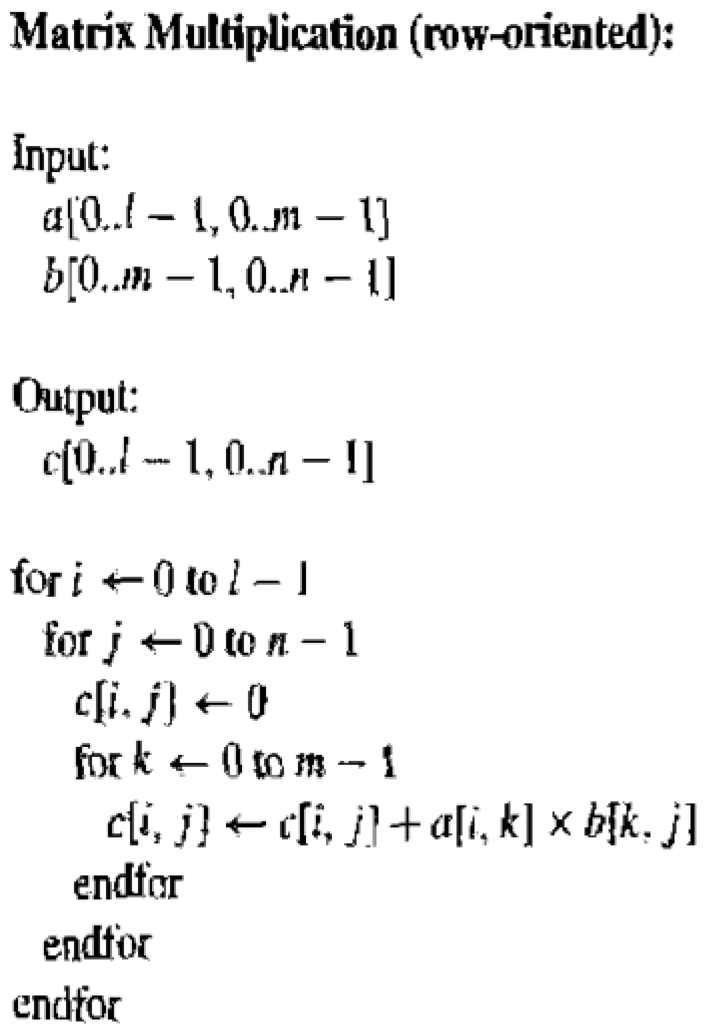
innovate achieve lead

BITS Pilani
Pilani Campus

Matrix Multiplication

Matrix-Matrix Multiplication

- Consider the problem of multiplying two n x n dense, square matrices A and B to yield the product matrix $C = A$ x B.
  - The serial complexity is $O(n^3)$
  - During each iteration of the outer i loop, every element of matrix B is read. If matrix B is too large for the cache, then later elements read into cache displace earlier elements read into cache, meaning that in the next iteration of the loop, all of the elements of B will need to be read into cache again. Hence once the matrices reach a certain size, the cache hit rate falls dramatically

