UNIT IV

CHAPTER 4

Analytical Modelling of Parallel Programs

Syllabus Topics

Analytical Models: Sources of overhead in Parallel Programs, Performance Metrics for Parallel Systems, and The effect of Granularity on Performance, Scalability of Parallel Systems, Minimum execution time and minimum cost, optimal execution time. Dense Matrix Algorithms: Matrix-Vector Multiplication, Matrix-Matrix Multiplication.

Syllabus Topic : Sources of Overhead in Parallel Programs

4.1 Sources of Overhead in Parallel Programs

- If I use two processors, shouldn't my program run twice as fast?
- No - a number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.
- The execution profile of a hypothetical parallel program executing on eight processing elements as shown in the Fig. 4.1.1. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

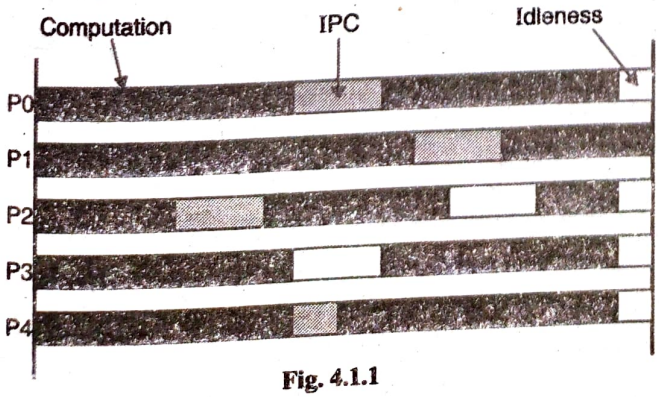


Fig. 4.1.1

- A program execution speedup automatically improves if multiple hardware resources are used instead of single copy of such resources. This rarely happen because of a number of reasons due to which overheads occurs in parallel program executions. The analysis of the performance of parallel programs is based upon the understanding of the overheads associated with the parallel programming.
- A parallel program contains statements for inter process communications, actual computations (problem solving through serial execution statements) and statements require for parallelization of the sequential computations (computations not performed by the serial formulation). This way a parallel program has to spend times in different activities also, this includes idleness and other just now mentioned computations in previous lines.

The following are the possible sources of overheads may occurs in parallel programs :

- **Inter process communications :** The processing elements in any parallel systems need to communicate with each other for exchange of controls and data (intermediate results). The time spent by the processing elements in communication with each other is generally considered as the source of overhead in parallel programming field.

- **Idling :** Processes may be idle because of load imbalance, synchronization, or serial components. E.g., the building-up phase in parallel quick sort.
- The idleness of processing elements involved to perform parallel computations occurs due to synchronization, load imbalance and strictly required to perform serial computations in the programs. The overall task may not be divided into equal number of subtasks and assigned to different processing elements. Due to this uneven distribution of tasks make some processors finish their computation earlier and wait from the reply from other processors. This means because of unequal workloads make some processors have to wait in idle for others and causes overheads in the system. The processing elements in some of the parallel programs must synchronize their activities in such a way that it should be clearly defined that they should meet at certain point for further processing. If all processing elements are not ready for synchronization at the same time, then the ones that are ready sooner will be idle until all the rest are ready.

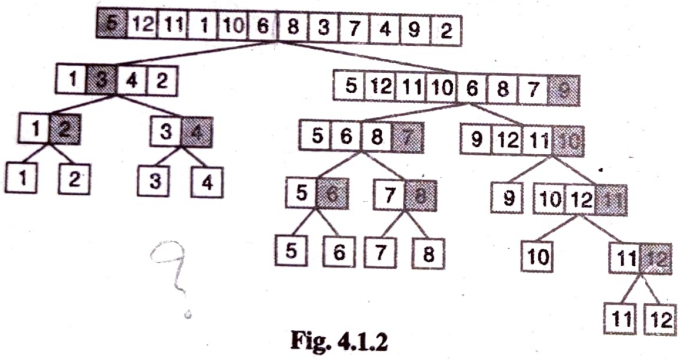


Fig. 4.1.2

- **Excess Computation :** This refers to the portion of the computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication. E.g, the consolidation of grid contents after row-sort to establish heap structure.
- The sequential programs which run efficiently and considered as fastest to solve a particular problem need extra steps so that it can run in parallel. The difference in computation performed by the parallel program and the best serial program is the excess computation overhead incurred by the parallel program.

Syllabus Topic : Performance Metrics for Parallel Systems

4.2 Performance Metrics for Parallel Systems

The performance of parallel programs are analyzed to determine the best algorithm and the system on which it can be executed most efficiently. There are a number of metrics used based on the analysis of the performance of parallel programs.

☞ Execution Time

- The serial runtime (denoted by T_S or T_1) of a program is the elapsed time between the start and the end of its execution on a sequential computer.
- The parallel runtime with p processing elements (denoted by T_P) is the elapsed time from the moment the first processor starts to the moment the last processor finishes its execution.
- Let T_{all} be the total time collectively spent by all the processing elements. We have $T_{all} = p \times T_P$
- Observe that $T_{all} - T_S$ is then the total time spent by all combined processors in non-useful work. This is called the total overhead. The overhead function (T_0) is therefore given by :

$$T_0 = p \times T_P - T_S$$

☞ Speedup

- While dealing with parallel computations we always ask a question that, what benefit we get from parallelism. After all, what we want is that the program execution should be completed at a shorter time. So we need to compute the speedup.
- Speedup is described differently for single processor system and the parallel processor systems. Speedup on a parallel system is considered on the basis of how fast a program finishes its executions on a parallel system as compare to serial system. Therefore Speedup (S) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processing elements.

$$S = \frac{T_S}{T_P}$$

We also express speedup as follows :

$$S(p) = \frac{T_1}{T_p}$$

- The secondary objective of using more than 1 PEs is to take advantage of the larger consolidated amount of cache memory to run a big program. Running a big program on a single PE can incur too much cache miss thus causing thrashing effect.

☞ Performance Metrics : Example

- Consider the problem of adding n numbers by using n processing elements. If n is a power of two, we can perform this operation in **log n** steps by propagating partial sums down a logical binary tree of processors.

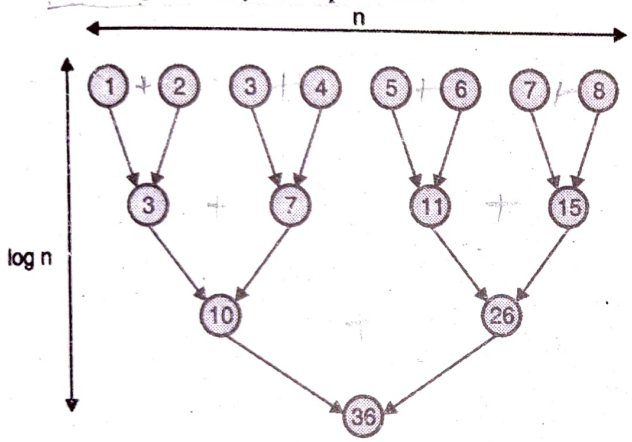


Fig. 4.2.1

- **In Summation Form :** Computing the global sum of 16 partial sums using 16 processing elements. Σ_{ji} denotes the sum of numbers with consecutive labels from i to j.

Fig. 4.2.2 : Initial data structure and the first communication step

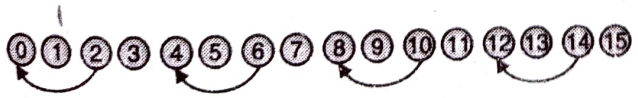


Fig. 4.2.3 : Second communication step

Fig. 4.2.4 : Third communication step

Fig. 4.2.5 : Fourth communication step

Fig. 4.2.6 : Accumulation of sum at processing element 0 after the final communication

- If an addition takes constant time, say, t_c and communication of a single word takes time $t_s + t_w$, we have the parallel time $T_P = \theta(\log n)$
- We know that $T_S = \theta(n)$, which can be implemented by a for loop as follows : for (i=0; i<n;i++) sum += a[i].

  Speedup S is given by :

$$S = \frac{T_S}{T_P} = \theta\left(\frac{n}{\log n}\right)$$

- For a given problem, there might be many serial algorithms available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees. What if T_S (or T_1) is not optimal? Do we over represent the speedup? For the purpose of computing speedup, we always consider the best sequential program as the baseline.

☞ Speedup Bounds

Speedup can be as low as 0 (the parallel program never terminates). This happens for infinite loop, dead lock and/or live lock situations. Speedup, in theory, should be upper bounded by p - after all, we can only expect a p-fold speedup if we use p times as many resources. A speedup greater than p is possible only if each processing element spends less than time T_S / p solving the problem. This is called super-linear speedup and cannot be guaranteed. Why?

Answer : Otherwise we can construct a new sequential algorithm using time-sliced approach to simulate the parallel algorithm to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup calculation.

☞ Super Performance Metrics : Super-linear Speedups

- One reason for super linearity is that the parallel version does less work than corresponding serial algorithm. Searching an unstructured tree for a node with a given label, 'S', on two processing elements using depth first traversal. The two-processor version with processor-0 searching the left subtree and processor-1 searching the right subtree expands only the shaded nodes before the solution is found.

- The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.

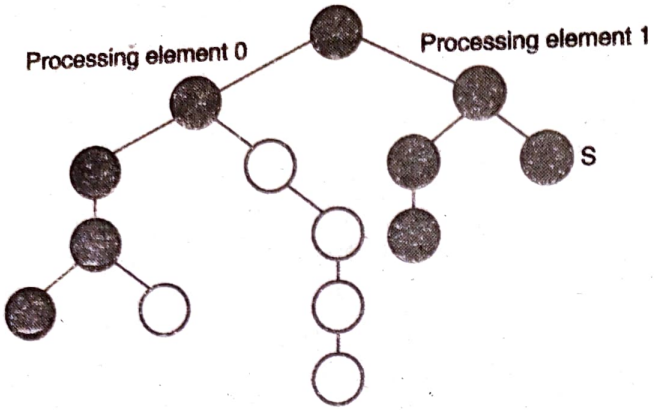


Fig. 4.2.7

$$T_1 = 14\,c$$

$$T_2 = 5\,c$$

$$S = \frac{14\,c}{5\,c}$$

$$= 2.8 \text{ (super-linear as } p = 2)$$

☞ Super-linear Speedup Due to linear Speedup Due to Higher Cache Hit Rate

- We call it resource-based super-linearity - the higher aggregate cache/memory bandwidth can result in better cache-hit ratios, and therefore super-linearity.

Example : A processor with 64KB of cache yields an 80% hit ratio. If cache access time is 2 ns, and DRAM access time is 100 ns. The effective memory access time is $2 \text{ ns} \times 0.8 + 100 \text{ ns} \times 0.2 = 21.6 \text{ ns}$.

- If the computation is memory bound and perform 1 FLOP/memory access, the processing rate is :

$$\frac{1}{21.6 \text{ ns}} = 46.3 \text{ MFLOPS}$$

- Now if two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory. If the remote memory access time is 400 ns, the effective memory access time is

$$2 \text{ ns} \times 0.9 + 100 \text{ ns} \times 0.08 + 400 \text{ ns} \times 0.02 = 17.8 \text{ ns}$$

The processing rate in each processor is

$1/17.8 \text{ ns} = 56.18$ MFLOPS

The total processing rate for two processors is

$56.18 \times 2 = 112.36$ MFLOPS,

and this corresponds to a speedup of $112.36/46.3 = 2.43$ (super-linear)

☞ Efficiency

Efficiency is a measure of the fraction of time for which a processing element is usefully employed.

Mathematically, it is given by

$$E = \frac{S}{P}$$

Following the bounds on speedup, efficiency in general can be as low as 0 and as high as 1. $E > 1$ can happen but cannot be guaranteed.

