Efficient Parallelization of Matrix-Matrix Multiplication

Sandeep Agrawal
HPC Technologies Group
C-DAC Pune

Plan

- Serial Code
- Block oriented recursive algorithm
- Naive parallel code
- Row oriented block striped algorithm
- Problems with it.

- Cannon's Matrix Multiplication Algorithm
  - Method
  - How it improves over the other algorithms
  - Analysis

- Bibliography

Introduction

Matrix-Matrix Multiplications

- A trivial but a fundamental algorithm
- Has wide range of applications.
- A simple example to explain parallel computing concepts

Aim

Aim

To solve $A_{nxm} X \, B_{mxp} = C_{nxp}$, where,

- n & m = number of rows and columns of Matrix A
- m & p = number of rows and columns of Matrix B
- n & p = number of rows and columns of Matrix C

Remember

- Number of Columns of A = Number of rows of B

Notations

- Consider ONLY Square Matrices of order n
- Row index $i=0,1,2,\ldots,n-1$ (n values)
- Column index $j=0,1,2,\ldots,n-1$ (n values)
- $(i,j)^{th}$ element of A,B & C matrix would be

$A_{i,j}$,

$B_{i,j}$,

$C_{i,j}$

Matrix A,B & C

$$A = \begin{matrix} a_{0,0} & a_{0,1} & \ldots. & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \ldots. & a_{1,n-1} \\ \ldots. & \ldots. & \ldots. & a_{2,n-1} \\ a_{n-1,0} & a_{n-1,1} & \ldots. & a_{n-1,n-1} \end{matrix}$$

$$B = \begin{matrix} b_{0,0} & b_{0,1} & \ldots. & b_{0,n-1} \\ b_{1,0} & b_{1,1} & \ldots. & b_{1,n-1} \\ \ldots. & \ldots. & \ldots. & b_{2,n-1} \\ b_{n-1,0} & b_{n-1,1} & \ldots. & b_{n-1,n-1} \end{matrix}$$

$$C = \begin{matrix} c_{0,0} & c_{0,1} & \ldots. & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \ldots. & c_{1,n-1} \\ \ldots. & \ldots. & \ldots. & c_{2,n-1} \\ c_{n-1,0} & c_{n-1,1} & \ldots. & c_{n-1,n-1} \end{matrix}$$

Naive Matrix Multiplication Algorithm

Serial Matrix-Matrix Multiplication Algorithm

$$C(i, j) = \sum_{k=0}^{n-1} A(i,k) * B(k, j)$$

for all i and j = 0, 1, …, n-1

Number of columns of A = Number of row of B

Pseudo Code

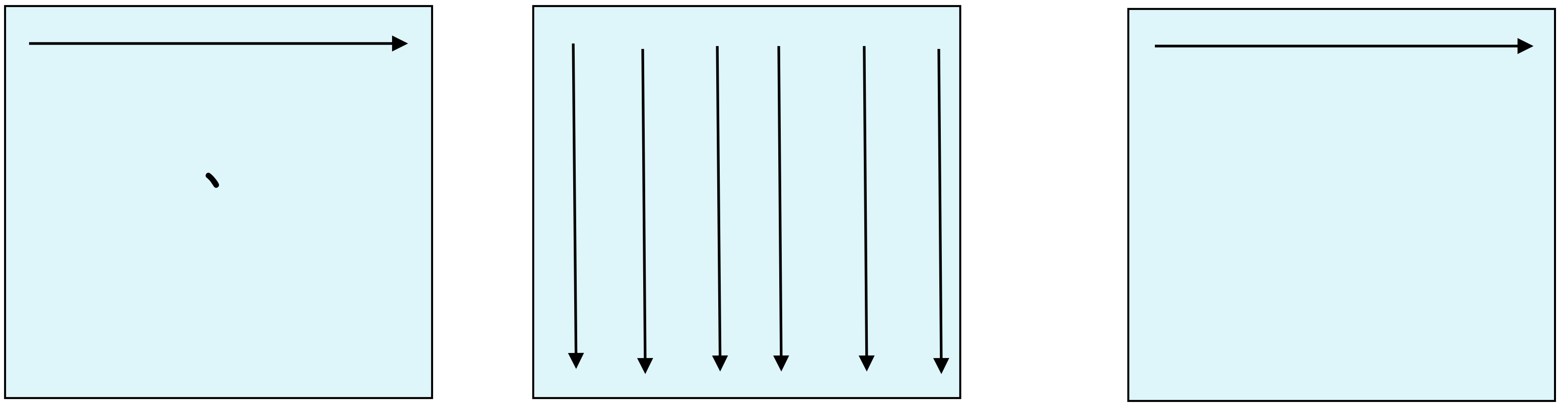
The Pseudo Code for Row wise Matrix Multiplication

```
1   Do i = 0 to n-1
2       Do j = 0 to n-1
3           C(i,j) = 0.0
4           Do k = 0 to n-1
5               C(i,j) = C(i,j)+ A(i,k) x B(k,j)
6           End k loop
7       End j loop
8   End i loop
```

- **Note steps 1, 2 & 4. Here, i,j & k takes n values each**
- **Time complexity serial matrix multiplication ~ $O(n^3)$**

Matrix Multiplication

- Schematic Matrix-Matrix Multiplication



- A_{nxn} B_{nxn} C_{nxn}

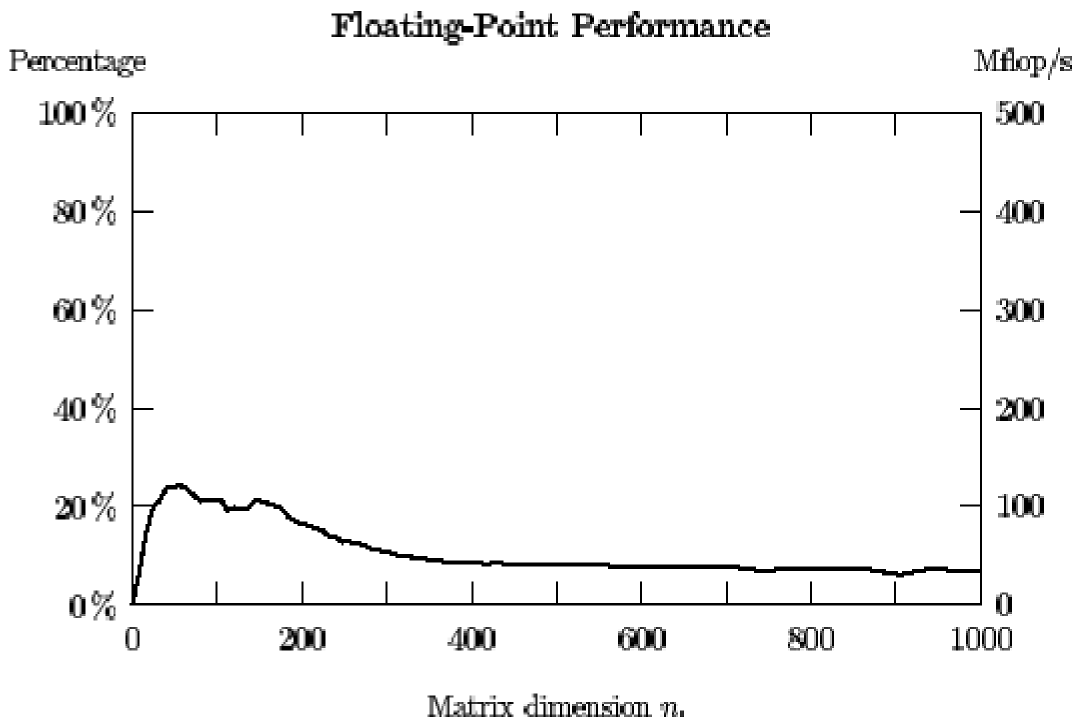
- **In a single iteration over row *i*, whole of B is read to produce single *i*^th row of C**

Analysis

- What is the performance of this serial code, if the matrix size increases beyond your cache memory limit?

Serial algorithm: Performance

- What is performance?



- Floating point performance of *i,j,k*–variant of n x n matrices on one processor of SGI Origin 2000

- Adapted from: ftp://ftp.vcpc.univie.ac.at/projects/aurora/reports/auroratr2002-08.ps.gz

Serial code: Analysis

- <u>Observations:</u>

- The performance of the code degrades dramatically for matrix size n > 150.