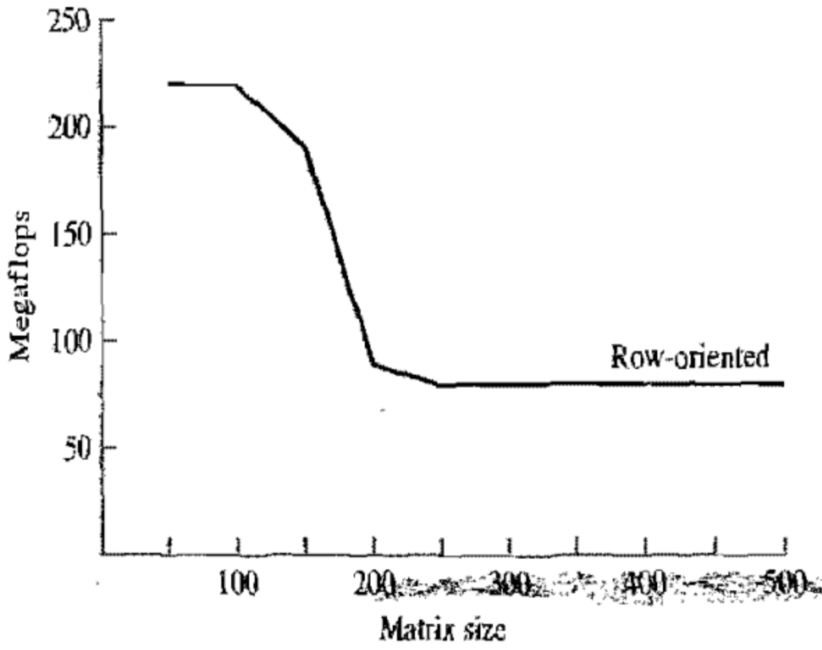


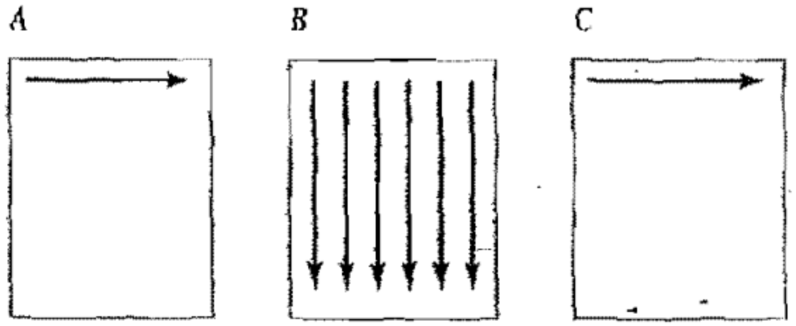
Matrix Multiplication (row-oriented):

```
Input:
  $a[0..l-1, 0..m-1]$
  $b[0..m-1, 0..n-1]$

Output:
  $c[0..l-1, 0..n-1]$

for $i \leftarrow 0$ to $l-1$
  for $j \leftarrow 0$ to $n-1$
    $c[i,j] \leftarrow 0$
    for $k \leftarrow 0$ to $m-1$
      $c[i,j] \leftarrow c[i,j] + a[i,k] \times b[k,j]$
    endfor
  endfor
endfor
```

Matrix-Matrix Multiplication

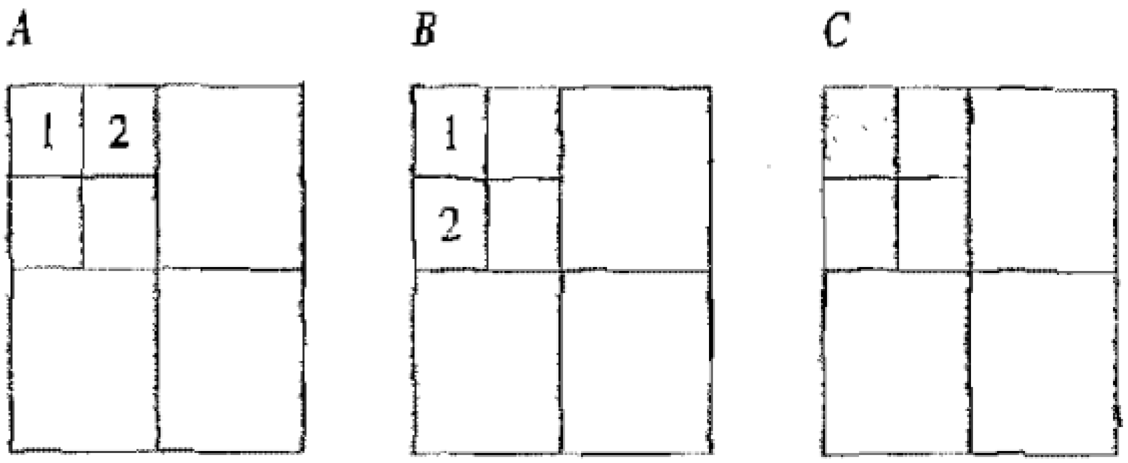
- During each iteration of the outer i loop, every element of matrix B is read. If matrix B is too large for the cache, then later elements read into cache displace earlier elements read into cache, meaning that in the next iteration of the loop, all of the elements of B will need to be read into cache again. Hence once the matrices reach a certain size, the cache hit rate falls dramatically





