3 Parallel Communication

Unit - III

Syllabus

Basic Communication : One-to-All Broadcast, All-to-One Reduction, All-to-All Broadcast and Reduction, All-Reduce and Prefix-Sum Operations, **Collective Communication using MPI :** Scatter, Gather, Broadcast, Blocking and non blocking MPI, All-to-All Personalized Communication, Circular Shift, Improving the speed of some communication operations.

3.1 One-to-All Broadcast and All-to-One Reduction

Q. Explain Broadcast and Reduction for multiplying matrix with vector. **MU - Oct. 18 (In Sem.), 6 Marks**

Q. Write a short note on All-to-one reduction with suitable example. **MU - Dec. 18, 6 Marks**

- Processes need to exchange data with other processes in most of the parallel algorithms.
- This exchange of data affects the efficiency of parallel programs by introducing interaction delays during their execution.
- Efficient implementations of these basic communication operations on various parallel architectures can improve performance and reduce development effort.
- Different architectures are :
  - One-to-all Broadcast (Scatter)
  - All-to-One Reduction (Gather)
  - All-to-All Broadcast and Reduction
  - All-Reduce and Prefix-sum
  - Scatter and gather
  - All-to-All personalized
  - Circular shift
- Communication operations will also be covered for linear arrays, meshes and hypercubes.
- We have learned in module 1 about communication cost in parallel machines.
- It takes roughly $t_s + mt_w$ time for a simple exchange of an m-word message between two processes running on different nodes of an interconnection network with cut-through routing.

  t_s : It is the latency or the startup time for the data transfer.

  t_w : It is the per-word transfer time, which is inversely proportional to the available bandwidth between the nodes.
- Also we have learned that data transfer time is roughly the same between all pairs of nodes. We are assuming all the networks that we are going to learn in this chapter are bidirectional and communication is single-ported.

3.1.1 One-to-All Broadcast

- Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them. This operation is known as **one-to-all broadcast**.
- One processor has a piece of data of size 'm', it needs to send to everyone.

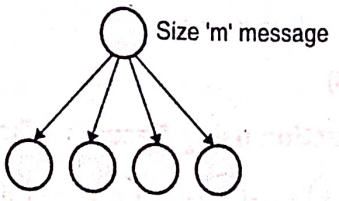


Fig. 3.1.1 : One to-All broadcast

3.1.2 All-to-One-Reduction

- Reduction can be performed by simply reversing the direction.
- In all-to-one reduction, each processor has m units of data. These data items must be combined piece-wise using some associative operator such as addition or min and accumulated at a single destination process into one message of size m.

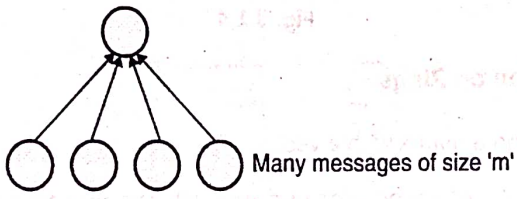


Fig. 3.1.2 : All-to-One Reduction

3.1.3 One-To-All Broadcast and All-To-One Reduction on Rings or Linear Array

- Simplest way is to send p – 1 messages from the source to the other p – 1 processors (there are p processes), this are not very efficient as the source process will be overloaded.
- Also as only the connection between a single pair of nodes is used at a time, the communication network is underutilized.
- Solution is to use **Recursive doubling** technique for broadcasting.
- Initially the root process only has the data of size m. Source process first sends a message to a selected process. Now these both independent processes (source and selected process) can simultaneously send the message to two other processes. Continue this procedure until all the processes have received data.
- After completing this operation, there are p copies of the data.

3.1.4 Example of One-To-All Broadcast using Recursive Doubling on Ring

As showing Fig. 3.1.3, Node 1 is the source of the broadcast. The message is first sent to the farthest node (here node 5) from the source (1).

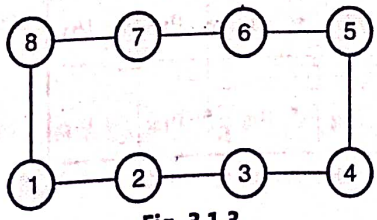


Fig. 3.1.3

Iteration1 :: {1 →5}

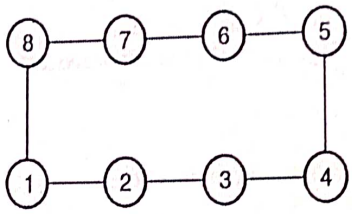
In the second step, the distance between the sending and the receiving nodes is halved. If node 1 sent the message to node 5 and then nodes 1 and 5 sent messages to 3 and 7 respectively in the second step.

Iteration 2 :: {1 →3, 5 →7}

This procedure will continue till the last node in the ring.

Final Iteration 3 :: {1 →2, 3 →4, 5 →6, 7 →8}

3.1.5 Example of All-To-One Reduction using Recursive Doubling on Ring



{2 →1; 4 →3;6 →5;8 →7}
{7 →5; 3 →1}
{5 →1}

Fig. 3.1.4

Example: Broadcast and Reduction on Rings

- Consider a problem of multiplying a matrix with a vector.
- The n* n matrix is assigned to an n*n processor grid (virtual). The vector is assumed to be on the first row of processors. The first step of the product requires a one-to-all broadcast of the vector element along the corresponding column of processors. This can be done concurrently for all n columns.
- The processors compute local product (multiplication) of the vector element and the local matrix entry.
- In the final step, the results of these products are accumulated to the first row using n concurrent all-to-one reduction operations along the columns.
- Fig. 3.1.5 shows one-to-all broadcast and all-to-one reduction in the multiplication of 4*4 matrix with a 4*1 vector.

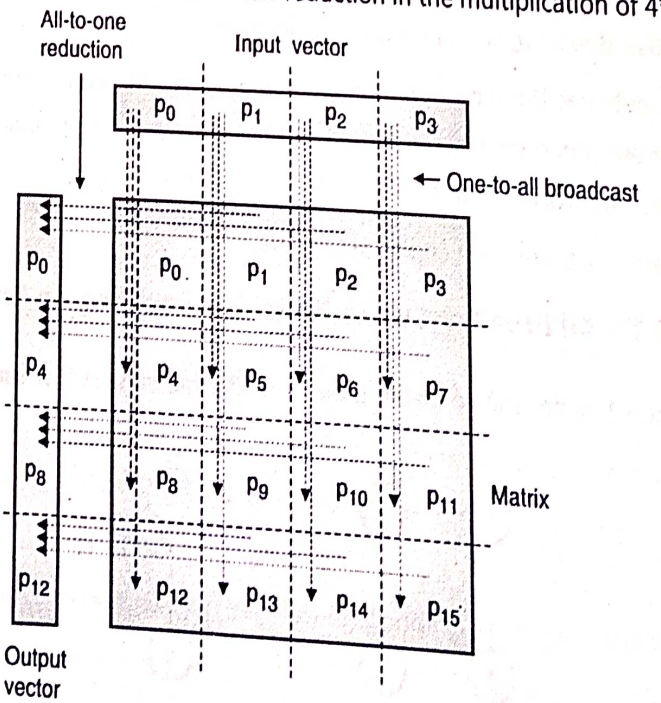


Fig. 3.1.5

3.1.6 Cost Analysis of One-to-All Communication

p processes → log p steps

Each transfer has a time cost of $t_s + t_w m$.

Total time for communication $T = (t_s + t_w m) \log p$.

Note : The One-to-all broadcast and All-to-one reduction operations can also be performed on mesh, matrix, Hypercube and on binary tree (same way that we have discussed with ring)

3.2 All-to-All Broadcast and Reduction

3.2.1 All-to All Broadcast

- All-to-All broadcast is generalization of broadcast in which each processor is the source as well as destination.
- Here, simultaneously every process out of p processes initiates a broadcast.
- A source process sends the same m-word message to every other process, but different processes may broadcast different messages.

Example

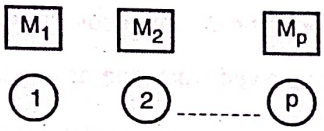


Fig. 3.2.1

Here source node 1 sends its message M_1 to all other nodes also other p - 1 nodes send their messages (node 2 send M_2, node 3 sends M_3 and node p sends M_p message) to other nodes including source node.

Here each processor is source as well as destination.

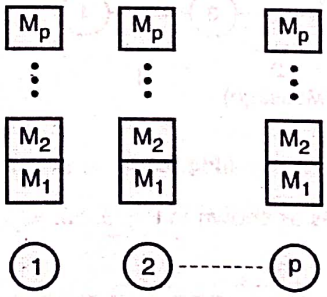


Fig. 3.2.2

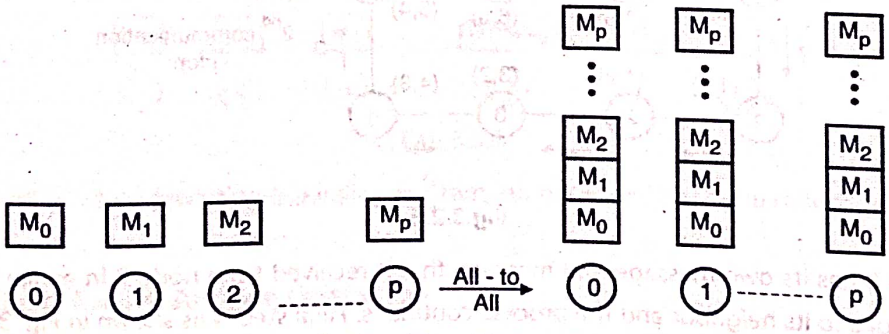


Fig. 3.2.3 : All-to-All Broadcast

3.2.2 All-to-All Reduction

All-to-All reduction is the reverse of all-to-all broadcast, in which every node is the destination of an all-to-one reduction.

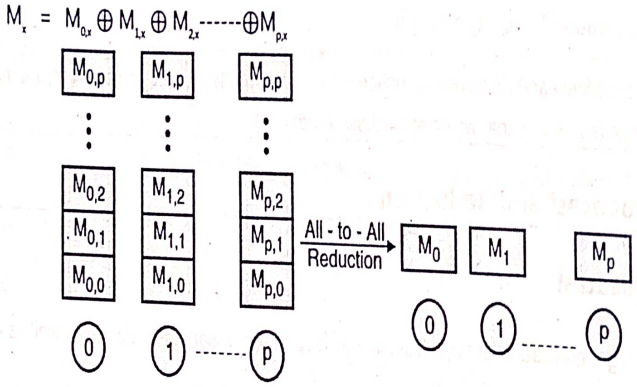


Fig. 3.2.4

Combine incoming messages with a subset of the local message using operator $\oplus$.

3.2.3 Example of All-To-All Broadcast Operation on Ring

- Each node first sends to one of its neighbours the data it needs to broadcast.
- In subsequent steps, it forwards the data received from one of its neighbours to its other neighbour as shown in Fig. 3.2.5.

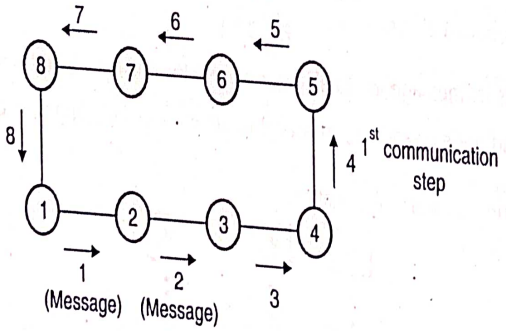


Fig.3.2.5

Now in 2nd step, every node has 2 messages as shown in Fig. 3.2.6.

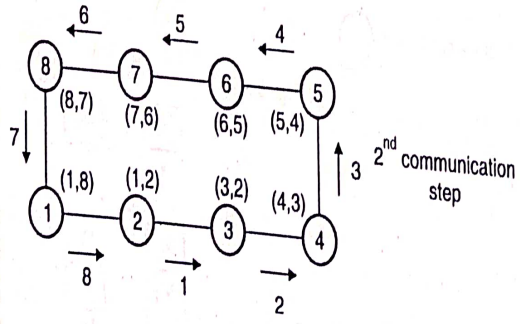


Fig.3.2.6

For example, node 8 has its own message and message that it received from node 7 in earlier step. Now this node broadcast these messages to its neighbor and the process continues. Final step is as shown in Fig. 3.2.7

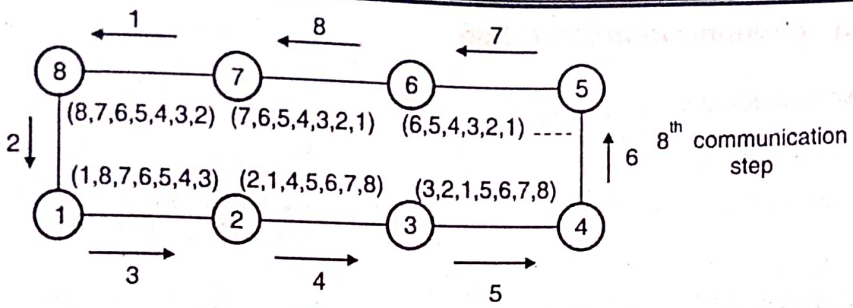


Fig. 3.2.7

3.2.4 Cost Analysis for All-To-All Operation on Ring

- **P processes :** in $p - 1$ steps
- **Total time :** $(t_s + t_w m)(p - 1)$

Note : The All-to-all broadcast and All-to-all reduction operations can also performed on mesh and hypercube (same way that we have done with Ring) in

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

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

$T = 2t_s (\sqrt{p} - 1) + t_w m (p - 1)$ and respectively.

3.3 All-Reduce and Prefix-Sum Operations

3.3.1 All-Reduce Communication Operation

- It is identical to All-to-one reduction followed by One-to-all broadcast.

Step 1 : All-to-one reduction

Step 2 : One-to-all broadcast

- This formulation is not the most efficient.
- The faster way to perform All-reduce by using pattern of All-to-All broadcast. The only difference is that message size does not increase here. As shown in Fig. 3.3.1 we are combining the messages instead of concatenating them.

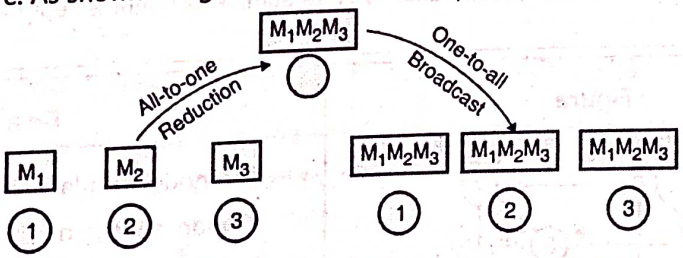


Fig. 3.3.1

In reduction operation while executing a parallel program, no node can finish the reduction before each node has contributed a value.

3.3.2 Cost Analysis for All Reduce Operation

- **P processes :** log p steps
- **Total time :** $T = (t_s + t_w m) \log p$

3.3.3 Prefix-Scan Communication Operation

- It is also called as scan operation.
- **Example :**

  Input : 1 2 3 4

  Operation : ADD
- The input to scan is the list of numbers 1,2,3,4 and an operation such as add.
- The output is the running sum of those numbers.

  Output : 1 3 6 10
- Each output is the sum of all numbers in the input up to that given point.
- Prefix sum 6 is the sum of 1,2 and 3.(sum of all elements from the beginning to that position)
- Same prefix sum 3 is the sum of 1 and 2
- Prefix sum at $i = \sum_{i=0}^{k} ni$ for all k between 0 to p – 1.
- Histogram uses this prefix scan. It is also used for data compression.
- A scan operation performs a parallel prefix with respect to a commutative and associative combining operator on message in a process.

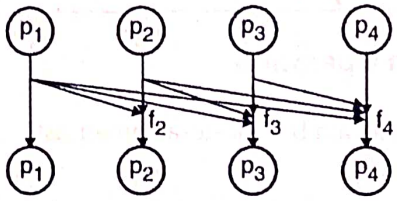
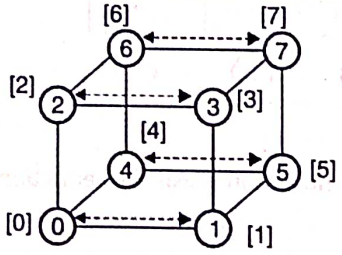


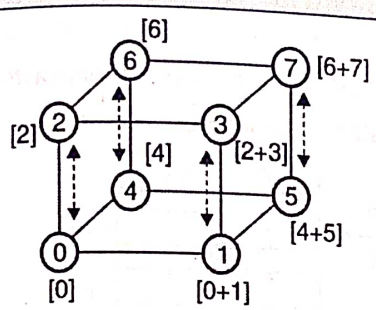
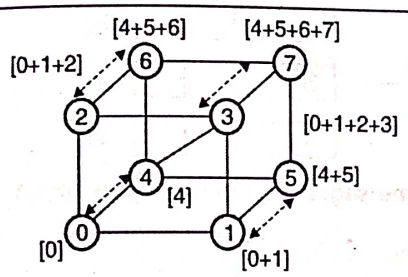
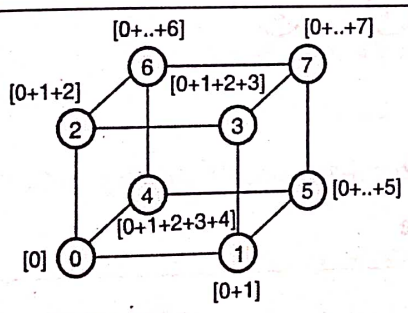
Fig. 3.3.2 : Scan communication

- Fig. 3.3.2 shows a scan operation in a four member process group with respect to associative operator f.

3.3.4 Example of the Prefix Sum Operation on Hypercube

Note :	Every node maintains an additional result buffer to store sum. This buffer is represented using square bracket in the Fig. 3.3.3.

Steps	Figure	Description
Step 1	 **Fig. 3.3.3 : Initial distribution of values**	• The node with label K uses information from only k-node subset of those nodes whose labels are less than or equal to k.

Steps	Figure	Description
Step 2	 **Fig. 3.3.4 : Distribution of sum before second step (consider forward direction from Fig. 3.3.3)**	• Here, content of incoming node is added to the result buffer only if the message comes from a node with smaller label than that of the recipiend node. • So here at node 7, message from 6 is added but at node 6, message of 7 is not added. Same result buffer of node 3, 5 and 1 is updated.
Step 3	 **Fig. 3.3.5 : Distribution of sums before third step(consider upward and downward direction from Fig.3.3.4)**	Here, result buffer of node 7 [4 + 5 + 6 + 7] 6 [4 + 5 + 6] 3 [0 + 1 + 2 + 3] is updated.
Final Step	 **Fig. 3.3.6 : Final distribution of prefix sums**	Here, result buffer of all nodes are updated except Node 0. Node 1 : [0 + 1] Node 2 : [0 + 1 + 2] Node 3 : [0 + 1 + 2 + 3] Node 4 : [0 + 1 + 2 + 3 + 4] Node 5 : [0 + 1 + 2 + 3 + 4 + 5] Node 6 : [0 + 1 + 2 + 3 + 4 + 5 + 6] Node 7 : [0 + 1 + 2 + ... + 7]

3.4 Scatter and Gather

- Scatter and gather are personalized operations. (Personalized means each process gets different data)

1. **Scatter** : Single node sends a unique message (different message) of size m to every other node. It is also called as one-to-all personalized communication.

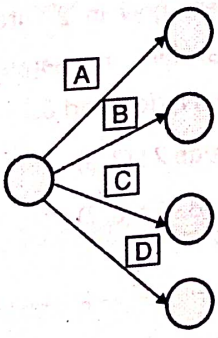


Fig. 3.4.1 : scatter

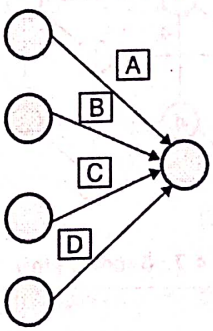


Fig. 3.4.2 : Gather

2. **Gather** : A single node collects unique messages from each node.

- While the scatter operation is fundamentally different from broadcast, the algorithmic structure is similar, except for different personalized messages broadcast (not like single same message broadcasting) and differences in message sizes (messages get smaller in scatter and stay constant in broadcast).

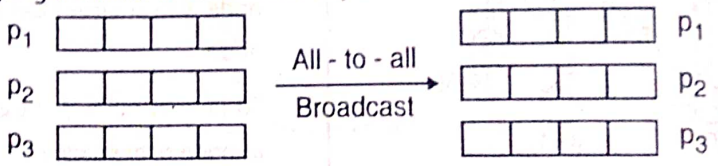


Fig. 3.4.3 : All-to-all broadcast (Same message is broadcast in all-to-all)

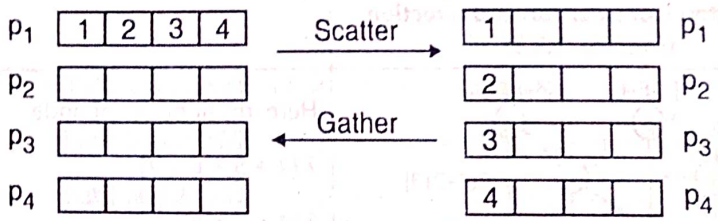


Fig. 3.4.4 : Scatter and Gather (Different messages are broadcast in scatter)

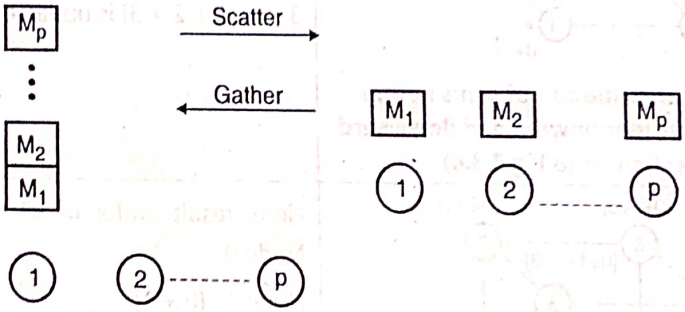
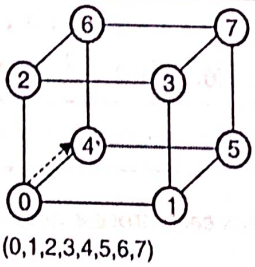
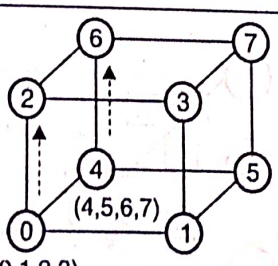
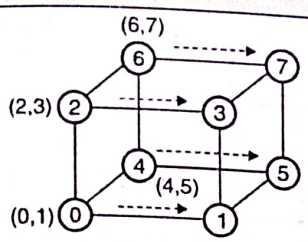
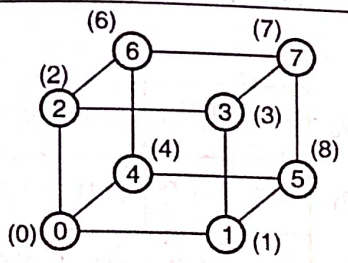


Fig. 3.4.5

3.4.1 Example of Scatter Operation on Hypercube

Step	Figures	Description
Step 1	 (0,1,2,3,4,5,6,7) **Fig. 3.4.6 : Initial distribution of message**	• In this step, the source transfers half of the messages to one of its neighbors. • Node 4 is neighbor to Node 0 so half of the messages (4, 5, 6, 7) are transfers to Node 4 from Node 0.
Step 2	 (4,5,6,7) (0,1,2,3) **Fig. 3.4.7 : Second step**	• Here now in 2nd iteration, Node 0 and Node 4 transfers messages (half) to their neighbor i.e. Node 2 and 6. • Node 2 : (2, 3) • Node 6 : (6, 7)

Step	Figures	Description
Step 3	 Fig. 3.4.8 : Third step	Now in 3rd iteration, Node 0, 4, 2 and 6 will transfers messages to their neighbors i.e. Node 1, 5, 3 and 7.
Final Step	 Fig. 3.4.9 : Final step	Now all messages are scattered to every node.

3.4.2 Cost Analysis of Scatter and Gather Communication

- **P processes :** number of steps log P
- **Total time :** $T = t_s \log p + t_w m (P - 1)$

Note : These operations can also be performed on a linear array and on a 2-D square mesh in time $t_s \log p + t_w m (p - 1)$

3.5 All-to-All Personalized Communication

- This communication is also called as **Total exchange**.
- We have learned that in all-to-all broadcast communication, each node sends the same message to all other nodes. But in all-to-all personalized communication, each node sends a distinct message of size m (same length) to every other node.
- These operations are used in many parallel algorithms including parallel sorting, Fast Fourier Transform (FFT) and some matrix operation.

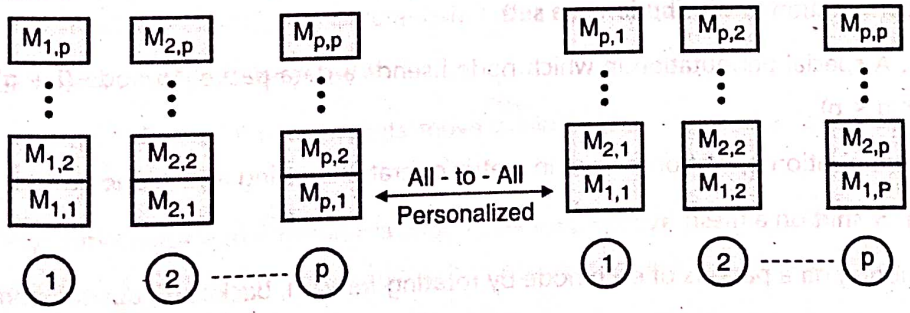
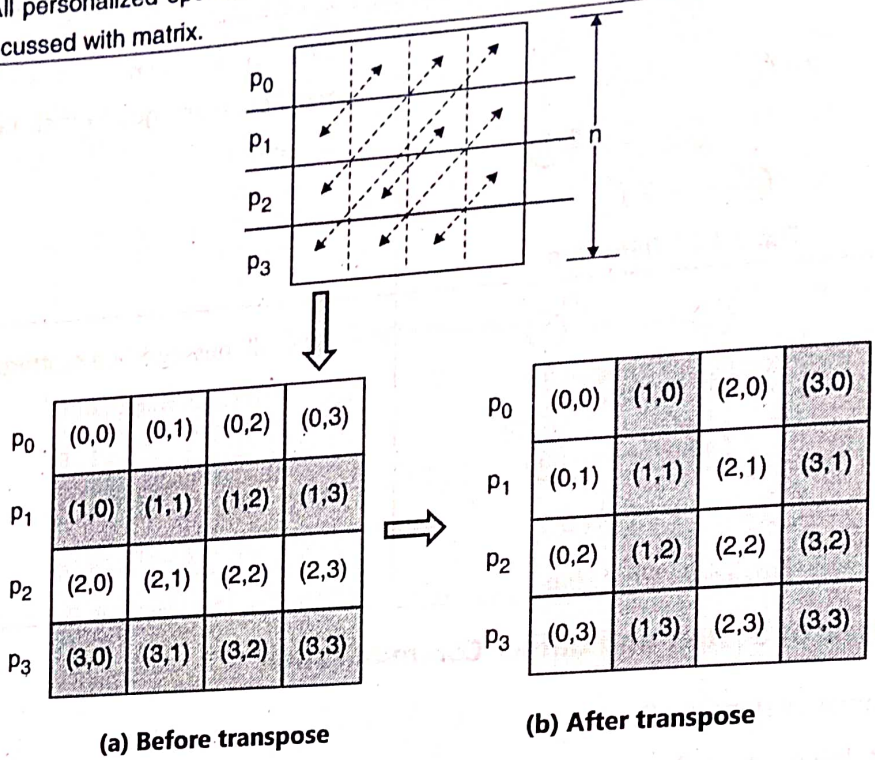


Fig. 3.5.1

3.5.1 Example : Problem of Transposing a Matrix

Hear the full row of the matrix is represented by each processes. The transpose operation is identical to an all-to-all personalized communication operation.

Note : The All-to-All personalized operation can also be performed on ring, mesh and hypercube in the same way that we have discussed with matrix.