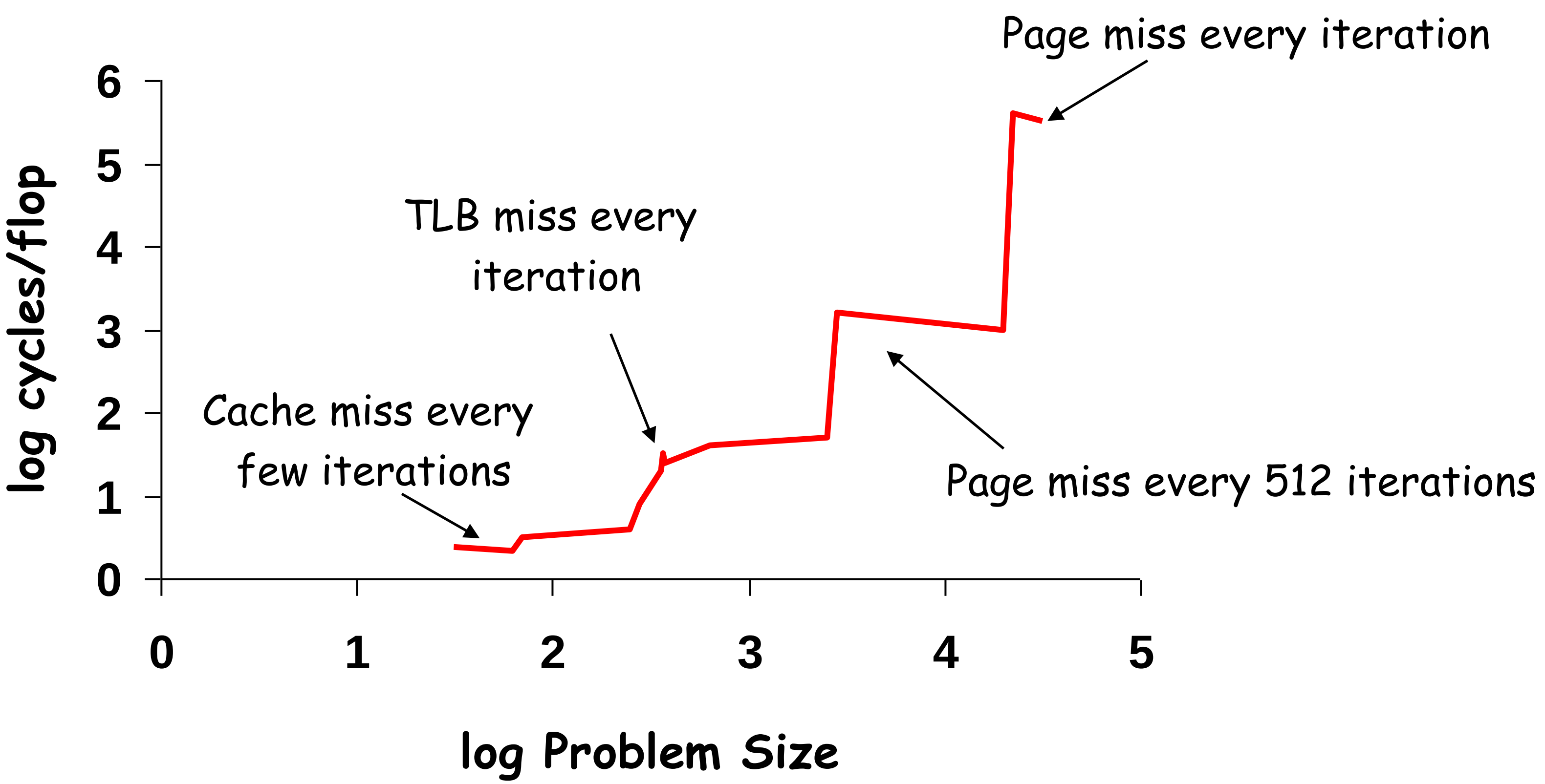
- Even for n < 150, the performance of the code is < 150 Megaflop/s. (10^6 floating point operations per second)

- <u>Reason:</u>

- Matrix B with size n > 150, no longer fits in the cache. So the the cache hit rate is small !

- So it's a FLOP-SHOW!!

Other Dynamics



Adapted from: www.sdsc.edu/~allans/cs260/lectures/matmul.ppt

Memory efficient method

- Solution

- Strip Matrix A and B so as to fit in your cache memory!!
- Divide Matrix $A_{n,n}$ and $B_{n,n}$ into, say, FOUR smaller sub-matrices.

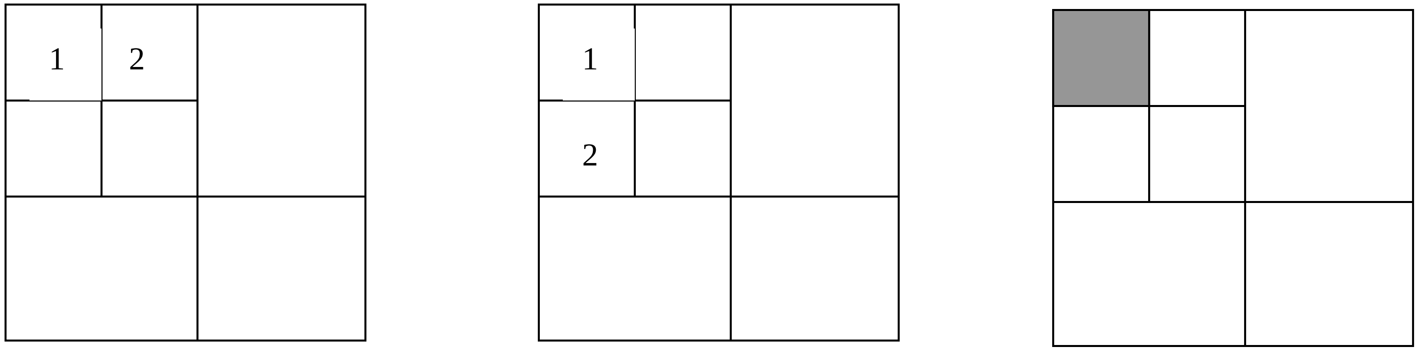
$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \qquad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

represents Matrix Additions

$$C = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{11} + A_{11}B_{11} \end{pmatrix}$$

Block striped Matrices

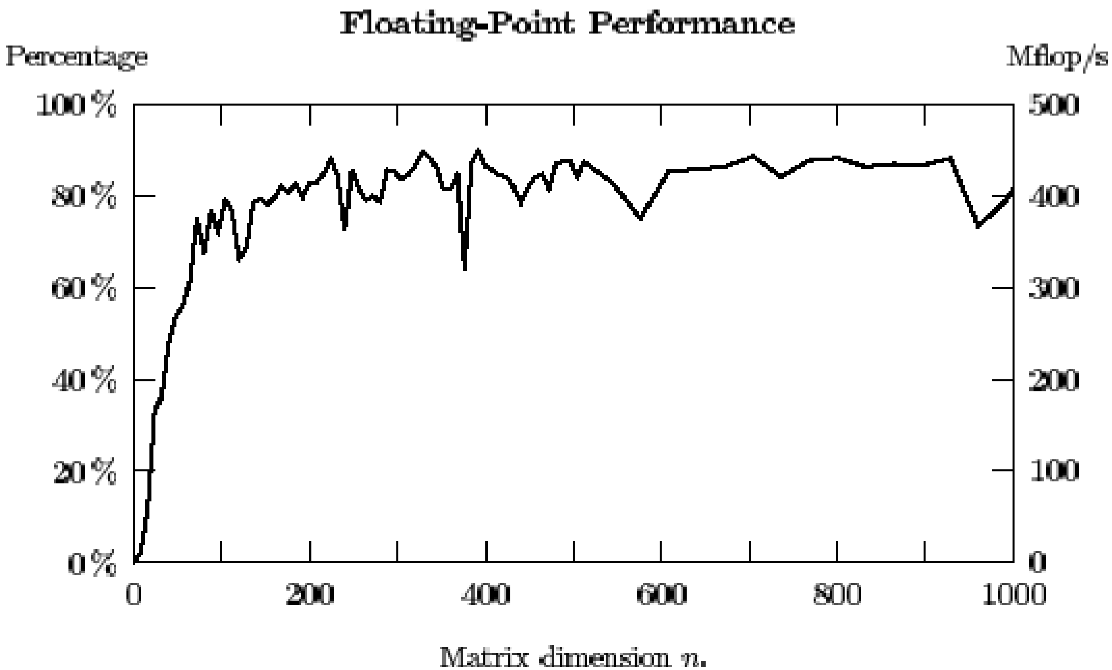
- Recursive Block oriented matrix multiplication algorithm



$A_{n,n}$ $B_{n,n}$ $C_{n,n}$

Break the matrices into smaller and smaller blocks until they fit in your cache memory!