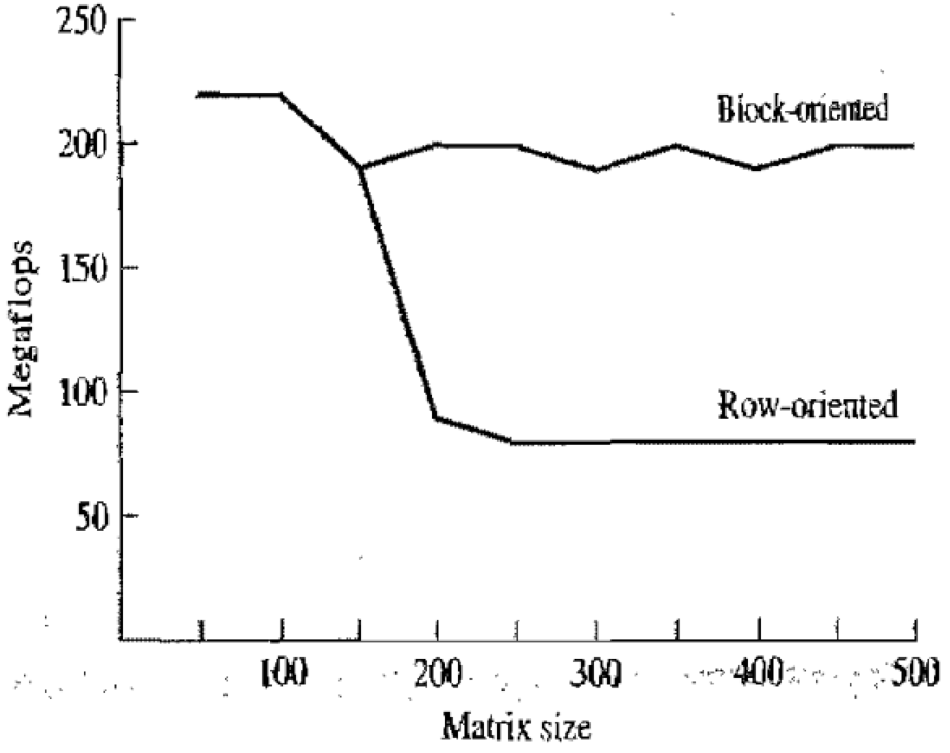
Matrix-Matrix Multiplication

- A useful concept in this case is called ***block* operations**.
  - In this view, an n x n matrix A can be regarded as a q x q array of blocks $A_{i,j}$ ($0 \leq i, j < q$) such that each block is an (n/q) x (n/q) submatrix.
  - In this view, we perform q^3 matrix multiplications, each involving (n/q) x (n/q) matrices.
- A recursive matrix multiplication algorithm breaks the matrices into smaller and smaller blocks until they can fit in cache



Matrix-Matrix Multiplication

- Row-oriented approach vs block oriented approach
  - The block-oriented matrix multiplication algorithm keeps the cache hit rate high and achieves better performance than the row-oriented algorithm



Rowwise Block·striped Parallel Algorithm

- Identifying Primitive Tasks
  - Each element of the product matrix C is a function of elements in A and B
  - Since A and B are not modified during the algorithm, it is possible to compute every element of C simultaneously
  - As a first step in our parallel design, then, we can associate one primitive task with every clement of C
  - Computing element $c_{i,j}$ of the product matrix involves finding the inner product (dot product) of row i of A and column j of B.
- Agglomeration
  - It is natural to agglomerate tasks associated with either a row of C or a column of C, since they share a need for either a row of A or a column of B, respectively,

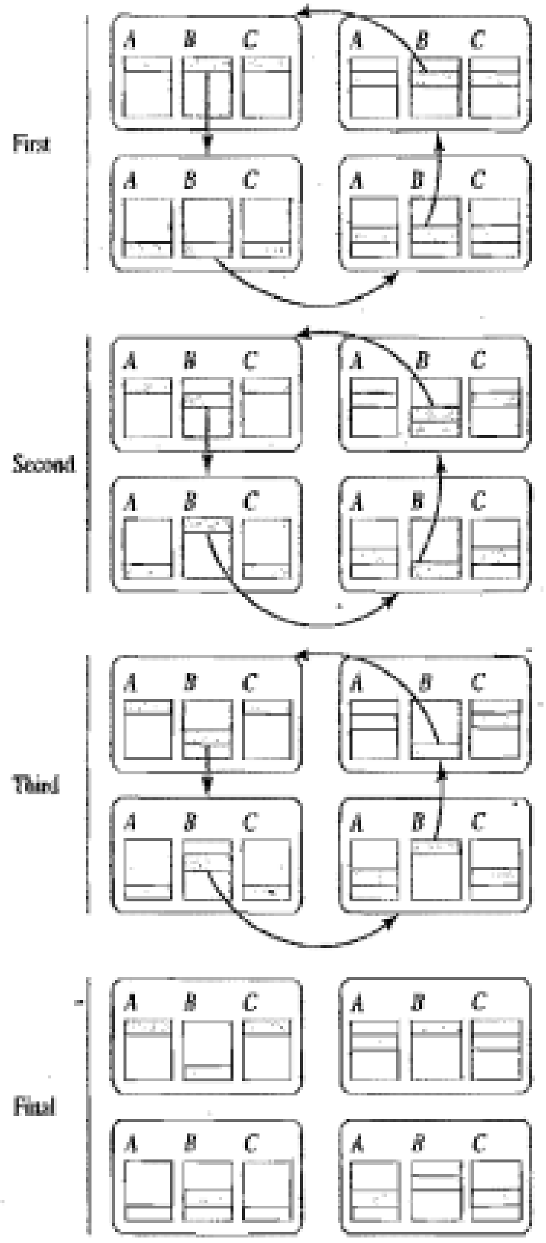
Rowwise Block·striped Parallel Algorithm

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

- Agglomeration
  - Let's think about what task i can do with row i of A, row i of B and row i of C
  - the task can perform n multiplications that represent partial sums for the n elements of row i of C

Rowwise Block-striped Parallel Algorithm

- Communication of B in row-oriented parallel matrix multiplication algorithm
  - Each task is responsible for a row of A, a row of B, and a row of C. If B has m rows, then after m-1 communication steps each task has had access to every row of B
- Computation time
  - $(n/p)(n/p)(n)=n^3/p^2$
- Communication: communicate n/p entries of B to next process.
  - $t_s+t_w(n/p)(n)$
- Total parallel time
  - p iterations
  - $T_p=p(n^3/p^2+t_s+t_w(n/p)(n))$
  - $T_p=n^3/p+pt_s+n^2t_w$