The speedup of adding numbers on processors is given by

$$S = \frac{n}{\log n}$$

and the efficiency is given by

$$E = \frac{\theta\left(\frac{n}{\log n}\right)}{n}$$

$$= \theta\left(\frac{1}{\log n}\right)$$

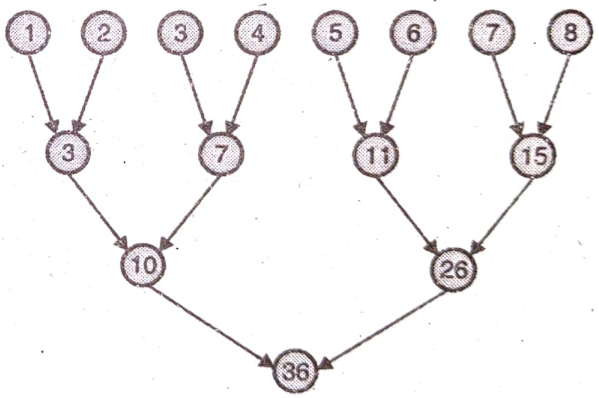
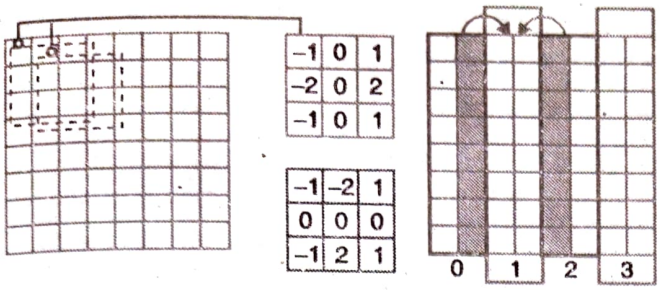


Fig. 4.2.8

☞ Parallel Time, Speedup, and Efficiency Example

Consider the problem of edge-detection in images. The problem requires us to apply a 3×3 template to each pixel. If each multiply-add operation takes time t_c, the serial time for an $n \times n$ image is given by

$$T_S = t_c n^2$$



(a) (b) (c)

Fig. 4.2.9 : Example of edge detection

(a) An 8×8 image

(b) Typical template for detecting edges

(c) Partitioning of the image across four processors with shaded regions indicating image data that must be communicated from neighboring processors to processor 1

☞ Edge Detection

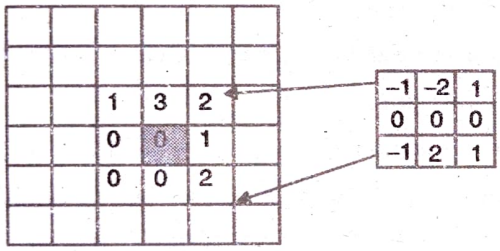


Fig. 4.2.10

$$c = 1 \times (-1) + 3 \times (-2) + 2 \times 1 + 0 \times 0 + 0 \times 0 + 1 \times 0 + 0 \times (-1) + 0 \times 2 + 2 \times 1 ;$$

if (c > threshold edge = true) ;

else edge = false ;

- One possible parallelization partitioning scheme is to divide the image equally into vertical segments, each with n2 / p pixels – which is the number of pixels to be processed by each PE.
- The boundary of each segment is 2n pixels (n pixels on the left, and n pixels on the right). This is also the number of pixel values that will have to be communicated. This takes time $2(t_s + t_w n)$.
- Templates (with 9 values) may now be applied to all n^2 / p pixels in time $9\, t_c n^2 / p$.

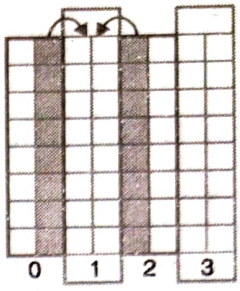


Fig. 4.2.11

- The total time for the edge detection algorithm is therefore given by the sum of computation time and communication time :

$$T_P = 9\, t_c \frac{n^2}{p} + 2 (t_s + t_w n)$$

The corresponding values of speedup (T_1/T_P) and efficiency (S/P) are given by :

$$S = \frac{9\, t_c n^2}{9\, t_c \frac{n^2}{p} + 2 (t_s + t_w n)}$$

$$\text{and} \quad E = \frac{1}{1 + \frac{2p (t_s + t_w n)}{9\, t_c n^2}}$$

☞ Cost of a Parallel System

Cost is the product of parallel runtime and the number of processing elements used $(p \times T_P)$. Cost reflects the sum of the time that each processing element spends solving the problem. A parallel system is said to be cost-optimal if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost.

$$\text{Since } E = \frac{(T_S) / (T_P)}{p}$$

$$= \frac{(T_s)}{p \times T_P},$$

for cost optimal systems $E = \frac{p}{p} = 0(1)$

Cost $(p \times T_P)$ is sometimes referred to as work or processor-time product.

☞ Cost of a Parallel System: Example Cost of a Parallel System

Consider the problem of adding numbers on processors. We have, $T_P = \log n$ (for $p = n$).

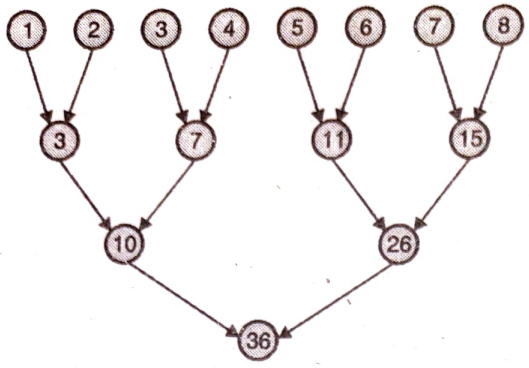


Fig. 4.2.12

The cost of this system is given by $p\ T_P = n \log n$.

Since the serial runtime of this operation is $\theta(n)$, the algorithm is not cost optimal.

Impact of Non-Cost Optimality

- Consider a sorting algorithm that uses n processing elements to sort a list in time $(\log n)^2$ – e.g., Odd-Even Merge Sort. Since the serial runtime of a (comparison-based) sort is n log n, the speedup and efficiency of this algorithm are given by n/log n and 1/log n respectively. The $p\ T_P$ product of this algorithm is $n (\log n)^2$ showing that this algorithm is not cost optimal by a factor of log n.
- If $p < n$, assigning n tasks to p processors gives $T_P = n (\log n)^2 / p$. The corresponding speedup (T_1 / T_p) of this formulation is p / log n. This speedup goes down as the problem size n is increased for a given p.

Syllabus Topic : Effect of Granularity on Performance

4.3 Effect of Granularity on Performance

- Often, using fewer processors improves performance of parallel systems. Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called scaling down a parallel system. (In real life this is called downsizing.)
- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign (or to cluster) these virtual processors equally to scale down the available processors.
- Since the number of processing elements decreases by a factor of n / p, the computation at each processing element increases by a factor of n / p.
- The communication cost should not increase by this factor since some of the virtual processors assigned to a physical processors might communicate to each other (intra-processor communication). This is the basic reason for the improvement from building granularity.

Example of Building Granularity

- Consider the problem of adding **n** numbers on **p** processing elements such that $p < n$ and both **n** and **p** are powers of 2.
- Each of the **p** processors is now assigned **n / p** virtual processors because we assume the use of virtual processors here. The first **log p** of the **log n** steps of the original algorithm are simulated in **(n / p) log p** steps on **p** processing elements. Subsequent **log n - log p** steps do not require any communication.
- The overall parallel execution time of this parallel system is : $\theta((n / p) \log p)$.
- The cost is θ **(n log p)**, which is asymptotically higher than the θ**(n)** cost of adding **n** numbers sequentially. Therefore, the parallel system is not cost-optimal.

Can we build granularity in the example in a cost-optimal fashion?

- Each processing element locally adds its **n / p** numbers in time θ **(n / p)**. The **p** partial sums on **p** processing elements can be added in time θ **(n /p)**. The Fig. 4.3.1 describes a cost-optimal way of computing the sum of 16 numbers using four processing elements.

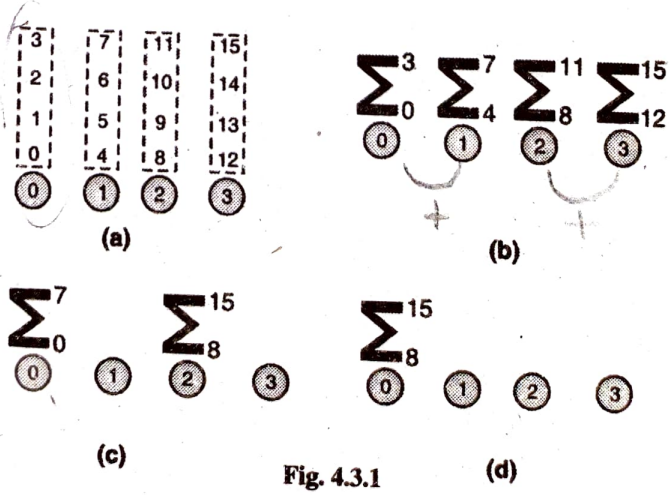


Fig. 4.3.1

- The parallel runtime of this algorithm is given by the following equation :

$$T_p = \theta (n / p + \log p),$$

The cost is $n = \Omega (p \log p)$

This is cost-optimal, so long as

$\theta (n + p \log p)$

- The parallel system is cost optimal or not can be checked by assumptions of some examples. The example discussed here demonstrates that the mapping of computations onto processing elements can determine the parallel system is cost - optimal or not. The important point to be noted here is that not all systems can be managed as a cost – optimal by scaling down the number of processing elements.

Syllabus Topic : Scalability of Parallel Systems

4.4 Scalability of Parallel Systems

- It is general practice that the testing of the program is done on smaller problems and fewer processing elements are used for their executions. But in reality these programs are executed on large number of processing elements to solve bigger problems. The techniques for evaluating the scalability of parallel programs using analytical tools are investigated here.
- How do we extrapolate performance from small problems and small systems to larger problems on larger configurations? Consider three parallel algorithms for computing an n-point Fast Fourier Transform (FFT) on 64 processing elements. Asymptotic effect is observed.

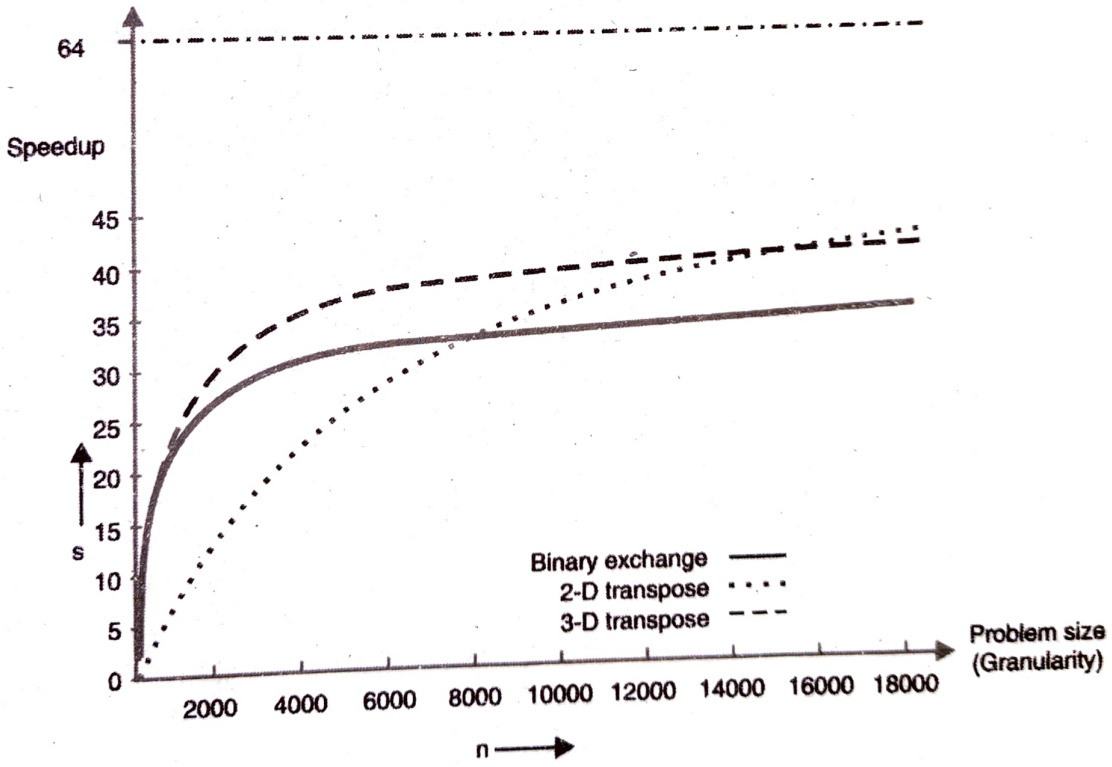


Fig. 4.4.1

Scaling Characteristics of Parallel Programs

- The efficiency of a parallel program can be written as below in Equation (4.4.1).

$$E = \frac{S}{p} = \frac{T_S}{p\,T_P} \qquad \ldots(4.4.1)$$

- The Equation (4.4.1) can be rewritten as follows in Equation (4.4.2) by including the expression for parallel overhead.

$$E = \frac{1}{1 + \frac{T_0}{T_S}} \quad \ldots(4.4.2)$$

The total overhead function T_o is an increasing function of p.