(a) Before transpose **(b) After transpose**

Fig. 3.5.2

3.5.2 Cost Analysis of All-To-All Personalized Communication

- **P processes :** number of steps : $p - 1$
- **Transmitted message size** : $m(p-1), m(P-2) \ldots\ldots m.$
- **Total time :** $T = t_s (p-1) + \sum_{i=1}^{p-1} i\, t_w m$

3.6 Circular Shift

Q. Explain Circular shift operation on mesh and hyper cube network. **SPPU - May 19, 8 Marks**

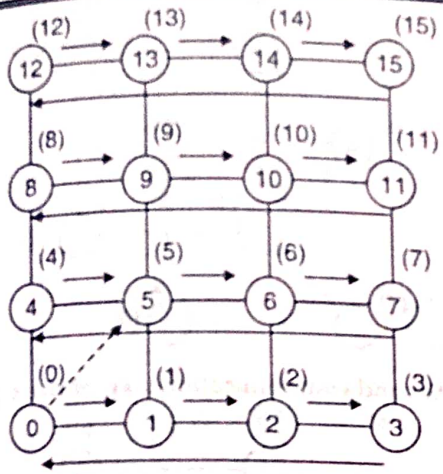
- It is a particular permutation (redistribution in a set).
- **Circular q-shift :** A special permutation in which node i sends a data packet to node **(i + q) mod p** in a set of p nodes where $(0 \leq q \leq p)$.
- Circular shift communication operation is used in matrix operations, string and image pattern matching.

Example : Circular 5-shift on a mesh

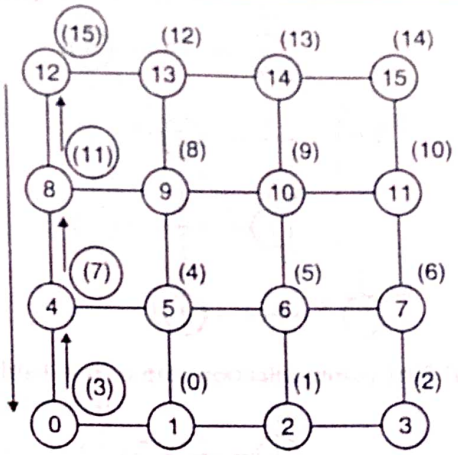
- Here we are permuting data packets of each node by rotating forward, backward, upward and downward.

Example : Circular shift operation on hypercube

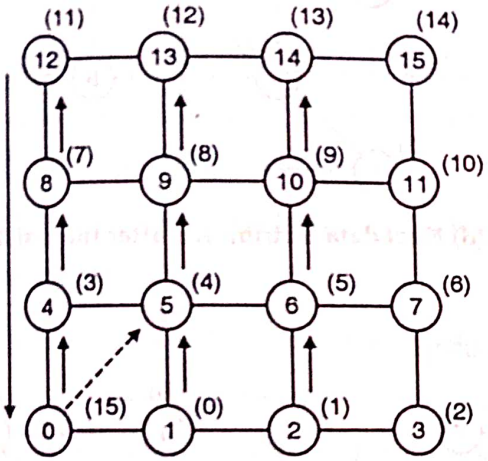
- To develop a hypercube algorithm for the shift operation, we map a linear array with 2d nodes onto a d-dimensional hypercube.
- To perform this mapping, assign node i of the linear array to node j of the hypercube.



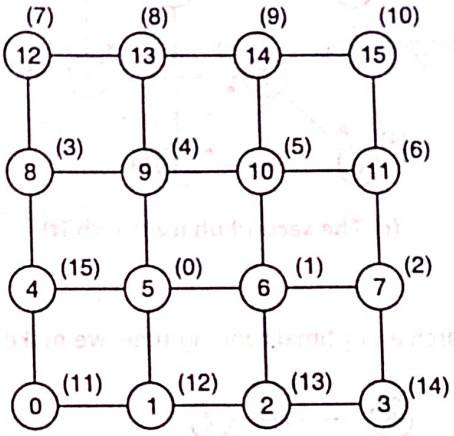
(a) Initial data distribution and the first communication step



(b) Step to compensable for backward row shifts



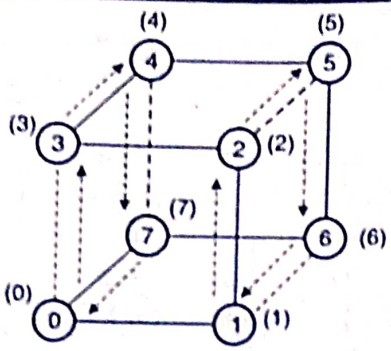
(c) Column shifts in the third communication step



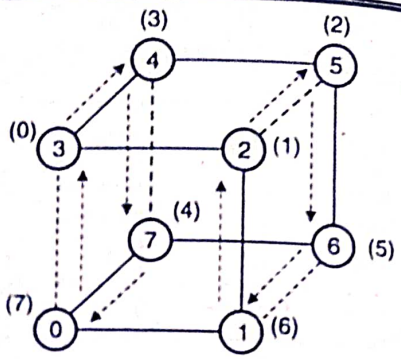
(d) Final distribution of the data

Fig. 3.6.1

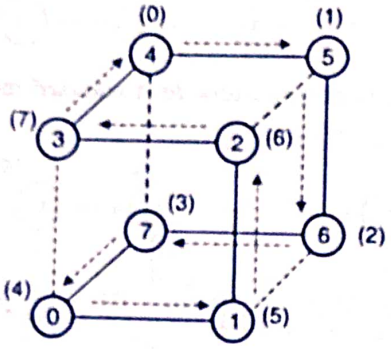
- A property of mapping is that any two nodes at a distance of 2i on the linear array are separated by exactly two links on the hypercube. An exception is i = 0 (that is, directly-connected nodes on the linear array) when only one hypercube link separates the two nodes.
- Expand q shift as a sum of powers of 2 (e.g. 5-shift = 20+22).
- If q is the sum of s distinct powers of 2, the circular qshift on a hypercube is perform in s phase.
- **Example :** Fig. 3.6.2 shows mapping for 8 nodes.
- In each phase of communication, all data packets move closer to their respective destinations by shot cutting the linear array in leaps of power of 2.
- All nodes can freely communicate in a circular fashion in their respective subarrays. This is shown in Fig. 3.6.2(a), in which nodes labeled 0, 3, 4, and 7 form one subarray and nodes labeled 1, 2, 5, and 6 form another subarray.
- Two communication steps in each phase, except the 1-shitf.
- Hence, the total number of steps for any q in p-node hypercube is at most 2log (P – 1)
- Time: $T = (t_s + t_w m)(2\log p - 1)$ (where m is m-word packets(m message length) on p-node hypercube)



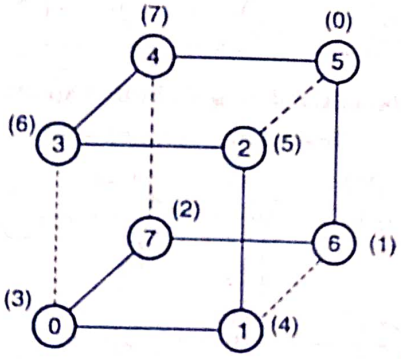
(a) First communication step of the 4-shift



(b) Second communication step of the 4-shift



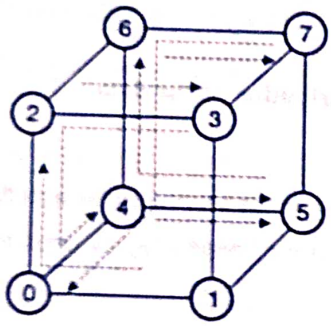
(c) The second phase(a 1-shift)



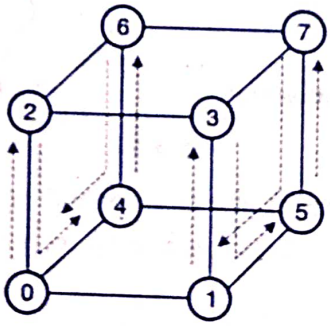
(d) Final data distribution after the 5-shift

Fig. 3.6.2

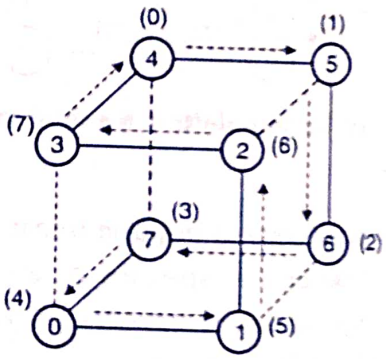
- To search an optimal running time, we make use of the E-cube routing.



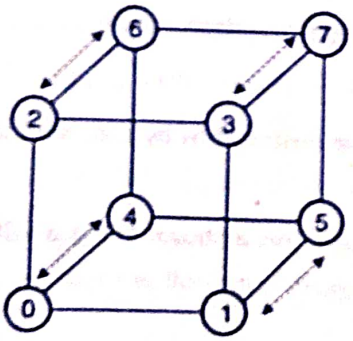
(a) 1-shift



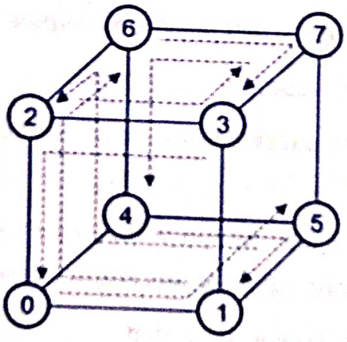
(b) 2-shift



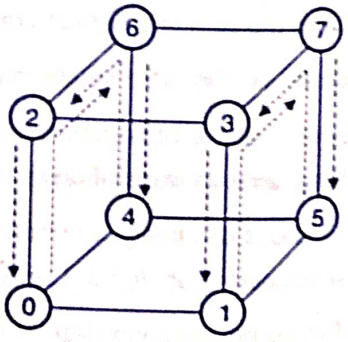
(c) 3-shift



(d) 4-shift

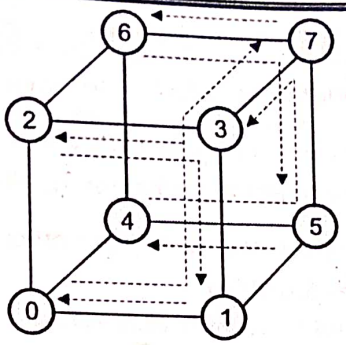


(e) 5-shift



(f) 6-shift

Fig. 3.6.3 (Cont...)



(g) 7-shift

Fig. 3.6.3 : Circular q-shifts on an 8-node hypercube for $1 \leq q < 8$.

- Total communication time for message of length m using E-cube routing is : $T = (t_s + t_w m)$

3.7 Improving the Speed of Some Communication Operation

Q. How to improve speed of communication operations? **MU - May 19, 8 Marks**

- Now we will discuss how to improve the performance of communication in parallel computing by improving the speed of communication operations.
- We will consider three operations, One-to-all broadcast, All-to-one reduction and All reduce.

1. One-to-all broadcast

- Splitting and routing messages into parts.

  Scatter + All-to-all broadcast

- So to improve the speed of One-to-all , we first spilt the message into p parts i.e perform scatter operation followed by All-to-all broadcast operation.

  Scatter – It will divide the message into n parts and every node will carry each part of the message.

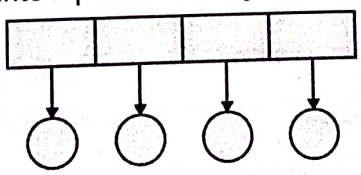


Fig. 3.7.1 : One-to-All broadcast

- And now we perform All-to-all broadcast. So every part of the message will be broadcast from each node to all other nodes and we will get a complete message at each node.

Cost analysis

$$T = 2 \times \left(t_s \log p + t_w (p-1)\frac{m}{p} \right)$$

$$\approx 2 \times (t_s \log p + t_w m)$$

2. All-to-one reduction

- It can be performed by performing All-to-all reduction followed by a gather operation.

  All-to-all reduction + Gather

- All messages are reduced to one message and then gathers reduced message.

3. All Reduce

- It can be performed by performing All-to-one reduction followed by One-to-all broadcast.
- The asymptotically optimal algorithms for these two operations (all-to-one reduction followed by one-to-all broadcast) can be used to construct a similar algorithm for the all-reduce operation.
- The intervening gather and scatter operations cancel each other. Therefore, an all-reduce operation requires an all-to all reduction and an all-to-all broadcast.

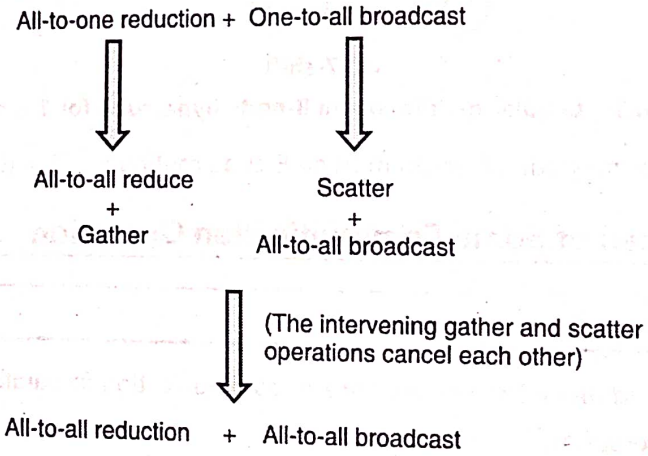


Fig. 3.7.2 : All Reduce Operation

Review Questions

Q. 1 Explain one-to-all broadcast and all-to-one reduction communication operation. **(8 Marks)**

Q. 2 Explain all-to-all broadcast and reduction. **(4 Marks)**

Q. 3 Explain all-reduce and prefix sum operations. **(8 Marks)**

Q. 4 Explain scatter and gather communication operation. **(6 Marks)**

Q. 5 Explain all-to-all personalized communication **(6 Marks)**

Q. 6 Explain circular shift operation **(4 Marks)**

□□□

4 Analytical Modeling of Parallel Programs

Unit - IV

Syllabus

Sources of Overhead in Parallel Programs, **Performance Measures and Analysis :** Amdahl's and Gustafson's Laws, Speedup Factor and Efficiency, Cost and Utilization, Execution Rate and Redundancy, The Effect of Granularity on Performance, Scalability of Parallel Systems, Minimum Execution Time and Minimum Cost, Optimal Execution Time, Asymptotic Analysis of Parallel Programs. **Matrix Computation :** Matrix-Vector Multiplication, Matrix-Matrix Multiplication.

4.1 Analytical Models : Sources of Overhead in Parallel Programs

Q. Explain the sources of overhead in parallel program. **SPPU - Oct. 18 (In Sem.), 7 Marks**

- A sequential algorithm is evaluated by its runtime (in general, asymptotic runtime as a function of input size).
- The parallel runtime of a program depends on the input size, the number of processors, and the communication parameters of the machine.
- The execution time of parallel algorithm = Input size + number of processing elements
- Therefore the underlying platform is used to analyze parallel algorithm.
- Whenever we are talking about a parallel system it is a combination of a parallel algorithm and parallel architecture (an underlying platform).
- If we increase the number of processors then can we get execute the program faster ?
- The answer is number Practically in parallel programs this is rarely possible due to a number of overheads including wasted computation, communication, idling and contention cause degradation in performance.
- Parallel programs in practice do not achieve linear speedup or desired efficiency because of parallel overheads.
- What are the different overheads when we are using parallel programming ?
  1. Inter process communication
  2. Idling (Load imbalance)
  3. Excess computation

1. Interprocess communication

- Processors working on any non-trivial parallel problem will need to talk to each other.
- The major source of parallel processing overhead is the **time spent to interact** and **communicate data** between processing elements.
- If there is data dependency exists in problem and the number of processing elements working on the same data then it requires interprocess communication. So even if two processing elements are independent still they cannot work in parallel.

2. Idling (Load imbalance)

- Processes may become idle because of many reasons.
- There are different factors that contribute to the idling of processing elements like load imbalance, synchronization, presence of serial components in program, dynamic subtasks generated.
- In dynamic task generation application, it is very difficult to predict the size of the subtasks assigned to processors in advance.
- Therefore to maintain uniform workload among processors, we cannot divide the problem statically between the processing elements.

3. Excess computation

- This is 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.

  Excess computation = (Computation performed by parallel program) – (Computation for best serial algorithm)

  In other words,

  Excess = Parallel – Best serial
- So we try to generate a parallel program that works like a best serial program but because of different factors parallel code takes more time than the best serial program and the net time is called as excess computation.

4.2 Performance Metrics for Parallel Systems

Q. Explain for performance metrics parallel systems. **SPPU - Dec. 18, May 19, 8 Marks**

- It is frequently necessary to compare the performance of parallel computers.
- Accurate predictions of the performance of a parallel algorithm helps determine whether coding it is worthwhile.

The performance of a parallel algorithm is also measured with various parameters listed as follows :

4.2.1 Execution Time

- It is a time to complete a task and denoted by T_{exe}.
- Our aim here is that for add parallel hardware should preferably reduce the time.

(i) Sequential or serial runtime (T_S) : The time elapse between the start of the execution and the end of the execution by the sequential computer is serial runtime.

(ii) The parallel runtime (T_p) : It is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution.

4.2.2 Total Overhead

- It is denoted by T_o.
- It is time spent by parallel program – Time spent by fastest serial program.
- Overhead $T_o = p\,T_p - T_s$

Where, p is processing elements (number of processors) in parallel program.

4.2.3 Speedup

- How much faster your code runs in parallel over serial execution.
- It evaluates the benefit of solving a problem in parallel.
- It is the ratio of the time taken to solve a problem on a single processing element (using the best sequential algorithm) to the time required to solve the same problem on a parallel computer with p identical processing elements.

$$S = T_S / T_p$$

Where, T_s is the execution time on a single processor, using the fastest known sequential algorithm.

T_p is the execution time using a parallel processor.

$$\text{Speedup} = \frac{\text{Time for serial}}{\text{Time for parallel}}$$

- 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 up a logical binary tree of processors.
- If an addition takes constant time, T_c and communication of a single word takes time $T_s + T_w$, we have the parallel time $T_p = \theta (\log n)$

$$\text{We know that } T_s = \Theta (n)$$

$$\text{Speedup S is given by } S = \Theta (n / \log n)$$

For Example

Addition of 16 numbers

$$\text{Time} = T_s = 16 \text{ unit, i.e. } T_s = \Theta (n) \quad n = 16$$

$$\text{Time} = 4 \text{ steps} = 4 \text{ units} = \log 16 = \log n = 4$$

$$\text{Speedup} = \Theta (n / \log n)$$

$$= 16/ 4 = 4$$

So, parallel program is 4 times faster than serial.

- Speedup can be as low as 0 (the parallel program never terminates).
- Speedup, in theory, should be upper bounded by p as 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.
- Linear speedup is maximum speedup which is usually p with p processors.
- Super linear speedup is maximum speedup which is usually greater than p.

4.2.4 Efficiency

- It is a measure of fraction of time for which processing element (processor) is engaged.
- Ratio of speedup to number of processing elements,

$$\text{Efficiency, } E = \frac{\text{Speedup}}{\text{Processing elements}} = S / p$$

4.4 Scalability of Parallel Systems

- Scalability is another important measure of performance for a parallel algorithm.
- Algorithms are said to be scalable if the level of parallelism increases at least linearly with size of the problem.
- The architecture to implement an algorithm is scalable if it continues to yield the same performance per processor, even if the problem size increases and also the number of processors increase.
- It is seen that data parallel algorithms are more scalable as compared to the architecture scalable algorithms.
- **A parallel computer system is said to be scalable if its efficiency can be fixed by simultaneously increasing the machine size and the problem size.**
- Scalability of a parallel system is the measure of its capacity to increase speedup in proportion to the machine size.
- Increasing the number of processors decreases the efficiency. And increasing the amount of computation per processor, increase the efficiency.
- To keep the efficiency fixed, both the size of problem and the number of processors must be increased simultaneously.

4.4.1 Isoefficiency Metric of Scalability

Isoefficiency function

- The isoefficiency function can be used to measure scalability of the parallel computing systems.
- It shows how the size of problem must grow as a function of the number of processors and in order to maintain some constant 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 isoefficiency function does not exist for unscalable parallel computing systems. This is because the efficiency in such systems cannot be kept at any constant value as machine size increases, no matter how fast the problem size is increased.
- What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?
- This rate determines the scalability of the system. 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.

We can write parallel runtime as,

$$T_p = W + T_o(W, p) / p$$

The resulting expression for speedup is,

$$S = W / T_p = \frac{W p}{W + T_o(W, p)}$$

A large isoefficiency function indicates a poorly scalable system.

4.5 Minimum Execution Time and Minimum Cost-optimal Execution Time

Q. Write a note on minimum and cost optimal execution time. SPPU - May 19, 8 Marks

4.5.1 Minimum Execution Time

- If the number of processors is not a constraint then we can find out how fast a problem can be solved or what the minimum possible execution time of parallel algorithm is.
- As we increase the number of processing elements for a given problem size, either the parallel runtime continues to decrease and asymptotically approaches a minimum value or it starts rising after attaining a minimum value.
- The equation to find the number of processing elements for which T_p (parallel runtime) is minimum is :

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

Let, p_0 be the value of the number of processing elements. The value of T_p^{min} (minimum parallel time) can be determined by substituting p_0 for p in the expression for T_p.

- For Example : Minimum execution time for adding n numbers.
- We know that the parallel run time for the problem of adding n numbers on p processing elements is,

$$T_p = n / p + 2 \log p$$

- Equating the derivative with respect to *p* of the right-hand side to zero,

$$-\frac{n}{p^2} + \frac{2}{p} = 0$$

$$-n + 2p = 0$$

$$P = \frac{n}{2}$$

Substitute, $p = n / 2$ we get,

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

- For this example, the processor-time product for $p = p_0$ is Q(n log n), which is higher than the Q(n) serial complexity of the problem.
- Hence, the parallel system is not cost-optimal for the value of *p* that yields minimum parallel runtime.

4.5.2 Minimum Cost-optimal Execution Time

- Let T_p^{min} be the minimum time in which a problem can be solved by a cost-optimal parallel system. Let $T_p^{cost\text{-}opt}$ be the minimum cost-optimal parallel time.
- From the previous sections, the isoefficiency function f(p) of parallel system is Θ (p log p).
- If, W = n = f (p) = p log p, then log n = log p + log p. Ignoring the double logarithmic term, log n log p. If, n = f (p) = p log p, then p = f – 1(n) = n/log p n/log n.

Hence, $f^{-1}(W) = \Theta$ (n/log n).

- As a consequence of the relation between cost-optimality and the isoefficiency function, the maximum number of processing elements that can be used to solve this problem cost-optimally is Θ(n/log n).

Put, $p = n/\log n$ in equation $T_p = \Theta(\log n)$ we get,

$$T_p^{cost\text{-}opt} = \log n + \log\left(\frac{n}{\log n}\right)$$

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

- Make a note that both T_p^{min} and $T_p^{cost\_opt}$ for adding n numbers are same $\Theta(\log n)$.
- Thus for the problem of adding n numbers, a cost-optimal solution is also the as asymptotically fastest solution.
- The parallel execution time cannot be reduced asymptotically by using a value of *p* greater than that suggested by the isoefficiency function for a given problem size.
- But it is possible that $T_p^{cost\text{-}opt} > T_p^{min}$ for parallel systems.

4.6 Principles of Scalable Performance

There are some laws that govern the performance of the parallel processing system. These laws will be studied in this section.

4.6.1 Amdahl's Law

This law is used for a fixed workload parallel system. There are systems that have fixed computational workload. Hence as the number of processing elements or processors increase, the time required for execution must reduce.

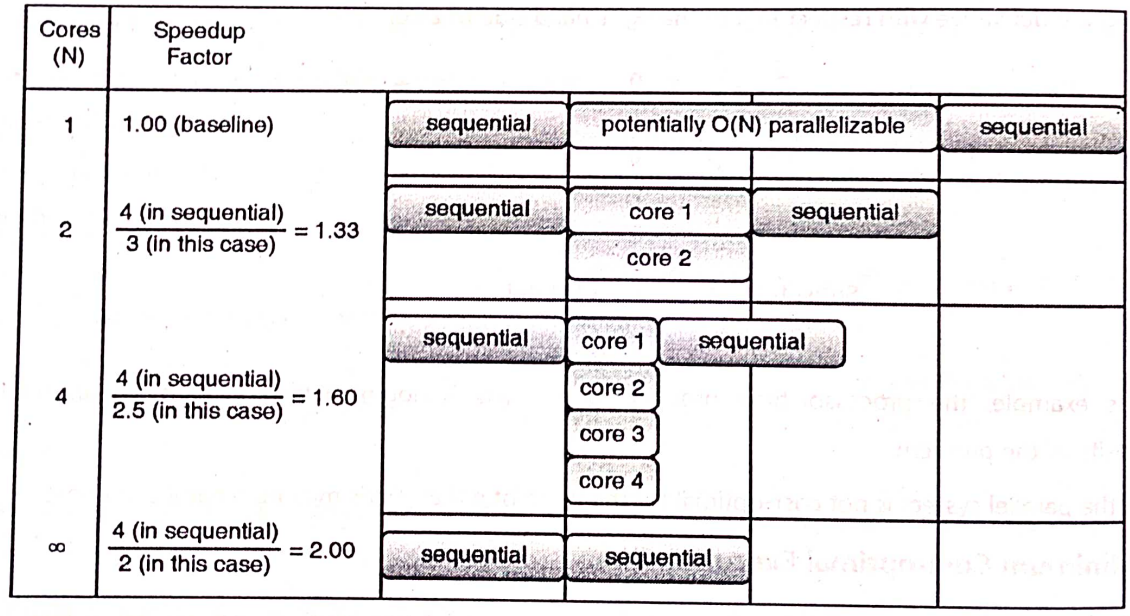


Fig. 4.6.1 : Amdahl's law

- Let the time requires for executing a task on a sequential system is t_s. If the 'f' fraction of the task is serial and $(1 - f)$ is non-serial or parallel, then we can make the second part work in parallel on multiple processors but the first part has to work serially.
- For example if a part of code is serial, it has to always work serially on whatever number of processors be there. The part that is parallelizable will require lesser and lesser time when the number of processor increases.
- Thus the Speed-up also keeps on increasing. This can be explained with the Fig. 4.6.1. In the Fig. 4.6.1, four time units as shown are required to execute the program in case of a sequential system.

- When there are two processors, the sequential time remains the same, while the parallelizable time requires half the time i.e. one time unit, as there are two processors. When the number of processors are 4, the time required further halves to just a half time unit.
- Thus the speed-up for n-processors according to the Amdahl's law can be given as below :

$$S(n) = \frac{t_s}{ft_s + (1 - f)\, t_s / n} \quad \text{...(4.6.1)}$$

where, t_s is the total time required on sequential system f is that fraction of the code which is sequential, hence 1-f is the parallelizable part of the task and 'n' is the number of processors.

- Equation (4.6.1) is called as Amdahl's law. The limitation of Amdahl's law is shown in Fig. 4.6.1, the last case.
- If we keep on increasing the number of processors to infinity, the speed-up cannot go beyond 1/f (2 in this case), as 'f' is the fraction of the code which is serial.
- The graph for the speed-up factor vs. number of processors 'n' for Amdahl's law is as shown in Fig. 4.6.2.

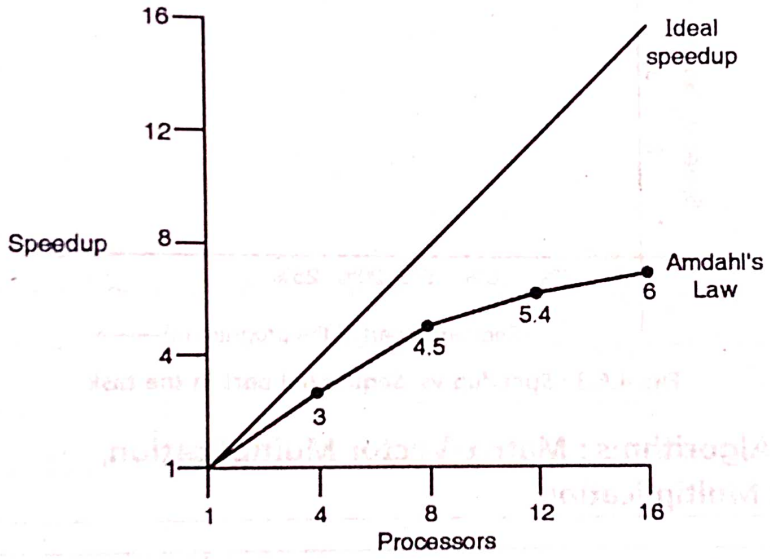


Fig. 4.6.2 : Speed-up vs number of processors for Amdahl's law

4.6.2 Gustafson's Law

- Gustafson law overcame the drawback of the Amdahl's law. Gustafson relaxed the problem size from being fixed to be of any size.
- Gustafson said that instead of having a fixed problem size or fixed workload we must assume that we have a fixed execution time. This is because in case of huge problem size, we will need to increase our system size i.e. number of processors instead of the execution time.
- The time for execution for whatever huge the workload is, the execution time must be fixed. According to this the speed-up factor will be numerically different as compared to the Amdahl's speed-up factor, and hence this is termed as scaled speed-up factor (S'(n)).
- Thus in case of Gustafson's law the execution time is fixed. Thus let the serial execution time be 's' and the parallel execution time be 'p' for 'n' processor system. Thus total execution time is s + p.
- If the execution is to be done on a sequential system, the execution time will be hence s + np. Thus the scaled speedup factor can be given as below :

$$S'(n) = \frac{s + np}{s + p} = \frac{s + np}{1} \quad \ldots(4.6.2)$$

assuming the time required on parallel system is 1 i.e. $s + p = 1$.