Rowwise Block·striped Parallel Algorithm

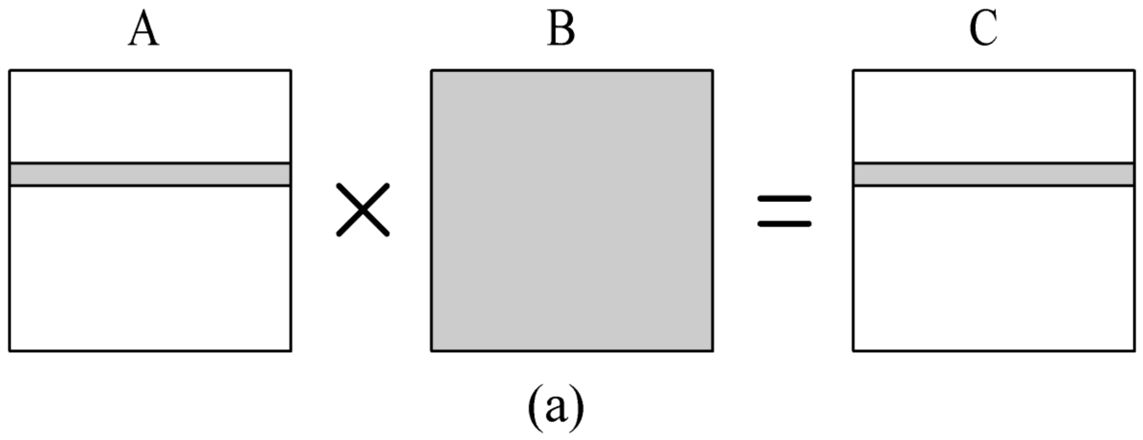
- Scalability
  - $T_0 = pTp - W$
  - $T_0 = p(n^3/p + pt_s + n^2t_w) - n^3 \rightarrow T_0 = (p^2t_s + pn^2t_w)$
  - $W = Kpn^2t_w$
  - $n^3 = Kpn^2t_w$
  - $\rightarrow n = Kt_wp$
  - $\rightarrow n^3 = K^3t_w{}^3p^3$
  - $\rightarrow W = K^3t_w{}^3p^3$
  - This rate of $\theta(p^3)$ is the asymptotic isoefficiency function

$$W = \left(\frac{E}{1-E}\right) * T_0(W, p)$$

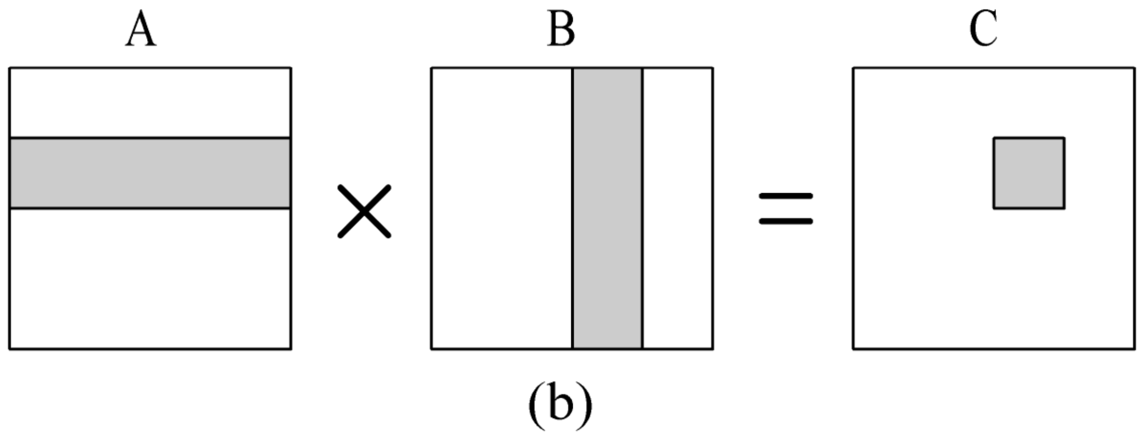
Weakness

- Blocks of B being manipulated have *p* times more columns than rows
- Each process must access every element of matrix B
- Ratio of computations per communication is poor: only $2n / p$
- Cannon's Algorithm
  - Associate a primitive task with each matrix element
  - Agglomerate tasks responsible for a square (or nearly square) block of C
  - Computation-to-communication ratio rises to $n / \sqrt{p}$