- The total overhead function here is represented by T_0 and it is an increasing function of p because some serial components included in every program. At the time of serial computations all other processing elements simply sits idle. The duration of time in which serial computation executes is represented by tserial. In consideration of serial computation time the total overhead function is given by $(p - 1) \times$ tserial.
- This indicates that the total overhead function T_0 at-least grows linearly with p. The different parameters like communication, idling and excess computations are considered and because of these the total overhead function can grow superlinearily with respect to the number of processing elements.
- The Equation (4.4.2) gives us several interesting insights about scaling of parallel programs. For a given problem size (i.e., the value of T_S remains constant), as we increase the number of processing elements, T_o increases. The overall efficiency of the parallel program goes down. This is the case for all parallel programs.

Example

- Consider the problem of adding n numbers on p processing elements.
- Suppose we assume a unit time for addition of two numbers. The local summations termed as phase-1 of the algorithm takes n/p time. The second phase of the algorithm is based on the communication and an addition at each step and it involves log p steps. The time for phase-2 is 2 log p if a single communication takes a unit time. Therefore, we can derive parallel time, speedup, and efficiency.

We have seen that :

$$T_P = \frac{n}{p} + 2 \log p$$

$$S = \frac{n}{\frac{n}{p} + 2 \log p}$$

$$E = \frac{1}{1 + \frac{2p \log p}{n}}$$

- Total overhead function T_o is a function of both problem size T_s and the number of processing elements p. In many cases, T_o grows sublinearly with respect to T_s.
- In such cases, the efficiency increases if the problem size is increased keeping the number of processing elements constant.
- For such systems, we can simultaneously increase the problem size and number of processors to keep efficiency constant. We call such systems scalable parallel systems. As we know that the cost-optimal parallel systems have an efficiency of $\theta(1)$. Scalability and cost-optimality are therefore related. A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately.

Isoefficiency Metric of Scalability

The following two points are worth noting from the above discussions of speedup and efficiency :

- For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down for all systems.
- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.
- The Fig. 4.4.2 illustrates the above two phenomena. Based upon the above two observations in phase-1 and phase-2 of the algorithms a scalable parallel system can be defined. A scalable parallel system is a system in which efficiency of the system remains constant as the number of processing elements increased, provided that the problem size is also increased.
- It is important to find out the rate at which the problem size is required to increase with respect to the increase in the number of processing elements to keep efficiency remains fixed.
- In order to maintain a fixed efficiency as the number of processing elements increases the problem size must increase at different rates for parallel systems. This rate actually determines the scalability of the parallel systems. The slower

this rate, the better. Before we formalize this rate, we define the problem size W as the asymptotic number of operations associated with the best serial algorithm to solve the problem.

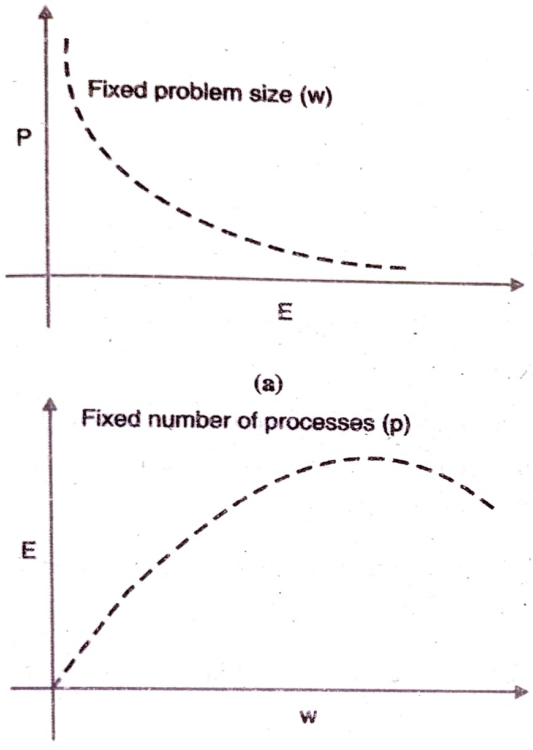


Fig. 4.4.2

Variation of efficiency : (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.

☞ The Isoefficiency Function

The execution time of a parallel system can be expressed as a function of problem size, overhead function, and the number of processing elements.

We can write parallel runtime as :

$$T_P = \frac{W + T_0(W, p)}{p}$$

The resulting expression for speedup is

$$S = \frac{W}{T_P}$$

$$= \frac{W_p}{W + T_0(W, p)}$$

Finally, we write the expression for efficiency as

$$E = \frac{S}{P}$$

$$= \frac{W_P}{W + T_0(W, p)}$$

$$= \frac{1}{1 + T_0(W, p) / W}$$

Efficiency can be maintained at a fixed value between 0 and 1 for a scalable system if the ratio T_0/W is maintained at a constant value.

For a desired value E of efficiency :

$$E = \frac{1}{1 + T_0(W, p) / W},$$

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

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

If $K = E / (1 - E)$ is a constant depending on the efficiency to be maintained, since T_o is a function of W and p, we have

$$W = KT_0(W, p)$$

The problem size W in above equation can usually be obtained as a function of p by algebraic manipulations to keep efficiency constant. This function is called the ***isoefficiency function***. This function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processing elements

☞ Isoefficiency Metric : Example

The overhead function for the problem of adding n numbers on p processing elements is approximately $2p \log p$.

Substituting T_o by $2p \log p$, we get

$$W = K\,2p \log p$$

Thus, the asymptotic isoefficiency function for this parallel system is

$$\theta(p \log p)$$

If the number of processing elements is increased from p to p', the problem size (in this case, n) must be increased by a factor of $(p' \log p') / (p \log p)$ to get the same efficiency as on p processing elements.

Consider a more complex example where

$$T_0 = p^{3/2} + p^{3/4} W^{3/4}$$

Using only the first term of T_o in Equation below

$$W = K T_0 (W, p)$$

We get

$$W = K 2p \log p$$

Thus, the asymptotic isoefficiency function for this parallel system is

$$\theta (p \log p)$$

If the number of processing elements is increased from p to p′, the problem size (in this case, n) must be increased by a factor of (p′ log p′) / (p log p) to get the same efficiency as on p processing elements.

Consider a more complex example where

$$T_0 = p^{3/2} + p^{3/4} W^{3/4}$$

Using only the first term of T_o in Equation below

$$W = K T_0 (W, p)$$

we get,

$$W = K p^{3/2}$$

Using only the second term, Equation ($T_0 = p^{3/2} + p^{3/4} W^{3/4}$) yields the following relation between W and p :

$$W = K p^{3/4} W^{3/4}$$

$$W^{1/4} = K p^{3/4}$$

$$W = K^4 p^3$$

The larger of these two asymptotic rates determines the isoefficiency. This is given by $\theta (p^3)$

☞ Cost-Optimality and the Isoefficiency Function

A parallel system is cost-optimal if and only if

$$p T_P = \theta (W)$$

From this, we have :

$$W + T_0 (W, p) = \theta (W)$$

$$T_0 (W, p) = O (W)$$

$$W = \Omega (T_0 (W, p))$$

If we have an isoefficiency function $f(p)$, then it follows that the relation $W = \Omega(f(p))$ must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up.

☞ Lower Bound on the Isoefficiency Function

For a problem consisting of W units of work, no more than W processing elements can be used cost-optimally.

The problem size must increase at least as fast as $\theta(p)$ to maintain fixed efficiency; hence, $\Omega(p)$ is the asymptotic lower bound on the isoefficiency function.

☞ Degree of Concurrency and the Isoefficiency Function

- The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency.*
- If $C(W)$ is the degree of concurrency of a parallel algorithm, then for a problem of size W, no more than $C(W)$ processing elements can be employed effectively.

☞ Degree of Concurrency and the Isoefficiency Function : Example

- Consider solving a system of equations in variables by using Gaussian elimination ($W = \theta (n^3)$)
- The n variables must be eliminated one after the other, and eliminating each variable requires $\theta(n^2)$ computations. At most $\theta(n^2)$ processing elements can be kept busy at any time. Since $W = \theta (n^3)$ for this problem, the degree of concurrency $C(W)$ is $\theta(W^{2/3})$. Given p processing elements, the problem size should be at least $\Omega(p^{3/2})$ to use them all.

Syllabus Topic : Minimum Execution Time and Minimum Cost-Optimal Execution Time

4.5 Minimum Execution Time and Minimum Cost-Optimal Execution Time

- There are some of things always takes into considerations in the field of High performance computing, such as the pace in which a problem can be solved and the minimum execution time of a parallel algorithm. These are based on the assumptions that the number of processing elements is not a constraint. There are two possibility of increasing the number of processing elements : either the parallel execution time decreases or the computation time increases. The increase in

number of processing elements on a given problem size decreases the execution time and it approaches asymptotically to a minimum value. The execution time starts rising after attaining a minimum value.

- Often, we are interested in the minimum time to solution.
- We can determine the minimum parallel runtime T_P^{min} for a given W by differentiating the expression for T_P w.r.t. p and equating it to zero.

$$\frac{d}{d_p} T_P = 0$$

If p_0 is the value of p as determined by this equation, $T_P(p_0)$ is the minimum parallel time.

☞ Minimum Execution Time : Example

Consider the minimum execution time for adding n numbers in parallel is given as :

$$T_P = \frac{n}{p} + 2 \log p$$

Setting the derivative with respect to p to zero, we have $p = n/2$. The corresponding runtime is

$$T_P^{min} = 2 \log n$$

One may verify that this is indeed a min by verifying that the second derivative is positive. Note that at this point, the formulation is not cost-optimal.

☞ Minimum Cost-Optimal Parallel execution Time for addition of n numbers

Let $T_P^{cost\_opt}$ be the minimum cost-optimal parallel time. If the isoefficiency function of a parallel system is $\theta(f(p))$, then a problem of size W can be solved cost-optimally if and only if $W = \Omega(f(p))$.

In other words, for cost optimality, $p = O(f^{-1}(W))$.

For cost-optimal systems, $T_P = \theta(W/p)$,

therefore represented by the below equation :

$$T_P^{cost\_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right)$$

Example

Consider the problem of adding n numbers. The isoefficiency function $f(p)$ of this parallel system is $\theta(p \log p)$. From this, we have $p \approx n / \log n$. At this processor count, the parallel runtime is given as :

$$T_P^{cost\_opt} = \log n + \log\left(\frac{n}{\log n}\right)$$

$$= 2 \log n - \log \log n$$

Note that both T_P^{min} and $T_P^{cost\_opt}$ for adding n numbers are $\theta(\log n)$. This may not always be the case.

Syllabus Topic : Dense Matrix Algorithms

4.6 Dense Matrix Algorithms

- Due to their regular structure, parallel computations involving matrices and vectors readily lend themselves to data-decomposition. Typical algorithms rely on input, output, or intermediate data decomposition. Most algorithms use one and two-dimensional block, cyclic, and block-cyclic partitioning.
- The run-time performance of such algorithms depends on the amount of overheads incurred as compared to the computation workload. As a rule of thumb, good speedup can be achieved if the computation granularity is able to outweigh the overheads such as the communication cost, consolidation cost - algorithm penalty, data packaging, etc.