Thus, $S'(n) = \frac{s + n(1 - s)}{s + (1 - s)}$ (since, $s + p = 1$, therefore $p = 1 - s$)

$$= \frac{n + s(1 - n)}{1}$$

$$S'(n) = n + s(1 - n) \quad \ldots(4.6.3)$$

- This equation is called as Scaled speed-up factor of Gustafson's law. The graph of speedup factor vs. sequential part of the task can be shown as in Fig. 4.6.3. Fig. 4.6.3 shows that there is no effect of the sequential part on the speedup factor of a system.

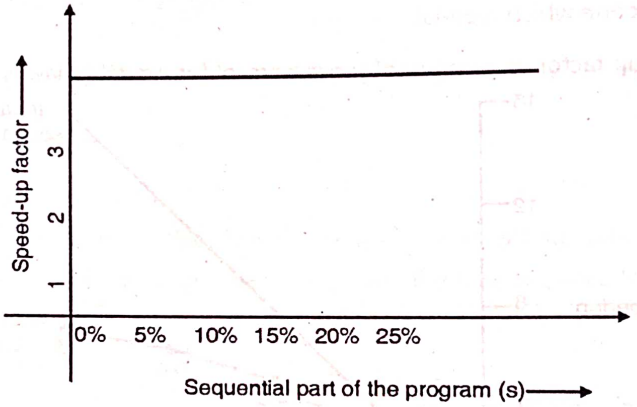


Fig. 4.6.3 : Speedup vs. Sequential part in the task

4.7 Dense Matrix Algorithms : Matrix-Vector Multiplication, Matrix-Matrix Multiplication

Q. Explain Matrix-Matrix Multiplication in detail. **SPPU - May 19, 8 Marks**

- Dense or full matrices that have no or few known usable zero entries. Sparse matrix is a matrix in which most of the elements are zero.
- Parallel computations involving matrices and vectors due to their regular structure, used for data decomposition.
- Any algorithms rely on input, output and intermediate data decomposition.
- Depending on the computation, the decomposition may be induced by partitioning the input, the output, or the intermediate data.
- The algorithms discussed in this module use one- and two-dimensional block (1-D and 2-D), cyclic, and block-cyclic partitioning.

4.7.1 Matrix-Vector Multiplication

- This section covers the details about the matrix and vector multiplication.
- We are going to multiply a dense $n \times n$ matrix A with an $n \times 1$ vector x to yield an $n \times 1$ result vector y.
- Compute $y = y + A^* x$, where A is a dense matrix.

- The serial algorithm requires n^2 multiplications and additions. Assuming that multiplication and addition pair takes unit time, the sequential run time is,

$$W = n^2$$

- Depending on whether row wise 1-D, column wise 1-D, or a 2-D partitioning is used, three distinct parallel formulations of matrix-vector multiplication are possible.

Algorithm

A sequential algorithm for multiplying an n × n matrix A with an n × 1 vector x to yield an n × 1 product vector y.

```
procedure MatrixVectorMul(A, x, y)
begin
  for i := 0 to n – 1 do
  begin
    y[i]:=0;
    for j:= 0 to n – 1 do
      y[i]:= y[i] + A[i, j] x x[j];
  endfor;
end MatrixVectorMul;
```

Row wise 1-D Partitioning

- Now we will discuss how to use row wise 1-D partioning to implement matrix-vector multiplications parallel.
- The n x n matrix is partitioned among *p* processors, with each processor storing n/p complete rows of the matrix.
- The n x 1 vector x is distributed such that each process owns n/p of its elements.
- **We can implement parallelism either by considering one row per process or fewer row than n processes.**

One row per process

- Consider the case in which the n×n matrix is partitioned among n processes so that each process stores one complete row of the matrix.
- The n x 1 vector x is distributed such that each process owns one of its elements.
- The initial distribution of the matrix and the vector for row wise block 1-D partitioning is shown in Fig. 4.7.1

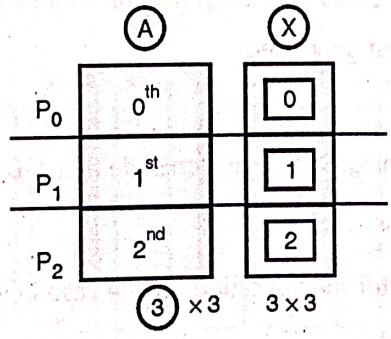


Fig. 4.7.1

- Here we have first row of matrix A to process P_0 and also 0^{th} element of vector x is given to process P_0. In the same manner, we have assigned 1^{st} element of matrix A to process P_1 and 1^{st} element of vector to process P and so on.

- **It is called 1-D because we have considering only rows for partitioning of matrix elements into processes.**
- **Also it is called one row per process because we have assigned one row to each process.**
- In Fig. 4.7.1 process P_1 initially owns x [i] and A[i,j], A[i,1] ,.....A[i,n – 1] and is responsible for computing y[i].
- Vector x is multiplied with each row of the matrix hence every process needs the entire vector.
- Since each process starts with only element of x, an all-to-all broadcast is required to distribute all the elements to all the process.

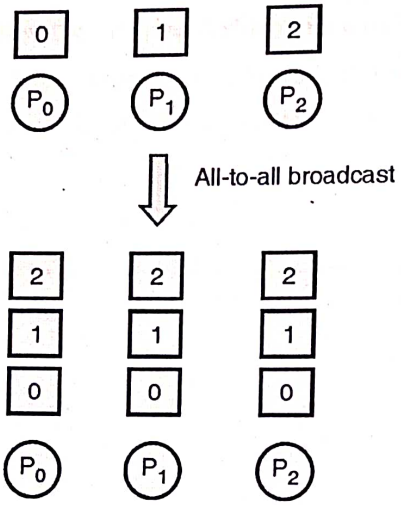


Fig. 4.7.2 : After all-to-all broadcast

- So every process will have row and vector elements as shown in Fig. 4.7.3.

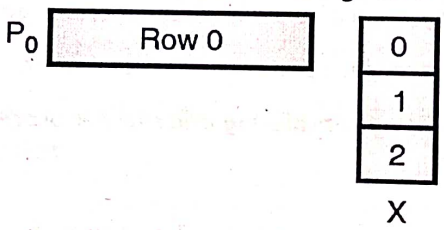


Fig. 4.7.3

- Now as every process performed partitioning on matrix, and also got vector, we can perform computation. That multiply each row with vector.
- As soon P_0 finish computation (multiplication), will store results in vector y (0^{th} location).
- Process-time Product- Whatever the cost generated.
- Process-time product cost is $\Theta(n^2)$.
- This time is same as sequential partitioning. So we can conclude that 1-D partitioning is cost-optimal.

Rowwise 2-D Partitioning

- Now we will discuss parallel matrix-vector multiplication for the case in which the matrix is distributed among the processes using a block 2-D partitioning.
- Here we consider 2 dimensions for portioning i.e. row and column.
- When we are following 1-D partitioning, we have p = n (every process will execute one **row** of matrix) but in 2-D partitioning , as we two dimensions , so $p = n^2$ (one element per process)

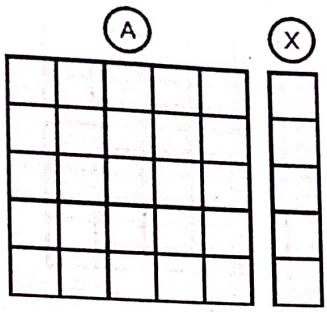


Fig. 4.7.4

- The n*1 vector x is distributed only in the last column of n processors.
- We must align the vector with the matrix appropriately.
- The first communication step for the 2-D partitioning aligns the vector x along the principal diagonal of the matrix.
- The second step copies the vector elements from each diagonal process to all the processes in the corresponding column using n simultaneous broadcasts among all processors in the column.
- Finally, the result vector is computed by performing an all-to-one reduction along the columns.
- Three basic communication operations are used in this algorithm :

(i) One-to-one communication (ii) One-to-all broadcast

(iii) All-to-one reduction

(i) One-to-one communication

It is use to align the vector along the main diagonal.

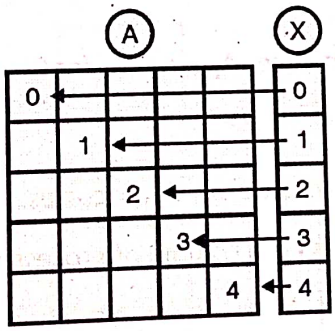


Fig. 4.7.5

(ii) One-to-all broadcast of each vector element among the n processes of each column

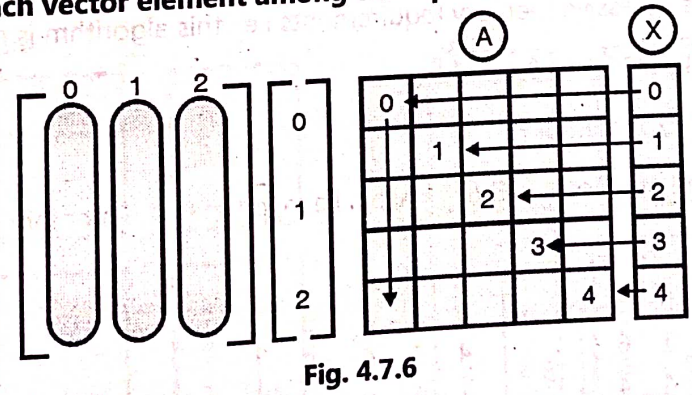


Fig. 4.7.6

- Here 0^{th} element of vector is only multiply with the 0^{th} column elements of matrix.
- In the same way, 1^{st} element of vector can multiply with 1^{st} column elements only.
- And then 1^{st} element of 0^{th} column will broadcast it all other rows of 0^{th} column only.

(iii) All-to-one reduction in each row

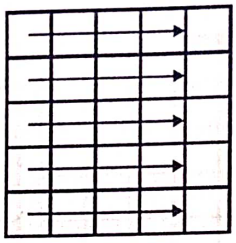


Fig. 4.7.7

- Whatever results are generated in a row by a particular process in a given element that will be reduced in the last row.
- Perform the addition operation with all the last row elements and this will give us the resultant vector.
- Cost complexity is $\Theta(n^2 \log n)$. It is not a cost-optimal algorithm.

4.7.2 Matrix-Matrix Multiplication

Q. Explain parallel Matrix-Matrix multiplication algorithm with example. **SPPU - Dec. 18, 8 Marks**

- In this module we will discuss parallel algorithms for multiplying two $n \times n$ dense, square matrices A and B to yield the product matrix $C = A \times B$.
- The serial complexity is O (n3).
- Partition A and B into p square blocks $A_{i,j}$ and $B_{i,j}$ $(0 \le i,j < p)$ of size $(n / p) \times (n / p)$ each.
- Use Cartesian topology to set up process grid. Process $P_{i,}$ initially stores $A_{i,}$ and B_i and computes block $C_{i,}$ of the result matrix.
- Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \le k < \sqrt{p}$

Algorithm

- Perform all-to-all broadcast of blocks of A in each row of processes.
- Perform all-to-all broadcast of blocks of B in each column of processes.
- Each process $P_{i,j}$ perform $C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} \; B_{k,j}$
- Drawback of this algorithm is excessive memory requirements i.e. this algorithm is not memory efficient.
  - Each process $P_{i,j}$ has $2\sqrt{p}$ blocks of $A_{i,k}$ and $B_{k,j}$
  - Each process needs $\Theta\ (n^2/p)$ memory
  - Total memory over all the processes is $\Theta(n^2 \times \sqrt{p})$, i.e., $\sqrt{p}$ times the memory of the sequential algorithm.

For Example

$$A = \begin{bmatrix} 2 & 1 & 5 & 3 \\ 0 & 7 & 1 & 6 \\ 9 & 2 & 4 & 4 \\ 3 & 6 & 7 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 6 & 1 & 2 & 3 \\ 4 & 5 & 6 & 5 \\ 1 & 9 & 8 & -8 \\ 4 & 0 & -8 & 5 \end{bmatrix}$$

- Divide the matrices into blocks assigns each block to different processes.

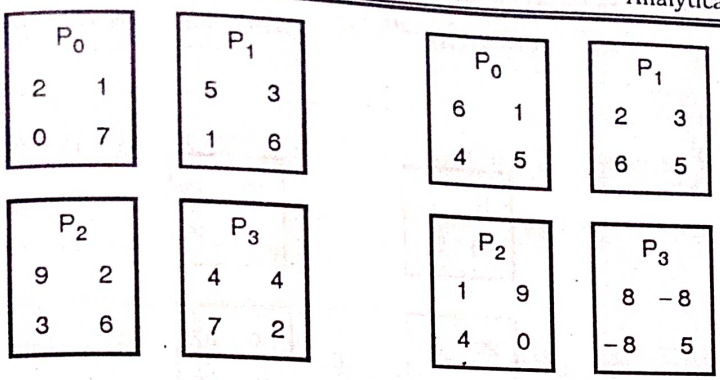


Fig. 4.7.8

- Now we have 4 processors P_0, P_1, P_2, P_3 before doing any computation, first we will implement left shift and up shift.
- Apply left shift on 1st matrix i.e. A and apply up shift on 2nd matrix i.e. B.

Left shift : Apply row wise from bottom to up

Up shift : Apply column wise from right to left.

(i) Apply left shift m bottom i.e. P_2 and P_3

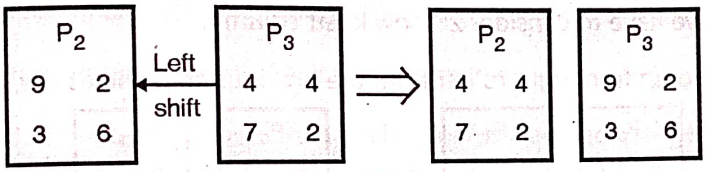


Fig. 4.7.9

(ii) Up shift

Here shift bottom column with up column

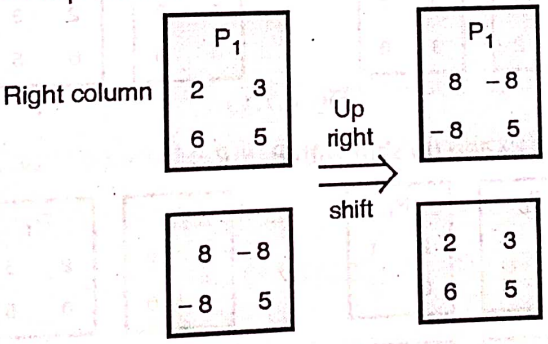


Fig. 4.7.10

(iii) Now the final values are,

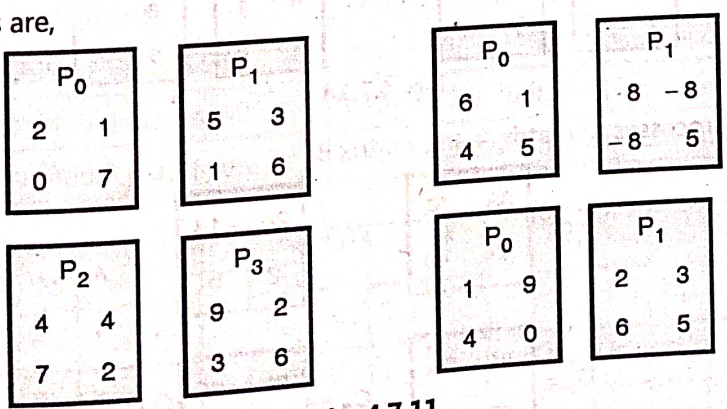


Fig. 4.7.11

(iv) Multiplication should be performed with the same index processes.

i.e. $A \cdot P_0 \times B \cdot P_0$

$(A's\ P_0) \times (B's\ P_0)$

$C_0 = \begin{bmatrix} 16 & 7 \\ 28 & 55 \end{bmatrix}$ $C_1 = \begin{bmatrix} 16 & -25 \\ -40 & 22 \end{bmatrix}$

$C_2 = \begin{bmatrix} 20 & 36 \\ 15 & 63 \end{bmatrix}$ $C_3 = \begin{bmatrix} 30 & 37 \\ 42 & 39 \end{bmatrix}$

Fig. 4.7.12

We have to perform these shifting by,

$\sqrt{P}$ times (P = Number of processors)

$\sqrt{4} = 2$

- So, we need to perform shifting operations two times.

(v) Now in 2nd iteration, we have to consider 2nd row k left column.

As shifting is bottom to up from right to left from the previous step values are,

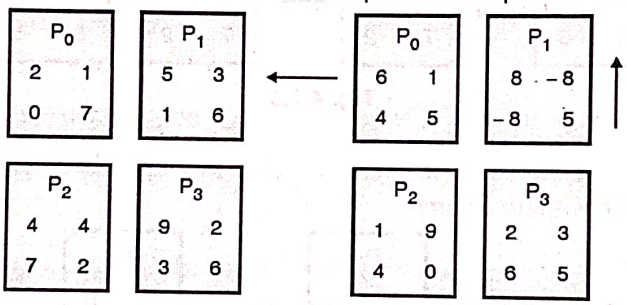


Fig. 4.7.13

So, after left shift with P_0 and P_1 and up shift with $P_3 \times P_1$

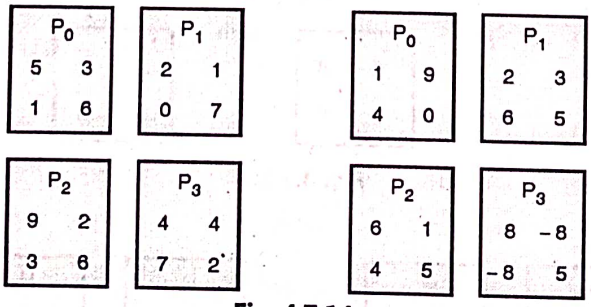


Fig. 4.7.14

(vi) Multiple same index processes of matrix A and matrix B.

$C_0 = \begin{bmatrix} 33 & 52 \\ 53 & 44 \end{bmatrix}$ $C_1 = \begin{bmatrix} 26 & -14 \\ 2 & 57 \end{bmatrix}$

$C_2 = \begin{bmatrix} 82 & 55 \\ 57 & 96 \end{bmatrix}$ $C_3 = \begin{bmatrix} 30 & 25 \\ 82 & -7 \end{bmatrix}$

Fig. 4.7.15

4.7.3 Cannon's Algorithm of Matrix-Matrix Multiplication

Q. Explain Canon's algorithm. **SPPU - Oct. 18 (In Sem.), 7 Marks**

- Cannon's algorithm is a memory efficient version of the above simple algorithm for matrix-matrix multiplication.
- In the previous multiplication, each process was broadcasting its element to the other processes. But in cannon's algorithm, shifting operation is used and not the broadcasting.
- And this is the reason it is memory efficient algorithm.
- The main advantage of the algorithm is that its **storage requirements remain constant** and are **independent of the number of processors**.

Let, $A = [a_{i,j}]\ n \times n$ $B = [b_{j,i}]\ n \times n$ be $n \times n$ matrices. Compute $C = AB$

Algorithm

1. Consider two $n \times n$ matrices A and B partitioned into p blocks.
2. A(i , j) and B(i , j) $(0 \le i,j \le \sqrt{p}\ )$ of size $(n/\sqrt{p}\ ) \times (n/\sqrt{p}\ )$ each.
3. Process P(i , j) initially stores A(i , j) and B(i , j) computes block C(i , j) of the result matrix.
4. The initial step of the algorithm regards the alignment of the matrixes.
5. Align the blocks of A and B in such a way that each process can independently start multiplying its local submatrices.
6. This is done by shifting all submatrices A(i , j) to the left (with wraparound) by i steps and all submatrices B(i , j) up (with wraparound) by j steps.
7. Perform local block multiplication.
8. Each block of A moves one step left and each block of B moves one step up (again with wraparound).
9. Perform next block multiplication, add to partial result, repeat until all blocks have been multiplied.

Consider iteration, i = 1, j = 2:

C [1,2] = A[1,0] * B [0,2] + A [1,1] * B [1,2] + A [1,2] * B [2,2]

Steps

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 1 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.

For Example

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

Fig. 4.7.16

- Divide matrix A and B into blocks (partition) such that each block contains each element of matrix.
- Now apply shifting operation
  - Row wise shifting (rap around circular shift) with A matrix and
  - Column wise shifting with matrix B
  - Row wise shifting is in left direction and column wise shifting is in up direction.

Row shifting consider i, index only i.e.

A(i,j) → Consider i only for shifting

Column shifting consider j index only i.e.

A(i,j) → Consider j only for shifting

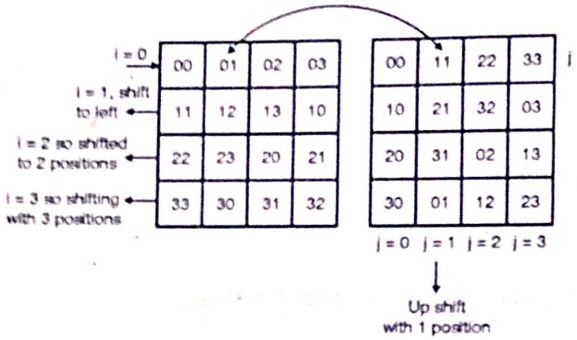


Fig. 4.7.17

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

Fig. 4.7.18

- After shifting, combine both the matrices (consider above positions in matrix)

  A → 00 and B → 00 add in same block C → 00
- Now one shift for each block. So rowwise - left shift (only 1 shifting) and columnwise- up shift (only 1 shifting)
  - **A matrix :** Left shift by 1 shift only.
  - **B matrix :** Up shift by 1 shift only.

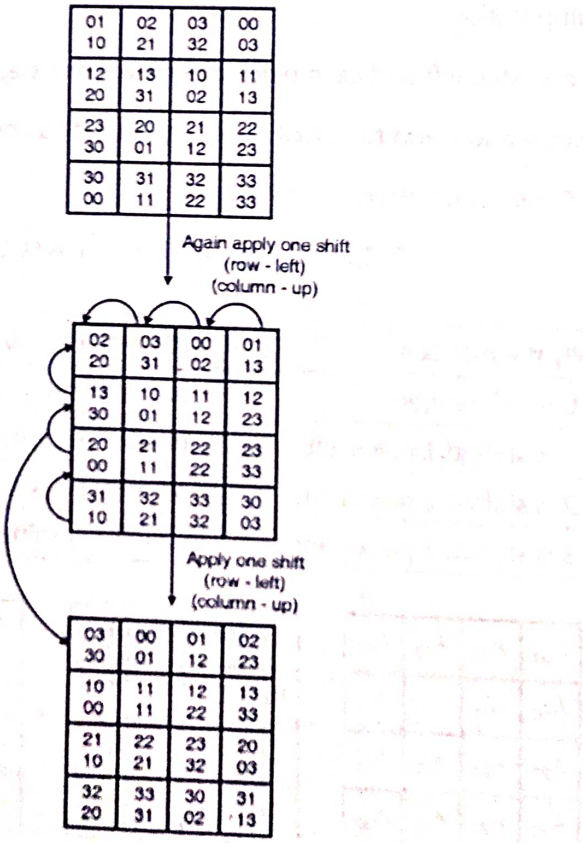


Fig. 4.7.19

We have to perform shifting $\sqrt{P}$ times where P is number of blocks.

In this example number of blocks are 16. So,

$$\sqrt{P} = \sqrt{16} = 4$$

4 shifts are (i) Initial shift

(ii) One shift

(iii) One shift

(iv) One shift

So, the result will be as shown in Fig. 4.7.20.

C =

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

Fig. 4.7.20

Scaling and distributing the subtasks among the processors

1. The sizes of the matrices blocks can be selected so that the number of subtasks will coincides the number of available processors p.
2. The most efficient execution of the parallel Canon's algorithm can be provided when the communication network topology is a two-dimensional grid.
3. In this case the subtasks can be distributed among the processors in a natural way : the subtask (i,j) has to be placed to the p(i,j) processor.

Complexity

Parallel time

$$T_p = \frac{n^3}{p} + 2\sqrt{p}t_s + 2\,t_w\frac{n^2}{p}$$

Where,

t_s is the startup cost.

t_w is the word transfer time.

4.7.4 The DNS Algorithm

- This algorithm is known as the DNS algorithm because it is due to Dekel, Nassimi and Sahni.
- In this module, we will learn how to parallelize matrix-matrix multiplication with the DNS algorithm.

- In the previous section we have learned to parallelize the matrix multiplication which uses block 2-D partitioning of the input and output.
- It uses a 3-D partitioning.
- Visualize the matrix multiplication algorithm as a cube – matrices A and B come in two orthogonal faces and result C comes out the other orthogonal face.
- Cube with faces corresponding to A,B,C.
- Each internal node in the cube represents a single add-multiply operation.
- DNS algorithm partitions this cube using a 3-D block scheme.
- We can use upto n^3 processes for better concurrency.

Algorithm

1. Assume an n*n*n mesh of processors.
2. Move the columns of A and rows of B and perform broadcast.
3. One-to-all broadcast along j and i axis.
4. All-to-one reduction (+) along k axis.
5. Each processor $P_{i,j,k}$ computes a single multiply :

   C[i,k] = A [i,k] * B [k,j]

   This is followed by an accumulation along the k dimension.
6. Since each add-multiply takes constant time and accumulation and broadcast takes log n time, the total runtime is Θ (log n), communication on groups of n processes.
7. Not cost optimal for n^3 processes.