Elements of A and B Needed to Compute a Process's Portion of C

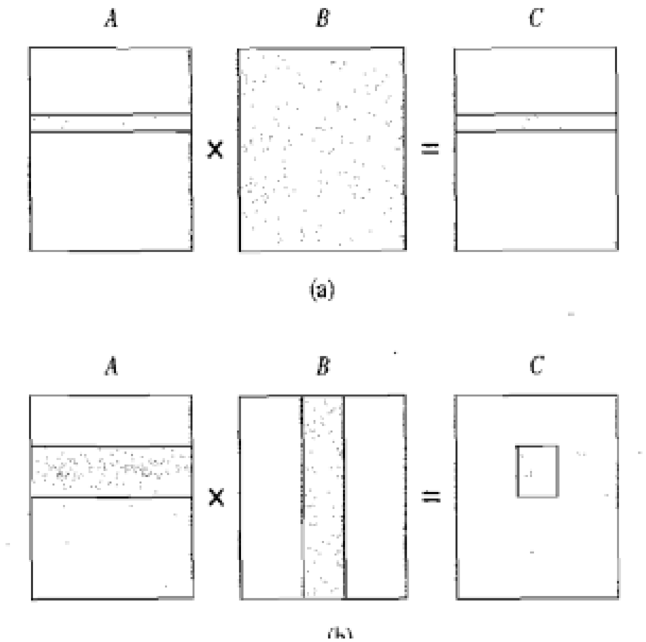


Algorithm 1



Cannon's Algorithm

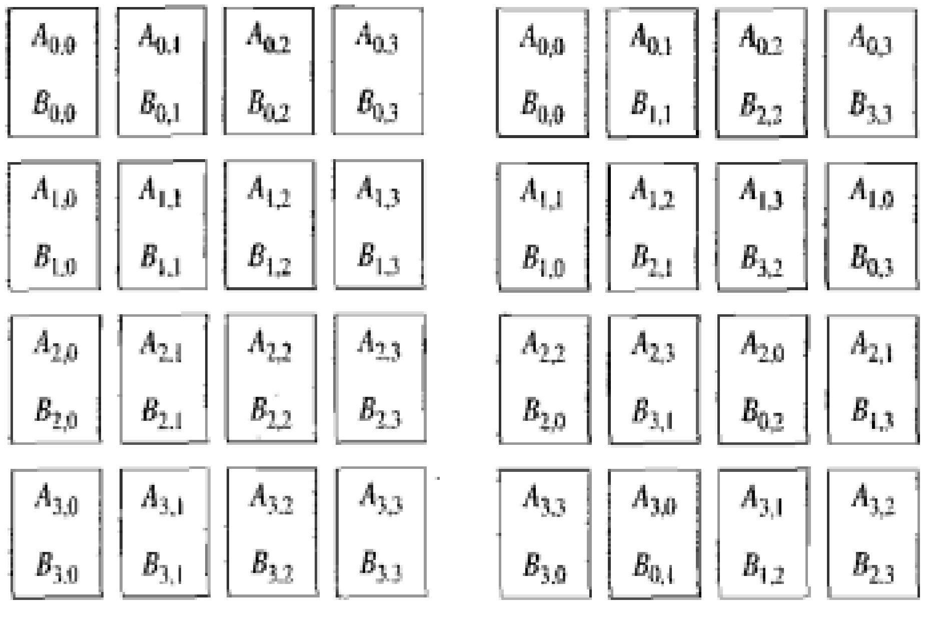
Cannon’s algorithm



- Comparison of number of elements' of A and B needed to compute a process's portion of C in - the two parallel matrix multiplication algorithms (a) row-oriented (b) Cannon’s algorithm

Cannon’s algorithm

- Each process in row i of the process mesh cycles its block of A to the process i places to its left. Each process in column j of the process mesh cycles its block of B to the process j places above it.

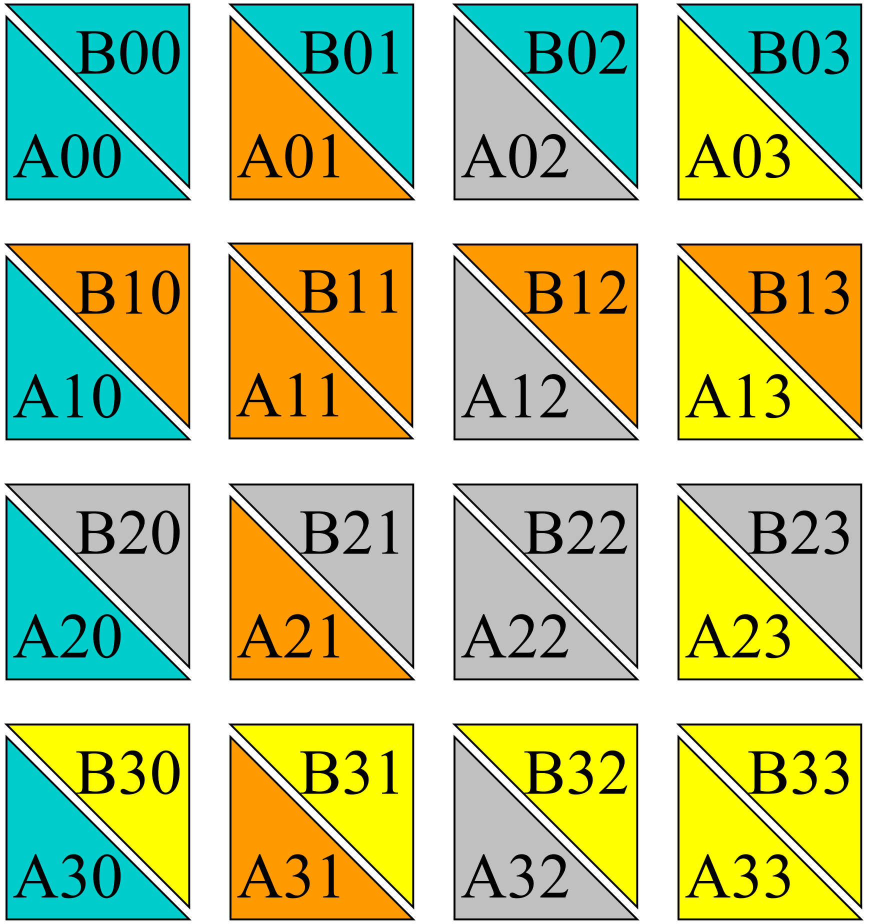


(a) (b)