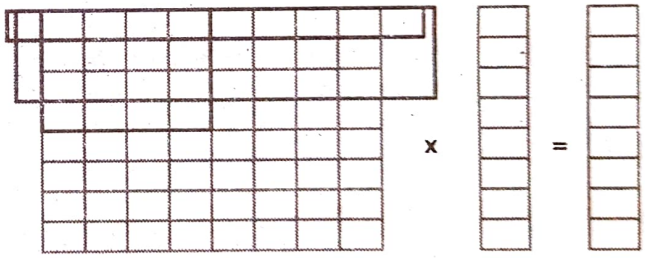


Fig. 4.6.1

Syllabus Topic : Matrix-Vector Multiplication

4.6.1 Matrix-Vector Multiplication

This section covers the details about the matrix and vector multiplication. Here the matrix A is a dense matrix of n x n which will be multiplied with the vector V of size n x 1. The result of this multiplication is also a vector let's call it as R.

☞ Sequential algorithm for the multiplication

```
Algorithm multiply_matrix_vector(A, V, R)
{
    for( I = 0 to n-1 )
```

```
{
    R[i] = 0;
    for(j = 0 to n-1 )
    {
        R[i] = R[i] + A[i][j] * V[j];
    }
}
```

This algorithm requires n^2 multiplications and additions. The sequential runtime of the algorithm is $W = n^2$ on the basis of assumption that the multiplication and addition pair takes unit time.

The different partitioning schemes such as row-wise 1-D, column-wise 1-D and a 2-D can be used and parallel formulation of matrix and vector multiplication can be performed.

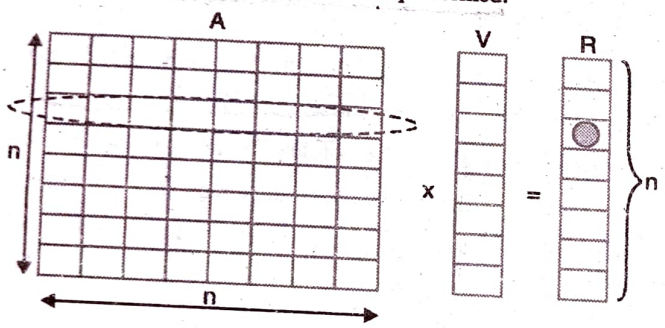


Fig. 4.6.2 : Multiplication of a matrix A and a vector V produces result in R.

4.6.2 Row-wise 1-D Partitioning

The Row-wise Block-Striped Decomposition approach for matrix – vector multiplication is covered in this section.

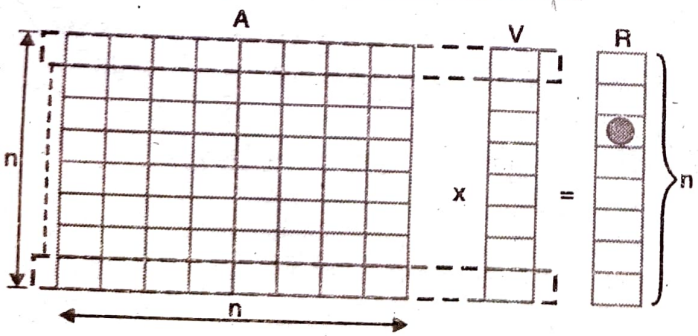


Fig. 4.6.3 : A complete row is stored in each processor

One Row Per Process

In the first case the matrix A of size $n \times n$ is partitioned among n processors in which a complete row of the matrix is stored in each processor. The vector V of size $n \times 1$ is distributed among processes such that each process owns one of its elements. The initial distribution of matrix and vector for row-wise block 1-D partitioning is shown in the Fig. 4.6.4.

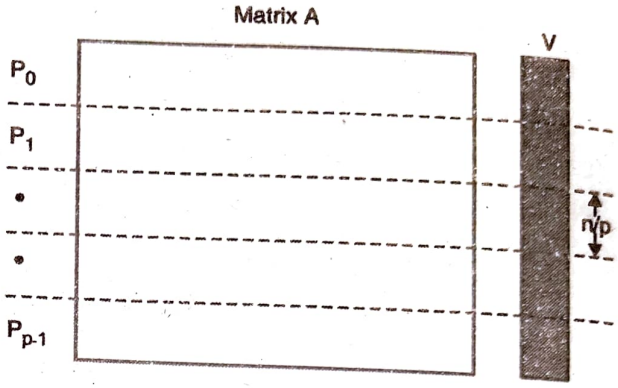


Fig. 4.6.4

- The distribution to perform multiplication of matrix and vector is described as, all elements of first row of matrix A and first element of vector V is assigned to process P_0. In general this is detailed as initially V[i] and A[i,0],A[i,1]...,A[i,n-1] are contained in process P_0 and it will compute first element in the result vector as R[i]. Here each row of the matrix is multiplied with vector V and therefore, it is required that entire vector is to be assigned to every process.

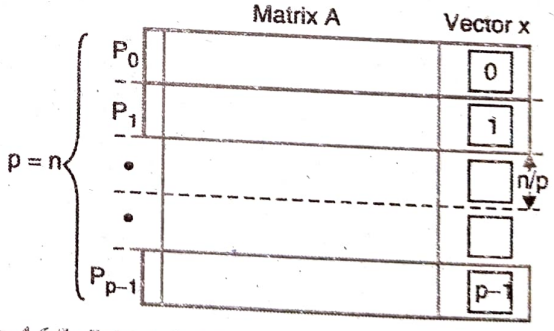


Fig. 4.6.5 : Initial distribution of matrix elements and the starting vector

- In each process initially only one element of the vector is required, this way all elements of the vector is required to all processes. This is because each process starts with only one element of Vector V, an all-to-all broadcast is required to distribute all the elements V[i] to all the processes before performing the computations shown in the following equation.

$$R[i] = \sum_{j=0}^{n-1} (A[i, j] \times v[j])$$

- This is made available through the distribution of elements using an all - to - all broadcast approach. The Fig. 4.6.6 shows this communication.

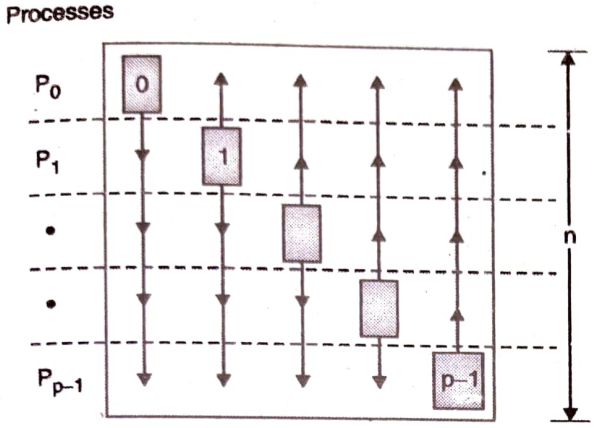


Fig. 4.6.6 : Distribution of vectors using all-to-all approach

- Once the elements of the vector V are distributed among all the processes, the elements of the resultant vector are computed using the approach shown below :

$$R[i] = \sum_{j=0}^{n-1} (A[i, j] \times v[j])$$

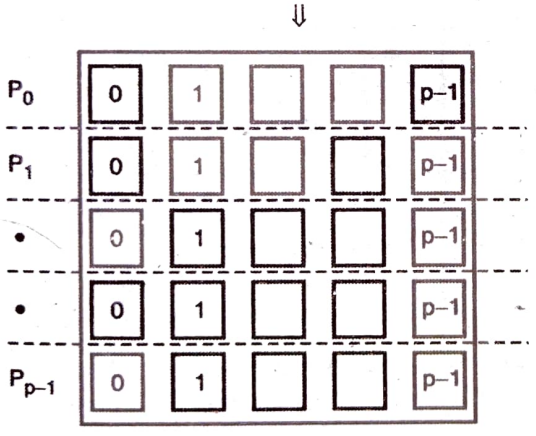


Fig. 4.6.7 : After the broadcast vector is distributed to each process

☞ Parallel Run Time

The time required by the all – to – all broadcast of the vector elements among n processes is $\theta(n)$ on any architecture. The time required for multiplication of a single row of A with V is also $\theta(n)$. Therefore, $\theta(n)$ time is required for completion of entire procedure by n processes. This results of time $\theta(n^2)$ in a process – time product. The sequential algorithm complexity is $\theta(n^2)$ and thus the parallel algorithm is cost – optimal. This way the parallel time of the computation is $\theta(n)$ as shown in Fig. 4.6.8.

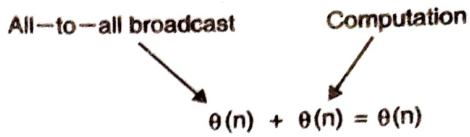


Fig. 4.6.8

☞ Using Fewer than n Processes (Row-wise 1-D Partitioning when p < n)

Now consider the case in which, number of processes used is less than the elements n that is p < n. Again the matrix is partitioned among the processes by using block 1-D partitioning. Initially n/p complete rows of the matrix and n/p size portion of the vector are stored in each process. In this case also, the entire vector is required in each process because vector V must be multiplied with each row of the matrix. Because of this fact again all – to – all broadcast approach is required. The Fig. 4.6.9 depicts this arrangements.

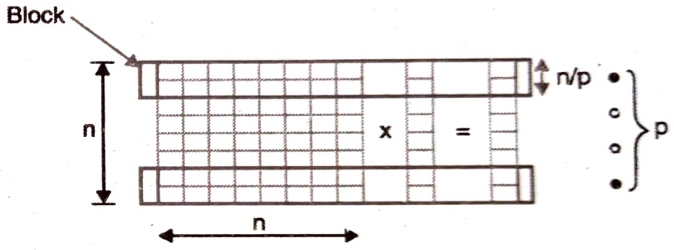


Fig. 4.6.9

☞ The important points in this case is listed below

- The case here is when p < n and we use blocks in partitions.
- Each process initially stores n/p complete rows of the matrix, and a portion of the vector of size n/p.
- The all-to-all broadcast takes place among p processes and involves messages of size n/p.
- This is followed by n/p rows of local dot products where each row takes O(n) computation time.
- The total computation time = n (per row) x n/p (rows in a block) = n^2/p

The messages of size n/p involves in this case and all - to - all broadcast approach takes place among p processes. Once the elements are made available after the communication, each process multiplies its n/p rows with the vector V to produce n/p elements of the result vector.

Parallel Run Time

An all - to - all broadcast of messages of size n/p among p processes takes time as below :

$$T = \sum_{i=0}^{\log p} (t_s + 2^{i-1} t_w m) \quad (m = \text{Message size})$$

$$= t_s \log p + t_w m (p - 1)$$

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^{\log p - 1}$$

$$= \frac{1 (2^{(\log p - 1) + 1} - 1)}{2 - 1} = p - 1$$

Now $m = \frac{n}{p}$, we have

$$T = t_s \log p + \left(t_w \times \frac{n}{p}\right) \times (p - 1)$$

For large p, this can be approximated

$$\simeq t_s \log p + t_w \times n$$

Thus, the parallel run time of matrix-vector multiplication based on rowwise 1-D partitioning ($p < n$) is

$$T_P = \underbrace{\frac{n^2}{p}}_{\text{Computation}} + \underbrace{t_s \log p + t_w n}_{\text{All-to-all broadcast of vector elements}}$$

4.6.3 2-D Partitioning

In this section the parallel matrix - vector multiplication is focused for the case where matrix elements are distributed among the processes using a block 2-D partitioning.

One Element Per Process

- First we will discuss the simple case in which a matrix of size n × n is partitioned among n^2 processes (processors). Where each process (processor) is assigned with a single element of the matrix. The vector V of size n × 1 is distributed only in the last column of n processors. Each one of the n processors owns one element of the vector. This arrangement is shown in the Fig. 4.6.10.
- The algorithm for the multiplication of vector and matrix performs the multiplication of elements of vector V with the corresponding elements in each row of the matrix.
- The computation can be perform according to the algorithm steps when the vector elements are distributed such that the i^{th} element of the vector is available to the i^{th} element of each row of the matrix.