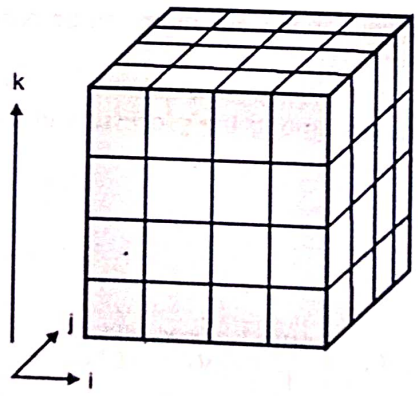
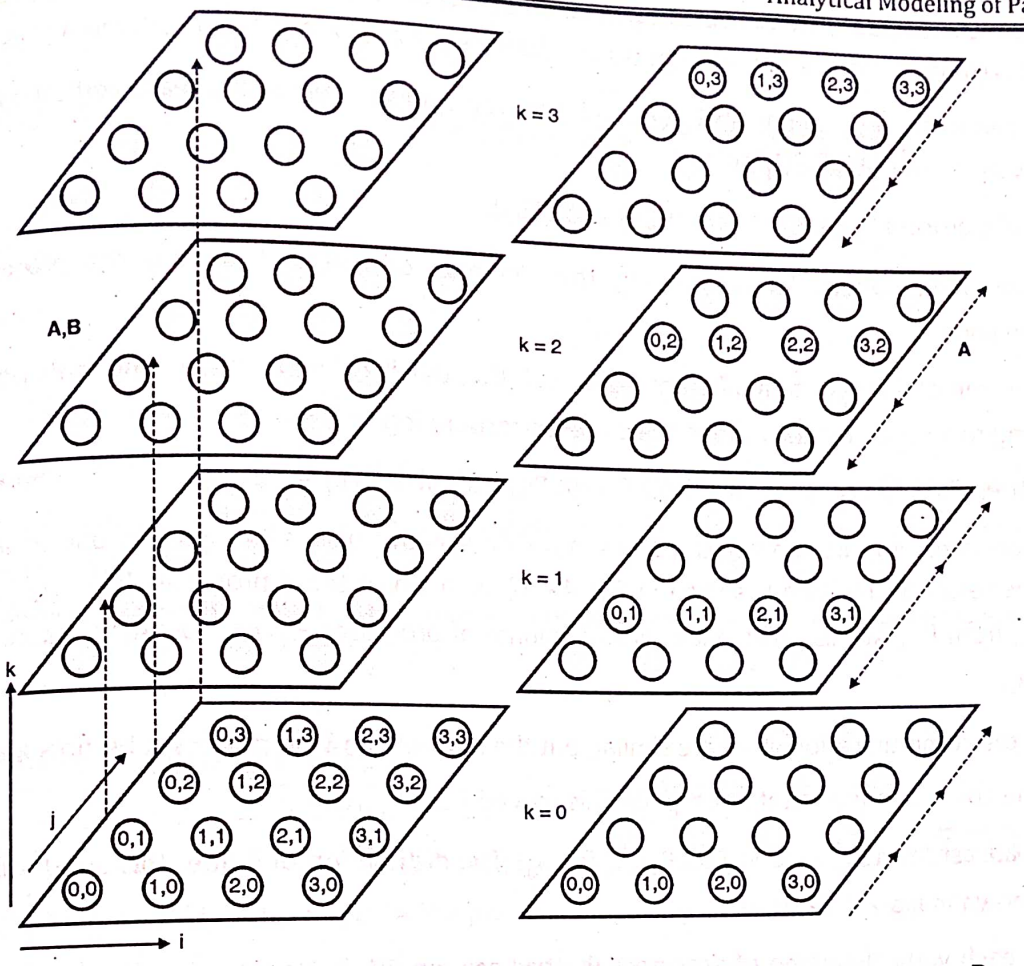


Fig. 4.7.21

For Example :

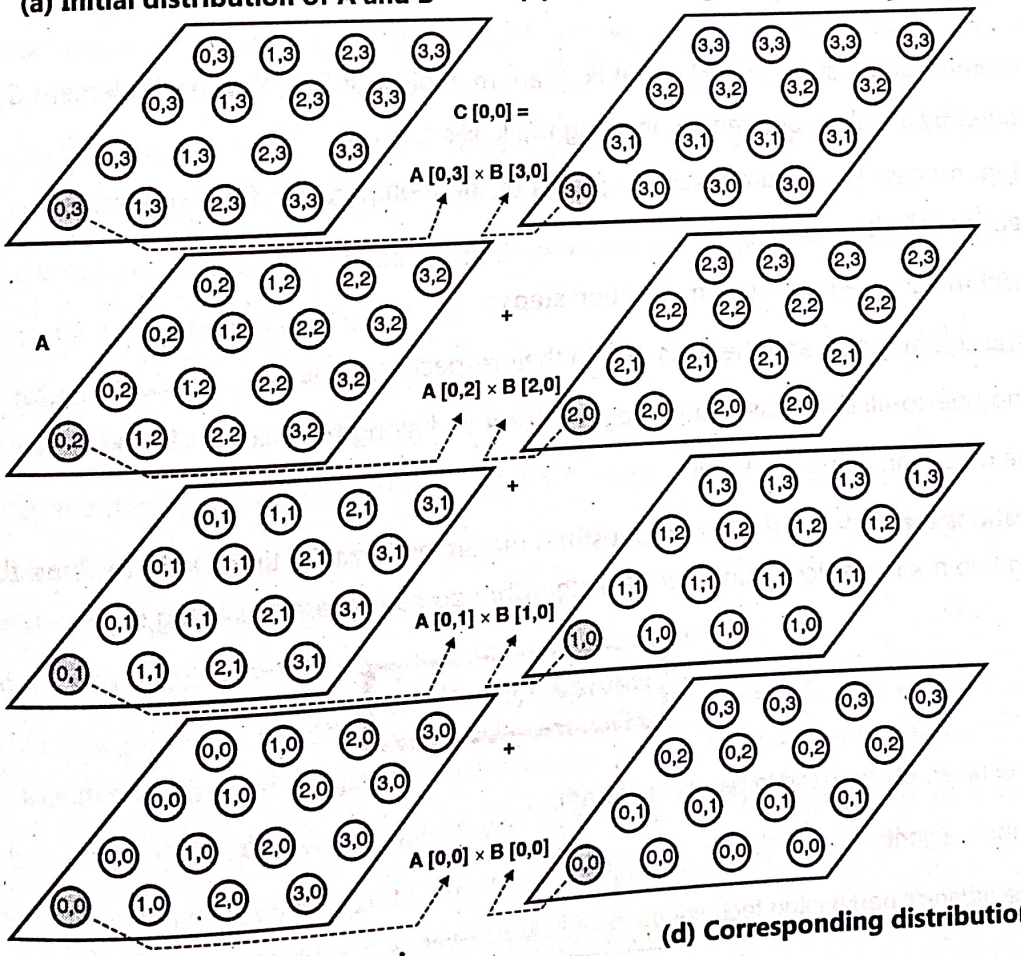
Now let us discuss a practical parallel implementation of matrix multiplication based on this DNA algorithm.

The process arrangement can be visualized as n planes of n × n processes each. Each plane corresponds to a different value of k. Initially, as shown in Fig. 4.7.22(a), the matrices are distributed among the n^2 processes of the plane corresponding to k = 0 at the base of the three-dimensional process array. Process $P_{i,j,0}$ initially owns A[i, j] and B[i, j].



(a) Initial distribution of A and B

(b) After moving A [i,j] from $P_{i,j,0}$ to $P_{i,j,j}$



(c) After broadcasting A [i,j] along j axis

(d) Corresponding distribution of B

Fig. 4.7.22

- The vertical column of processes $P_{i,j,*}$ computes the dot product of row A [i, *] and column B [*, j].
- Therefore, rows of A and columns of B need to be moved appropriately so that each vertical column of processes $P_{i,j,*}$ has row A[i, *] and column B[*, j].
- More precisely, process $P_{i,j,k}$ should have A[i, k] and B[k, j].
- The communication pattern for distributing the elements of matrix A among the processes is shown in Fig. 4.7.22 (a) and (c).
- First, each column of A moves to a different plane such that the j^{th} column occupies the same position in the plane corresponding to k = j as it initially did in the plane corresponding to k = 0.
- The distribution of A after moving A[i, j] from $P_{i,j,0}$ to $P_{i,j,j}$ is shown in Fig. 4.7.22(b).
- Now all the columns of A are replicated n times in their respective planes by a parallel one-to-all broadcast along the j axis. The result of this step is shown in Fig. 4.7.22(c), in which the n processes $P_{i,0,j}$, $P_{i,1,j}$, ..., $P_{i,n-1,j}$ receive a copy of A[i, j] from $P_{i,j,j}$. At this point, each vertical column of processes $P_{i,j,*}$ has row A[i, *]. More precisely, process $P_{i,j,k}$ has A[i, k].
- For matrix B, the communication steps are similar, but the roles of i and j in process subscripts are switched.
- In the first one-to-one communication step, B[i, j] is moved from $P_{i,j,0}$ to $P_{i,j,i}$.
- Then it is broadcast from $P_{i,j,i}$ among $P_{0,j,i}$, $P_{1,j,i}$, ..., $P_{n-1,j,i}$. The distribution of B after this one-to-all broadcast along the i axis is shown in Fig. 4.7.22(d).
- At this point, each vertical column of processes $P_{i,j,*}$ has column B[*, j]. Now process $P_{i,j,k}$ has B[k, j], in addition to A[i, k].
- After these communication steps, A[i, k] and B [k, j] are multiplied at $P_{i,j,k}$. Now each element C [i, j] of the product matrix is obtained by an all-to-one reduction along the k axis.
- During this step, process $P_{i,j,0}$ accumulates the results of the multiplication from processes $P_{i,j,1}$, ..., $P_{i,j,n-1}$. Fig. 4.7.22 shows this step for C[0, 0].
- The DNS algorithm has three main communication steps :
  (i) Moving the columns of A and the rows of B to their respective planes
  (ii) Performing one-to-all broadcast along the j axis for A and along the i axis for B, and
  (iii) All-to-one reduction along the k axis.
- All these operations are performed within groups of n processes and take time $\Theta(\log n)$. Thus, the parallel run time for multiplying two $n \times n$ matrices using the DNS algorithm on n^3 processes is $\Theta(\log n)$.

Review Questions

Q. 1 Explain the sources of overhead in parallel program. **(8 Marks)**

Q. 2 Explain Canon's algorithm. **(8 Marks)**

Q. 3 What are the different partitioning techniques used in Matrix-vector multiplication. **(8 Marks)**

❑❑❑

5

Unit - V

CUDA Architecture

Syllabus

Introduction to GPU : Introduction to GPU Architecture overview, Introduction to CUDA C- CUDA programming model, write and launch a CUDA kernel, Handling Errors, CUDA memory model, Manage communication and synchronization, Parallel programming in CUDA- C.

5.1 Introduction to GPU and CUDA

Q. What is CUDA ? **SPPU - May 19, 1 Mark**

- "Compute Unified Device Architecture". CUDA is a parallel computing architecture from NVIDIA.
- It is parallel computing platform which is implemented by GPU. It is also an interface API.
- It defines general purpose programming model in which user kicks off batches of threads on the GPU (dedicated super-threaded, massively data parallel coprocessor.)
- Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly.
- Using CUDA, the GPUs can be used for general purpose processing; this approach is known as GPGPU.
- It enables heterogeneous systems (i.e. CPU + GPU)
- CUDA gives program developers direct access to the virtual instructions set and memory of the parallel computational elements in CUDA GPUs.
- It supports shared memory and synchronization along the threads.
- NVIDIA developed the CUDA programming model and software environment to let programmers write scalable parallel programs using a straightforward extension of the C language.
- CUDA platform works with programming languages like C, C++ and Forton.
- The CUDA platform is accessible to software developers through CUDA-accelerated libraries, compiler directives, and extensions to industry standard programming languages, including C, C++ and Fortran.
- C/C++ programmers use 'CUDA C/C++', compiled with 'nvcc', NVIDIA's LLVM based C/C++ compiler.
- The initial CUDA SDK was made public in 2007, for Microsoft Windows and Linux.
- CUDA works with all Nvidia GPUs from the G8x series onwards, including GeForce, Quadro and the Tesla line.
- CUDA is compatible with most standard operating systems.

What are the benefits of CUDA ?

- CUDA gives us a way that we can efficiently process thousands of elements for a particular task in parallel.
- It gives us a way that nearby tasks can communicate and collaborate efficiently.
- Good for lots of computations and lots of data.

5.1.1 CUDA Architecture Overview

Q. Explain CUDA Architecture with Schematic Diagram.	**SPPU - Oct. 18 (In Sem.), 8 Marks**
Q. Describe CUDA Architecture in details with neat diagram.	**SPPU - Dec. 18, 8 Marks**
Q. Draw and explain CUDA architecture in detail.	**SPPU - May 19, 8 Marks**

- Parallel computing architecture developed by NVIDIA.
- It consists of C language extensions in our source code that is going to execute on the compute device.It splits into two parts, host and the device.
- Host is the computer- the machine, the CPU that running Windows, Linux or MAC OS.
- The device is going to be the GPU, the hardware that plugged into PCI Express bus or it's an onboard chip like a Tegra.
- For fast single thread execution CPU is optimized. It is designed to execute one thread or two threads concurrently.
- For high multi thread throughput, GPU is optimized. It is designed to execute many parallel threads concurrently.
- CUDA architecture is divided into two main parts CPU and GPU as shown in Fig. 5.1.1.

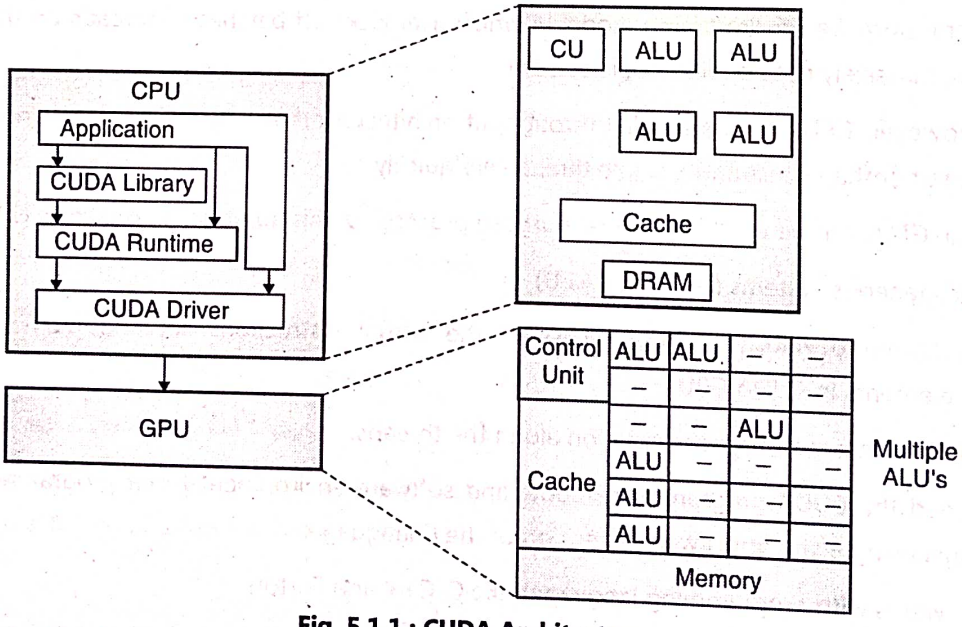


Fig. 5.1.1 : CUDA Architecture

1. CPU

- Host comprises of the CPU and its memory. The Host code is executed by CPU.
- Control intensive instructions are handled by CPU.
- Small data size tasks are executed by CPU.
- CPU consists of 3 main parts :

(i) CUDA Libraries

Advanced libraries that include BLAS, FFT, and other functions optimized for the CUDA Architecture.

(ii) CUDA Runtime

The runtime API is a higher-level API implemented on top of the driver API. Each function of the runtime API is broken down into more basic operations issued to the driver API. It handles dynamic behavior.

(iii) CUDA Driver

The driver API is a low-level API and is relatively hard to program, but it provides more control over how the GPU device is used.

Expanded version of CPU consists of following units as shown in Fig. 5.1.1.

(i) Control Unit

(ii) Arithmetic Logic Unit (ALU)

(iii) Cache memory

(iv) DRAM

Control Unit : The key component in high-performance computing is the Central Processing Unit (CPU), usually called the core.

2. GPU

- Graphic Processor Unit or GPU is a highly parallel, multithreaded, many core processor with tremendous computational power.
- GPU has many processors dedicated to computations.
- The GPU is viewed as a compute device which is coprocessor to the CPU or host and has its own DRAM (device memory). Therefore, GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus.
- Device comprises of the GPU and its memory. The Device code executed by GPU.
- Computation intensive tasks are handled effectively by GPU because of high processing power.
- Huge amount of data is handled by GPU not by CPU.
- Expanded version of GPU consists of Multiple processing units (transistors) that gives high processing power.
- NVIDIA introduced a programming model improvement called Unified Memory, which bridges the divide between host and device memory spaces. This improvement allows us to access both the CPU and GPU memory using a single pointer, while the system automatically migrates the data between the host and device.

5.1.2 CUDA Programming Models for High Performance Computing Architecture

Q. Explain CUDA Programming model for HPC Architecture. **SPPU - Oct. 18 (In Sem.), 7 Marks**

- The model used for programming in particular architecture acts as a bridge between an application and its implementation on available hardware.
- Fig. 5.1.2 shows layers of abstraction that lie between the program and the programming model implementation.

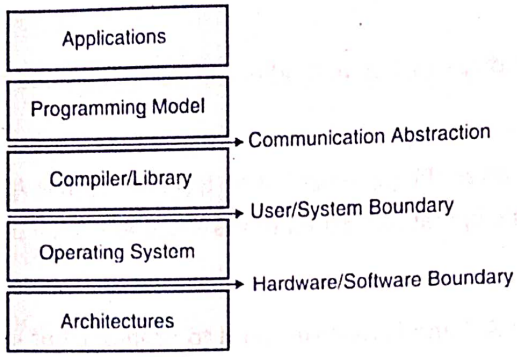


Fig. 5.1.2 : Layers of Abstraction

- The boundary between the program and the programming model implementation is communication abstraction. A compiler or libraries using system calls to access hardware services are used to provide such a communication abstraction.
- The components of the program share information and coordinates their activities by the instruction are provided through program.
- The CUDA programming model share many abstractions with other parallel programming model. It has following features of fully utilization of GPU.
  - A way to organize threads on the GPU through a hierarchy structure.
  - A way to access memory on the GPU through a hierarchy structure.
- We will discuss these features in the next section.

CUDA Programming Structure

- The CUDA programming model enables us to execute applications on heterogeneous computing systems.
- CUDA programming model is a combination of serial and parallel executions. Fig. 5.1.3 shows this heterogeneous type of programming.
- It is used to make an interaction between CPU and GPU programming models. Data may be copied from host memory to device memory and the results are copied back to host from the device memory.

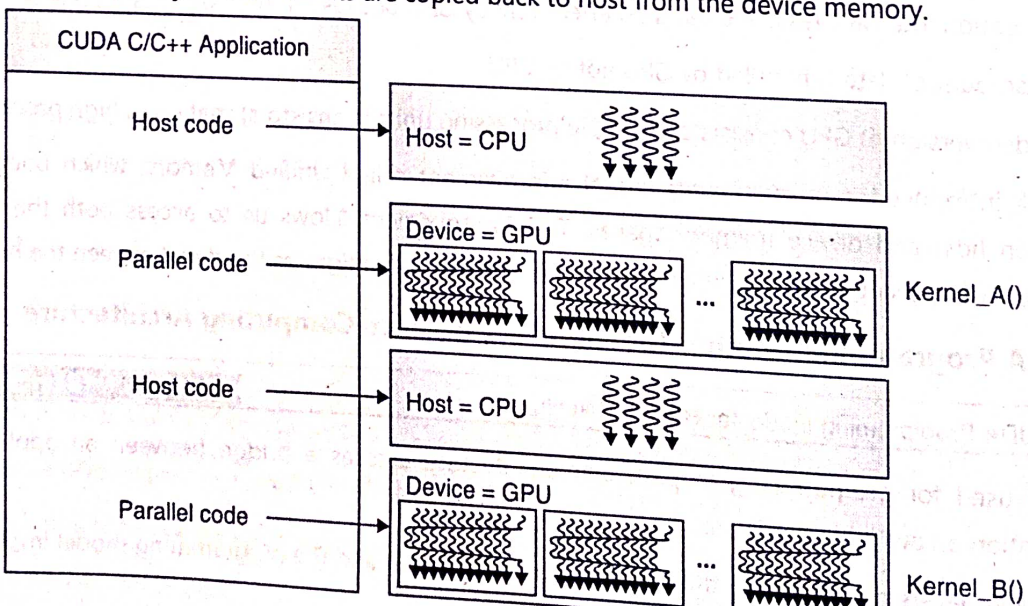


Fig. 5.1.3 : CUDA Programming Model

- It divides code into Host (CPU) and Device (GPU) Code.
- Serial code executes in a Host thread. Parallel code executes in many concurrent Device threads across multiple parallel processing elements.
- An application executing on a heterogeneous platform is typically initialized by the CPU. The CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks on the device.
- Processing flow of a CUDA program:
  1. Copy data from CPU memory to GPU memory
  2. Invoke kernel to run on the GPU.
  3. Copy data back from GPU to CPU memory
  4. Release the GPU memory and reset the GPU.
- CUDA code file name extension .cu
- The host code is written in ANSI C, and the device code is written using CUDA C.

5.2 Using the CUDA Architecture

- To facilitate parallel computing, NVIDIA added many features to its GPU chips, and these features we have to use using OpenGL or DirectX.
- Users have to write their computations in a graphics-oriented shading language such as OpenGL's GLSL or Microsoft's HLSL.
- To implement some of the features of CUDA architecture, NVIDIA developed a compiler using C language, called CUDA C.
- CUDA C became the first language specifically designed by a GPU company to facilitate general- purpose computing on GPU's
- To exploit the CUDA architecture's computational power, NVIDIA also provides a specialized hardware driver. So users are no longer required to have any knowledge of the OpenGL or DirectX graphics programming interfaces.

5.3 Advantages, Limitations and Applications of CUDA

Advantages of CUDA

Q. Write advantages of CUDA. **SPPU - Dec. 18, 3 Marks**

1. CUDA gives us a way that we can efficiently process thousands of elements for a particular task in parallel.
2. It gives us a way that nearby tasks can communicate and collaborate efficiently.
3. Good for lots of computations and lots of data.
4. CUDA provides ability to use high-level languages such as C to develop application that can take advantage of high level of performance and scalability that GPUs architecture offer.
5. CUDA also exposes fast shared memory (16KB) that can be shared between threads.
6. Full support for integer and bitwise operations.
7. Compiled code will run directly on GPU

CUDA program execution

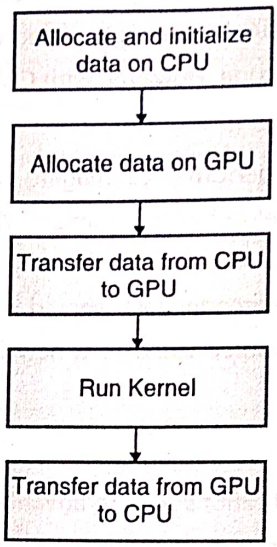


Fig. 5.4.1

5.4.1 First CUDA C Program

```
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

Output

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```

- Standard C that runs on the host.
- NVIDIA compiler (nvcc) can be used to compile programs with no device code.

```
//Program With Device Code
__global__ void mykernel(void)
{
}
int main(void)
{
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Output

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- With the help from above program we will learn what is writing and launching in CUDA C.

5.4.2 Writing

```
__global__ void mykernel(void)
{
}
```

Syntax

```
__global__ void kernelname(argument list)
```

- Here CUDA C/C++ keyword __global__ indicates a function that runs on the device and is called from host code.
- NVCC separates source code into host and device components.
- Device functions (eg. mykernel()) processed by NVIDIA compiler
- Host function (eg. main()) processed by standard host compiler.
- There are some restrictions with device code like
  - Kernel return type should be void
  - It accesses only device memory (GPU has its local memory). Even if we have declared kernel as global still kernel will execute on device only.
  - Static variables and function pointers are not supported by kernel.

5.4.3 Launching

- As we have created kernel, now how to launch it. Launching of kernel is done by Host(CPU).

```
mykernel<<<1,1>>>();
```

Syntax

```
kernelname<<<grid, block>>>(argument list);
```

- Triple angle brackets mark a call from host code to device code
- Also called a "**kernel launch**"
- We'll return to the parameters (1,1) in a next session.
- This way we execute a function on the GPU.

Note : So mykernel() will execute on the device and will be called from the host.

5.5 C Kernels

Q. Explain how the CUDA C program executes at the kernel level with example. **SPPU - May 19, 9 Marks**

- A CUDA kernel is the code executed by an array of threads.

- All threads run the same code. Each thread has an ID that it uses to compute memory addresses and make control decisions.
- CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.
- Remember, a kernel is just a name for a function that executes on the GPU.
- To invoke a kernel you use the following syntax :

  kernel_function<<<num_blocks, num_threads>>>(param1, param2, .)
- Parallel portions of an application are executed on the device as kernels. One kernel is executed at a time. Many threads execute each kernel.

Basic Terms

1. **Thread :** Same as java threads but more limited than Java threads.
2. **Kernel :** The code executed by thread.
3. **Thread Block :** A group of threads that executes simultaneously on the same processor. Each thread within the block uniquely identified by 1D,eg. (128,1,1), 2D, e.g. (16,16,1) or 3D,eg.(4,8,16)indexes. Fast shared memory and lightweight barrier synchronization among threads in the same block.
4. **Warps :** Set of 32 concurrent threads in a block. Only one (Fermi) or two (Kepler) instructions per cycle per warp.
5. **Grid :** A group of thread blocks, each block executing on a different processor. Each block within the grid uniquely identified by one (1-D) or two (2-D) indexes. Maximum 65,535 thread blocks in a 1-D grid.
6. **Stream :** A grid in the process of executing a kernel. The system schedules each thread block to execute on an available processor. Each thread in each block executes the same code (kernel). The thread blocks "**stream**" through the compute engine called "**stream programming**".

- The CUDA logical hierarchy (Fig. 5.5.1) explains the points discussed above with respect to grids, blocks and threads.

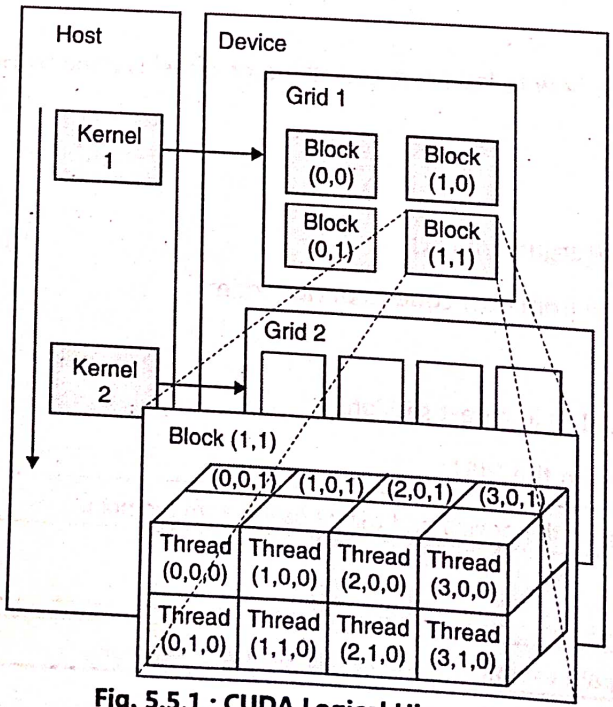


Fig. 5.5.1 : CUDA Logical Hierarchy

- A block contains a number of threads. A thread block or warp is a collection of threads that can use shared data through shared memory and synchronize their execution.
- Basically, Threads within a block cooperate via shared memory. Threads within a block can synchronize. Threads in different blocks cannot synchronize.
- Threads allow programs to transparently scale to different GPUs.
- Threads from different blocks operate independently, and can be used to perform different functions in parallel.
- Each block and each thread is identified by a build-in block index and thread index accessible within the kernel.

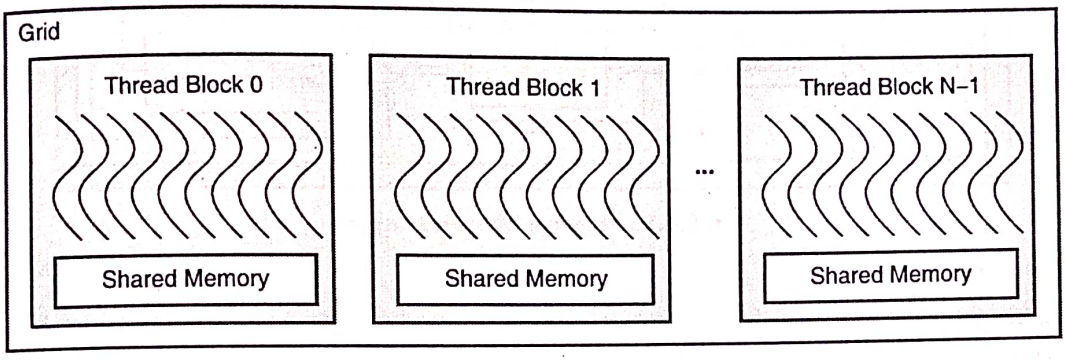


Fig. 5.5.2

- Each thread uses IDs to decide what data to work on as shown in Fig. 5.5.3.

