / \sqrt{p})$ [lower!]

➡ Overhead ratio= $O(n^2 / \sqrt{p}) / O(n^3/p) = O(\sqrt{p}/n)$

➡ **Super-scalable** since efficiency drops if we increase both *p* and *n*

Efficient Interconnection Networks for Cannon's MxM?



Memory need of each processor

*As a function of **n**, **p** and **b** (the number of bytes per element)*

- Sequential algorithm:
- One-step algorithm:
- Alternate algorithm:
- Cannon’s algorithm:

Special MPI functions

- MxV version 2: Reduce operation
- Cannon: Shift operation through a SendRecv_replace call

- Submatrix sending:
  - Consist of equally spaced blocks
  - DataType.Vector(int count, int blocklength, int stride, Datatype oldtype)

Overview

1. Matrix-vector Multiplication
2. Matrix-matrix Multiplication
3. Shared-memory

Shared-memory systems

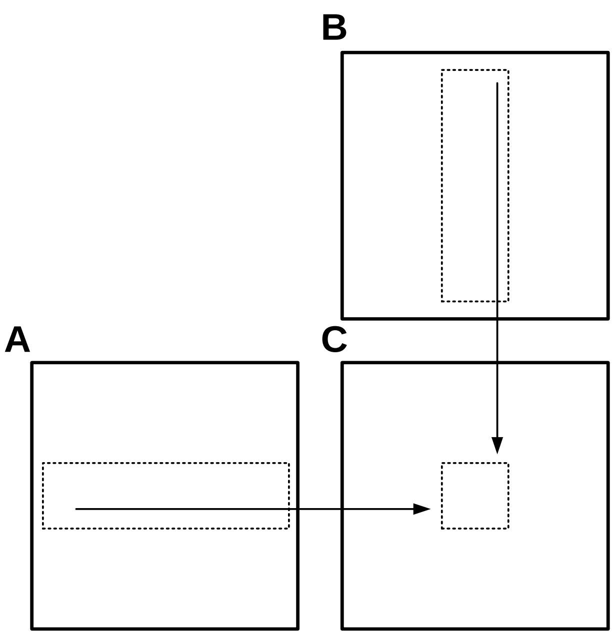
- Computation of each element is independent of the others
  - Can be done by a thread without the need of synchronization (see chapter of shared memory)
- Data is accessible by all threads
- However…

MxM on GPU

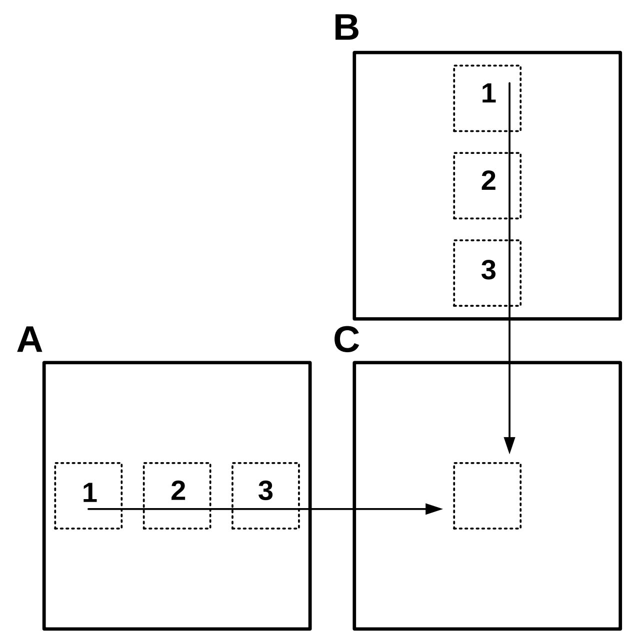
Example:
GPUmat toolbox in Matlab

- Initially, matrices are copied to GPU
  - If they are not still in memory from previous matrix operations, keep pointers in CPU to the data on the GPU
- Every thread computes 1 element of C.
- Not enough memory to put all data in shared memory (16K)
- On one multiprocessor, 1 block of threads computes 1 block of the C matrix
- Iteratively copy A row blocks and B column blocks to shared memory.
- *Result*: 200x speedup, 50x if compared to quadcore

MxM on GPU



Each multiprocessor computes 1 block

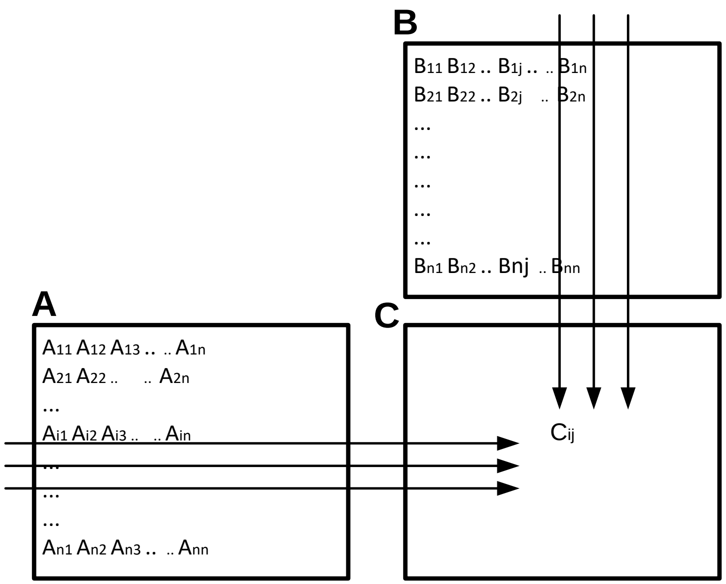


If rows and columns do not fit in local memory: compute block-by-block

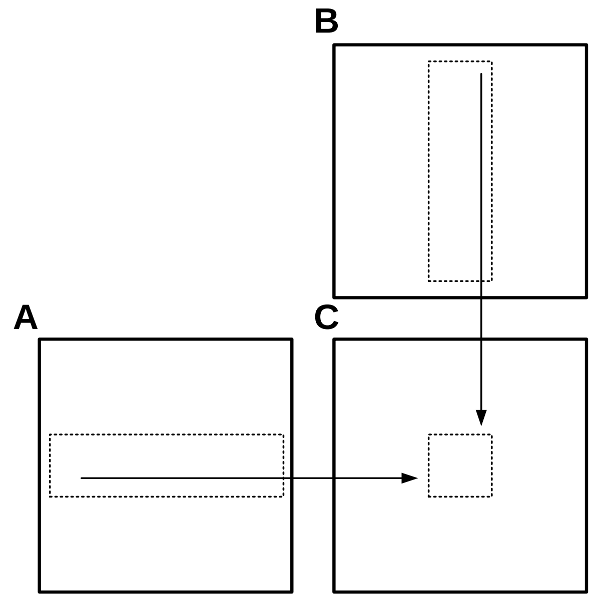
Cache efficiency?

- Multicores or GPUs with cache can directly access RAM/global memory
- Data is cached automatically
- However: *is the data in cache reused optimally?*
- Better is to organize the computations as with the GPU implementation
  - Multicores: each thread calculates the multiplication of 2 submatrices of A and B

Cache efficiency



- Row of A is reused in computing row of C
- Column of B is reused in computing column of C



To maximize cache usage, compute block by block (cf Cannon which minimizes communication)

If A rows and B columns do not fit in cache: alternate over A and B blocks like GPU version

Experimental results