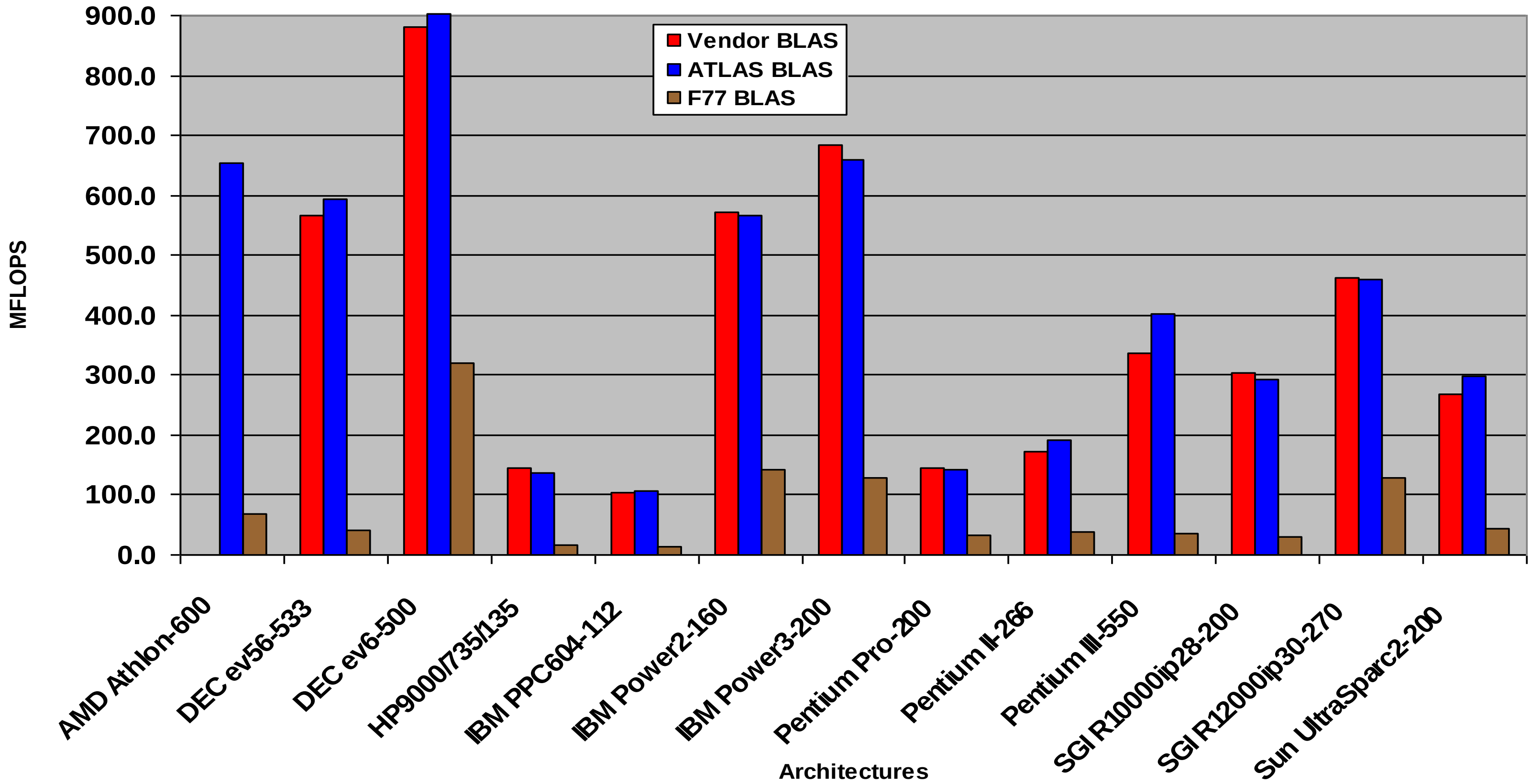
Performance of block oriented Matrix Multiplication



Performance of Block-oriented Matrix Multiplications using subroutine *dgemm* from vendor optimised BLAS library

Adapted from: ftp://ftp.vcpc.univie.ac.at/projects/aurora/reports/auroratr2002-08.ps.gz

ATLAS (DGEMM n = 500)

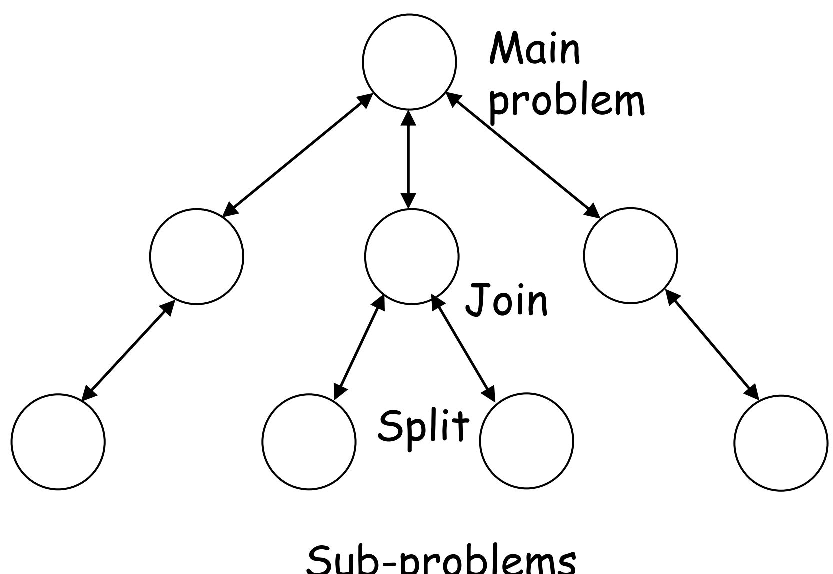


- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

Source: Jack Dongarra

Parallel Approach: Divide and Conquer

- Problem is divided into two or more sub-problems
- Each sub-problem is solved independently
- Sub-problem results are combined to give final results
- Problem decomposition and distribution are dynamic



Think *Parallel*

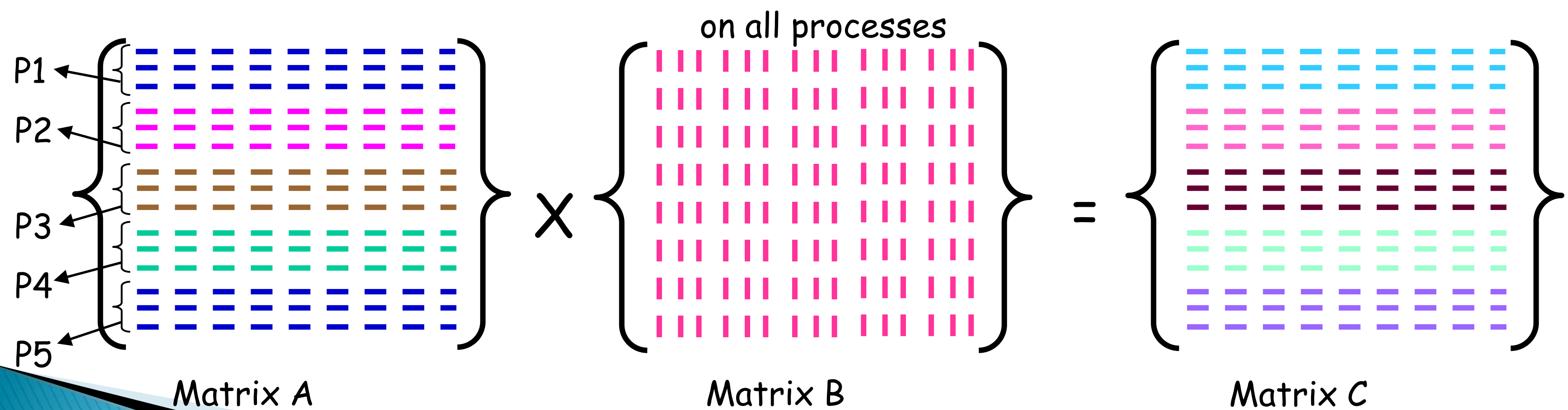
- Think *Parallel* !
- Identify tasks which are *independent* of each other!!

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \qquad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Defensive Matrix multiplication

- The master process is the controller process that
  - Distributes Matrix A (row-wise) to each worker process
  - Communicates complete Matrix B to each worker process
- Each worker multiplies assigned rows with matrix B
- Master process collects resultant Matrix C from each worker process.