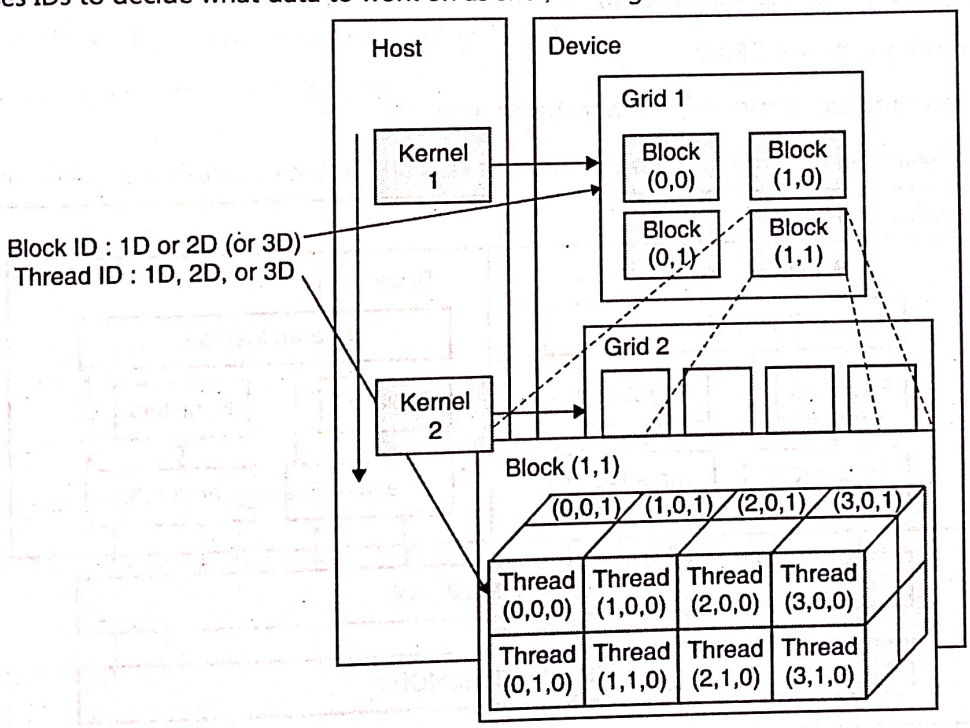


Fig. 5.5.3

- It simplifies memory addressing when processing multidimensional data.

5.6 Manage CUDA Memory Model

Q. How to manage GPU Memory. **SPPU - Oct. 18 (In Sem.), 5 Marks**

Q. Write a short note on : Managing GPU memory. **SPPU - May 19, 9 Marks**

- CPU and GPU have separate memory spaces.

- Device pointers point to GPU memory.
- Host pointers point to CPU memory.
- Host(CPU) code manages device(GPU) memory.

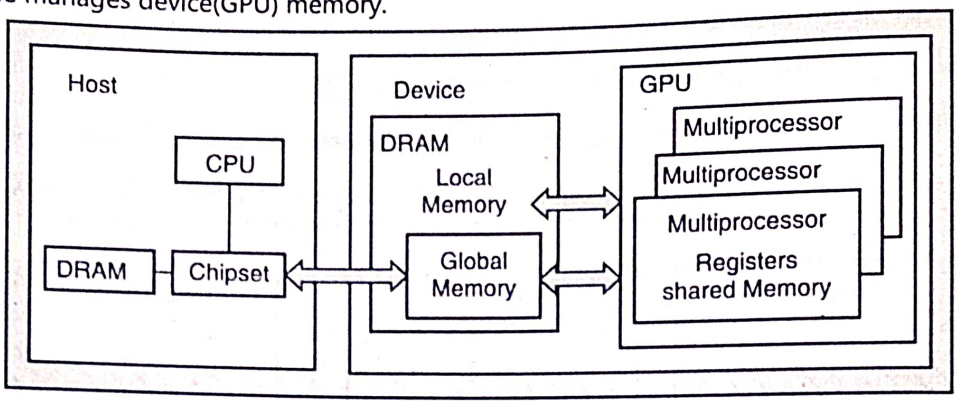


Fig. 5.6.1 : Physical Memory Layout

- It does following tasks
  - Allocate or free memory
  - Copy data to and from device
  - Applies to global device memory (DRAM)
- Local memory resides in device DRAM.
- Host can read and write global memory but not shared memory.
- CUDA threads may access data from multiple memory spaces during their execution as shown in Fig. 5.6.2.

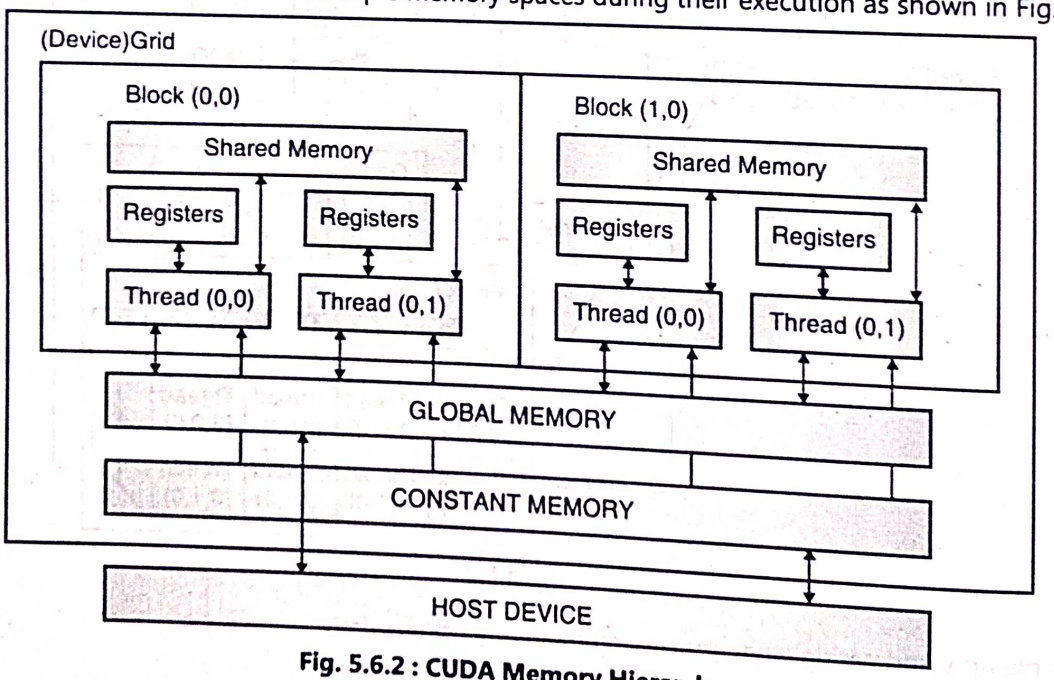


Fig. 5.6.2 : CUDA Memory Hierarchy

- Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

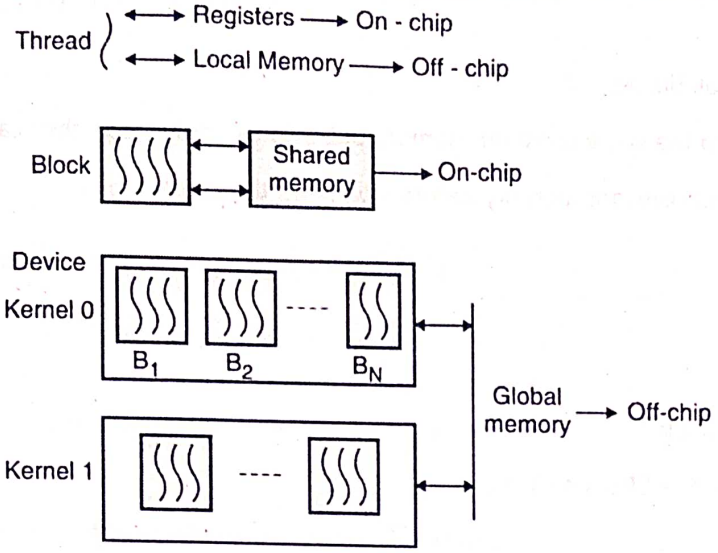


Fig. 5.6.3

Processor registers

- Local variables in the kernel code typically go in processor registers. The variables we declare in a kernel will use registers unless we run out or they cannot be stored in registers, then local memory will be used.
- Register scope is per thread.
- Unlike the CPU, there's thousands of registers in a GPU.
- Registers are the fastest memory on the GPU.

Local Memory

- Not shared
- The scope of local memory is per thread.
- Read-write
- Off the processor chip

Shared Memory

- Shared by all threads in the same block only
- Shared memory is used to enable fast communication between threads in a block.
- Read-write
- On the processor chip

Global Memory

- There is a large amount of global memory.
- Shared by all threads in all blocks
- Main means of communicating Read-Write Data between host and device.
- Its slower to access than other memories like shared and registers.
- Off the processor chip

- The main use of synchronization is to prevent RAW, WAR, WAW hazards during communication between multiple threads.
  - RAW(Read After Write)
  - WAR(Write After Read)
  - WAW(Write After Write)

  And Synchronize to commit all the memory writes, reads and computation.
- CUDA synchronizes threads using function __syncthreads()

  __syncthreads : It is sometimes necessary to ensure that all data from all threads is valid before threads read from shared memory which is written to by other threads.

  This function synchronizes all threads within a block. This function waits till all the threads execute.
- To ensure all threads are synchronized, we use a synchronization barrier.

  There are two types of synchronization in CUDA.

  1. Implicit synchronization
  2. Barrier synchronization or Explicit Synchronization
- In barrier synchronization, a programmer can place the synchronization barrier explicitly, to synchronize tasks.
- In implicit some functions are implicitly synchronized, which means one or all threads must complete before proceeding to the next section.

Let's summarize all these points:

(i) Threads within a warp communicate using shared or global memory and synchronize using implicit synchronization

(ii) Warps within a thread block communicate using shared or global memory and synchronize using barrier synchronization.

(iii) Thread blocks within a given grid or kernel communicate using global memory and synchronize using atomic memory operations

(iv) Thread blocks from different grids or kernels communicate using global memory and synchronize using implicit synchronization

5.8 Parallel Programming in CUDA C

Q. Design a simple CUDA kernel function to multiply two integers.

Q. Explain how the CUDA C program executes at the kernel level with example. SPPU - Dec. 18, 6 Marks SPPU - May 19, 9 Marks

- CUDA provides a scalable way to express parallelism. We simply write a program one data element and it gets automatically distributed across hundreds of cores for thousands of threads.GPU computing is about massive parallelism.

So how do we run code in parallel on the device ?

- Solution lies in the parameters between the triple angle brackets:

```
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
add<<< N, 1 >>>( dev_a, dev_b, dev_c );
```

Instead of executing add() once, add() executed N times in parallel.

- With add() running in parallel, let's do vector addition. Consider a case where we have 3 vectors A, B and C and we simply want to add these vectors and store the results.

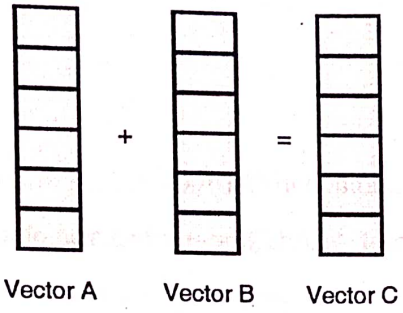


Fig. 5.8.1

- In C language, sequentially execution is simply look like a loop

```
for (i=0....N-1)                //N is length of the vector
{
    C[i] = A[i]+B[i];
}
```

In CUDA instead of this, we consider is each of these in individual additions is a thread and a thread operates it as a one data element.

Modified code

Instead of this being a loop, we write it like a function to which pass these three vectors and each thread has an index.

```
VecAdd(A,B,C)
{
    int i = threadIdx;
    C[i] = A[i]+B[i];
}
```

So now each thread is operating on exactly one element.

```
//Function Call
VecAdd<<<N>>>(A,B,C);
```

This function call instead of executing once per call executes N times as in separate threads, one thread per data element. Here N can become very large thousands perhaps even millions allowing us to scale up to even greater levels.

Basic CUDA program structure

```
int main(int argc, char **argv)
{
```

1. Allocate memory space in device (GPU) for data
2. Allocate memory space in host (CPU) for data
3. Copy data to GPU

4. Call "kernel" routine to execute on GPU(launching of kernel)
5. Transfer results from GPU to CPU
6. Free memory space in device (GPU)
7. Free memory space in host (CPU)

return;
}

Let's see the whole program of vector addition in CUDA C.

- **Terminology :** Each parallel invocation of VecAdd() referred to as a block. Kernel can refer to its block's index with variable i.
- Each block adds a value from a[] and b[] , storing the result in c[]:

```
__global__ void VecAdd(int *a, int *b, int *c)
{
        c[i]= a[i] + b[i];
}
```

By using i to index arrays, each block handles different indices

We write this code:

```
__global__ void VecAdd(int *a, int *b, int *c)
{
        c[i] = a[i] + b[i];
}
```

- This is what runs in parallel on the device:

Block 0 c[0] = a[0] + b[0];

Block 1 c[1] = a[1] + b[1];

Block 2 c[2] = a[2] + b[2];

Block 3 c[3] = a[3] + b[3];

- Parallel Addition: VecAdd()

Using our newly parallelized VecAdd()kernel:

```
__global__ void VecAdd( int *a, int *b, int *c )
{
        c[i] = a[i] + b[i];
}
```

Let's take a look at main()...

Parallel Addition: main()

```
#define N 512
int main( void )
{
    int *a, *b, *c; // host copies of a, b, c
```

```
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
}
// Program for addition of 2 vectors using CUDA C language.
#define N 512
int main( void )
{
 int *a, *b, *c;                    // host copies of a, b, c
 int *dev_a, *dev_b, *dev_c;        // device copies of a, b, c
 int size = N * sizeof( int );      // we need space for 512 integers

 // allocate device copies of a, b, c
 cudaMalloc( (void**)&dev_a, size );
 cudaMalloc( (void**)&dev_b, size );
 cudaMalloc( (void**)&dev_c, size );
 a = (int*)malloc( size );
 b = (int*)malloc( size );
 c = (int*)malloc( size );
 random_ints( a, N );
 random_ints( b, N );

 // copy inputs to device
    Destination      Source
 cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
 cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

 // copy inputs to device
 cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
 cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

 // launch add() kernel with N parallel blocks
 add<<< N, 1 >>>( dev_a, dev_b, dev_c );
 // copy device result back to host copy of c
     Destination    Source

 cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

 // Use regular C free routine to deallocate memory if previously allocated with malloc.
```

```
  free( a ); free( b ); free( c );

// Use CUDA cudaFree routine.
  cudaFree( dev_a );
  cudaFree( dev_b );
  cudaFree( dev_c );
 return 0;
}
```

Review Questions

Q. 1 What is CUDA ? Explain with the help of architecture. **(8 Marks)**

Q. 2 Explain CUDA programming structure/Model with the help of diagram. **(7 Marks)**

Q. 3 Explain various applications of CUDA. **(8 Marks)**

Q. 4 Write a Hello World program in CUDA C programming language. **(5 Marks)**

Q. 5 How to manage GPU memory? **(5 Marks)**

Q. 6 Design a simple CUDA kernel function to add two integers. **(5 Marks)**

❑❑❑

6

Unit - VI

High Performance Computing Applications

Syllabus

Scope of Parallel Computing, **Parallel Search Algorithms :** Depth First Search(DFS), Breadth First Search(BFS), **Parallel Sorting :** Bubble and Merge, **Distributed Computing :** Document classification, Frameworks – Kuberbets, GPU Applications, Parallel Computing for AI/ ML

6.1 Issues in Sorting on Parallel Computers

Q. Explain the issue in Sorting of Parallel Algorithm. **SPPU - Oct. 18 (In Sem.), 8 Marks**

Q. Discuss the issues in sorting for parallel computers. **SPPU - Dec. 18, 8 Marks**

- The important point to make any sequential algorithm parallel is we need to distribute **sorted** elements to the available processes.
- In this type of sorting on parallel processors, there are many issues that need to be addressed.
- There are two issues :
  1. Where the Input and output is stored?
  2. How comparisons are performed?

1. Where the Input and output is stored?

- Input and output i.e. sorted sequences are stored in processor's memory in sequential algorithms.
- But there are two places where input and output are stored in parallel algorithms.
  (i) Stored on only one of the processes.
  (ii) Distribute among the processes and store on each process.
- A method of distribution of the sorted output sequence among the processes is to list out the processes and use this list to specify a final sorted order.
- For example, if process P_i comes before process P_j in the list, then all the elements stored in Pi will be smaller than those stored in P_j.

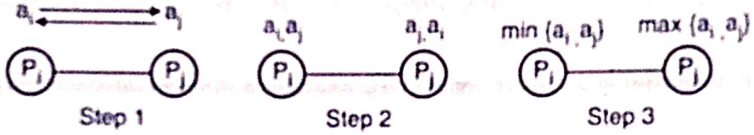
2. How comparisons are performed?

- Sorting can be comparison-based or non-comparison based.
- The fundamental operation of comparison-based sorting is compare-exchange.

- As elements are stored locally in the processor's memory, a sequential sorting algorithm can easily perform a compare-exchange on two elements.
- But this is difficult in parallel sorting algorithms, as the elements are stored on different processors.

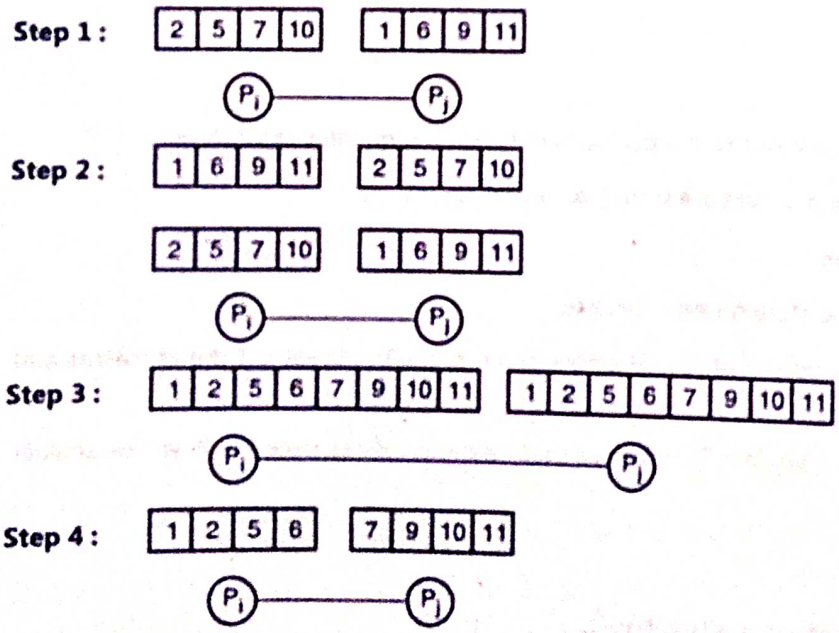
(i) Comparison if one element per process (compare-exchange)

- If each process holds only one element of the sequence to be sorted then comparison is called **compare-exchange.**
- Suppose a pair of processes (P_i, P_j) want to compare their elements a_i and a_j.
- Here process P_i and P_j send their elements to each other. Each process compares received element with its own and retains appropriate elements. Process Pi keeps min(a_i,aj) and Pj keeps max(a_i,a_j).



(ii) Comparison if more than one element per process (compare-split)

- If in a parallel algorithm a large sequence of elements with relatively small number of processors is to be sorted then each processor is assigned a block of n/p elements (Where, n is number of elements to be sorted and P is the number of processes).
- Let, Ai and Aj be the blocks stored in processes Pi and Pj respectively.
- Each process sends its block of size n/p to the other process. And this operation is called as **Compare-split operation.**
- Each process merges the received block with its own block and retains only the appropriate half of the merged block.
- Here process Pi retains the smaller elements and process Pj retains the larger elements.



6.2 Bubble Sort and its Variants

- Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

```
// Sequential bubble sort algorithm
begin
    for i:= n–1 down to 1 do
    for j:= 1 to i do
      compare_exchange (a j, aj+1);
end
```

- Time complexity is $O(n^2)$.
- To parallelize bubble sort is difficult as algorithm has no concurrency.
- Though, some variants of bubble sort uncover the concurrency.
- There are two variants of bubble sort that give parallelization.
  - Odd-Even Transposition
  - Shell sort

6.2.1 Odd-Even Transposition

Q. Explain odd-even transportation on bubble sort using parallel formulation. **SPPU - Dec. 18, May 19, 8 Marks**

- The odd-even transposition algorithm sorts n elements in n phases (n is even), each of which requires $n/2$ compare-exchange operations.
- This algorithm alternates between two phases, called the odd and even phases. Let $<a_1, a_2, ..., a_n>$ be the sequence to be sorted.
- During the odd phase, elements with odd indices are compared with their right neighbours, and if they are out of sequence they are exchanged; thus, the pairs (a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n) are compare-exchanged (assuming n is even).
- Similarly, during the even phase, elements with even indices are compared with their right neighbours, and if they are out of sequence they are exchanged; thus, the pairs (a_2, a_3), (a_4, a_5), ..., (a_{n-2}, a_{n-1}) are compare-exchanged.
- After n phases of odd-even exchanges, the sequence is sorted. Each phase of the algorithm (either odd or even) requires $Q(n)$ comparisons, and there are a total of n phases; thus, the sequential complexity is $Q(n^2)$.

Bubble sorting : Sequential algorithm of odd-even transposition

```
begin
    for ( i= 1; i<=n; i++)
    {
    if ( i%2==1) // odd iteration
```

```
        for (j=0; j<n/2-1;j++)
        {
          compare_exchange(A[2j+1],A[2j+2]);

          if (i%2==0) // even iteration

            for (j=1; j<n/2-1; j++)
          compare_exchange(A[2j],A[2j+1]);
      }
      }
end;
```

Ex. 6.2.1 : Sequence 3 2 3 8 5 6 4 1

Soln. :

Read all these elements into array A

i = 1 to 8

i →	①	②	③	④	⑤	⑥	⑦	⑧
	3	2	3	8	5	6	4	1

1. i = 1 (Odd)

So, execute 1st part of program

3	2	3	8	5	6	4	1

As, i = 1, algorithm form a pair with the immediate right element.

Here, A [1] and A [2] Forms the pair

A [3] and A [4] Forms the pair

A [5] and A [6]

A [7] and A [8]

- After forming pairs of group, Compare exchange method will execute.
- In A [1] = 3 and A [2]= 2, 2 is less than 3 so exchange the elements
- After 1st iteration elements of array are :

  2 3 3 8 5 6 1 4
- Here again between 4 and 1, 1 is less so swapped.

2. Now i = 2 (even)

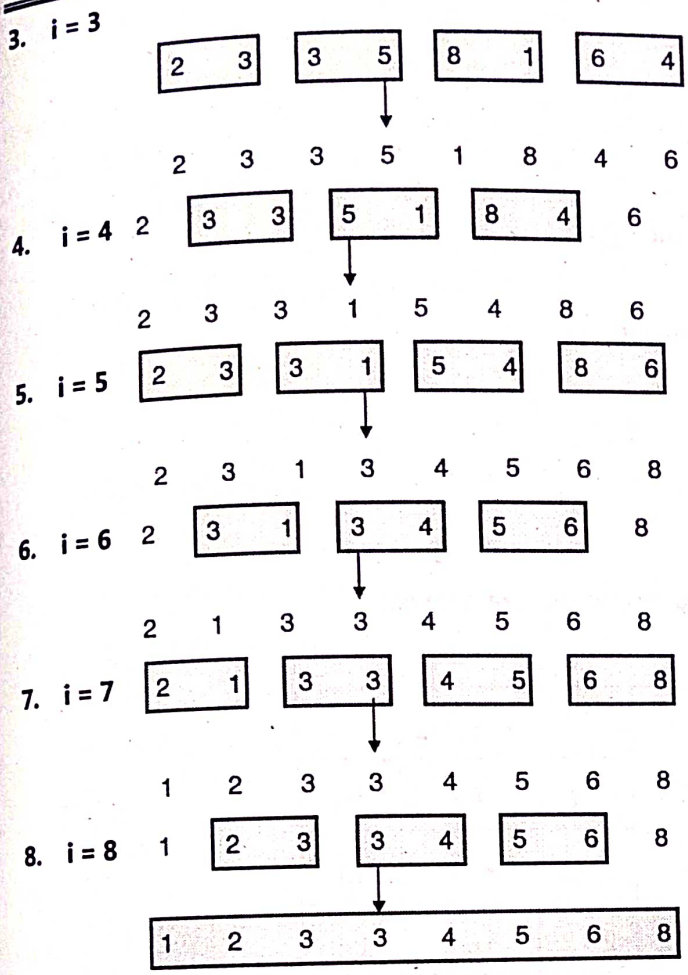
So, forms the pair with even and immediate right of even numbers.

→ 2 3 3 8 5 6 1 4

2	3	3	8	5	6	1	4

Compare and exchange

2 3 3 8 5 6 1 4



6.2.1(A) Parallel Odd-Even Algorithm

- Here different rules for carrying out odd and even iterations thus it is suitable for parallelizing.
- Drawback of sequential odd-even sort is a sequence which has a few elements out of order, still need $O(n^2)$ to sort.
- In parallel odd-even sorting, consider one item per processor. There are n iterations, in each iteration; each processor does one compare-exchange.
- The parallel runtime is O(n).

Parallel algorithm of odd-even transposition

```
begin
  id:= process's label
  for i:=1 to n do
  {
    if(i%2 == 1)    //odd iteration
    {
      if(id % 2 == 1) // odd process number
        compare_split_min(id+1);    // compare_exchange to the right
      else
        compare_split_max(id-1);    //compare_exchange to the left
```

```
    }

    if(i % 2 == 0)   //even iteration
    {
      if(id %2 == 0)   //even process number
         compare_split_min(id+1);    // compare_exchange to the right
      else
         compare_split_max(id-1);    //compare_exchange to the left
    }
  }
end;
```

For example

Odd - Even parallel transposition sort. We will take the sequence as per Example 6.2.1.

2 3 3 8 5 6 4 1

Time	Process		
	P_1	P_2	P_3
Start	2,3,3	8,5,6	4,1
After local sort	2,3,3	5,6,8	1,4
After phase 1 (odd phase) consider P_1 and P_2 and swap (Use algorithm)	2,3,3	5,6,8	1,4
After phase 2 (even phase) consider P_2 and P_3	2,3,3	1,4,5	6,8
After phase 3 (odd phase) consider P_1 and P_2	1,2,3	3,4,5	6,8

Note : To make parallel operation cost optimal always number of processes p should be less than n (number of elements) **p < n** and each process will hold or store **n/p** elements.

6.2.2 Shell Sort

- In previous algorithm, we have sorted elements with adjacent elements. But in shell sort, it compares elements that are distant apart rather than adjacent.
- Spacing between elements is known as Interval or gap.
- In every pass, gap is reduced by 1 till we reach last pass when gap is 1.
- Interval = floor (N/2), N is number of elements.

```
// Sequential algorithm of Shell sort

begin
  int incr = n/2;
  while( incr > 0 )
  {
    for ( int i=incr+1; i<n ; i++)
    j = i - incr;
    while(j > 0)
    {
      if( A[j] > A[j+incr] )
      {
        Swap( A[j] , A[j + incr ]);
        j = j - incr;
      }
      else j = 0;
    }
  incr = incr/2;
  }
end;
```

- Time complexity is $O(n \log_2 n)$

Ex. 6.2.2 : Consider sequence 15 19 20 38 24 41 30 31 12, sort this using shell sort.

Soln. :

Sequence 15 19 20 38 24 41 30 31 12

N = 9

gap (pass1) = floor (N/2)

= floor (9/2)

= floor (4.5)

= 4

0	1	2	3	4	5	6	7	8
15	19	20	38	24	41	30	31	12

1. As gap is 4, So we have to compare

   a [0] and a [4] → 15 < 24

   a [1] and a [5] → 19 < 41

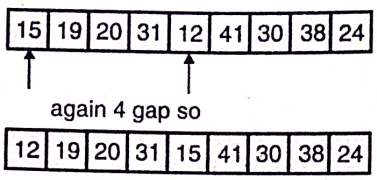
   a [2] and a [6] → 20 < 30

   a [3] and a [7] → 38 > 31

   a [4] and a [8] → 24 >1 2

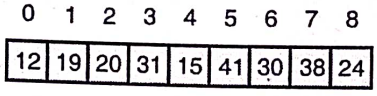
   } This need to swap

So, after computation



Now,

2. gap (pass2) = floor (gap/2)
   = 2

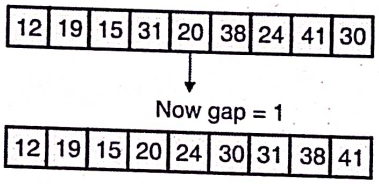


So now gap is of 2, compare

a [0] and a [2]

a [1] and a [3] and so on

Final sequence is,



6.2.2(A) Parallel Shell Sort Algorithm

- Let n be the number of elements to be sorted and p be the number of processes.
- During the first phase, processes that are far away from each other in the array compare-split their elements.
- During the second phase, the algorithm switches to an odd-even transposition sort.
- Parallel shell sort is basically a trick to make parallel odd-even transposition sort run faster. It is used to overcome the main limitation of odd-even transposition sort that is moves elements only one position at a time.