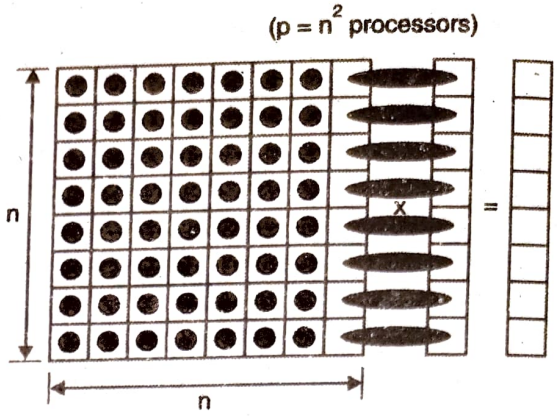


Fig. 4.6.10 : Vector V is distributed only in the last column of n processors

- For the requirements of the purpose we must first align the vector with the matrix appropriately. The first communication step for the 2-D partitioning aligns the vector V along the principal diagonal of the matrix. This is shown in the Fig. 4.6.11.

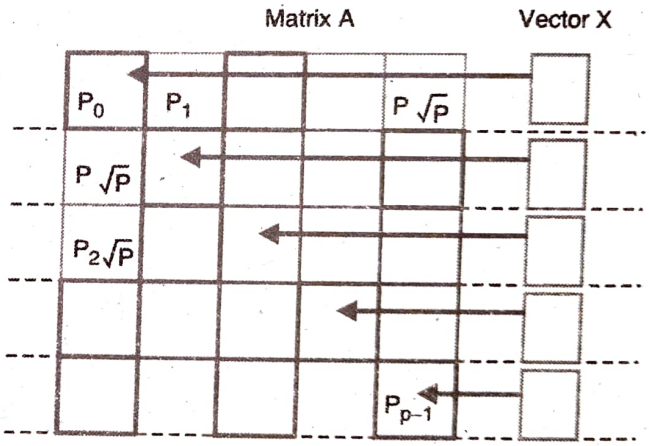


Fig. 4.6.11 : Initial data distribution and communication steps to align the vector along the diagonal

- The vector is most often stored along the diagonal instead of the last column. This step is not required in such situations.
- In the second step, the vector elements are copied from each diagonal process to all the processes in the corresponding column using n simultaneous broadcasts among all processors in the column.
- The Fig. 4.6.12 shows this step. In this step n simultaneous one - to - all broadcast operations, one in each column of processes are performed.

- Once these two communication steps are completed, each process multiplies its matrix element with the corresponding element of *V*.

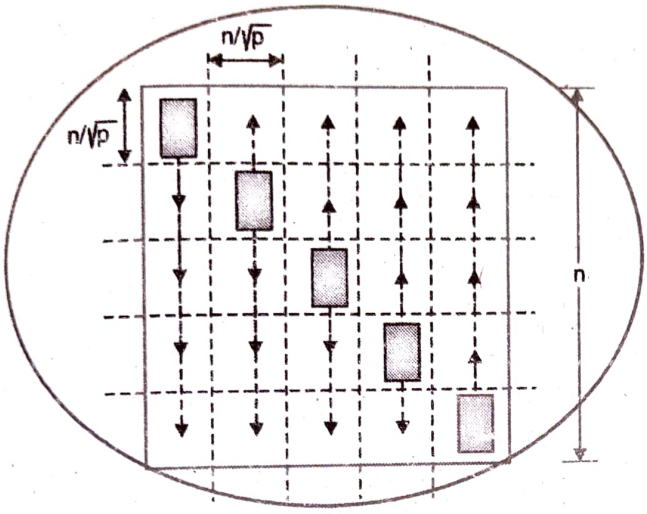


Fig. 4.6.12 : One - to - all broadcast of portions of the vector along process column

- Finally the result vector R is computed by performing the computation in which the products computed for each row must be added and the result sum is stored in the last column of processes. The steps of this operation is shown below.

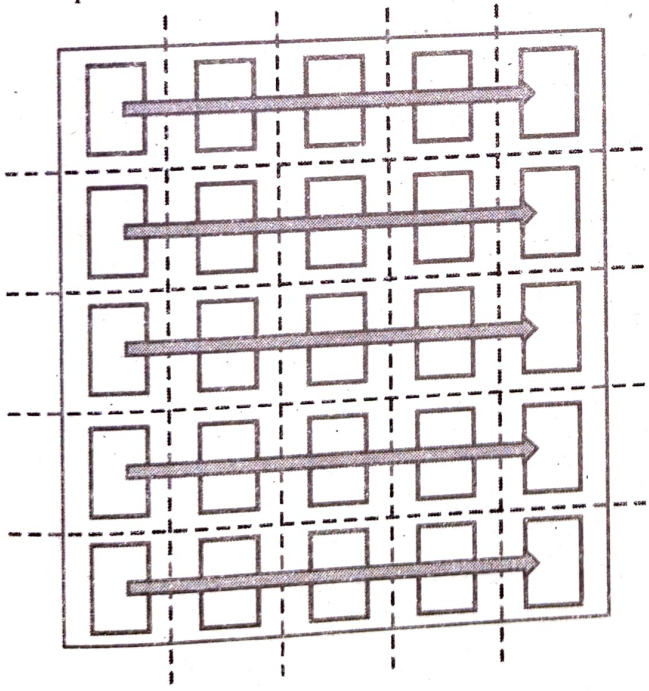


Fig. 4.6.13 : All - to - one reduction of the partial results

- After completion of the reduction step, the parallel matrix - vector multiplication is completed. The distribution of the result vector finally is done after reduction and it is like as shown below :

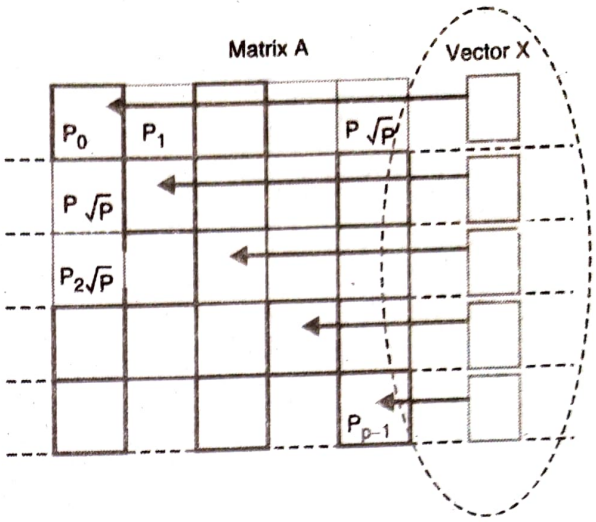
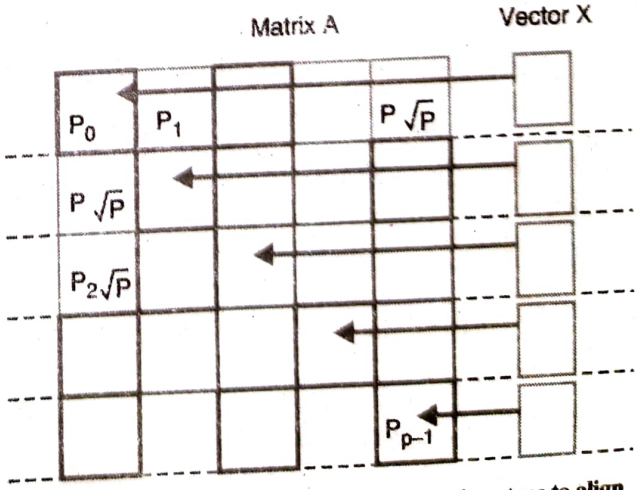


Fig. 4.6.14 : Final distribution of the result vector

Parallel Run Time

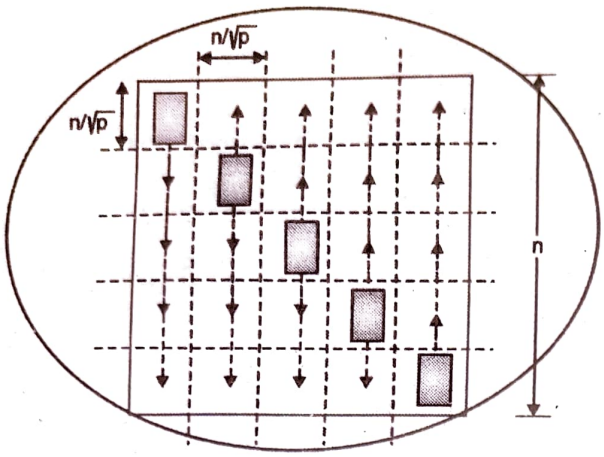
- Three communication operations used in this algorithm are namely one - to - one communication, one - to - all broadcast and all - to - one reduction.
- These communications are used for different purposes with different time complexities as shown in Fig. 4.6.15.
- The alignment of the vector along the main diagonal is handled using one - to - one communication with complexity of O(1). The complexity O(log n) is obtained in one-to-all broadcast of each vector element among the n processes of each column. The third communication operation that is all-to-one reduction in each row is done in (O(log n)).



(a) Initial data distribution and communication steps to align the vector along the diagonal

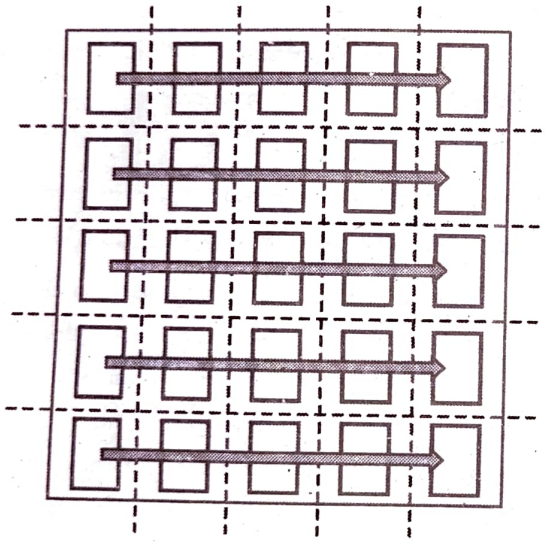
Time Complexity → O(1)

Fig. 4.6.15 : Contd....



(b) One-to-all broadcast of portions of the vector along process columns

Time Complexity → O(log n)



(c) All-to-one reduction to partial results

Time Complexity → O(log n)

Fig. 4.6.15 : Time complexity of different operations

- These communications take O(log n) time in total. Computation time is O(1). The parallel time of this algorithm is O(log n) + O(1) = O(log n).
- The cost (process-time product) is :

$n^2 \times \log n = O(n^2 \log n) > n^2$; hence, the algorithm is not cost-optimal.

4.6.4 Using Fewer than n^2 Processes

- The matrix - vector multiplication algorithm with block 2-D partitioning of the matrix can have a cost optimal implementation if the computation granularity at each process is increased by the use of fewer than n^2 processors.
- Each process owns a $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of matrix if a logical two - dimensional mesh of p processes is considered. Here p (i.e. $\sqrt{p} \times \sqrt{p}$) processors are used.
- It is required to distribute the entire vector on each row of processes before starting the multiplication. In the first step the vector is aligned on the main diagonal. In this step for alignment of vector elements, each process in the rightmost column sends its n/sqrt(p) vector elements to the diagonal process in its row. Thereafter, a column-wise one - to - all broadcast of n/sqrt(p) elements takes place. Each process then performs n^2/p multiplications and locally adds the sets of products. At the end of this step each process has, n/sqrt(p) partial sums that must be accumulated along each row to obtain the result vector. Hence, the last step of the algorithm is an all-to-one reduction of the n/sqrt(p)

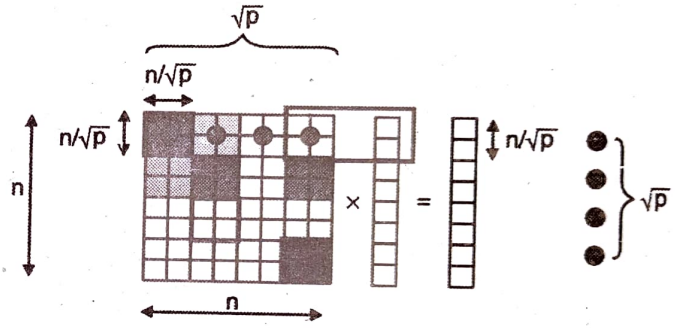


Fig. 4.6.16

☞ Parallel Run Time

The first alignment step takes time $t_s + t_w n/\sqrt{p}$.