. You may assume row dimension of Matrix A to be multiple of number of worker processes P_k



Problems !!

Observations:

- Each Process (p_k) must have
  - Whole of matrix B
  - All the elements of the given rows of $A_{i,k}$(i=0,n-1)
  - Communication time between processes ~ $O(n^2)$

Conclusions:

Not a very efficient Method !!

Row-wise Block-striped Parallel Algorithm

- Identify primitive tasks
  - Elements of A & B are not modified during Matrix-Multiplication
  - Compute every element of C simultaneously
  - Associate one primitive task with every element of C

- Agglomeration of tasks:
  - Agglomerate tasks associated with row of C
  - So each task is responsible for corresponding row of C

Primitive task

$$C(i,j) = \sum_{k=0}^{n-1} A(i,k) * B(k,j)$$

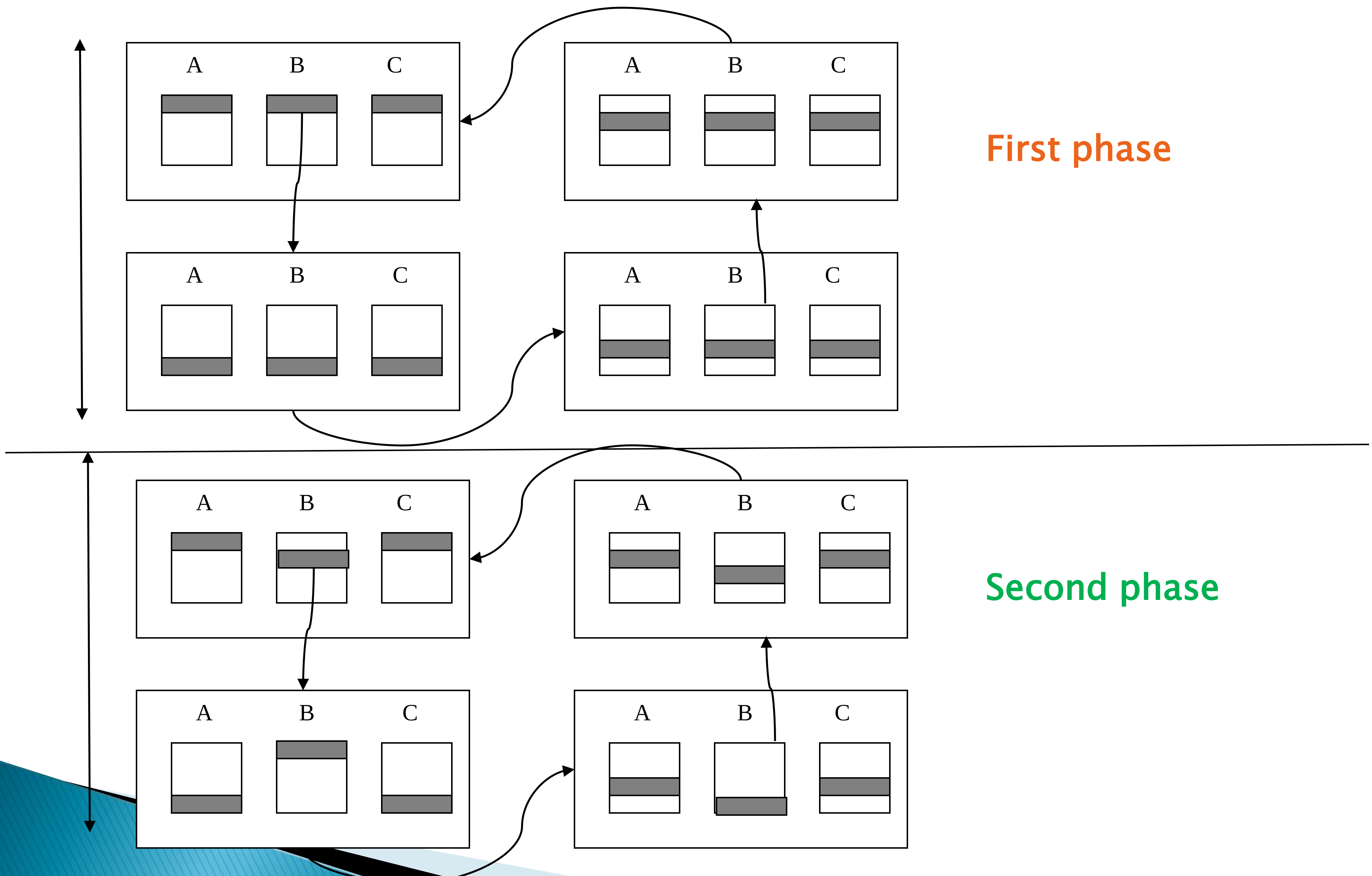
Agglomeration of tasks

C(i,j) for j = 0 to N-1

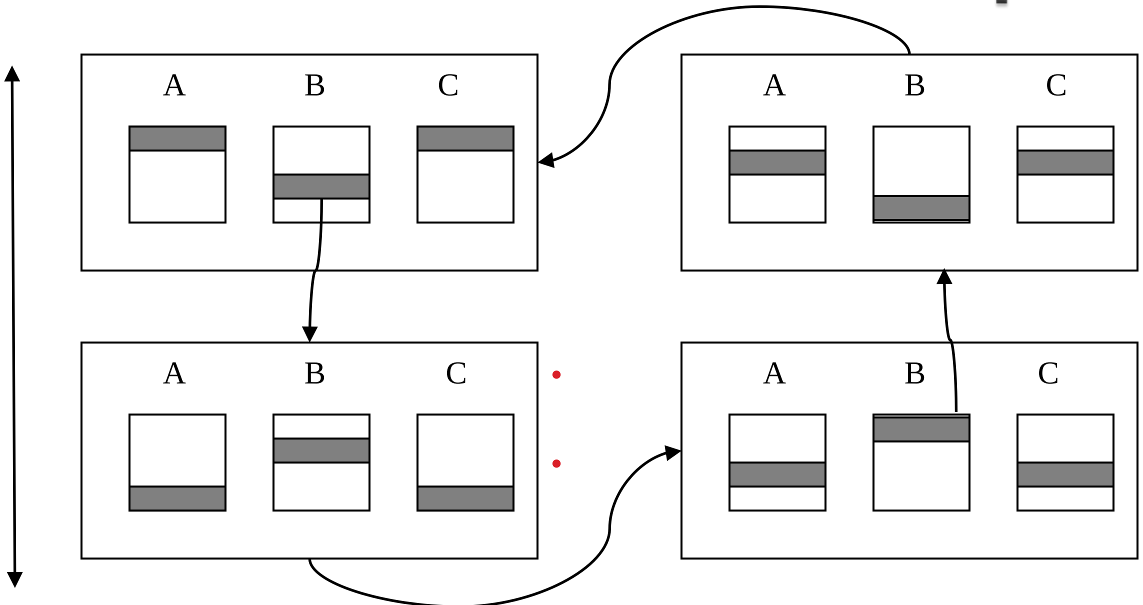
Ring Communication

- Communication and further agglomeration
- Let A, B & C be n x n matrix
- Organise the tasks as a ring
- Each task passes its row of B to next task on the ring
- After series of n iterations, every task will have every row of B
- Let A, B & C be divided into FOUR tasks