// Parallel shell sort algorithm

1. Initialize Data of size N elements and interval value h on P0
2. Broadcast data elements across P Processors, each holding (N/P) data
3. In Parallel perform Shell sort on P processor

   Each process performs log p compare-split operations. This can help the elements move closet to the final destination
4. Processes implements odd-even transposition. Fewer odd and even comparisons are expected.

Time complexity

Local sort First phase Second phase

$$T_P = \theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \theta\left(\frac{n}{p}\log p\right) + \theta\left(l\frac{n}{p}\right)$$

Here l refers to the number of odd-even sort comparisons.

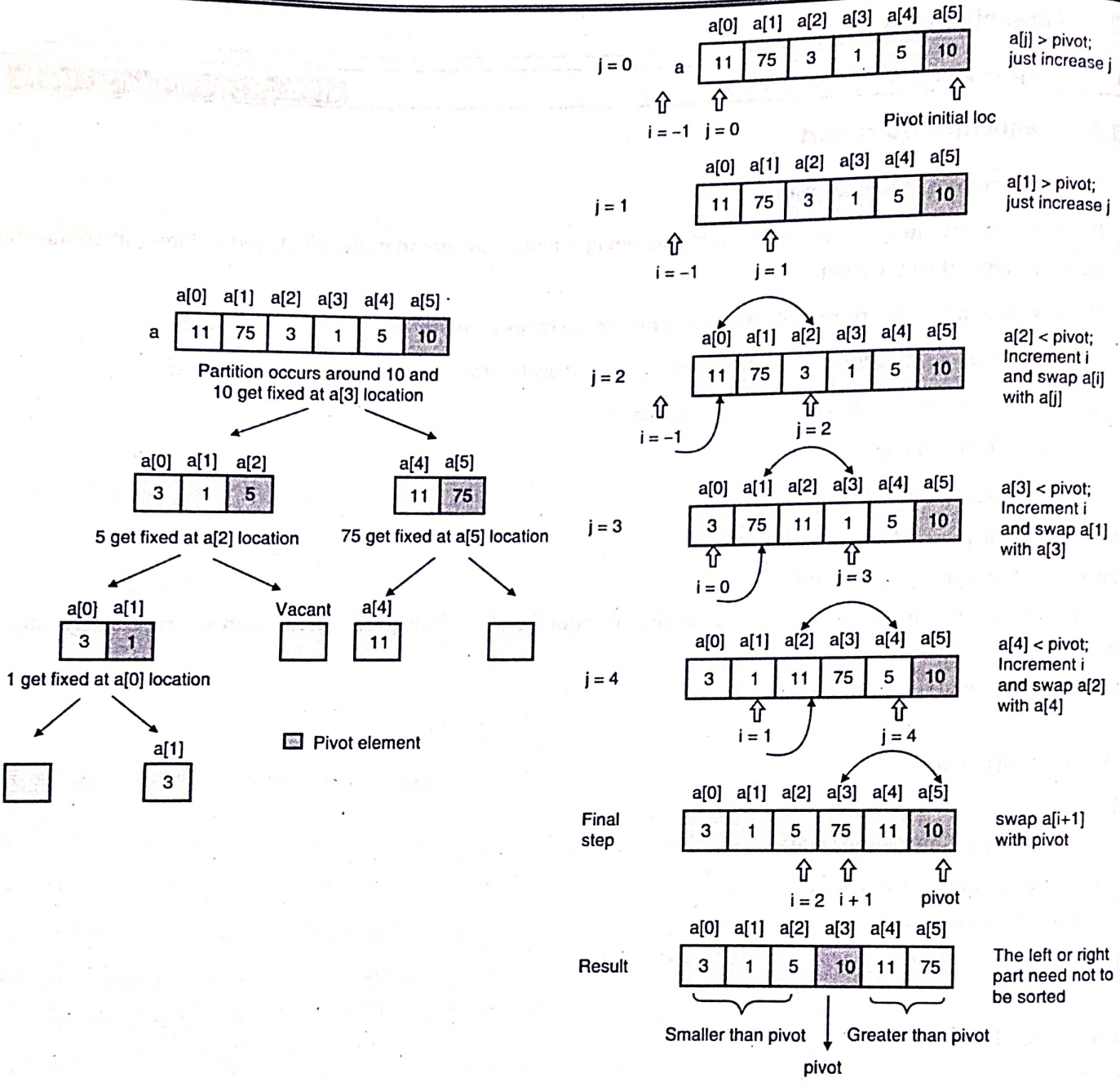
6.3 Parallelizing Quick sort

Q. Write a pseudo code for parallel Quick Sort. **SPPU- Oct. 18 (In Sem.), 7 Marks**

6.3.1 Sequential Quick sort

1. Select one of the numbers as pivot.
2. Divide the list into two sublists : a "low list" containing numbers smaller than the pivot, and a "high list" containing numbers larger than the pivot.
3. The low list and high list recursively repeat the procedure to sort themselves.
4. The final sorted result is the concatenation of the sorted low list, the pivot, and the sorted high list.
   (i) Find pivot that divides the array into two halves
   (ii) Quick sort Left half
   (iii) Quick sort Right half

```
//Sequential quick sort algorithm
//array is in the form a[start, ....,end]
/* divides the array into two parts based on the pivot location obtained from partition function and recursively calls them for sorting*/
quicksort(a,start,end)
{
  if(start < end) then
  {
    pivot_loc = partition(a,start,end)
    quicksort(a,start ,pivot_loc-1)
    quicksort(a, pivot_loc+1,end)
  }
}
partition (a,start,end)
{
 pivot = a[end]
 pivot_loc= start
 for i:= start to end-1
 {
   if( a[i] <= pivot)
   {
     swap( a[i],a[pivot_loc] )
     pivot_loc= pivot_loc + 1
   }
 }
}
```



Ex. 6.3.1 : Sequential quick sort Given sequences is {3 1 4 1 5 9 2 6 5 3 5 9}

Soln. :

Choose a number as pivot

N = 12

Pivot number is 3

Low list contains {3 1 1 2 3}

High list contains {4 5 9 6 5 5 9}

We have done partitioning of elements around the pivot value.

That means get all the elements that are less than or equal to the pivot and put them in low list.

And get all the elements that are greater than the pivot and put them on the other side i.e. in high list.

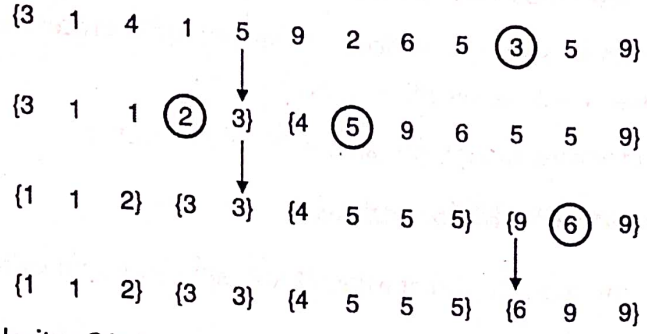
(1) For sublist {3 1 1 2 3}, choose pivot as 2.

{1 1 2} {3 3}

For sublist {4 5 9 6 5 5 9} choose 5 as pivot

{4 5 5 5} {9 6 9}

Just keep repeating the quick sort until you are done.



Average and worst case complexity : O(n logn)

6.3.2 Parallel Quicksort

- Each process holds a segment of the unsorted list. The unsorted list is evenly distributed among the processes.
- We randomly choose a pivot from one of the processes and broadcast it to every process.
- Each process divides its unsorted list into two lists: those smaller than (or equal) the pivot, those greater than the pivot.
- Each process in the upper half of the process list sends its "low list" to a partner process in the lower half of the process list and receives a "high list" in return.
- Now, the upper-half processes have only values greater than the pivot, and the lower-half processes have only values smaller than the pivot.
- Thereafter, the processes divide themselves into two groups and the algorithm recurses.

//Pseudo code: Parallel Quick sort algorithm

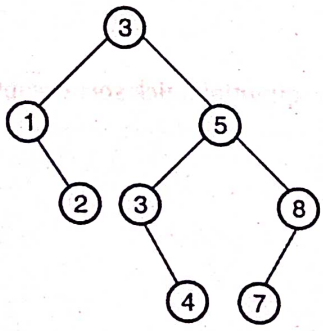
Pseudo code for parallel quick sort is same as sequential quick sort except the recursive sort function will be executed on different processors.

```
quicksort(a,start,end)
{
  if(start < end) then
  {
    pivot_loc = partition(a,start,end)
    quicksort(a,start ,pivot_loc-1)  }  This two functions will be executed concurrently on
    quicksort(a, pivot_loc+1,end)    }  different processors.
  }
}
```

- For example, take the same previous example, but recursive quick sort will be executed on different processors (A_L and A_R) . In example pivot 2 and pivot 5 list will be executed on two different processors instead of one.
- The main limitation of this parallel formulation is that it performs the partitioning step serially.
- And each of the subproblems is handled by a different processor.
- The time for this algorithm is lower-bounded by Ω(n).
- Can we parallelize the partitioning step – in particular, if we can use n processors to partition a list of length n around a pivot in O(1) time then it will be the efficient sorting.
- This is difficult to do on real machines, though we can use PRAM formulation.

6.3.3 Parallelizing Quicksort : PRAM Formulation

- We assume a CRCW (concurrent read, concurrent write) PRAM with concurrent writes resulting in an arbitrary write succeeding.
- The formulation works by creating pools of processors. Every processor is assigned to the same pool initially and has one element.
- Each processor attempts to write its element to a common location (for the pool).
- Each processor tries to read back the location. If the value read back is greater than the processor's value, it assigns itself to the 'left' pool, else, it assigns itself to the 'right' pool.
- Each pool performs this operation recursively.
- Note that the algorithm generates a tree of pivots. The depth of the tree is the expected parallel runtime. The average value is O(log n).
- A binary tree generated by the execution of the quicksort algorithm. Each level of the tree represents a different array-partitioning iteration.
- The average run time of this algorithm is O(log *n*) on an n-process PRAM. Thus, its overall process-time product is O(*n* log *n*), which is cost-optimal.



6.4 All-Pairs Shortest Paths

- It is used to find shortest paths between all pairs of vertices. To take each node in a graph and find out how far it is from all the other nodes in the graph.
- We have to find the shortest path between every pair of vertices in a directed graph G.
- The all pairs shortest path problem is to determine a matrix A such that A(i,j) is the length of shortest path from i to j.

- Given a directed graph G=(V,E), where each edge (v,w) has a non-negative cost c[v,w] , for all pairs of vertices (v,w) find the cost of the lowest cost path from v to w.
- It is a generalization of the single-source shortest path problem.
- Use Dijkstra's algorithm, varying the source node among all the nodes in the graph.
- Two approaches are there to solve the all-pairs shortest paths problem.
  1. Dijkstra's Algorithm
  2. Floyd's Algorithm

6.4.1 Dijkstra's Algorithm

Q. Explain Dijkstra's Shortest Path. SPPU - Dec. 18, 8 Marks

Q. Explain Dijkstra's Algorithm in parallel formulation. SPPU - May 19, 8 Marks

Dijkstra's Single Source Shortest Path (SSSP)

- Given the graph and the source, find the shortest path from source to all the nodes.
- Among vertices not already in the graph, it finds vertex u with the smallest sum.

$$d_v + w(v,u)$$

Where, v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree rooted at s).

dv is the length of the shortest path from source s to v

w(v,u) is the length (weight) of edge from v to u.

```
//Dijkstra's Single source shortest path algorithm
begin
 dist[s] ← 0  //distance to source vertex is 0
 for all v ∈ V- {s}
   do dist[v] ← ∞      //set all other distances to infinity
     S ← ø             // S , the set of visited vertices is initially empty
     Q ← V             // Q, the queue initially contains all vertices
   while Q ≠ ø         // while the queue is not empty
   do u ← mindistance(Q, dist)   // select the element of Q with the min distance
     S ← Su{ u}                  // add u to list of visited vertices
     for all v ∈ neighbors [u]
       do if dist[v] > dist[u] + w(u,v)  // if new shortest path found
          then  d[v] ← d[u]+w(u,v)  //set new value of shortest path
     return dist
end;
```

Here, if dist[v] > dist[u] + w(u,v)

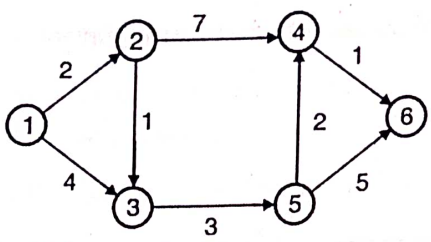
then d[v] ← d[u]+w(u,v) is called Relaxation.

We perform relaxation for the vertices. Whenever we select a shortest path we will try to relax other vertices.

For example

Dijkstra's algorithm : Single source shortest path

If a weighted graph is given then we have to find shortest path from some starting vertex to all of the other vertices.



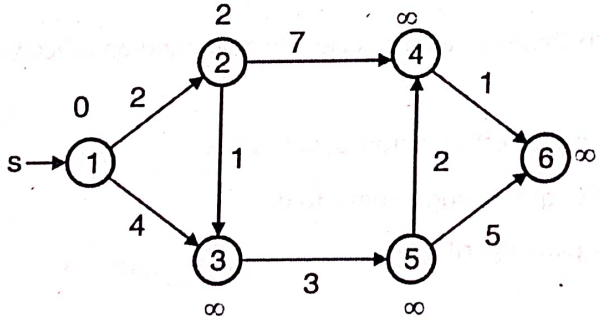
Suppose starting vertex is 1, then we have to find out shortest path to all the vertices, May be direct path or via other vertex.

We can select any one of the vertex as a source vertex.

As we have to find a shortest path so it's a minimization problem.

1. Select one as a starting vertex.

   Now give distance for all the vertices by considering just single edge.



   As there is no direct edge for 4,5 and 6 from starting vertex 1 so distance are ∞

2. Select the shortest distance from 1 among 2,4,∞,∞,∞

   So here 2 is the shortest distance and perform relaxation.

   Relaxation :

   if (d [u] + cost (u,v) < d [v]) then d [v] = d [u] + cost (u,v)

   Here 3 and 4 are connected to node 2

   Check d [2] + cost (2,3) ? d [3]

   d [2] + 1 ? 4

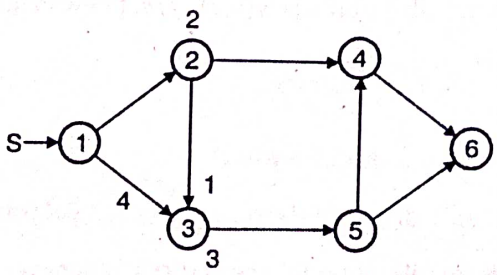
   2 + 1 4

   3 < 4

   Hence modify d [3]

   d [3] = d [2] + cost (2,3)

   = 2 + 1 = (3)



Now check with node 4

d [2] + cost (2,4) ? d [4]

$2 + 7 = < \infty$

$9 < \infty$

Modify d [6] = 11

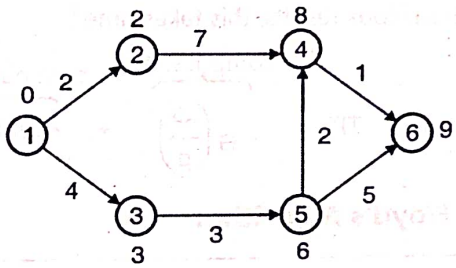
3. Out of node 4 and node 6 i.e. 8 and 11 value. Select node 4 with value 8.

D[4] + cost (4,6) ? d[6]

$8 + 1 < 11$

$9 < 11$

Modify d [6] = 9



Shortest path from 1 to all other vertices is

v	d [v]
2	2
3	3
4	8
5	6
6	9

i.e. 1 to 2 shortest path is 2

1 to 3 shortest path is 3

1 to 4 shortest path is 8

1 to 5 shortest path is 6

1 to 6 shortest path is 9

Dijkstra's All Pair Shortest Algorithm

- It executes n instances of the single-source shortest path problem, one for each of the source vertices.
- Time complexity is $O(n^3)$.

6.4.2 Parallel Dijkstra's Algorithm

Two parallelization strategies - execute each of the *n* shortest path problems on a different processor (source partitioned), or use a parallel formulation of the shortest path problem to increase concurrency (source parallel).

1. Dijkstra's Algorithm : Source Partitioned Formulation

- Use *n* processors, each processor P_i finds the shortest paths from vertex v_i to all other vertices by executing Dijkstra's sequential single-source shortest paths algorithm.

- It requires no interprocess communication (provided that the adjacency matrix is replicated at all processes).
- The parallel run time of this formulation is : $\Theta(n^2)$.

2. Dijkstra's Algorithm : Source Parallel Formulation

- In this case, each of the shortest path problems is further executed in parallel. We can therefore use up to n^2 processors.
- Given p processors ($p > n$), each single source shortest path problem is executed by p/n processors.

Using previous results, this takes time :

$$TP = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n \log P)}^{\text{communication}}$$

6.4.3 Floyd's Algorithm

Q. Explain Floyd algorithm in details. **SPPU - Oct. 18 (In Sem.), 8 Marks**

- Floyd's algorithm finds the shortest path from every vertex in a graph to every other vertex in the graph, if such paths exist. This algorithm requires that there be no negative cycles in the graph, however, negative edges may be included.
- We will make use of the adjacency matrix to implement Floyd's algorithm.

Adjacency matrix

- The adjacency matrix can be represented as follows :
  - $A_{ij} = 0$, if i == j
  - $A_{ij} = w_{ij}$ if an edge exists between vertices i and j
  - $A_{ij} = \infty$, if no edge exists between vertices i and j

Algorithm

1. The shortest path from vertex i to vertex j is either:
   - The length of the path "as is": A_{ij} or
   - The length of the paths from i to k and from k to j including the intermediate point k, so its $A_{ik} + A_{kj}$ (k is a intermediate vertex between i and j).
   - Thus at any given time step, $A_{ij}^{(t)}$ can be defined as,

$$A_{ij}^{(t)} = \min(A_{ij}^{(t-1)}, A_{ik}^{(t-1)} + A_{kj}^{(t-1)})$$

Ak [i, j] = min { Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j] }

Input: n – number of vertices

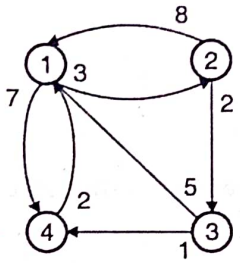
Adjacency matrix

Output: Transformed A that contains the shortest path lengths

```
begin
 for k:=1 to n do
```

```
{
for i :=1 to n do                    /* NOTE : For parallel implementation , the two
{                                       inner most loops can be executed in
 for j :=1 to n do                      parallel */
 {
  A[i,j] = min ( A[i,j], A[i,k]+A[k,j]);
 }
}
}
End
```

Floyd's algorithm example



$$A^{\circ}\text{ Original matrix} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array} \text{ Adjacency matrix}$$

No edge = ∞

Self loop = 0

To find the shortest path

1. Consider intermediate vertex as '1' so is there any shortest path (better) going via vertex 1.

So when we are saying via 1, then all the paths that belongs to vertex 1 will remain unchanged so we should note calculate them, directly take it.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{array}$$

$$\downarrow$$

$$A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 3 & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix}$$

Formula

$AK[i, j] = \min$

$\{AK-1[i, j], A^{K-1}[i, k] + A^{K-1}[k, j]\}$

Now consider path from 2 to 3 via 1

$A^{\circ}[2,3]$ $A^{\circ}[2,1] + A^{\circ}[1,3]$

$(2) < 8 + \infty$

So there is no better path via vertex

1. $(2 \rightarrow 1 \rightarrow 3$ – Not better)

$$A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ (8) & 0 & 3 & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix}$$

We will go with old path only i.e. 3

Consider path 2 and 4 via 1 and check whether it is shortest or not.

$A^{\circ}[2,4]$ $A^{\circ}[2,1] + A^{\circ}[1,4]$

∞ $8 + 7$

$\infty > 15$

So, now we have to update original path of ∞ to 15.

$2 \rightarrow 1 \rightarrow 4 \rightarrow (15)$

$$A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 3 & (15) \\ 5 & & & \\ 2 & & & \end{bmatrix}$$

Same procedure we will find (3 to 2)

(3, 4), (4, 2), (4, 3)

$A^{\circ}[3,2]$ $A^{\circ}[3,1] + A^{\circ}[1,2]$

∞ $5 + 3$

$\infty > (8)$

So Final,

$$A^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 3 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & \infty & 0 \end{bmatrix}$$

2. Consider intermediate vertex as '2'

$A^2 =$

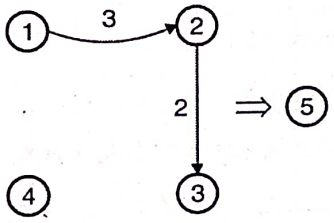
	1	2	3	4
1	0	3	5	
2	8	0	2	15
3		8		
4		8		

$A^1[1,3]$? $A^1[1,2] + A^1[2,3]$

$\infty > 3 + 2$

$\infty > 5$

Replace ∞ with 5 i.e. ($1 \rightarrow 3$ via 2 is 5)



Same,

$A^1[1,4]$ $A^1[1,2] + A^1[2,4]$

$7 < 3 + 15$

$7 < 18$

So, take 7 as shortest path from 1 to 4

$$A^2 = \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

3. $$A^3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ & & 5 & \\ & & 2 & \\ 5 & 8 & 0 & 1 \\ & & 7 & \end{bmatrix}$$

↓ Consider intermediate vertex as 3

$A^2[1,2]$ $A^2[1,3] + A^2[3,2]$

$3 < 5 + 8$

So $$A^3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

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

↓ Consider 4 as intermediate vertex

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

From the last matrix we got a shortest path between all pair of vertices.

- We have taken sequence of decisions at each stage, we were getting a matrix.
- Floyd's algorithm is an exhaustive and incremental approach.
- The entries of the A-matrix are updated n rounds.
- A[i, j] is compared with all n possibilities, that is, against A[i, k] + A[k, j], for $0 \le k \le n-1$
- n^3 of comparisons in total

6.4.4 Parallelization of Flyod's Algorithm

- The basic idea to parallelize the algorithm is to partition the matrix and split the computation between the processes.
- Each process is assigned to a specific part of the matrix.
- A common way to achieve this is **1-D** or **2-D Block Mapping**. Here the matrix is partitioned into squares of the same size and each square gets assigned to a process.
- Obviously, the two innermost loops of above sequential algorithm can be executed in parallel. For computation, each process needs some elements from the k^{th} row and column in the matrix $A^{k-1}[i,j]$.
- Let, *p* be the number of processes available. Matrix $A^k[i,j]$ is partitioned into *p* parts, and each part is assigned to a process. Each process computes the $A^k[i,j]$ values of its partition. To accomplish this, a process must access the corresponding segments of the k^{th} row and column of matrix $A^{k-1}[i,j]$.

Parallel formulation using 2-D block mapping

Pi, j is the process that is assigned to the square in i^{th} row and j^{th} column.

```
begin
for k:= 1 to n do
begin
  each process Pi, j that has a segment of the kth row of Ak-1;
     broadcasts it to the P*, j processes;( broadcast A[k, j] to A[0,j], A[1,j],.....A[n-1,j])
  each process Pi, j that has a segment of the kth column of Ak-1;
     broadcasts it to the Pi, * processes;( broadcast A[i, k] to A[i,0], A[i,1],.....A[i,n-1])
```

each process waits to receive the needed segments;

each process Pi, j computes its part of the Ak matrix ;

end

end

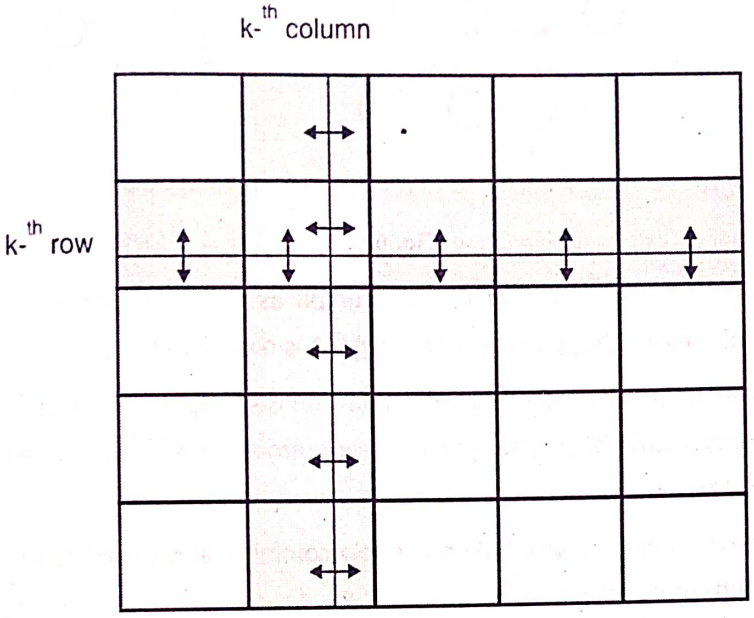


Fig. 6.4.1

How above algorithms works ?

- $P_{i,j}$ which has elements of the k^{th} row after the $k\text{-}1^{th}$ iteration sends the part of the A^{k-1} matrix to processors $P_{i,j+1}$, $P_{i,j-1}$;
- Similarly, $P_{i,j}$ which has elements of the k^{th} column after the $k\text{-}1^{th}$ iteration sends the part of the A^{k-1} matrix to processors $P_{i-1,j}$, $P_{i+1,j}$;
- Each of these values are stored and forwarded to the all the processors in the same row and column.
- The forwarding is stopped when mesh boundaries are reached.
- For n iterations the computation time is $O(n^3 / P)$.

6.5 Algorithm for Sparse Graph

In previous sections we have learned parallel algorithms for dense graphs. However, we have yet to address parallel algorithms for sparse graphs.

Sparse Graph

A graph is called sparse if the number of edges is much smaller than the square of the number of vertices. Otherwise it is called dense.

- For the dense graph, use the array representation.
- For a sparse graph, use list representation. For each $v \in V$, store the adjacency list

$Adj[v] = \{u \in V | (v, u) \in E\}$.

For examples

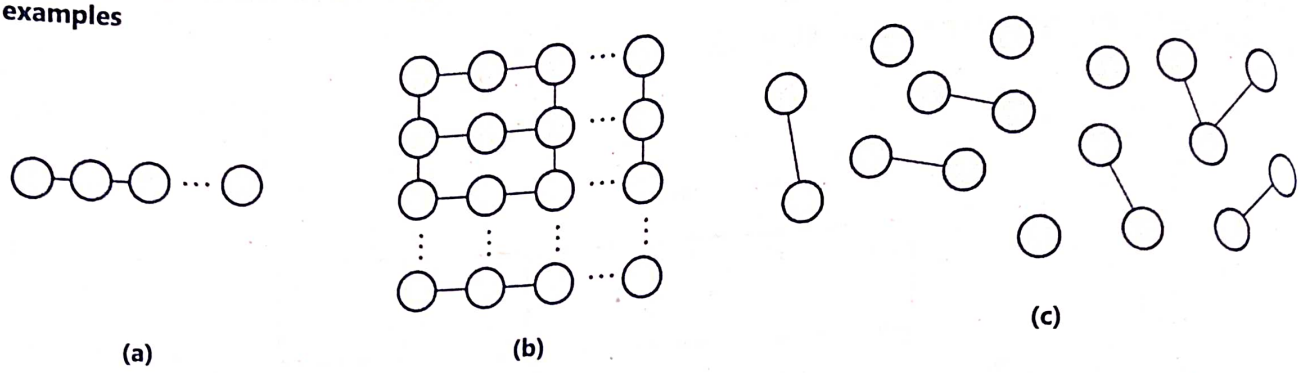


Fig. 6.5.1

- Any dense-graph algorithm works correctly on sparse graph as well. However, it is usually possible to obtain significantly better performance if the sparseness of the graph is considered.
- In the parallel formulation of sequential algorithms for dense graphs, we obtained good performance by partitioning the adjacency matrix of a graph so that each process performed an equal amount of work and communication was localized.
- But it is difficult to achieve even work distribution and low communication overhead for sparse graphs as we are representing sparse graphs using adjacency list.
- Thus it is hard to derive efficient parallel formulations for sparse graphs. But we can often derive efficient parallel algorithms if the sparse graph has a certain structure.
- The different types of sparse graphs for which efficient parallel

6.5.1 Johnson's Algorithm

It is easy to modify Dijkstra's single-source shortest paths algorithm so that it finds the shortest paths for sparse graphs efficiently. The modified algorithm is known as Johnson's algorithm.