The broadcast (one-to-all) and reductions (all-to-one) each take time

$$(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})$$

Local matrix-vector products take time $t_c n^2/p$, i.e., $O(n/\sqrt{p})^2$

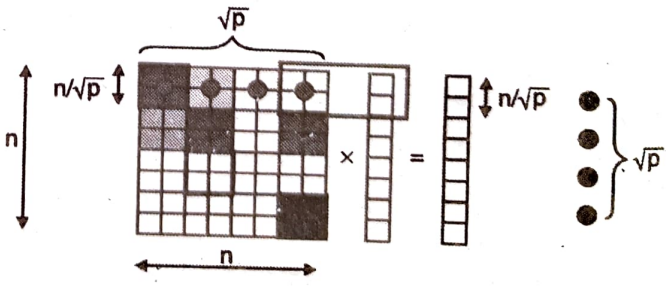


Fig. 4.6.17

☞ Total time

$$t_c n^2/p + t_s + t_w n/\sqrt{p} + 2(t_s + t_w n/\sqrt{p}) \log\sqrt{p}$$

$$= \frac{t_c n^2}{p} + t_s(1 + 2\log\sqrt{p}) + \frac{t_w n}{\sqrt{p}}(1 + 2\log\sqrt{p})$$

Syllabus Topic : Matrix Multiplication

4.7 Matrix Multiplication

Q. 4.7.1 Write short note on matrix multiplication. (Refer section 4.7) **(5 Marks)**

Almost in all areas of scientific research matrix operations like matrix multiplications are commonly used. Matrix multiplications are applied in many fields including scientific computing and pattern recognition, graph theory, signal processing, digital control and in many numerical algorithms. Many matrix algorithms have been designed to be implemented on different types of hardware including parallel and distributed systems, where the computational work is spread over multiple processors (perhaps over a network).

Let A and B be matrices of size n × m and m × l, respectively. The product matrix C = A * B is an n × l matrix, for which elements are defined as follows :

$$c_{ij} = \sum_{k=0}^{m-1} aik \cdot bkj \quad \text{where } 0 \le i \le n, 0 \le j < l$$

4.7.1 Sequential Algorithm for Matrix Multiplication

```
Algorithm MAT_MULT_SEQUENTIAL(A, B, C)
{
    for (i = 0 to n – 1)
    {
        for (j = 0 to n – 1)
        {
            C[i, j] = 0;
            for (k = 0 to n – 1)
            {
                C[i, j] = C[i, j] + A[i, k] * B[k, j];
            }
        }
    }
}
```

- The algorithm performs the matrix C rows calculation sequentially. At every iteration of the outer loop on i variable a single row of matrix A and all columns of matrix B are processed. This can be described as shown in Fig. 4.7.1 :

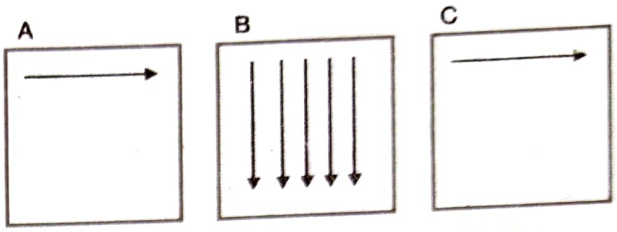


Fig. 4.7.1 : Sequential matrix multiplication

- We can see that there are (**m·l**) inner products calculated to perform the matrix multiplication. The complexity of the matrix multiplication is **O(mnl)** and assuming that n=m=l it gives complexity as $O(n^3)$.

4.7.2 Partitioning into Submatrices

- The block matrix operations are useful in matrix multiplications as well as in other operations. The matrix is divided into number of submatrices or blocks and the algebraic operations are applied on these submatrices like in original matrices. The algebraic operations applied on the submatrices are called block matrix operations.

- The block operation is a useful concept for matrix computations. In this view an n x n matrix A can be regarded as a q × q array of blocks **Ai,j(0 ≤ i, j <s)** such that each block is an (n/s) x (n/s) submatrix. For example, a matrix is divided into S^2 submatrices. Fig. 4.7.2 shows the block in the matrix.

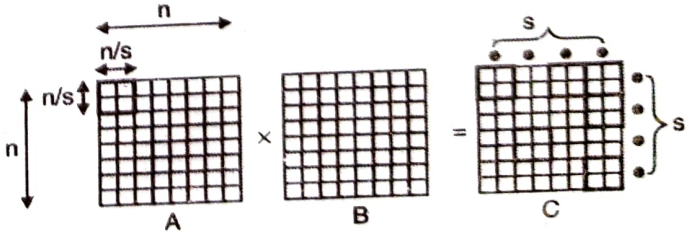


Fig. 4.7.2 : Matrix block operations

Each submatrix has **n/s × n/s** elements. It can be expressed in the following way using the notation $A_{p,q}$ as the sbmatrix in submatrix row **p** and submatrix column **q** :

```
for (p=0; p<s; p++)
{
    for (q=0; q<s; q++)
    {
        C[p,q]=0;/*clear elements of sumbatrix*/
        for (r=0; r<m ; r++) /*submatrix multiplication
        and add to accumulating submatrix*/
        {
            C[p,q] = C[p,q] + A[p,r] * B[sq];
        }
    }
}
```

In above algorithm the line $C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q}$ means multiply submatrix $A_{p,r}$ and $B_{r,q}$ using matrix multiplication and add to submatrix $C_{p,q}$ using matrix addition. This is described in the Fig. 4.7.3.

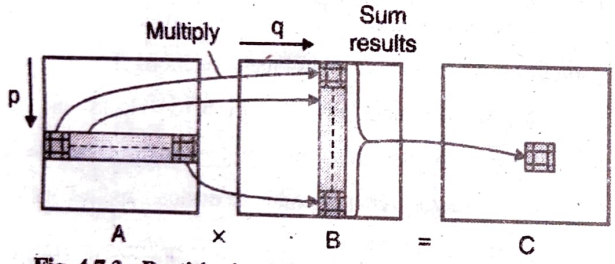


Fig. 4.7.3 : Partitioning into submatrices and perform multiplication

4.7.3 The Steps of the Parallel Algorithm for Dense Matrix Multiplication

Two square matrices, A and B of size n that have to be multiplied :

1. Partition these matrices in square blocks p, where p is the number of processes available.
2. Create a matrix of processes of size $p^{1/2} \times p^{1/2}$ so that each process can maintain a block of A matrix and a block of B matrix.
3. Each block is sent to each process, and the copied sub blocks are multiplied together and the results added to the partial results in the C sub-blocks.
4. The A sub-blocks are rolled one step to the left and the B sub-blocks are rolled one step upward.
5. Repeat steps 3 and 4 ($\sqrt{p}$) times

The Fig. 4.7.4 shows different partition of blocks of matrices A and B.

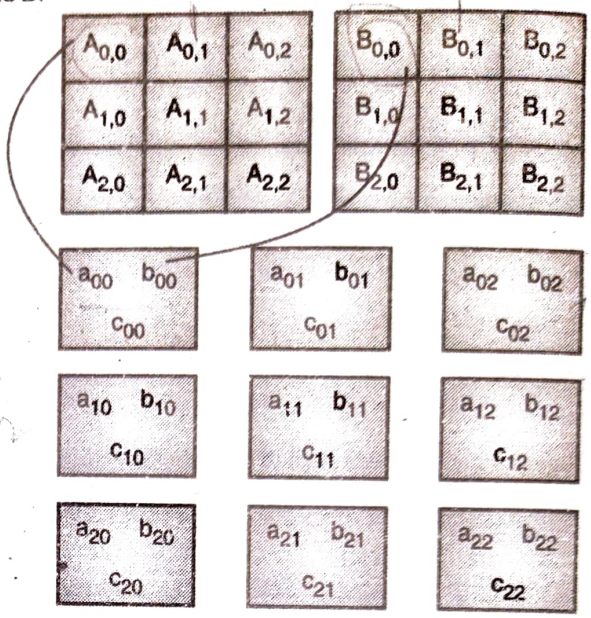


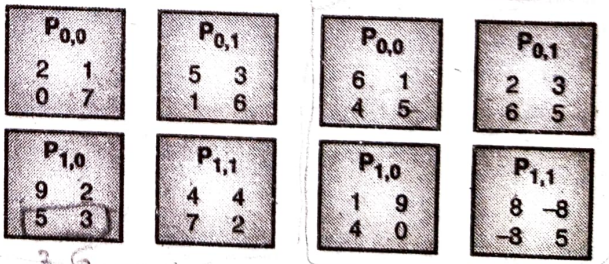
Fig. 4.7.4

Example : consider following two matrixes A and B to be multiplied using block operations.

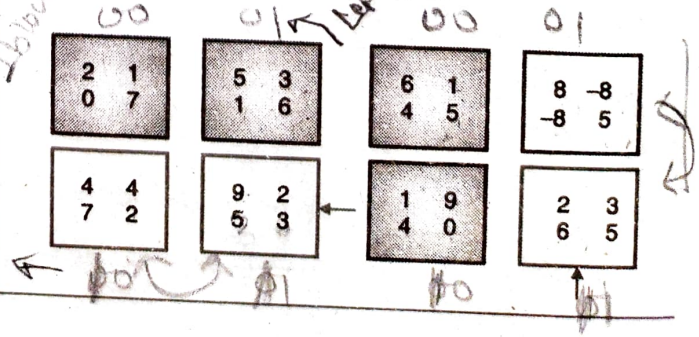
$$A = \begin{bmatrix} 2 & 1 & 5 & 3 \\ 0 & 7 & 1 & 6 \\ 9 & 2 & 4 & 4 \\ 3 & 6 & 7 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 6 & 1 & 2 & 3 \\ 4 & 5 & 6 & 5 \\ 1 & 9 & 8 & -8 \\ 4 & 0 & -8 & 5 \end{bmatrix}$$

These matrices are divided into 4 square blocks as follows :



The matrices A and B are initially aligned and the matrices after the initial alignment are shown below :



Now perform the Local matrix multiplication as shown below :

$C_{0,0}$ = $\begin{bmatrix}2 & 1\\0 & 7\end{bmatrix}$ × $\begin{bmatrix}6 & 1\\4 & 5\end{bmatrix}$ = $\begin{bmatrix}16 & 7\\28 & 35\end{bmatrix}$

$C_{0,1}$ = $\begin{bmatrix}5 & 3\\1 & 6\end{bmatrix}$ × $\begin{bmatrix}8 & -8\\-8 & 5\end{bmatrix}$ = $\begin{bmatrix}16 & -25\\-40 & 22\end{bmatrix}$

$C_{1,0}$ = $\begin{bmatrix}4 & 4\\7 & 2\end{bmatrix}$ × $\begin{bmatrix}1 & 9\\4 & 0\end{bmatrix}$ = $\begin{bmatrix}20 & 36\\15 & 63\end{bmatrix}$

$C_{1,1}$ = $\begin{bmatrix}9 & 2\\5 & 3\end{bmatrix}$ × $\begin{bmatrix}2 & 3\\6 & 5\end{bmatrix}$ = $\begin{bmatrix}30 & 37\\42 & 39\end{bmatrix}$

In next step shifting operation is performed on A and B. Shift A one step to left, shift B one step up

$\begin{bmatrix}5 & 3\\1 & 6\end{bmatrix}$ $\begin{bmatrix}2 & 1\\0 & 7\end{bmatrix}$ $\begin{bmatrix}1 & 9\\4 & 0\end{bmatrix}$ $\begin{bmatrix}2 & 3\\6 & 5\end{bmatrix}$