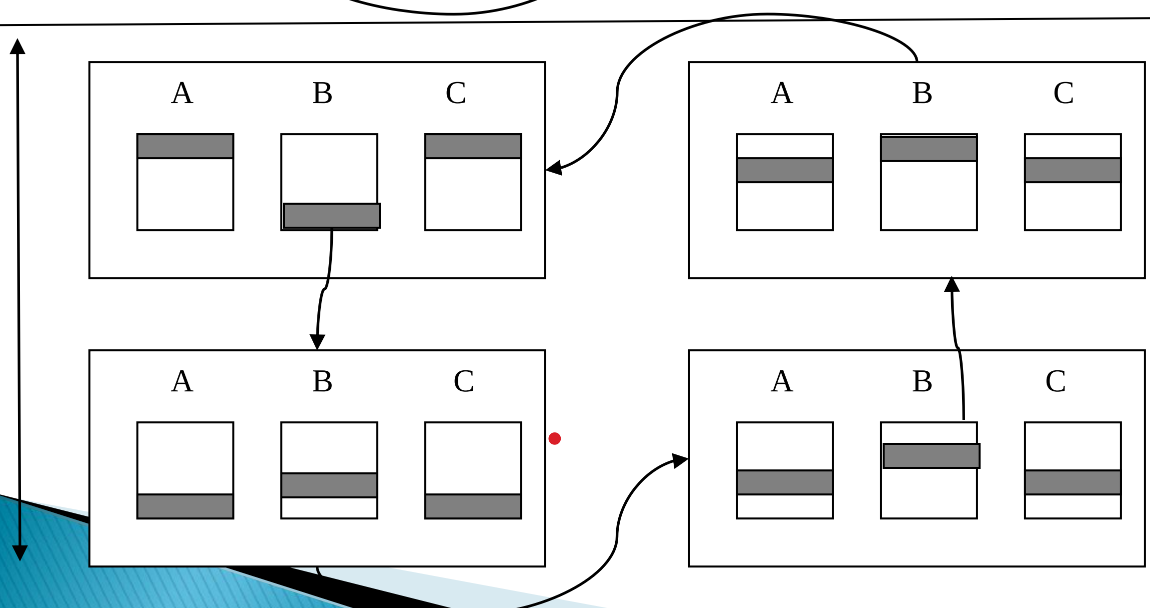
Parallel Matrix multiplication



Parallel Matrix multiplication



Third phase



Final phase

Disadvantages

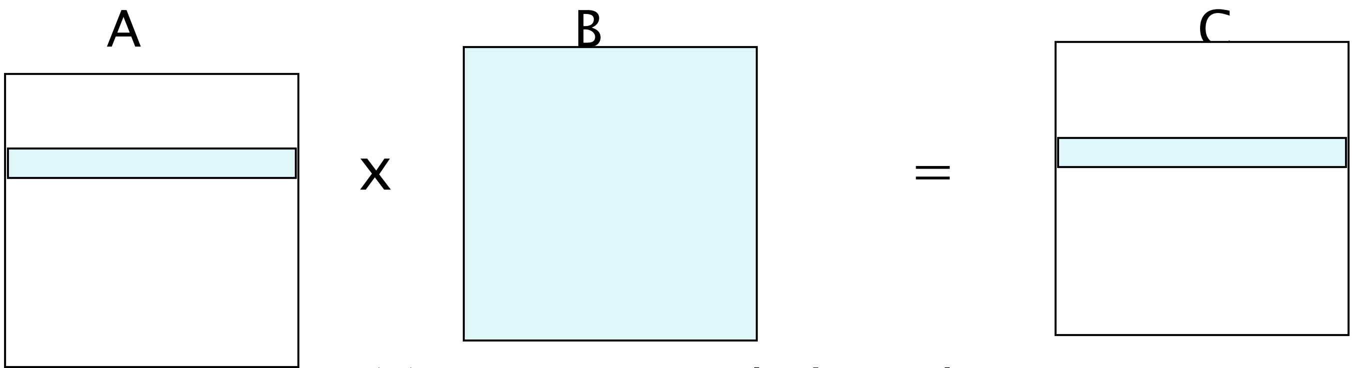
- Computation/Communication ratio:
- Multiply two nxn matrices on p processes
- Each process iterates through p phases, in each of which
  - Multiples (n/p) x n submatrix of A
    with (n/p) x n submatrix of B

- Ratio of computations to communication per process is
  - $$\frac{2n^3/p^2}{n^2/p} = \frac{2n}{p}$$

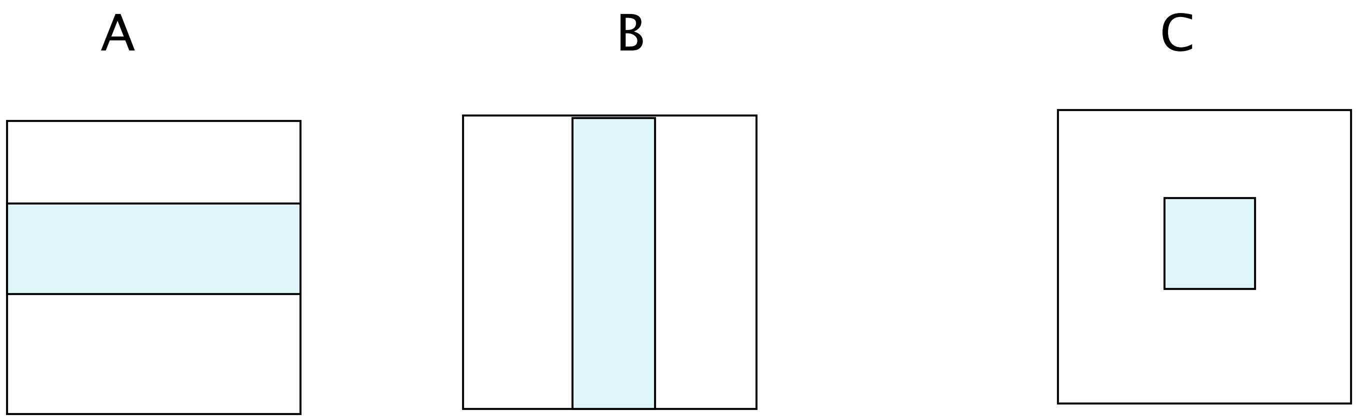
Cannon's Algorithm: Strategy

- Memory efficient version
- Let A_{nxn}, B_{nxn}, C_{nxn} matrices, such that
  - Total process *p* is a square number
  - *n* is multiple of $\sqrt{p}$
- Partition A_{nxn} & B_{nxn} into *p* square (or nearly square) blocks.
- Computation/communication per process reduces drastically!!