- Dijkstra's algorithm, modified to handle sparse graphs is called Johnson's algorithm.
- The modification accounts for the fact that the minimization step in Dijkstra's algorithm needs to be performed only for those nodes adjacent to the previously selected nodes.
- Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in a sparse, edge-weighted, directed graph.
- Johnson has defined an algorithm that takes advantage of the sparse nature of most graphs, with directed edges and no cycles.
- It bases on Bellman ford algorithm, reweighting method and Dijkstra's algorithm.
- It allows some of the edge weights to be negative numbers. It works by using the bellman-ford algorithm to a compute a transformation of the input graph that removes all negative weights (has to reweight edges for Dijkstra's to be used) allowing Dijkstra's algorithm to be used on the transformed graph.
- Bellman-ford is called to calculate the reweighting. The Dijkstra's is called repeatedly using the modified graph.
- **Reweight : reweight edge weights to eliminate negative weight edges and serve shortest path.**

- Original edge weight is w(u,v)

  New edge weight :

  $w'(u,v) = w(u,v) + h(u) - h(v)$

  h(v) is a function mapping vertices to real numbers.

- Complexity $O(n^2 \log n + ne)$ where n is set of all the vertex and e is the set of all the edges.

Algorithm Steps

The Johnson's algorithm includes three main steps.

1. In the first step, an extra node, called s, is added to the graph with zero weighted edges to all other existing nodes. Then, the shortest path from s to all other nodes is calculated by the modified version of Bellman-Ford algorithm.
2. In the second step, the costs of the edges are re-weighted by using the costs of the shortest path calculated in the first step.
3. Then, in the last stage, the extra node is removed from the reweighted graph and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the reweighted graph.

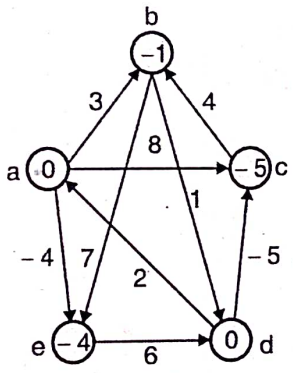
Before discussing pseudocode and parallelization, let's see one example.

Johnsons Algorithm Example

G = |VE|

|v| = 5

|E| = 9

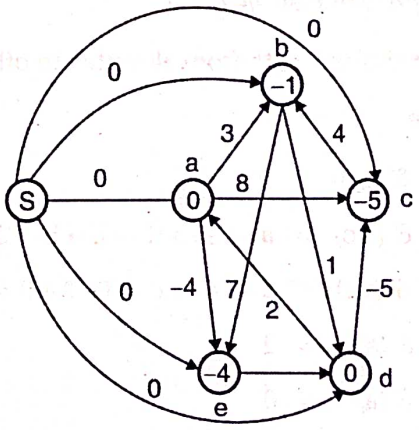


First step is add new vertex in the graph, such a that weight of the each path should be zero.

Now graph is G' = |V' E'|

|V'| = 6

|E'| = 14



Now find the shortest path from s to all the vertices,

h(a) = δ (S,a) = (0)

h(b) = δ (s,b)

= s →d→c→b

= 0 – 5 + 4 = (–1) (shortest path)

h(c) = δ (s,c) = s → d → c = 0 – 5 = (–5)

h(d) = δ (s,d) = s → d = (0)

h(e) = δ (s,e) = s → a → e = 0 – 4 = (–4)

So, $h(a) = 0$

$h(b) = -1$

$h(c) = -5$

$h(e) = -4$

Reweight

$$W'(u,v) = w(u,v) + h(u) - h(v)$$

So for the given graph,

$W'(a,b) = w(a,b) + h(a) - h(b)$

$= 3 + 0 - (-1) = 4$

$W'(a,c) = w(a,c) + h(a) - h(c)$

$= 8 + 0 - (-5) = 13$

$W'(a,e) = w(a,e) + h(a) - h(e) = -4 + 0 - (-4) = 0$

$W'(b,d) = 0$

$W'(b,e) = 10$

$W'(d,c) = 0$

$W'(d,a) = 2W'(c,b) = 0$

$W'(e,d) = 2$

$W'(s,a) = 0$

$W'(s,b) = 1$

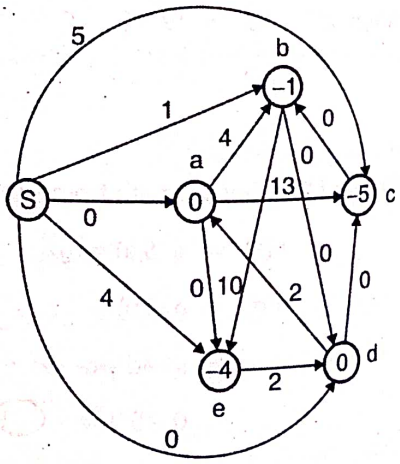
$W'(s,c) = 5$

$W'(s,d) = 0$

$W'(s,e) = 4$

New graph with new weight is,

Now find shortest path from all vertex to other vertex.



For vertex 'a'

$\delta'(a,a) = 0$

$\delta'(a,b) = a \rightarrow e \rightarrow d \rightarrow c \rightarrow b = 0 + 2 + 0 + 0 = 2$

$\delta'(a,c) = a \rightarrow e \rightarrow c \rightarrow 0 + 2 + 0 = 2$

$\delta'(a,d) = 2$

$\delta'(a,e) = 0$

To find 'δ'

$$\delta(u,v) = \delta'(u,v) - h(u) + h(v)$$

$\delta(a,a) = \delta'(a,a) - h(a) + h(a)$

$= 0 - 0 + 0 = \textcircled{0}$

$\delta(a,b) = \delta'(a,b) - h(a) + h(b)$

$= 2 - 0 + (-1) = \textcircled{1}$

$\delta(a,c) = \delta'(a,c) - h(a) + h(c)$

$= 2 - 0 + (-5) = \textcircled{-3}$

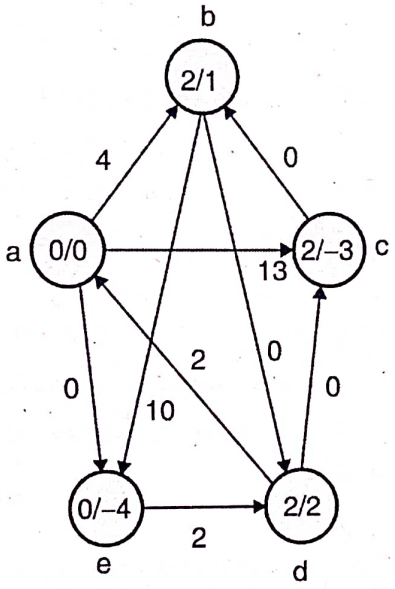
$\delta(a,d) = \delta'(a,d) - h(a) + h(d)$

$= 2 - 0 + 0 = \textcircled{2}$

$\delta(a,e) = \delta'(a,e) - h(a) + h(e)$

$= 0 - 0 + (-4) = -4$

Now, consider these values in the graph.



Now shortest path from,

$\delta(a,b) = 1$

$\delta(a,c) = -3$

$\delta(a,d) = 2$

$\delta(a,e) = -4$

Some way we will find the shortest path from b to all other vertex.

$\delta'(b,b) = 0$

$\delta'(b,a) = 2$

$\delta'(b,c) = 0$

$\delta'(b,d) = 0$

$\delta'(b,e) = 2$

Find δ :

$\delta(b,b) = \delta'(b,b) - h(b) + h(b) = 0 - (-1) + 1 = 0$

$\delta(b,a) = \delta'(b,a) - h(b) + h(a) = 2 - (-1) + 0 = 3$

δ (b,c) = -4

δ (b,d) = 1

δ (b,e) = -1

Shortest path from b to all other vertex

δ (b,b) = 0/0 = (old value/new value) = (δ'/δ)

δ (b,a) = 2/3

δ (b,c) = $0/-4$

δ (b,d) = $0/1$

δ (b,e) = $2/-1$

Same for vertex 'C'

δ' (c,c) = 0	δ (c,c) = 0
δ' (c,a) = 2	δ (c,a) = 7
δ' (c,b) = 0	δ (c,b) = 4
δ' (c,d) = 0	δ (c,d) = 5
δ' (c,e) = 2	δ (c,e) = 3

Same for vertex 'd'

δ' (d,d) = 0	δ (d,d) = 0
δ' (d,a) = 2	δ (d,a) = 2
δ' (d,b) = 0	δ (d,b) = -1
δ' (d,c) = 0	δ (d,c) = -5
δ' (d,e) = 2	δ (d,e) = -2

Same for vertex 'e'

δ' (e,e) = 0	δ (e,e) = 0
δ' (e,a) = 4	δ (e,a) = 8
δ' (e,b) = 2	δ (e,b) = 5
δ' (e,c) = 2	δ (e,c) = 1
δ' (e,d) = 2	δ (e,d) = 6

// Johnson algorithm (1-to-all shortest path)

1. Create G' where G'.V = G.V $\cup$ {s},
   G'.E = G. E $\cup$ { (s, v) : v $\in$ G,V } and
   weight(s, v) = 0 for all v $\in$ G.V

2. for each vertex v ∈ G'.V
   set h(v) to the value of δ (s, v) // h(v)=distance(s, v) computed by Bellman-ford
   // computed by the Bellman-Ford algorithm

   for each edge (u, v) ∈ G'.E
      weight'(u, v) = weight(u, v) + h(u) – h(v)

3. New matrix D = (duv) initialized to infinity
   for each vertex u∈ G.V
   run Dijkstra(G, weight', u) to compute δ'(u, v) for all v ∈ G.V
   for each vertex v ∈ G.V
   duv = δ'(u, v) + h(v)- h(u)
   return D.

The adjacency list representation is used to store the graph. With adjacency lists, a graph G(E,V) is an array A of size n = |V| such that, for each vertex vi ∈ V, A[i] is a pointer to a linked list of nodes (v_1, v_2, . . . , v_k) such that (A[i],v_j) is an edge in E for each j = 1, 2, . . . , k. Vertices of graph are represented as array, say Va; another array of adjacency list stores the edges with edges of vertex i+1 immediately following the edges of vertex i for all i in V.

Johnson's algorithm uses a priority queue Q to store the value *l*[*v*](path cost of each vertex) for each vertex *v* (*V* – *Vt*) where V_t ={s} (the extra node). The priority queue is constructed so that the vertex with the smallest value in *l* is always at the front of the queue. A common way to implement a priority queue is as a binary min-heap. A binary min-heap allows us to update the value *l*[*v*] for each vertex *v* in time *O* (log *n*).

6.5.2 Parallel Johnson's Algorithm

- An efficient parallel formulation of Johnson's algorithm must maintain the priority queue Q efficiently.
- A simple strategy is for a single process. For example, *P*0, to maintain Q. All other processes will then compute new values of *l*[*v*] for *v* (*V* – V_T), and give them to *P*0 to update the priority queue.
- There are two main limitation of this scheme.
  - First, because a single process is responsible for maintaining the priority queue.
  - Second, during each iteration, the algorithm updates roughly |*E*|/|*V*| vertices. As a result, no more than |*E*|/|*V*| processes can be kept busy at any given time, which is very small for most of the interesting classes of sparse graphs.
- These limitations can be overcome by distributing the maintenance of the **priority queue to multiple processes.**
- We use multiple queues, one for each processor. Each processor builds its priority queue only using its own vertices.
- When process Pi extracts the vertex u ∈ Vi, it sends a message to processes that store vertices adjacent to u.
- Process Pj , upon receiving this message, sets the value of l[v] stored in its priority queue to min{l[v],l[u] + w(u,v)}.
- If a shorter path has been discovered to node v, it is reinserted back into the local priority queue.
- The algorithm terminates only when all the queues become empty.

6.6 Parallel Depth-First Search

Q. Describe Parallel DFS or BFS in details?	SPPU - Oct. 18 (In Sem.), 7 Marks
Q. Write a short note on Parallel-Depth-First-Search.	SPPU - Oct. 18 (In Sem.), 7 Marks
Q. Modify Depth First Search for parallel execution and analyze its complexity.	SPPU - Dec. 18, 8 Marks
Q. Explain parallelism in best first search algorithm. Give an appropriate example.	SPPU - Dec. 18, 8 Marks
Q. Explain parallel First Search for solving 8 puzzle problem.	SPPU - May 19, 8 Marks

- DFS is a general technique used in Artificial Intelligence for solving a variety of problems in planning, decision making, theorem proving, expert system etc.
- A major advantage of DFS strategy is that it requires very little memory.
- Depth-first search can be performed in parallel by partitioning the search space into many small, disjunct parts (subtrees) that can be explored concurrently.
- It uses idea of backtracking.

6.6.1 Parallel Depth-Search Algorithm

The critical issue in parallel depth-search algorithm is the distribution of the search space among the processors.

We need dynamic load balancing between processors to implement parallel DFS.

Dynamic load balancing : when a processor runs out of work it gets more work from another processor that has work.

A parallel formulation of DFS based on dynamic load balancing is as follows

- Each processor performs DFS on a disjoint part of the search space.
- When a processor finishes searching its part of the search space, it requests an unsearched part from other processors.
- Whenever a processor finds a goal node, all the processors terminate.
- If the search space is finite and the problem has no solutions, then all the processors eventually run out of work, and the algorithm terminates.

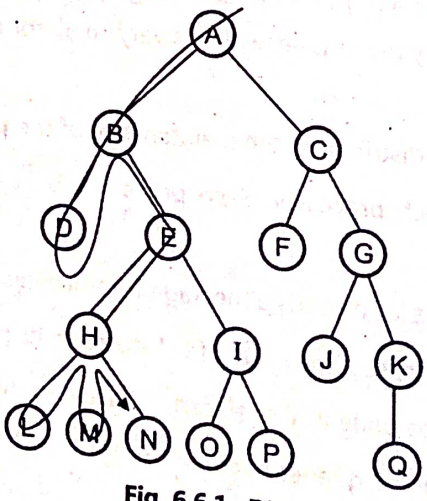


Fig. 6.6.1 : DFS

At the beginning only one processor contains the whole state space. The space is distributed as the processors requests work. : **Donor and recipient processors.**

Problems :

- How to split the work? => work splitting strategies
- How to determine the donor? => load-balancing schemes

Dynamic load balancing scheme :

Important parameters of parallel DFS :

- Splitting up the work
  - How much work should you give to another processor?
- Determining a donor processor
  - Who do you request more work from?

Work-splitting strategy

- Donor's stack is split into two stacks and one stack is sent to the recipient.
  - If two little work is sent => recipient quickly becomes idle
  - If two much work is sent => donor quickly becomes idle
  - Ideal: split the stack into equal parts => half-split
- Nodes at the bottom of the stack are the roots of bigger subtrees
- Nodes at the top of the stack are the roots of smaller subtrees
- Nodes beyond a cutoff depth are not sent => Avoids sending to little work

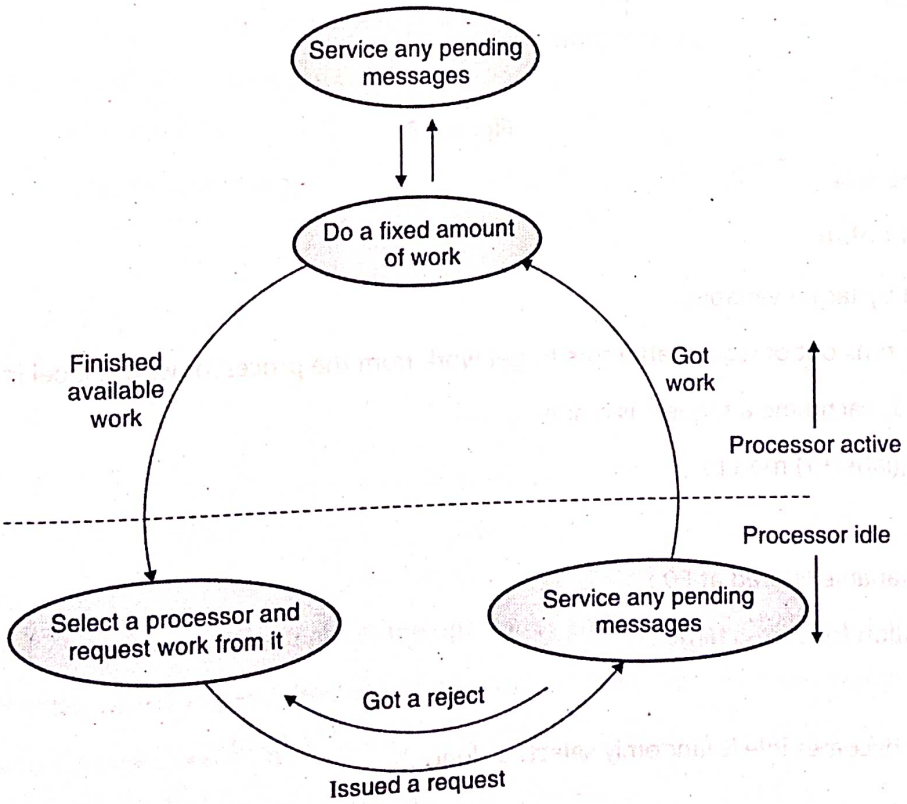


Fig. 6.6.2

Strategies

1. Send nodes near the bottom of the stack.
2. Send nodes near the cutoff depth.
3. Send half the nodes between the bottom of the stack and the cutoff depth.

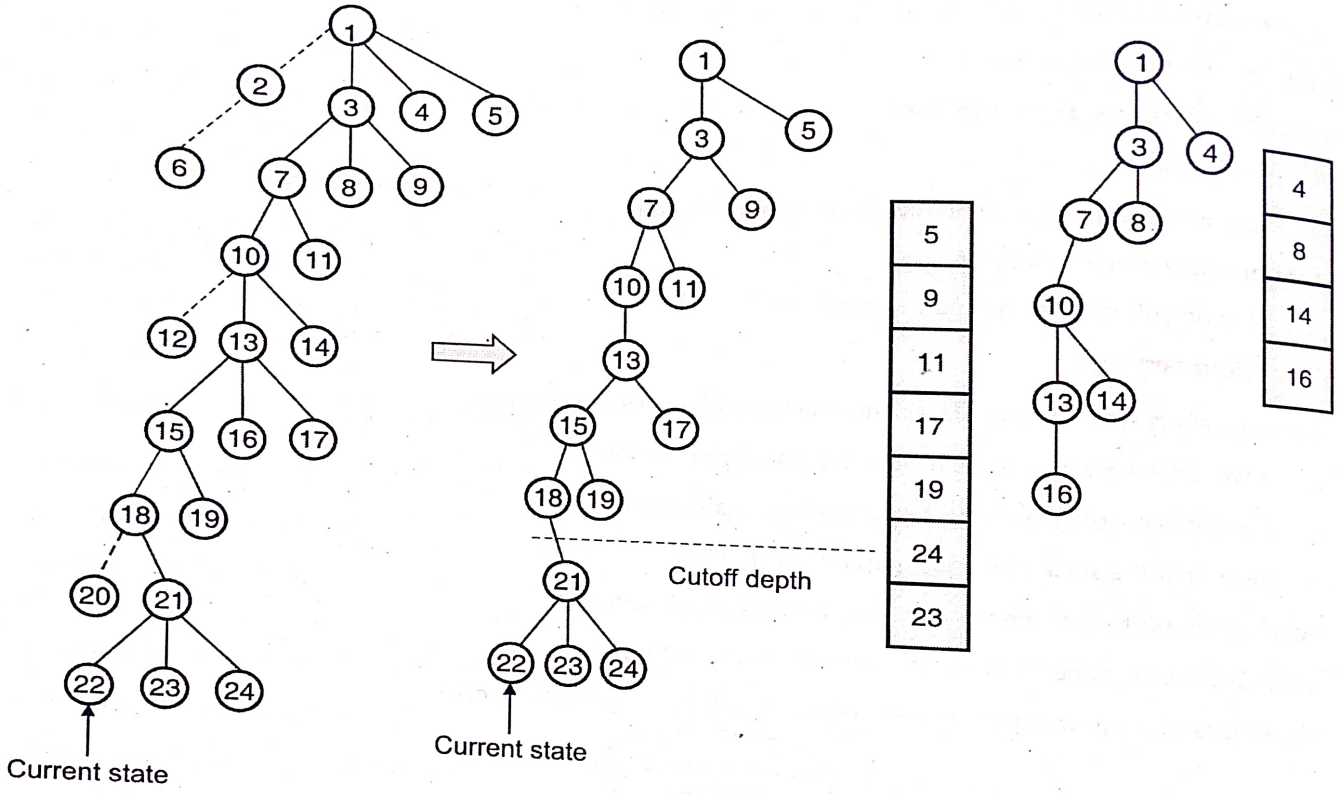


Fig. 6.6.3

Load –balancing schemes

Asynchronous Round Robin

- Donor is specified by target variable.
- When a processor runs out of work it attempts to get work from the processor whose label is target.
- Target = target + 1, each time a request is made.
- Initially : target = ((label+1) mod p)

Global Round Robin

- target is a global variable (stored at P0)
- Drawback- contention for access target.

Random Polling

- When a processor becomes idle it randomly selects a donor.

6.7 Parallel Best First Search

Q. Write a short note on Parallel-Best-First-Search. **SPPU - Oct. 18 (In Sem.), 7 Marks**

- It is graph search algorithm that begins at the root vertex and explores all the connected vertices, traversing all vertices of a particular level before traversing of the next level.
- At the end of the BFS we can find out the level of a vertex if it is connected to the root element and also its predecessor.

  Node should be traversed as :
  - o First move horizontally and visit all the nodes.
  - o Move to the next layer.
- It is useful in logistics and supply chains, e-commerce, social media, fraud detection etc.

  Why BFS?
  - o Least work of the graph algorithm
  - o Building blocks for many other graph problems
- The best-first search method is commonly employed in searching a dynamic search space with heuristic information which allows the method to locate the goal state or solution by generating only a small part of the search space.
- This search is efficient because it always considers the most promising node or state for expansion or exploitation.
- In this approach of search we have some knowledge about problem.
- In BFS, heuristic values are used which is associated with each and every node and with the help of the heuristic values we will find the path to reach the goal node.
- The important component of BFS is the open list and the closed list. Open list maintains the unexpanded nodes in the search graph according to their cost value.
- Then the most promising node from the open list is removed and expanded and newly generated nodes are added to the open list.

For example :

Goal state is I.

1. open = [ ]

   Closed = [ ]
2. open = [s]

   Closed = [ ]

Here we start with the root node, S and will put in the open array and will check whether s is the goal state or not.

As 's' is not the goal state, we will expand the children of s and will place it in the open array.

Children of s will place it in the open array.

Children of s are C, B and A

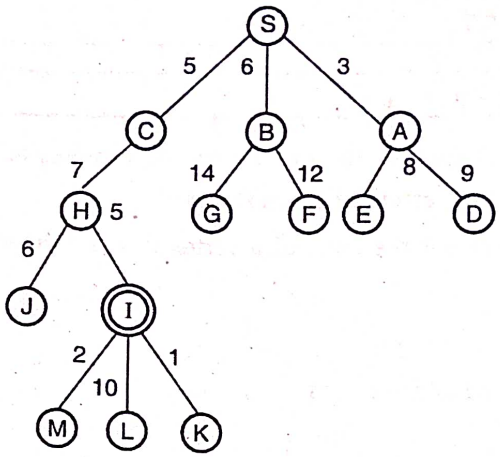


Fig. 6.7.1

Note : While putting this children into open array, we have to sort this 's' children's based on the heuristic value in ascending order.

Here A with heuristic value 3 will come first and next is c with value s and last is B with 6 value.

3. open = [A3, C5, B6]

   closed = [S]

Here consider A, as it is not the goal state. So will placed in closed array and expand A.

Children of A are E 8 and D9

4. open = [C5, B6, E8, D9]

   closed = [S, A3]

The same procedure will follow till goal state.

5. open = [B6, H7, E8, D9]

   closed = [S, A3, C5]

6. open = [H7, E8, D9, F12, G14]

   closed = [S, A3, C5, B6]

7. open = [I5, J6, E8, D9, F12, G14]

   closed = [S, A3, C5, B6, H7]

As I is the goal state, will place it closed.

8. open = [J6, E8, D9, F12, G14]

   closed = [S, A3, C5, B6, H7,I5]

So the final best path to reach goal I is **S, A, C, B, H, I**

6.7.1 Parallel Best First Search Algorithm

- In parallel formulation of BFS, the different nodes from the open list are expanded concurrently by different processors.

- Given P processors, the simplest parallel strategy is to let each parallel processor work on one of the current best nodes in the Open list.
- This is called centralized strategy because each processor gets work from a single global open list.
- Fig. 6.7.2 shows a parallel best first search using a centralized strategy.

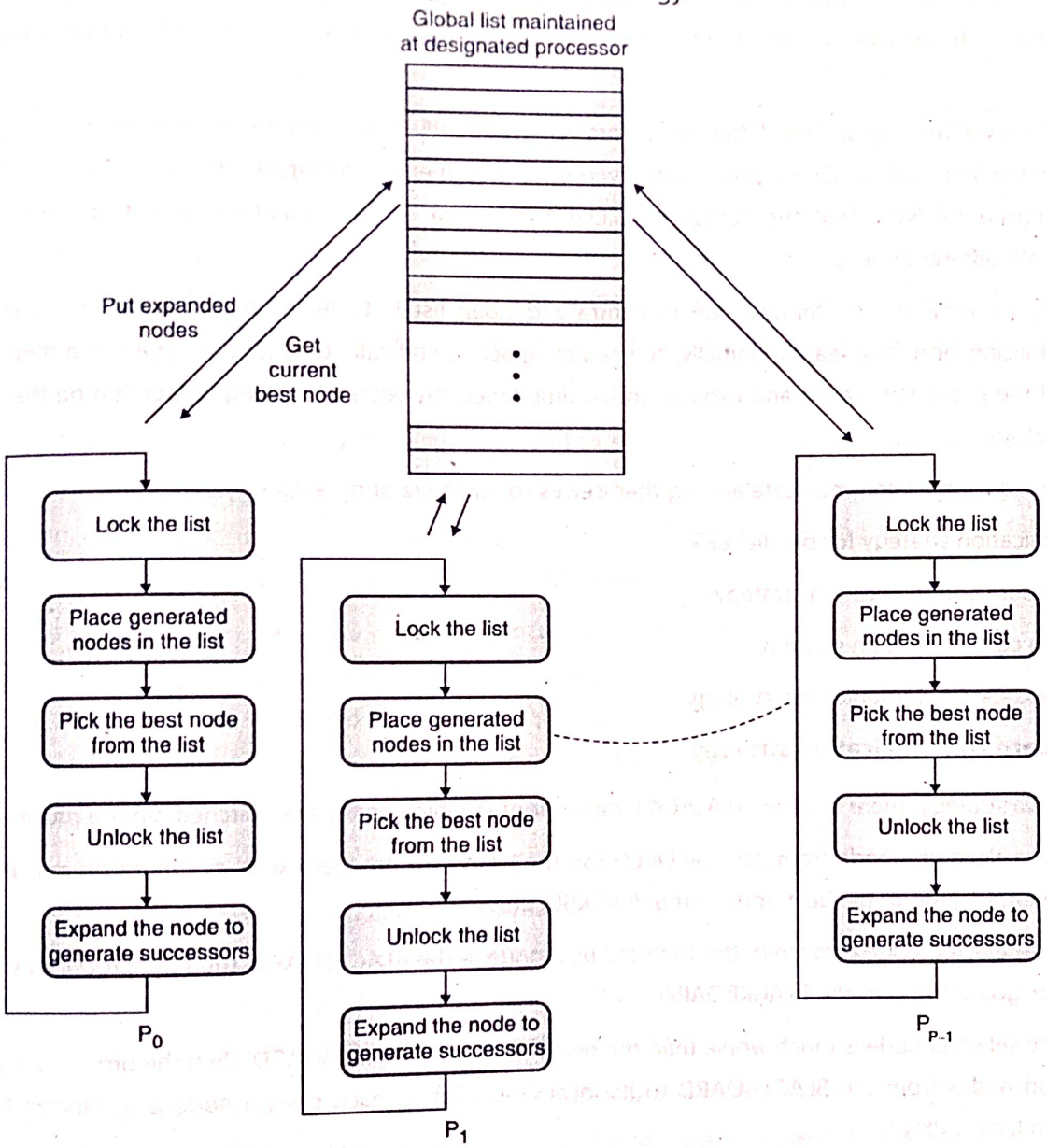


Fig. 6.7.2

- In a parallel formulation of Best first search, each processor searches a disjoint part of the search space in **BFS fashion.**
- When a processor has finished searching its part of the search space, it tries to get an unsearched part of the search space from other processors.
- When a goal node is found, all the processors quit.
- If the search space is finite and the problem has no solutions, then all the processors eventually run out of work, and the algorithm terminates.
- A centralized lead to congestion as open list is locked when accessed.