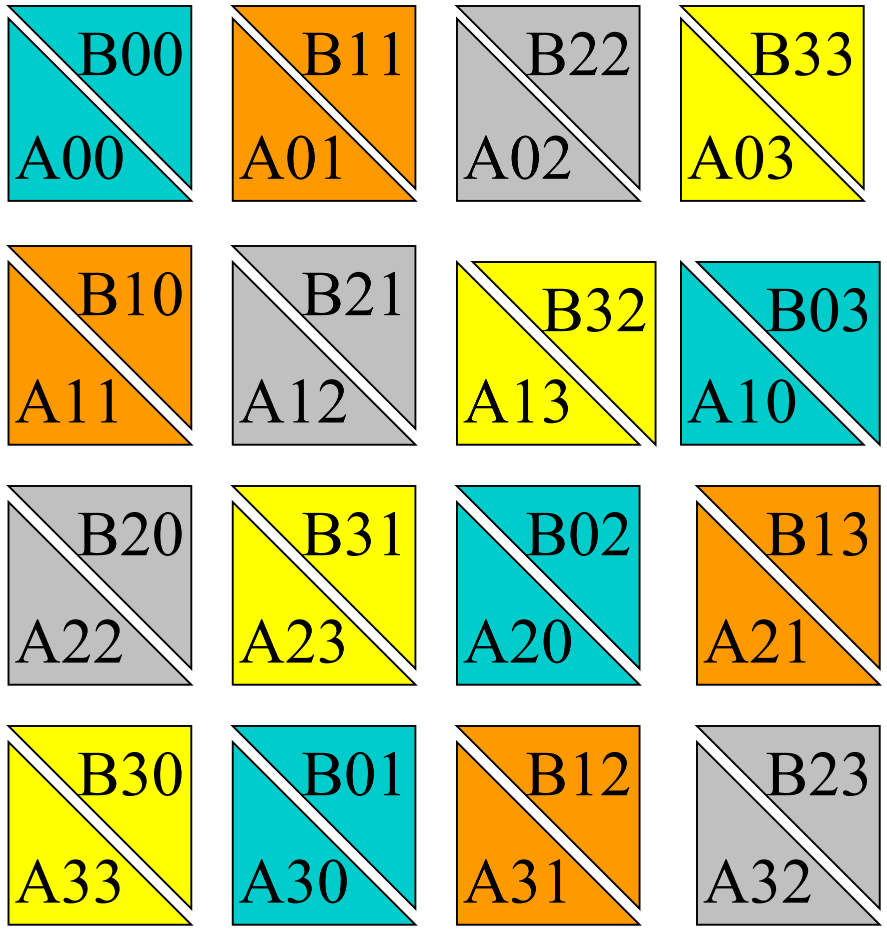
Blocks Need to Be Aligned

Each triangle represents a matrix block

Only same-color triangles should be multiplied



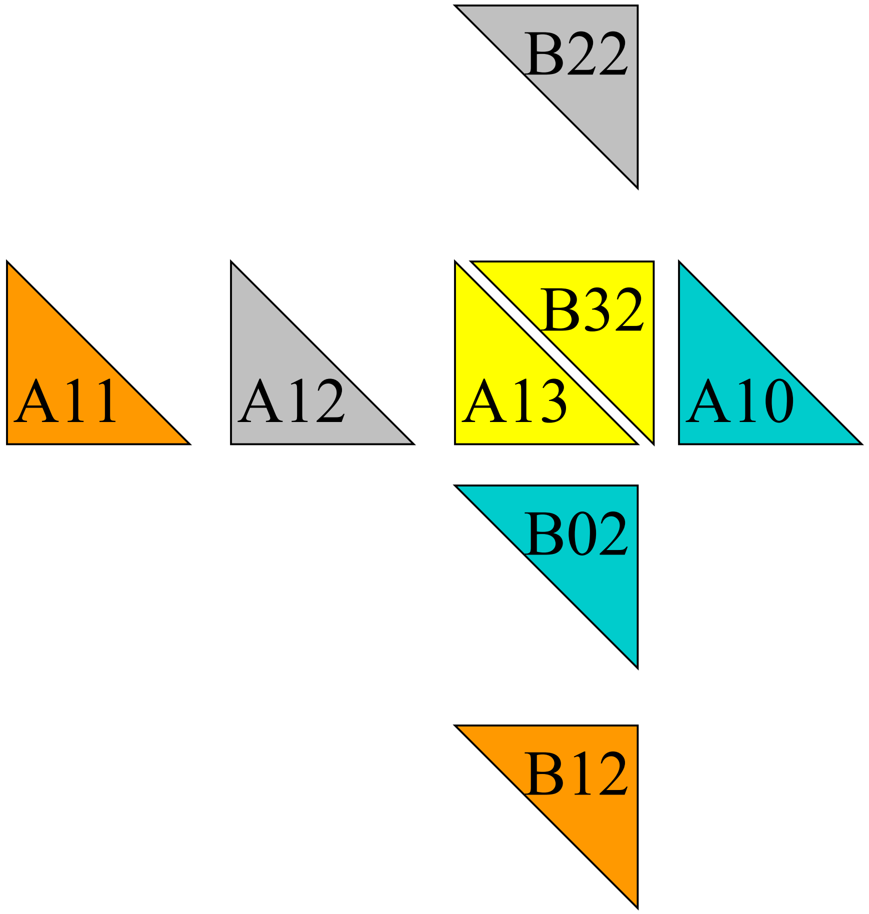
Rearrange Blocks



Block Aij cycles left i positions

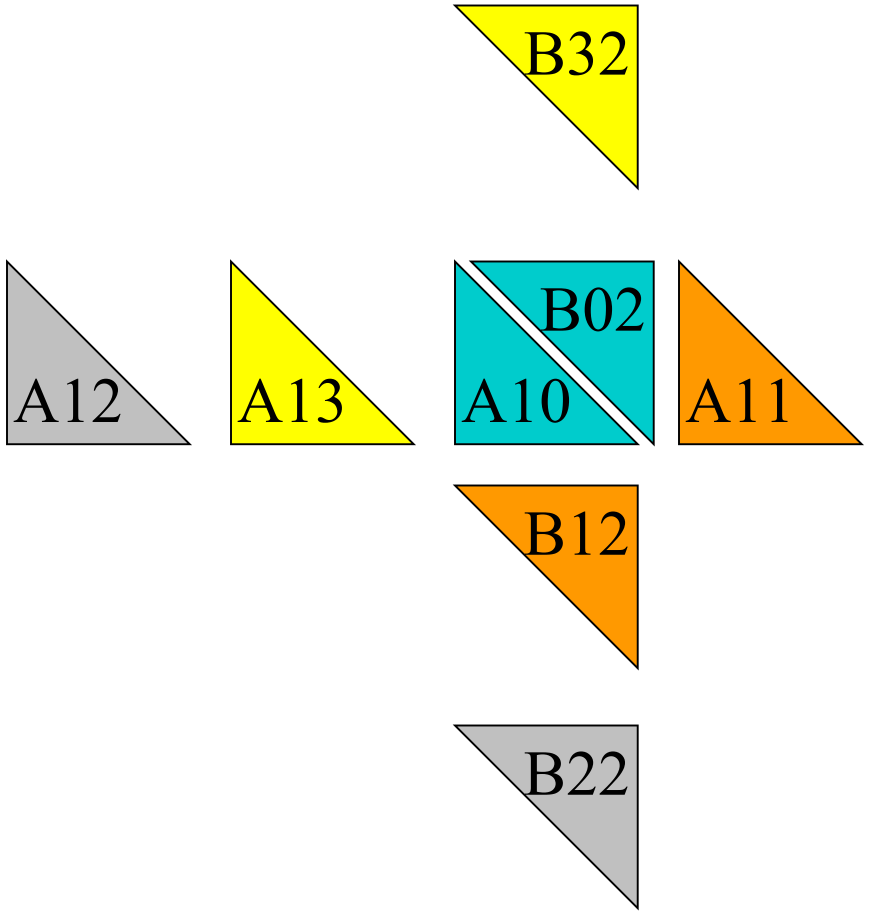
Block Bij cycles up j positions

Consider Process $P_{1,2}$



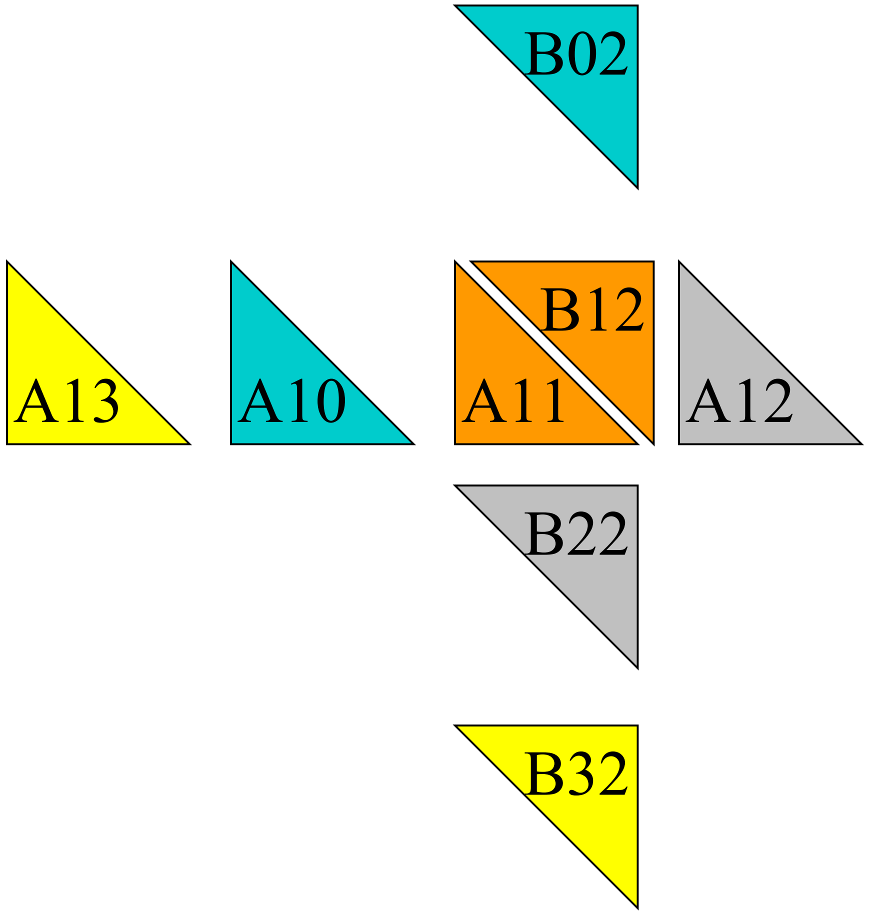
Step 1

Consider Process $P_{1,2}$