Comparison of Algorithms



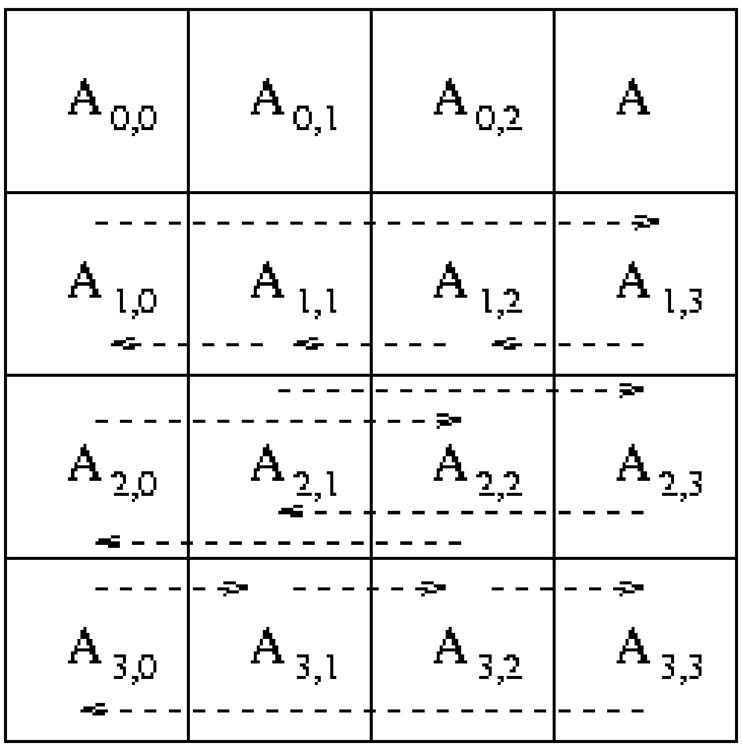
(a) Row oriented algorithm



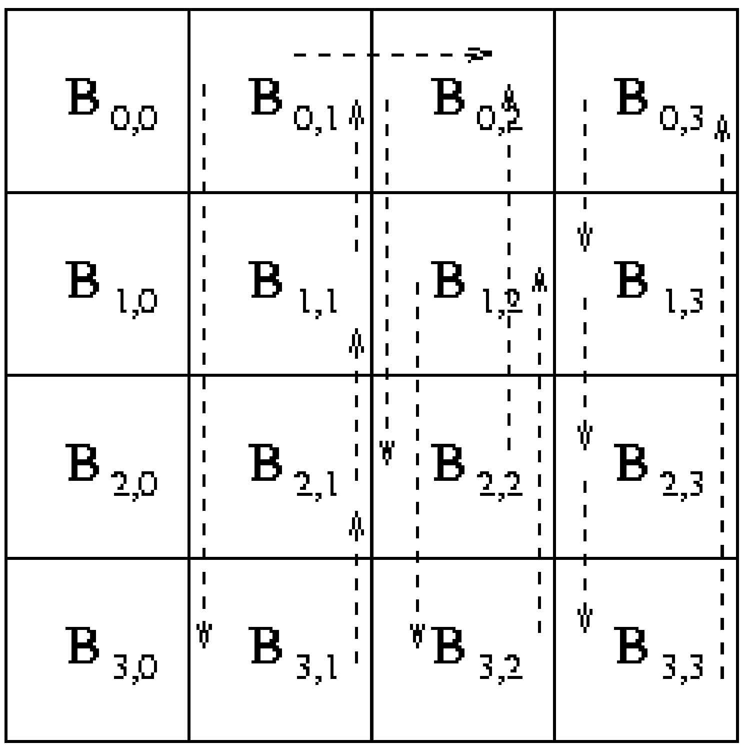
(b)Cannon's Algorithm

Number of elements of A & B to compute portion of C

Cannon's algorithm



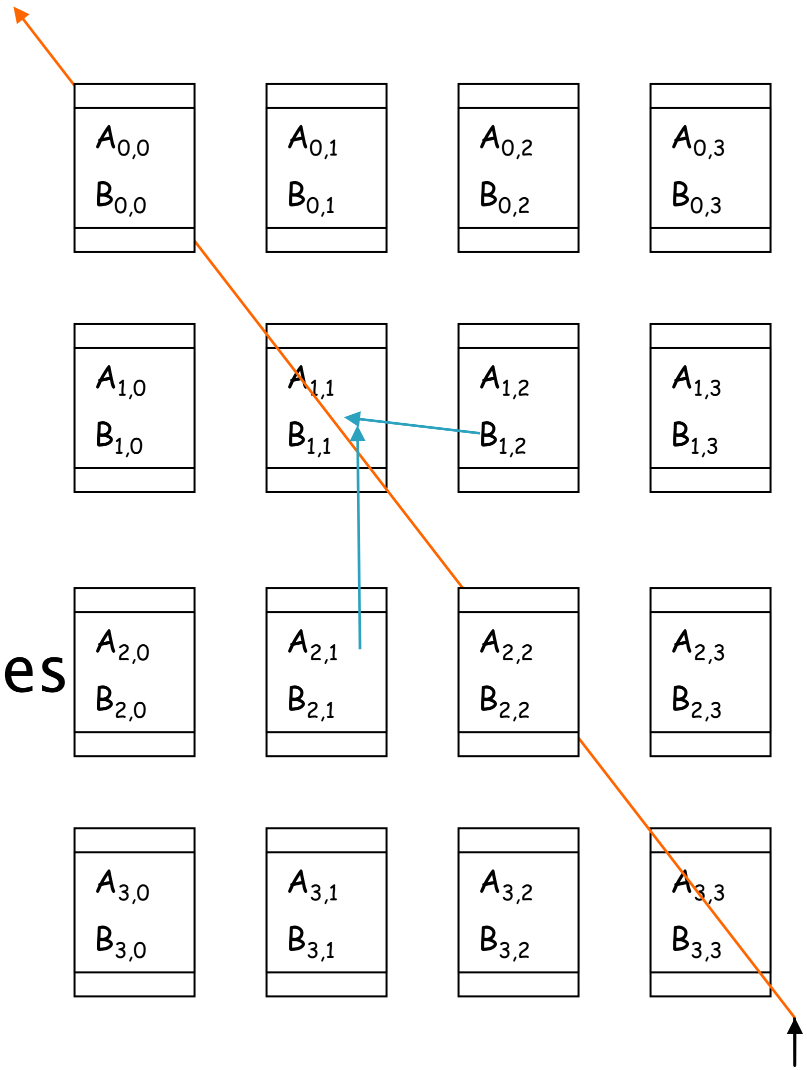
(a) Initial alignment of A



(b) Initial alignment of B

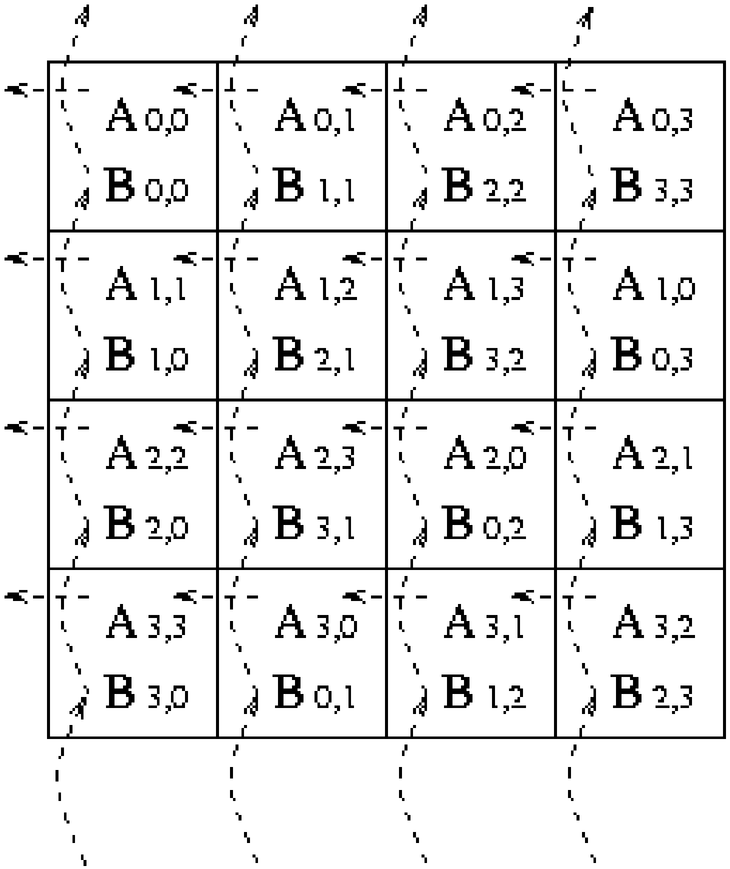
Initial Alignment

- Initial alignment of blocks for matrix multiplication. Process $P_{i,j}$ contains block $A_{i,j}$ & $B_{i,j}$. Only diagonal processes will satisfy $A_{i,k} \times B_{k,j}$