$\begin{bmatrix}9 & 2\\5 & 3\end{bmatrix}$ $\begin{bmatrix}4 & 4\\7 & 2\end{bmatrix}$ $\begin{bmatrix}6 & 1\\4 & 5\end{bmatrix}$ $\begin{bmatrix}8 & -8\\-8 & 5\end{bmatrix}$

Now again perform local matrix multiplication as shown below :

$C_{0,0} = C_{0,0} + \begin{bmatrix}5 & 3\\1 & 6\end{bmatrix} \times \begin{bmatrix}1 & 9\\4 & 0\end{bmatrix} = \begin{bmatrix}16 & 7\\28 & 35\end{bmatrix} + \begin{bmatrix}17 & 45\\25 & 9\end{bmatrix} = \begin{bmatrix}33 & 52\\53 & 44\end{bmatrix}$

$C_{0,1} = C_{0,1} + \begin{bmatrix}2 & 1\\0 & 7\end{bmatrix} \times \begin{bmatrix}2 & 3\\6 & 5\end{bmatrix} = \begin{bmatrix}16 & -25\\-40 & 22\end{bmatrix} + \begin{bmatrix}10 & 11\\42 & 35\end{bmatrix} = \begin{bmatrix}26 & -14\\2 & 57\end{bmatrix}$

$C_{1,0} = C_{1,0} + \begin{bmatrix}9 & 2\\5 & 3\end{bmatrix} \times \begin{bmatrix}6 & 1\\4 & 5\end{bmatrix} = \begin{bmatrix}20 & 36\\15 & 63\end{bmatrix} + \begin{bmatrix}62 & 19\\42 & 33\end{bmatrix} = \begin{bmatrix}82 & 55\\57 & 96\end{bmatrix}$

$C_{1,1} = C_{1,1} + \begin{bmatrix}4 & 4\\7 & 2\end{bmatrix} \times \begin{bmatrix}8 & -8\\-8 & 5\end{bmatrix} = \begin{bmatrix}30 & 37\\42 & 39\end{bmatrix} + \begin{bmatrix}0 & -12\\40 & -46\end{bmatrix} = \begin{bmatrix}30 & 25\\82 & -7\end{bmatrix}$

4.7.4 A Simple Parallel Algorithm

– Now we will discuss a simple parallel algorithm for matrix multiplication. In this algorithm the n × n matrices A and B are partitioned into P blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ each. These blocks are allotted to p processes. This is similar to 2-D partitioning, discussed earlier. For simplicity, we consider the processes in a two-dimensional notation and are labelled from $P_{0,0}$ to $P_{\sqrt{p}-1, \sqrt{p}-1}$.

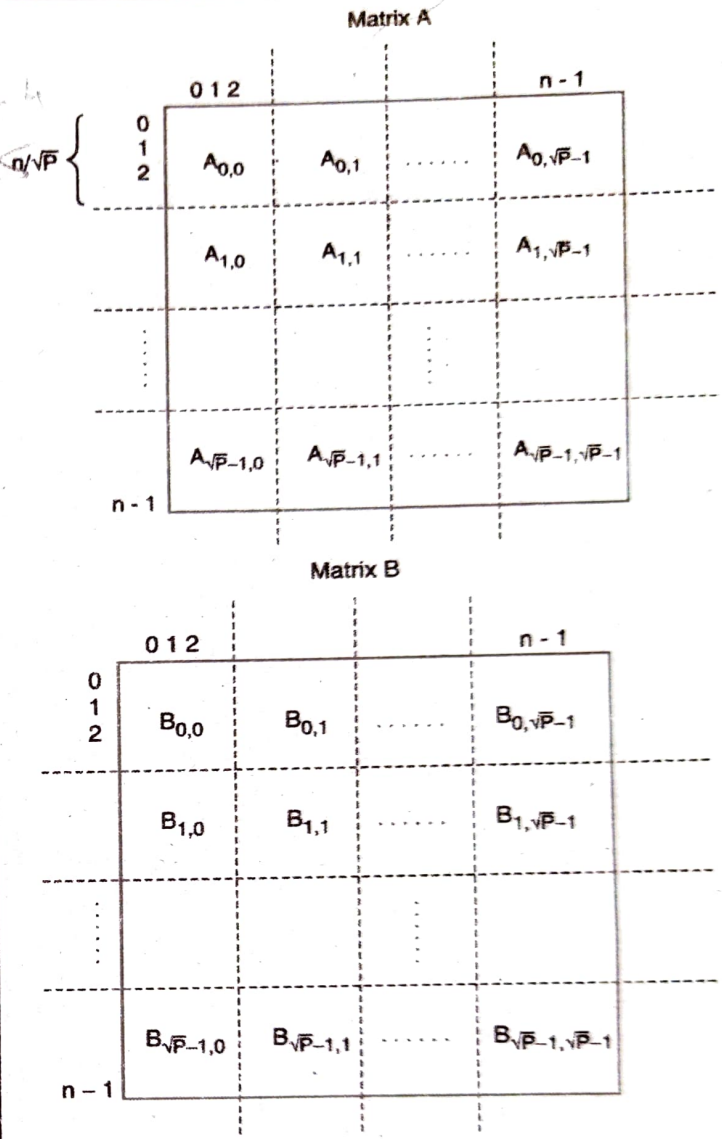


Fig. 4.7.5 : Initial allotment of blocks of matrices A and B to processes

- A process $P_{i,j}$ is allotted with the $A_{i,j}$ and $B_{i,j}$ blocks, where $0 \le i, j \le \sqrt{p} - 1$. Fig. 4.7.5 shows this now in each row of processes, an all-to-one broadcast of matrix A's blocks is done, and in each column of processes, an all-to-all broadcast of matrix B is done. This results into the process $p_{i,j}$ to have $A_{i,0}, A_{i,1}, \ldots, A_{i,\sqrt{p}-1}$ and $B_{0,j}, B_{1,j}, \ldots B_{\sqrt{p}-1,j}$ blocks.

Now, each process $P_{i,j}$ computes the resulting sub block $c_{i,j}$ as follows :

$$C_{ij} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j}$$

$P_{0,0}$	$P_{0,1}$		$P_{0,\sqrt{p}-1}$
$A_{0,0}, A_{0,1}, \ldots A_{0,\sqrt{p}-1}$ $B_{0,0}, B_{1,0}, \ldots B_{\sqrt{p}-1,0}$	$A_{0,0}, A_{0,1}, \ldots A_{0,\sqrt{p}-1}$ $B_{0,1}, B_{1,1}, \ldots B_{\sqrt{p}-1,1}$	. . .	$A_{0,0}, A_{0,1}, \ldots, A_{0,\sqrt{p}-1}$ $B_{0,\sqrt{p}-1}, B_{1,\sqrt{p}-1}, \ldots$ $B_{\sqrt{p}-1,\sqrt{p}-1}$
$P_{1,0}$ $A_{1,0}, A_{1,1}, \ldots A_{1,\sqrt{p}-1}$ $B_{0,0}, B_{1,0}, \ldots B_{\sqrt{p}-1,0}$	$P_{1,1}$ $A_{1,0}, A_{1,1}, \ldots A_{1,\sqrt{p}-1}$ $B_{0,1}, B_{1,1}, \ldots B_{\sqrt{p}-1,1}$	. . .	$P_{1,\sqrt{p}-1}$ $A_{1,0}, A_{1,1}, \ldots, A_{1,\sqrt{p}-1}$ $B_{0,\sqrt{p}-1}, B_{1,\sqrt{p}-1}, \ldots$ $B_{\sqrt{p}-1,\sqrt{p}-1}$
.	.		.
.	.		.
.	.		.
$P_{\sqrt{p}-1,0}$ $A_{\sqrt{p}-1,0}, A_{\sqrt{p}-1,1}, \ldots, A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{0,0}, B_{1,0}, \ldots B_{\sqrt{p}-1,0}$	$P_{\sqrt{p}-1,1}$ $A_{\sqrt{p}-1,0}, A_{\sqrt{p}-1,1} \ldots A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{0,1}, B_{1,1}, \ldots B_{\sqrt{p}-1,1}$	. . .	$P_{\sqrt{p}-1,\sqrt{p}-1}$ $A_{1,0}, A_{1,1}, \ldots, A_{1,\sqrt{p}-1}$ $B_{0,\sqrt{p}-1}, B_{1,\sqrt{p}-1}, \ldots$ $B_{\sqrt{p}-1,\sqrt{p}-1}$

Fig. 4.7.6 : Broadcasting A's block through rows and B's blocks through columns

The multiplication of sub blocks can be done by using sequential algorithm discussed earlier.

The all-to-all broadcasting through each row and column requires, $2\left(t_s \log \sqrt{p} + t_w\left(\frac{n^2}{p}\right)(\sqrt{p}-1)\right)$ time complexity. The multiplication of sub matrices of size $\left(\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}\right)$ require $\frac{n^3}{(\sqrt{p})^3} = \frac{n^3}{p\sqrt{p}}$. Each process has to do such $\sqrt{p}$ multiplication.

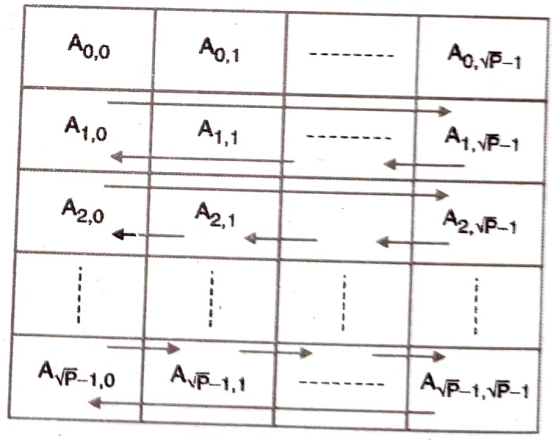
$$\therefore \text{ complexity} = \sqrt{p} \times \frac{n^3}{p\sqrt{p}} = \frac{n^3}{p}$$

Thus total time complexity is given by $\frac{n^3}{p} + t_s \log p + 2t_w\frac{n^3}{\sqrt{p}}$

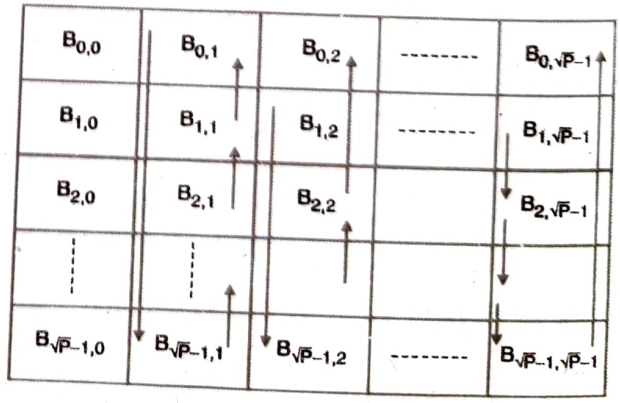
4.7.5 Cannon's Algorithm

Q. 4.7.2 Explain Cannon's algorithm. (Refer section 4.7.5) **(5 Marks)**

- Cannon's algorithm works similar to the simple parallel algorithm discussed in the previous section. But it is more memory efficient. The initial assignment of the block of matrices A and B to the process is as shown in Fig. 4.7.7.
- In the previous algorithm, the blocks of A were broadcasted though the rows and blocks of B were broadcasted through the columns. In cannon's algorithm, instead of broadcasting, the blocks are shifted.
- Blocks of A are shifted through the rows and B are shifted through columns. See Fig. 4.7.7.
- While assigning the bocks to the processes, the row i of blocks in A are shifted i blocks and column i of blocks in A are shifted i blocks.
- Thus after initial allocation, the process $P_{i,j}$ has modified blocks $A_{x,y}$ and $B_{y,z}$. It will find the partial result by multiplying $A_{x,y}$ and $B_{y,z}$ and stores in $C_{i,j}$.
- After first shift, the process Pi, j has $A_{x,(y+1)}$ mod $\sqrt{P}$ and $B_{(x+1)}$mod $\sqrt{P}$, y.



Initial blocks of A



Initial blocks of B

The arrow line indicates the initial shifting

Fig. 4.7.7 : The initial assignment of the block of matrices A and B to the process

- The initial allotment to the processes is shown in Fig. 4.7.8.