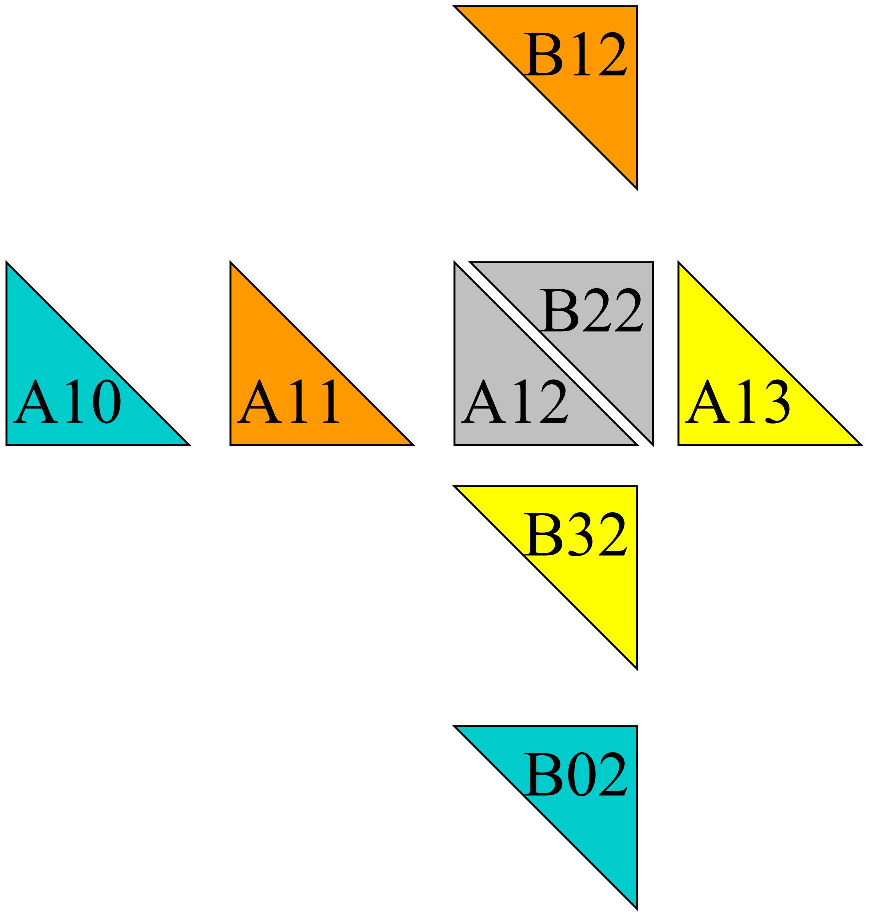
Step 2

Consider Process $P_{1,2}$



Step 3

Consider Process $P_{1,2}$



Step 4

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: $\Theta(n^3 / p)$
- During each iteration process sends and receives two blocks of size $(n / \sqrt{p}) \times (n / \sqrt{p})$
- Communication complexity: $\Theta(n^2 / \sqrt{p})$
- Total parallel runtime
  - $T_p = n^3/p + n^2/\sqrt{p}$

Isoefficiency Analysis

- Sequential algorithm: W= $\Theta(n^3)$
- Overhead=$T_0=pT_p-W$
- Parallel overhead: $T_p=\Theta(\sqrt{p}n^2)$

Isoefficiency relation: $n^3 \geq C \sqrt{p} n^2 \Rightarrow n \geq C \sqrt{p}$

Worksize $W=Cp^{3/2}$

Memory per process:

$$M(C\sqrt{p})/p = C^2 p/p = C^2$$

- This system is highly scalable

References

BITS Pilani

- Chapter 11 Quinn’s book “Parallel programming in C with MPI and OPeMP”

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You