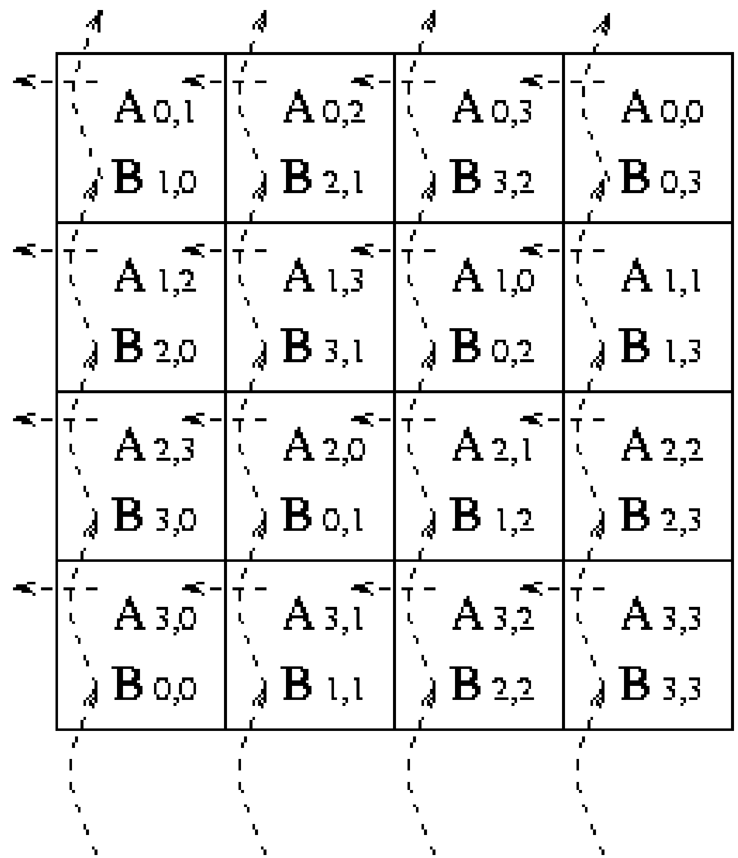


,0

Cannon's algorithm

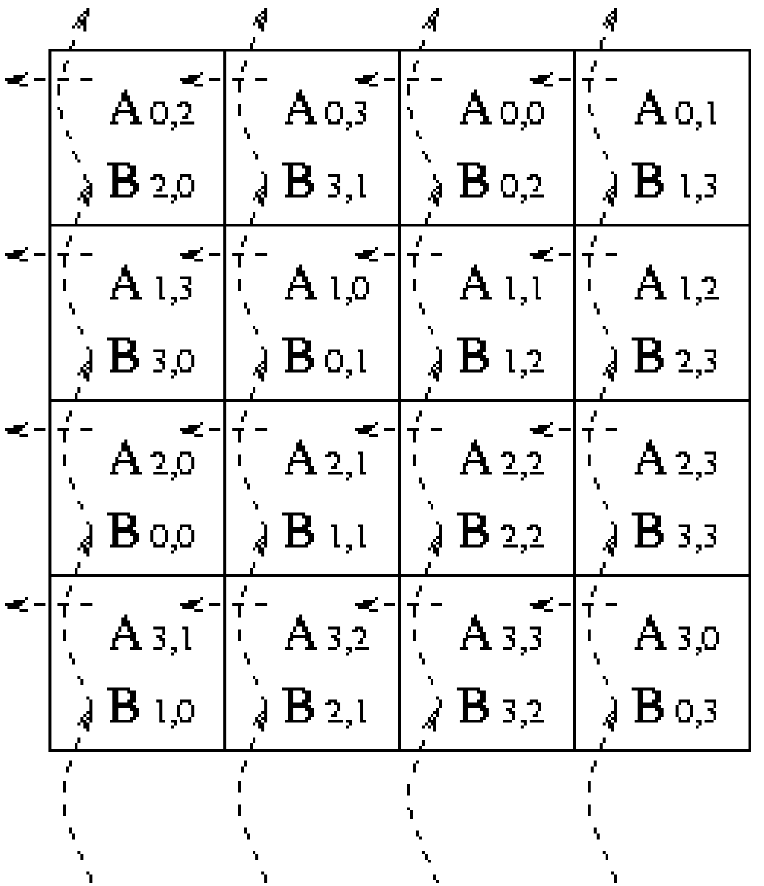


(c) A and B after initial alignment

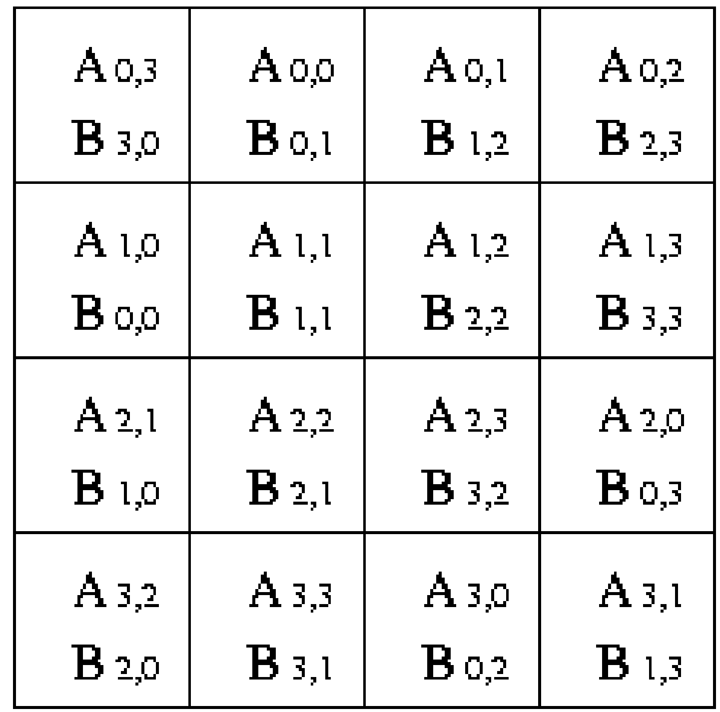


(d) Submatrix locations after first shift

Cannon's algorithm



(e) Submatrix locations after second shift



(f) Submatrix locations after third shift

Process: $P_{1,2}$

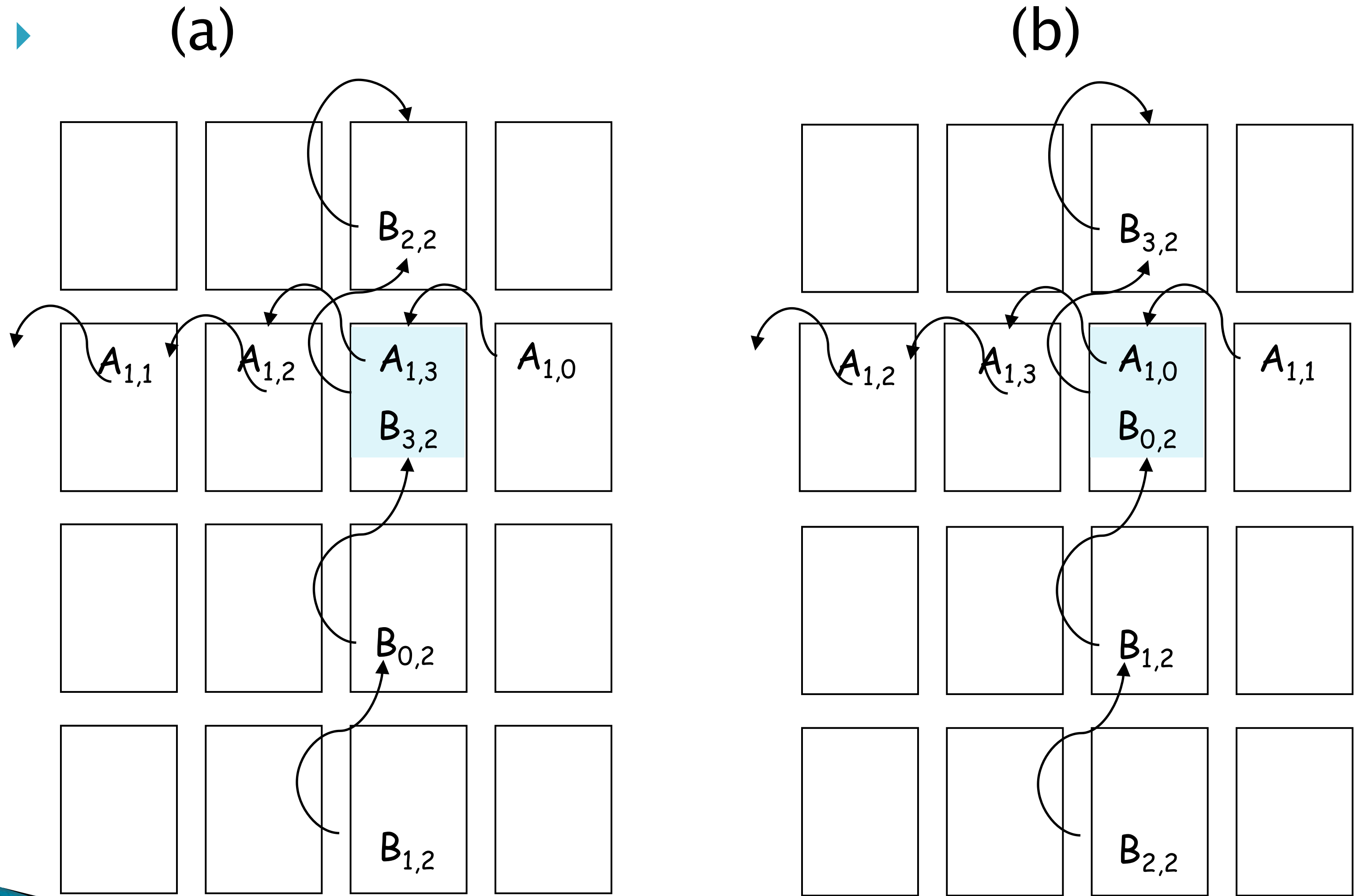
- Let us now look what happens to a

  very specific process,

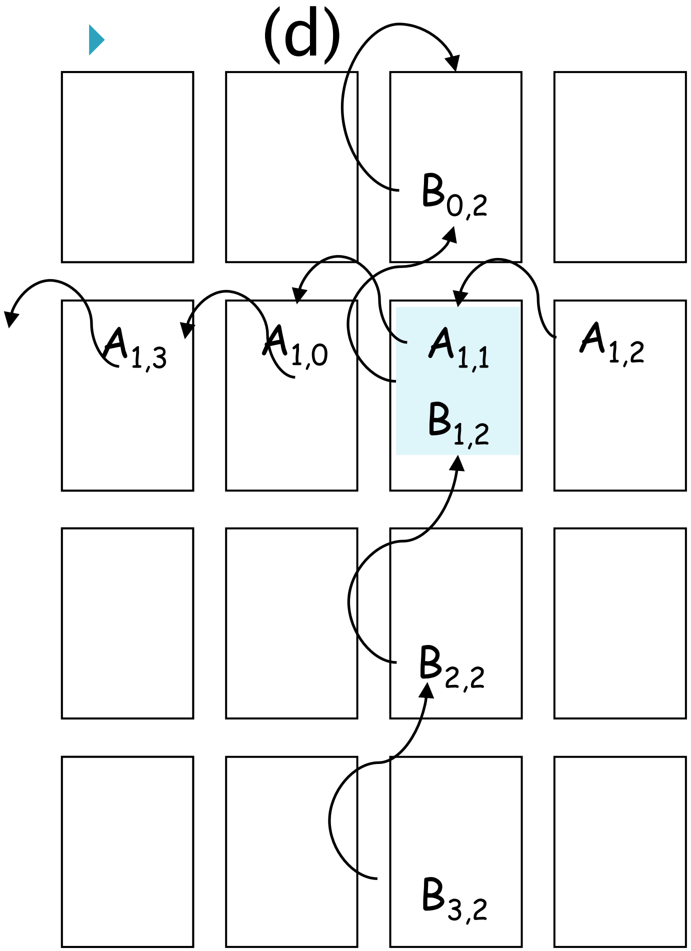
  say,

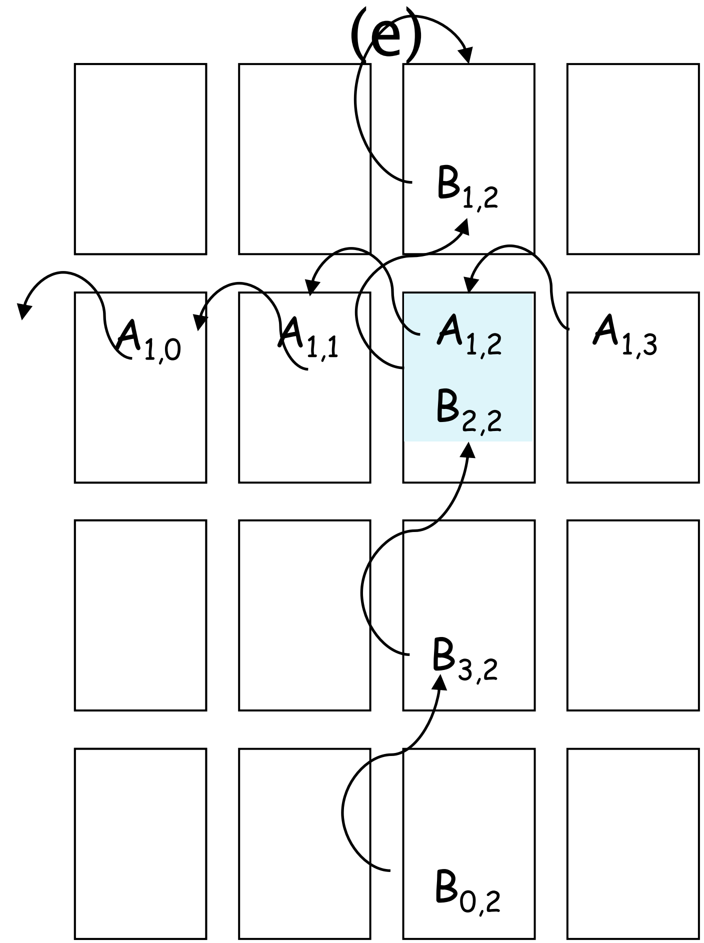
  $P_{1,2}$

Cannon: Process $P_{1,2}$



Cannon: Process $P_{1,2}$





Agglomeration

- Let A_{nxn}, B_{nxn}, C_{nxn} matrices, such that
  - Total process *p* is a square number
  - *n* is multiple of $\sqrt{p}$
- Each process computes $(n/\sqrt{p}) \times (n/\sqrt{p})$ of Matrix C.
- Requires $\sqrt{p}$ phases

Computation/Communication ratio

- Number of computations on each process = $2n^3 / p\sqrt{p}$
- Number of elements each process needs to access = $2\,(n/\sqrt{p}) \times (n/\sqrt{p})$
- Computation to communication ratio = $\dfrac{n}{\sqrt{p}}$

Comparison: two methods

Computation to Communication ratio for Cannon's algorithm with that of Row wise algorithm

$$\frac{n}{\sqrt{p}} > \frac{2n}{p} \Rightarrow \sqrt{p} > 2 \Rightarrow p > 4$$

Cannon's algorithm holds more promise for process > 4

Naïve Parallel Code: Matrix-Matrix Multiplication

```
#define NRA 50 /* Rows in Matrix A */
#define NCA 40 /* Columns in Matrix A */
#define NCB 30 /* Columns in Matrix B */
#include "mpi.h"
#include <stdio.h>

int main(int argc, int *argv[])
{
   int numtasks,taskid; /* No.of tasks and task identifier */
   int source,dest      /* Task id of message source and destination*/
   double a[NRA][NCA],b[NCA][NCB],c[NRA][NCB]; /* Matrix A, B, C */
   rows,averow,extra,offset,numworkers,i,j,k,rc; /* Miscellaneous */
   MPI_Status status;

   rc = MPI_Init(&argc,&argv);
   rc|= MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   rc|= MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   if (rc != 0)printf("\nError initializing MPI or Task ID\n");
   else printf("\nTask ID = %d\n", taskid);
   numworkers = numtasks-1;
```

Master process

```
if(taskid == 0)
{
  printf("Number of worker tasks = %d\n",numworkers);
  for (i=0; i<NRA; i++)     /* Generate data for Matrix A & B */