$P_{0,0}$ $A_{0,0}$ $B_{0,0}$	$P_{0,1}$ $A_{0,1}$ $B_{1,1}$	$P_{0,2}$ $A_{0,2}$ $B_{2,2}$	...	$P_{0,\sqrt{p}-1}$ $A_{0,\sqrt{p}-1}$ $B_{\sqrt{p}-1,\sqrt{p}-1}$
$P_{1,0}$ $A_{1,1}$ $B_{1,0}$	$P_{1,1}$ $A_{1,2}$ $B_{2,1}$	$P_{1,2}$ $A_{1,3}$ $B_{3,2}$	...	$P_{1,\sqrt{p}-1}$ $A_{1,0}$ $B_{0,\sqrt{p}-1}$
$P_{2,0}$ $A_{2,2}$ $B_{2,0}$	$P_{2,1}$ $A_{2,3}$ $B_{3,1}$	$P_{2,2}$ $A_{2,2}$ $B_{4,2}$	...	$P_{2,\sqrt{p}-1}$ $A_{2,1}$ $B_{:,\sqrt{p}-1}$
.	.	.		.
$P_{\sqrt{p}-1,0}$ $A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{\sqrt{p}-1,0}$	$P_{\sqrt{p}-1,1}$ $A_{\sqrt{p}-1,0}$ $B_{0,1}$	$P_{\sqrt{p}-1,2}$ $A_{\sqrt{p}-1,1}$ $B_{1,2}$	...	$P_{\sqrt{p}-1,\sqrt{p}-1}$ $A_{\sqrt{p}-1,\sqrt{p}-2}$ $B_{\sqrt{p}-1,\sqrt{p}-1}$

Fig. 4.7.8 : Initial allotment to the processes

- Now multiply these blocks and add the result to $C_{i,j}$. This process is continued for $\sqrt{P}$ times. This is shown in Fig. 4.7.9.
- While assigning the blocks to the processes, the row i of blocks in A one shifted i blocks, and column i of blocks in A one shifted i blocks.

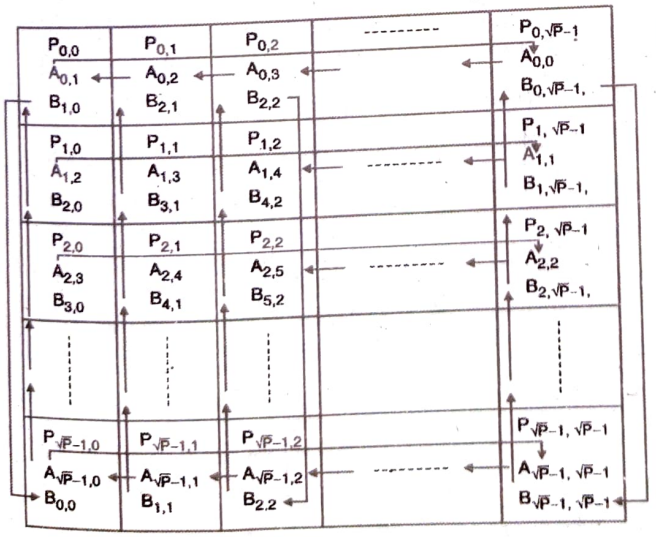


Fig. 4.7.9 : Blocks of A and B after first shift

- The arrow lines show the next shift. This is repeated $\sqrt{p}$ times.
- The time complexity of Cannon's algorithm is same as that of simple parallel algorithm. But when space is considered, in the simple algorithm each process had to store $\sqrt{p}$ blocks of A and $\sqrt{p}$ blocks of B.

Now we will see the steps of the cannon's algorithm. Here matrices A and B are partitioned among P processors.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

A

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

B

1. Initially processor $P_{i,j}$ has elements $a_{i,j}$ and $b_{i,j}$ $(0 \le I < n, 0 \le j < n)$

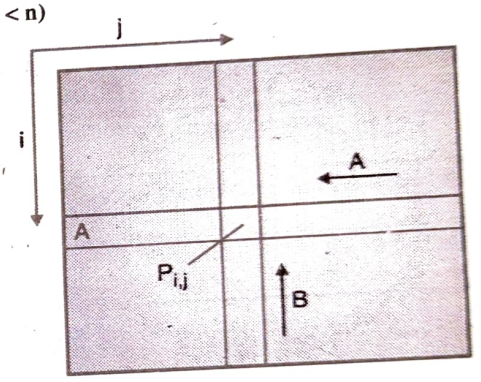


Fig. 4.7.10

2. Elements are moved from their initial position to an aligned position. The complete i^{th} row of A is shifted i places left and the complete j^{th} column of B is shifted j places upward.

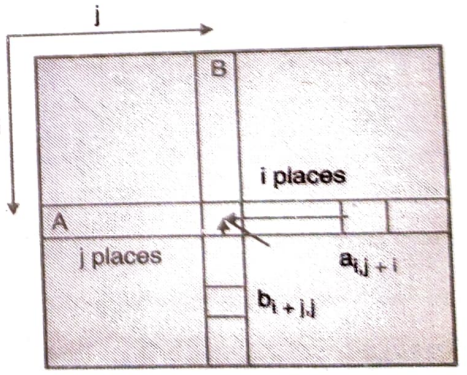


Fig. 4.7.11

3. Each processor $P_{i,j}$ multiply its elements.
4. The i^{th} row of A is shifted one place left, and the j^{th} column of B is shifted *one* place upward.
5. Each processor $P_{i,j}$ multiplies its elements brought to it and adds the results to the accumulating sum.
6. Step 4 and 5 are repeated until the final result is obtained (n-1 shifts with n rows and n columns of elements).

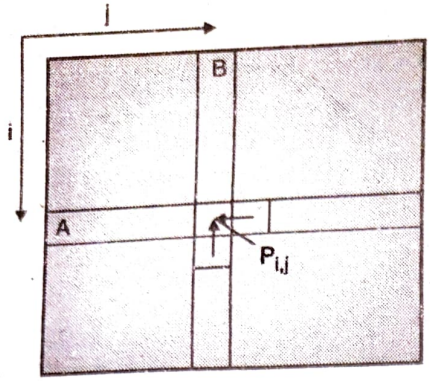


Fig. 4.7.12

This can be summarized for matrix A and B as in Table 4.7.1.

Table 4.7.1

Initially the matrix A	Initially the matrix B
Row 0 is unchanged.	Column 0 is unchanged
Row 1 is shifted 1 place left.	Column 1 is shifted one place up.
Row 2 is shifted 2 places left.	Column 2 is shifted 2 places up.
Row 3 is shifted 3 places left.	Column 3 is shifted 3 places up.

Consider two matrices A and B to perform matrix multiplication using Cannon's algorithm. The third matric C is used to store the result of multiplications as shown below. The different steps of the algorithm can be applied to perform the multiplication and described here in terms of diagrammatic representation.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}$$

A B C

Step 1 : in cannon algorithm :

a_{00}	a_{01}	a_{02}	a_{03}	$c_{00} = a_{00}$	b_{00}	$c_{10} = a_{11}$	b_{10}
b_{00}	b_{11}	b_{22}	b_{33}	$c_{01} = a_{01}$	b_{11}	$c_{11} = a_{12}$	b_{21}
a_{11}	a_{12}	a_{13}	a_{10}	$c_{02} = a_{02}$	b_{22}	$c_{12} = a_{13}$	b_{32}
b_{10}	b_{21}	b_{32}	b_{03}	$c_{03} = a_{03}$	b_{33}	$c_{13} = a_{10}$	b_{03}
a_{22}	a_{23}	a_{20}	a_{21}	$c_{20} = a_{22}$	b_{20}	$c_{30} = a_{33}$	b_{30}
b_{20}	b_{31}	b_{02}	b_{13}	$c_{21} = a_{23}$	b_{31}	$c_{31} = a_{30}$	b_{01}
a_{33}	a_{30}	a_{31}	a_{32}	$c_{22} = a_{20}$	b_{02}	$c_{32} = a_{31}$	b_{12}
b_{30}	b_{01}	b_{12}	b_{23}	$c_{23} = a_{21}$	b_{13}	$c_{33} = a_{32}$	b_{23}

Step 2 : in cannon algorithm :

a_{01}	a_{02}	a_{03}	a_{00}	$c_{00} += a_{01}$	b_{10}	$c_{10} += a_{12}$	b_{20}
b_{10}	b_{21}	b_{32}	b_{03}	$c_{01} += a_{02}$	b_{21}	$c_{11} += a_{13}$	b_{31}
a_{12}	a_{13}	a_{10}	a_{11}	$c_{02} += a_{03}$	b_{32}	$c_{12} += a_{10}$	b_{02}
b_{20}	b_{31}	b_{02}	b_{13}	$c_{03} += a_{00}$	b_{03}	$c_{13} += a_{11}$	b_{13}
a_{23}	a_{20}	a_{21}	a_{22}	$c_{20} += a_{23}$	b_{30}	$c_{30} += a_{30}$	b_{00}
b_{30}	b_{01}	b_{12}	b_{23}	$c_{21} += a_{20}$	b_{01}	$c_{31} += a_{31}$	b_{11}
a_{30}	a_{31}	a_{32}	a_{33}	$c_{22} += a_{21}$	b_{12}	$c_{32} += a_{32}$	b_{22}
b_{00}	b_{11}	b_{22}	b_{33}	$c_{23} += a_{22}$	b_{23}	$c_{33} += a_{33}$	b_{33}

Step 3 : in cannon algorithm :

a_{02}	a_{03}	a_{00}	a_{01}	$c_{00} += a_{02}$	b_{20}	$c_{10} += a_{13}$	b_{30}
b_{20}	b_{31}	b_{02}	b_{13}	$c_{01} += a_{03}$	b_{31}	$c_{11} += a_{10}$	b_{01}
a_{13}	a_{10}	a_{11}	a_{12}	$c_{02} += a_{00}$	b_{02}	$c_{12} += a_{11}$	b_{12}
b_{30}	b_{01}	b_{12}	b_{23}	$c_{03} += a_{01}$	b_{13}	$c_{13} += a_{12}$	b_{23}
a_{20}	a_{21}	a_{22}	a_{23}	$c_{20} += a_{20}$	b_{00}	$c_{30} += a_{31}$	b_{10}
b_{00}	b_{11}	b_{22}	b_{33}	$c_{21} += a_{21}$	b_{11}	$c_{31} += a_{32}$	b_{21}
a_{31}	a_{32}	a_{33}	a_{30}	$c_{22} += a_{22}$	b_{22}	$c_{32} += a_{33}$	b_{32}
b_{10}	b_{21}	b_{32}	b_{03}	$c_{23} += a_{23}$	b_{33}	$c_{33} += a_{30}$	b_{03}

Step 4 : in cannon algorithm:

a_{03}	a_{00}	a_{01}	a_{02}	$c_{00} += a_{03}$	b_{30}	$c_{10} += a_{10}$	b_{00}
b_{30}	b_{01}	b_{12}	b_{23}	$c_{01} += a_{00}$	b_{01}	$c_{11} += a_{11}$	b_{11}
a_{10}	a_{11}	a_{12}	a_{13}	$c_{02} += a_{01}$	b_{12}	$c_{12} += a_{12}$	b_{22}
b_{00}	b_{11}	b_{22}	b_{33}	$c_{03} += a_{02}$	b_{23}	$c_{13} += a_{13}$	b_{33}
a_{21}	a_{22}	a_{23}	a_{20}	$c_{20} += a_{21}$	b_{10}	$c_{30} += a_{32}$	b_{20}
b_{10}	b_{21}	b_{32}	b_{03}	$c_{21} += a_{22}$	b_{21}	$c_{31} += a_{33}$	b_{31}
a_{32}	a_{33}	a_{30}	a_{31}	$c_{22} += a_{23}$	b_{32}	$c_{32} += a_{30}$	b_{02}
b_{20}	b_{31}	b_{02}	b_{13}	$c_{23} += a_{20}$	b_{03}	$c_{33} += a_{31}$	b_{13}

□□□