    for (j=0; j<NCA; j++)a[i][j]= i+j;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)b[i][j]= i*j;

  averow = NRA/numworkers;
  extra = NRA%numworkers;
  offset = 0;
  for (dest=1; dest<=numworkers; dest++)
  {
    rows = (dest <= extra) ? averow+1 : averow;
    printf("\nSending %d rows to task %d\n",rows,dest);
    MPI_Send(&offset,       1,        MPI_INT,    dest,1,MPI_COMM_WORLD);
    MPI_Send(&rows,         1,        MPI_INT,    dest,1,MPI_COMM_WORLD);
    MPI_Send(&a[offset][0],rows*NCA,MPI_DOUBLE,dest,1,MPI_COMM_WORLD);
    MPI_Send(&b,            NCA*NCB, MPI_DOUBLE,dest,1,MPI_COMM_WORLD);
    offset = offset + rows;
  }
```

Master process (contd...)

```
  for (i=1; i<=numworkers; i++) /* Wait for results from workers */
  {
    source = i;
    MPI_Recv(&offset,1,MPI_INT,source,2,MPI_COMM_WORLD,&status);
    MPI_Recv(&rows,  1,MPI_INT,source,2,MPI_COMM_WORLD,&status);
    MPI_Recv(&c[offset][0],rows*NCB,MPI_DOUBLE,source,2,
                MPI_COMM_WORLD, &status);
  }

  printf("Here is the result matrix\n"); /* Print Results */
  for (i=0; i<NRA; i++)
  {
    printf("\n");
    for (j=0; j<NCB; j++)
      printf("%6.2f ", c[i][j]);
  }
  printf ("\n");
}
```

Worker process

```
if (taskid > 0) /* Worker Tasks */
{
  MPI_Recv(&offset,1,        MPI_INT,    0,1,MPI_COMM_WORLD,&status);
  MPI_Recv(&rows,  1,        MPI_INT,    0,1,MPI_COMM_WORLD,&status);
  MPI_Recv(&a,     rows*NCA,MPI_DOUBLE, 0,1,MPI_COMM_WORLD,&status);
  MPI_Recv(&b,     NCA*NCB, MPI_DOUBLE, 0,1,MPI_COMM_WORLD,&status);
  for (k=0; k<NCB; k++)
    for (i=0; i<rows; i++)
    {
      c[i][k] = 0.0;
      for (j=0; j<NCA; j++)
        c[i][k] = c[i][k] + a[i][j] * b[j][k];
    }
  MPI_Send(&offset, 1,        MPI_INT,    0, 2, MPI_COMM_WORLD);
  MPI_Send(&rows,   1,        MPI_INT,    0, 2, MPI_COMM_WORLD);
  MPI_Send(&c,      rows*NCB, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
}
 MPI_Finalize();
} /* End main */
```

Bibliography

Text Books

1. "Parallel Programming in C with MPI and OpenMP", by Michael J Quinn, Tata McGraw-Hill, New Delhi (2005)

2. "Introduction to Parallel Computing", by Anantha Grama, Anshul Gupta, George Karypis & Vipin Kumar, Pearson Education, New Delhi (2004)
.

3. "Parallel Programming with MPI", by Peter Pacheco, Morgan Kaufmann, San Francisco, USA (1997).

- Schematic Graphics adopted from Ref. 1 & 2

Thank You

Q&A