Problems with Centralized Strategy

Termination Criterion

- The termination criterion of sequential best first search does not work anymore; i.e., if a current processor picks up a goal node m for expansion, then the node m is no longer guaranteed to be the best goal node. But the termination criterion can be easily modified to ensure that termination occurs only after a best solution has been found.
- Since OPEN will be accessed by all the processors very frequently, it will have to be maintained in a shared memory that is easily accessible to all the processors. Even on shared memory architectures, contention for OPEN list limits the performance. Note that the access to CLOSED does not cause contention, as different processors would manipulate different nodes.
- One way to avoid the contention due to centralized open list is to let each processor have a local open list (decentralizing best first search). Initially, the search space is statically divided and given to different processors .Now all the processors select and expand nodes simultaneously without causing contention on the shared OPEN list as before.
- The processors must communicate among themselves to minimize unnecessary search.
- Communication strategy for parallel BFS
  - Random communication strategy
  - Ring communication strategy
  - Blackboard communication strategy

1. Blackboard communication strategy

- In this strategy, there is a shared BLACKBOARD through which nodes are switched among processors.
- After selecting a node from its local OPEN list, the processor proceeds with its expansion only if it is within a "tolerable" limit of the best node in the BLACKBOARD.
- If the selected node is much better than the best node in the BLACKBOARD, then the processor transfers some of its good nodes to the BLACKBOARD.
- If the selected node is much worse than the best node in the BLACKBOARD, then the processor transfers some good nodes from the BLACKBOARD to its local OPEN list. In each case, a node is reselected for expansion from local OPEN.
- Select best node from open list
  - If I-value is OK then expand
  - If I-value is BAD then get some from blackboard
  - If I-value is GREAT then give some to blackboard

2. Random communication strategy

Periodically send some of the best nodes to a random processor.

3. Ring communication strategy

Periodically exchange best nodes with neighbors.

6.8 Distributed Computing : Document Classification

- Distributed computing is a method of processing large amounts of data by breaking it down into smaller parts that are processed simultaneously by multiple computers or nodes in a network. This allows for faster processing and increased efficiency compared to traditional centralized computing.
- Document classification is a process in which large amounts of text documents are organized and classified based on their content. In a distributed computing environment, document classification can be performed by dividing the documents into smaller groups and processing each group on a separate node in the network. The results from each node can then be combined to produce a final classification of the entire document collection.
- Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. Kubernetes can be used in a distributed computing environment to manage the deployment of applications and services on multiple nodes, ensuring that resources are used efficiently and that applications are available when needed.
- GPU (Graphics Processing Unit) applications are designed to take advantage of the parallel processing capabilities of GPUs. In a distributed computing environment, GPU applications can be run on multiple nodes to take advantage of the increased processing power provided by the GPUs. This is particularly useful for tasks such as image and video processing, which require a lot of processing power.
- Parallel computing for AI/ML is the use of parallel processing techniques to perform machine learning and artificial intelligence tasks. In a distributed computing environment, parallel computing can be used to speed up the training of machine learning models by dividing the training data into smaller parts that are processed simultaneously by multiple nodes. This can significantly reduce the time required to train a machine learning model, making it possible to train models on large amounts of data in a reasonable amount of time.
- In conclusion, distributed computing is a method of processing large amounts of data by breaking it down into smaller parts that are processed simultaneously by multiple nodes in a network. This allows for faster processing and increased efficiency compared to traditional centralized computing. Applications such as document classification, Kubernetes, GPU applications, and parallel computing for AI/ML are examples of how distributed computing can be used to solve complex problems in various fields.
- Distributed computing is a computing architecture that enables multiple processors to work together in a coordinated manner to perform a common task. In the field of document classification, distributed computing can be used to speed up the process of classifying large amounts of documents.
- Document classification is the process of categorizing documents into predefined classes based on their content. In traditional document classification, the process is performed on a single computer, which can be time-consuming and computationally intensive when dealing with large amounts of data. By using distributed computing, the process of document classification can be split into smaller tasks, with each task being performed by a different processor. This allows for the computations to be performed in parallel, reducing the classification time significantly.

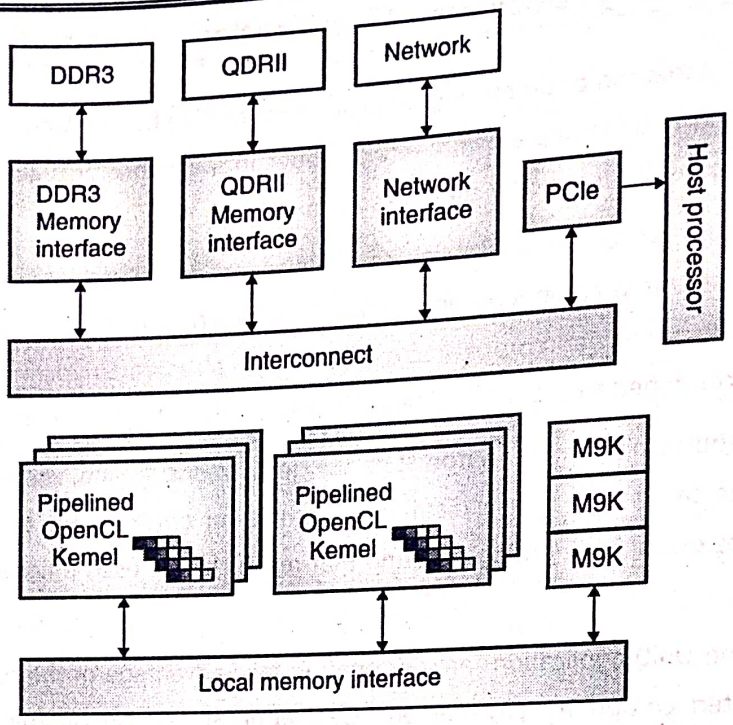


Fig. 6.8.1 : Distributed computing : document classification

- In a distributed computing system for document classification, each processor is assigned a set of documents to classify. The processors work independently, classifying the documents and sending the results back to a central node. The central node collects the results from all the processors and combines them to form the final classification.
- One advantage of using distributed computing for document classification is scalability. As the amount of data grows, more processors can be added to the system, allowing for faster processing times. Additionally, the use of multiple processors makes it possible to process data in real-time, making it possible to classify new documents as they arrive.
- In conclusion, distributed computing is a useful technology for speeding up the process of document classification. By using distributed computing, the classification process can be split into smaller tasks, with each task being performed by a different processor. This allows for faster processing times, making it possible to classify large amounts of data more efficiently. The use of distributed computing in document classification is leading to significant advancements in this field, making it possible to classify large volumes of data in real-time.

6.9 Frameworks : Kubernetes

- The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation result from counting the eight letters between the "K" and the "s". Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.
- Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It is a highly popular and widely used framework for managing and deploying

- Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.
- Kubernetes is designed to help developers and administrators manage their applications and services in a highly scalable, resilient, and efficient manner. It provides a unified platform for deploying and managing containers, ensuring that applications and services are always available and running optimally.
- Kubernetes works by dividing applications and services into smaller, isolated components called containers. Containers are packages that contain the application code, runtime, libraries, and other dependencies, and are used to deploy applications in a consistent manner across multiple environments.
- Kubernetes automates the deployment, scaling, and management of containers, ensuring that applications and services are always available and running optimally. It provides a range of features, including automatic scaling, self-healing, and rolling updates, that allow applications and services to be managed efficiently and effectively.
- Kubernetes also provides a unified platform for monitoring and logging, allowing developers and administrators to quickly identify and resolve issues with their applications and services. It integrates with a range of tools and technologies, including cloud services, continuous integration and delivery pipelines, and databases, to provide a complete solution for managing and deploying cloud-native applications.
- In conclusion, Kubernetes is a highly popular and widely used framework for automating the deployment, scaling, and management of containerized applications. It provides a unified platform for deploying and managing containers, ensuring that applications and services are always available and running optimally.
- Kubernetes provides a range of features, including automatic scaling, self-healing, and rolling updates, and integrates with a range of tools and technologies to provide a complete solution for managing and deploying cloud-native applications.
- As shown in Fig. 6.9.1 shows why Kubernetes is so useful by going back in time.

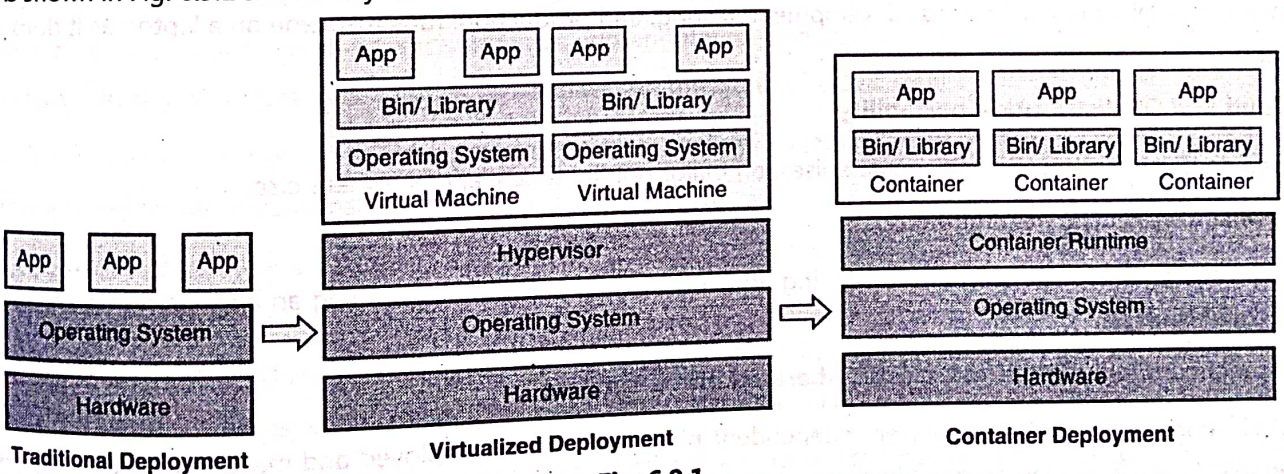


Fig. 6.9.1

1. **Traditional deployment era :** Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

2. **Virtualized deployment era :** As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application. Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines. Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.
3. **Container deployment era :** Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

6.9.1 Why Containers have become Popular ?

Containers provides additional benefits as following :

1. **Agile application creation and deployment**

   Increased ease and efficiency of container image creation compared to VM image use. Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and efficient rollbacks (due to image immutability).
2. **Dev and Ops separation of concerns**

   Create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
3. **Observability**

   It contains not only surfaces OS-level information and metrics, but also application health and other signals. Environmental consistency across development, testing, and production: runs the same on a laptop as it does in the cloud.
4. **Cloud and OS distribution portability**

   Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
5. **Application-centric management**

   Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
6. **Loosely coupled, distributed, elastic, liberated micro-services**

   Applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
7. **Resource isolation**

   Predictable application performance.
8. **Resource utilization**

   High efficiency and density.

6.9.2 Need of Kubernetes

- Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?
- That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example: Kubernetes can easily manage a canary deployment for your system.

6.9.3 Architecture of Kubernets

- Fig. 6.9.2 shows a high-level architecture of Kubernets.
- From a high level, a Kubernetes environment consists of a **control plane (master)**, a **distributed storage system** for keeping the cluster state consistent (**etcd**), and a number of **cluster nodes (Kubelets).**
- The control plane is the system that maintains a record of all Kubernetes objects. It continuously manages object states, responding to changes in the cluster; it also works to make the actual state of system objects match the desired state.
- As shown in Fig. 6.9.2, the control plane is made up of three major components: **kube-apiserver, kube-controller-manager** and **kube-scheduler.** These can all run on a **single master node**, or can be **replicated** across multiple master nodes for high availability.
- The **API Server** provides APIs to support lifecycle orchestration (scaling, updates, and so on) for different types of applications. It also acts as the gateway to the cluster, so the API server must be accessible by clients from outside the cluster.
- Clients authenticate via the API Server, and also use it as a proxy/tunnel to nodes and pods (and services).
- Most resources contain metadata, such as labels and annotations, desired state (specification) and observed state (current status). Controllers work to drive the actual state toward the desired state.
- There are various controllers to drive state for nodes, replication (autoscaling), endpoints (services and pods), service accounts and tokens (namespaces).
- The **Controller Manager** is a daemon that runs the core control loops, watches the state of the cluster, and makes changes to drive status toward the desired state.
- The **Cloud Controller Manager** integrates into each public cloud for optimal support of availability zones, VM instances, storage services, and network services for DNS, routing and load balancing.
- **The Scheduler** is responsible for the scheduling of containers across the nodes in the cluster; it takes various constraints into account, such as resource limitations or guarantees, and affinity and anti-affinity specifications.
- Pods are one of the crucial concepts in Kubernetes, as they are the key construct that **developers interact with.**
- This logical construct packages up a **single application**, which can consist of multiple containers and storage volumes. Usually, a single container (sometimes with some helper program in an additional container) runs in this configuration.

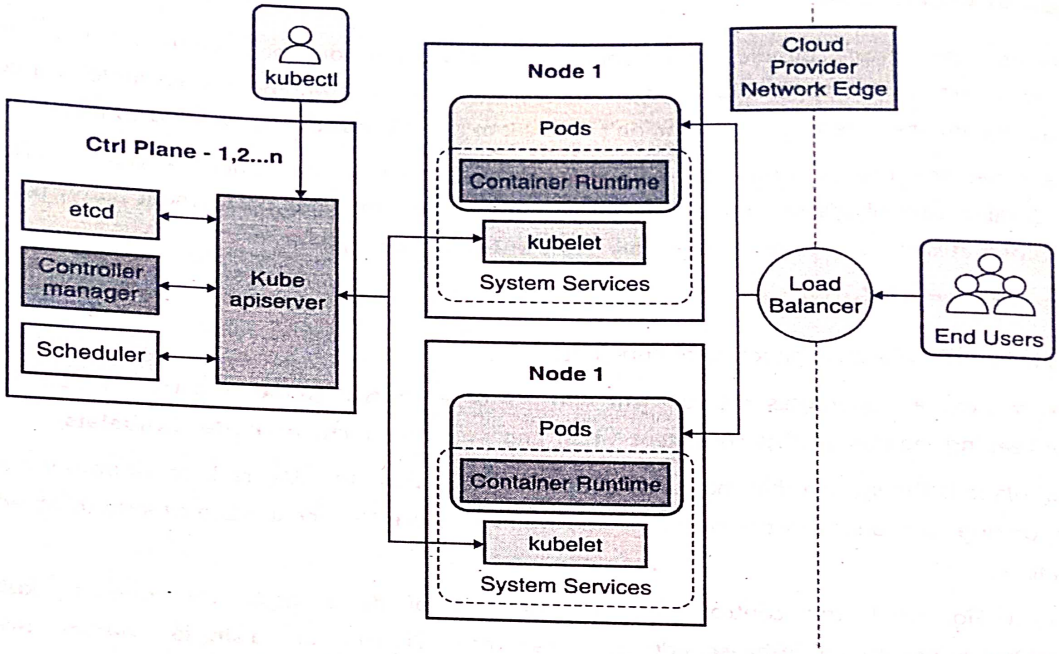


Fig. 6.9.2 : High-level architecture of Kubernets

- Alternatively, pods can be used to host vertically-integrated application stacks, like a WordPress LAMP (Linux, Apache, MySQL, PHP) application. A pod represents a running process on a cluster.
- Pods are ephemeral, with a limited lifespan. When scaling back down or upgrading to a new version, for instance, pods eventually die. Pods can do horizontal autoscaling (i.e., grow or shrink the number of instances), and perform rolling updates and canary deployments

6.9.4 Applications of Kubernetes

1. Service discovery and load balancing Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
2. Storage orchestration Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
3. Automated rollouts and rollbacks You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
4. Automatic bin packing You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
5. Self-healing Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

6. Secret and configuration management Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

6.9.5 Limitations of Kubernetes

1. Does not limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
2. Does not deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
3. Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.
4. Does not dictate logging, monitoring, or alerting solutions. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
5. Does not provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
6. Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.
7. Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

6.10 GPU Applications

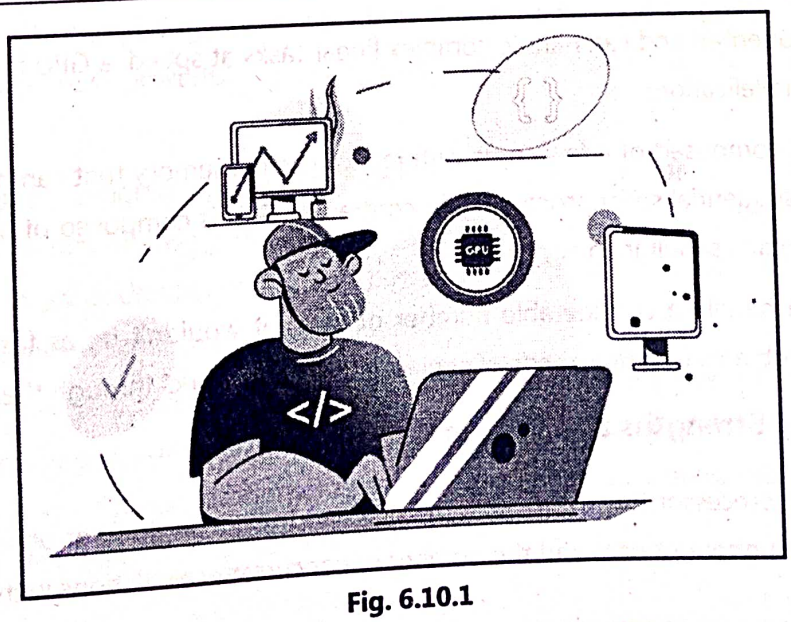


Fig. 6.10.1

- GPU computing is the use of a graphics processing unit (GPU) to perform highly parallel independent calculations that were once handled by the central processing unit (CPU).

6.10.1 History of GPU Computing

- Traditionally, GPUs have been used to accelerate memory-intensive calculations for computer graphics like image rendering and video decoding. These problems are prone to parallelization. Due to numerous cores and superior memory bandwidth, a GPU seemed to be an indispensable part of graphical rendering.
- While GPU-driven parallel computing was essential to graphical rendering, it also seemed to work real well for some scientific computing jobs. Consequently, GPU computing started to evolve more rapidly in 2006, becoming suitable for a wide array of general purpose computing tasks.
- Existing GPU instruction sets were improved and more of them were allowed to be executed within a single clock cycle, enabling a steady growth of GPU computing performance. Today, as Moore's law has slowed, and some even say it's over, GPU computing is keeping its pace.

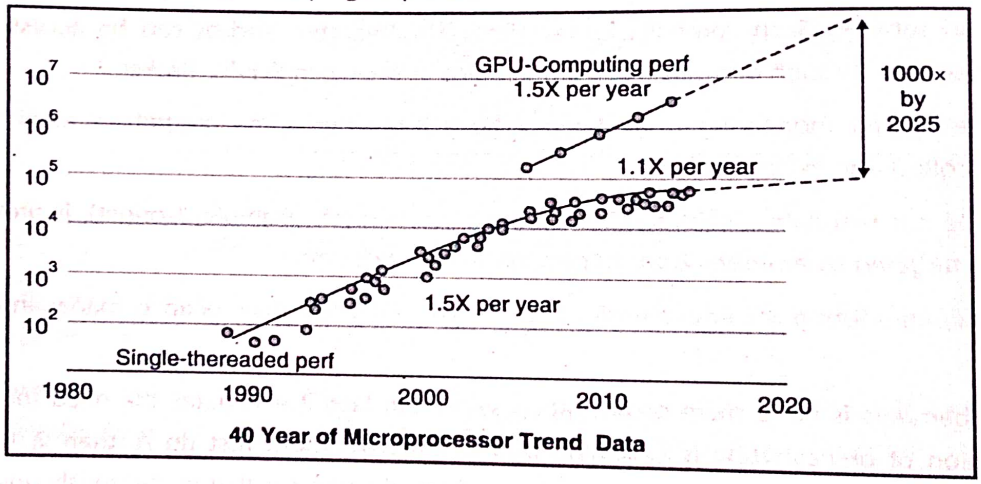


Fig. 6.10.2

(Fig. 6.10.2* - Nvidia Investor Day 2017 Presentation. Huang's law extends Moore's law - the performance of GPUs will more than double every two years).

6.10.2 CPU Vs. GPU : What's The Difference?

1. While a CPU is latency-oriented and can handle complex linear tasks at speed, a GPU is throughput-oriented, which allows for enormous parallelization.
2. Architecturally, a CPU is composed of a few cores with lots of cache memory that can handle few software threads at the same time using sequential serial processing. In contrast, a GPU is composed of thousands smaller cores that can manage multiple threads simultaneously.
3. Even though a CPU can handle a considerable number of tasks, it wouldn't be as fast as GPU doing so. A GPU breaks down complex problems into thousands of separate tasks and works through them simultaneously.

6.10.3 GPU Computing Strengths and Weaknesses

1. A GPU is a specialized co-processor that excels at some tasks and is not so good at others. It works in tandem with a CPU to increase the throughput of data and the number of concurrent calculations within the application.

2. **Arithmetic Intensity :** GPUs can cope extremely well with high arithmetic intensity. The algorithm is a good candidate for a GPU acceleration, if its ratio of math to memory operations is at least 10:1. If this is the case, your algorithm can benefit from the GPU's basic linear algebra subroutines (BLAS) and numerous arithmetic logic units (ALU).

3. **High Degree of Parallelism :** Parallel computing is a type of computation where many independent calculations are carried out simultaneously. Large problems can often be divided into smaller pieces which are then solved concurrently. GPU computing is designed to work like that. For instance, if it is possible to vectorize your data and adjust the algorithm to work on a set of values all at once, you can easily reap the benefits of GPU computing.

4. **Sufficient GPU Memory :** Ideally your data batch has to fit into the native memory of your GPU, in order to be processed seamlessly. Although there are workarounds to use multiple GPUs simultaneously or streamline your data from system memory, limited PCIe bandwidth may become a major performance bottleneck in such scenarios.

5. **Enough Storage Bandwidth :** In GPU computing you typically work with large amounts of data where storage bandwidth is crucial. Today the bottleneck for GPU-based scientific computing is no longer floating points per second (FLOPS), but I/O operations per second (IOPS). As a rule of thumb, it's always a good idea to evaluate your system's global bottleneck. If you find out that your GPU acceleration gains will be outweighed by the storage throughput limitations, optimize your storage solution first.

6.10.4 GPU Computing Applications

GPU computing is being used for numerous real-world applications. Many prominent science and engineering fields that we take for granted today would have not progressed so fast, if not GPU computing.

1. **Deep Learning :** Deep learning is a subset of machine learning. Its implementation is based on artificial neural networks. Essentially, it mimics the brain, having neuron layers work in parallel. Since data is represented as a set of vectors, deep learning is well-suited for GPU computing. You can easily experience up to 4x performance gains when training your convolutional neural network on a Dedicated Server with a GPU accelerator. As a cherry on top, every major deep learning framework like TensorFlow and PyTorch already allows you to use GPU computing out-of-the-box with no code changes.

2. **Drug Design :** The successful discovery of new drugs is hard in every respect. We have all become aware of this during the Covid-19 pandemic. Eroom's law states that the cost of discovering a new drug roughly doubles every nine years. Modern GPU computing aims to shift the trajectory of Eroom's law. Nvidia is currently building Cambridge-1 - the most powerful supercomputer in the UK - dedicated to AI research in healthcare and drug design.

3. **Seismic Imaging :** Seismic imaging is used to provide the oil and gas industry with knowledge of Earth's subsurface structure and detect oil reservoirs. The algorithms used in seismic data processing are evolving rapidly, so there's a huge demand for additional computing power. For instance, the Reverse Time Migration method can be accelerated up to 14 times when using GPU computing.

4. **Automotive design :** Flow field computations for transient and turbulent flow problems are highly compute-intensive and time-consuming. Traditional techniques often compromise on the underlying physics and are not very efficient. A new paradigm for computing fluid flows relies on GPU computing that can help achieve significant speed-ups over a single CPU, even up to a factor of 100.

5. **Astrophysics :** GPU has dramatically changed the landscape of high performance computing in astronomy. Take an N-body simulation for instance, that numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. You can accelerate the all-pairs N-body algorithm up to 25 times by using GPU computing rather than using a highly tuned serial CPU implementation.
6. **Options pricing :** The goal of option pricing theory is to provide traders with an option's fair value that can then be incorporated into their trading strategies. Some type of Monte Carlo algorithm is often used in such simulations. GPU computing can help you achieve 27 times better performance per dollar compared to CPU-only approach.
7. **Weather forecasting :** Weather forecasting has greatly benefited from exponential growth of mere computing power in recent decades, but this free ride is nearly over. Today weather forecasting is being driven by fine-grained parallelism that is based on extensive GPU computing. This approach alone can ensure 20 times faster weather forecasting models.

6.10.5 GPU Computing in the Cloud

- Even though GPU computing was once primarily associated with graphical rendering, it has grown into the main driving force of high performance computing in many different scientific and engineering fields.
- Most of the GPU computing work is now being done in the cloud or by using in-house GPU computing clusters. Here at Cherry Servers we are offering Dedicated GPU Servers with high-end Nvidia GPU accelerators. Our infrastructure services can be used on-demand, which makes GPU computing easy and cost-effective.
- Cloud vendors have democratized GPU computing, making it accessible for small and medium businesses world-wide. If Huang's law lasts, the performance of GPU will more than double every two years, and innovation will continue to sprout.
- GPU (Graphics Processing Unit) applications refer to software programs that are designed to take advantage of the parallel processing capabilities of GPUs. GPUs are specialized processors designed to perform complex mathematical calculations and visual processing tasks at high speeds. They are commonly used in computer graphics and video games, but their parallel processing capabilities make them well suited for a wide range of applications beyond graphics and gaming.
- One of the key applications of GPUs is in the field of Artificial Intelligence and Machine Learning. GPUs can be used to perform complex matrix calculations and vector operations, which are fundamental components of many AI and ML algorithms. By using GPUs for these tasks, the processing time can be significantly reduced, allowing for more complex models to be trained and for faster predictions to be made.
- Another application of GPUs is in scientific simulations and data processing. GPUs can be used to perform large-scale simulations in fields such as molecular dynamics, fluid dynamics, and weather forecasting, allowing scientists to study complex systems and make predictions faster than ever before. They can also be used to process large amounts of data in fields such as astronomy, genomics, and finance, making it possible to analyze vast amounts of data in a matter of minutes.
- GPUs are also widely used in video processing and streaming applications. They can be used to encode and decode video in real-time, making it possible to stream high-quality video content to a wide range of devices. They are also used to perform real-time video rendering and post-production, allowing video editors to make changes to their content in real-time.

- In conclusion, GPU applications are software programs designed to take advantage of the parallel processing capabilities of GPUs. GPUs are widely used in a range of fields, including Artificial Intelligence and Machine Learning, scientific simulations and data processing, and video processing and streaming. By leveraging the parallel processing capabilities of GPUs, these applications can perform complex tasks faster and more efficiently, leading to significant advancements in a wide range of fields.

6.10 Parallel Computing for AI/ ML

- Parallel computing is a computing architecture that enables multiple processors to perform computations simultaneously, allowing for faster processing times and improved performance. In the field of Artificial Intelligence (AI) and Machine Learning (ML), parallel computing can be used to speed up the training of complex models, make faster predictions, and process large amounts of data.
- In AI and ML, parallel computing is typically achieved by distributing the computations across multiple GPUs or CPU cores. For example, in deep learning, the process of training a neural network can be split into multiple tasks, with each task being performed by a different GPU or CPU core. This allows for the computations to be performed in parallel, reducing the training time significantly.
- Another area where parallel computing is used in AI and ML is in data processing. In many AI and ML applications, large amounts of data need to be processed, which can be time-consuming and computationally intensive. By using parallel computing, the data can be split into smaller chunks, with each chunk being processed by a different GPU or CPU core. This allows for faster processing times, making it possible to analyze vast amounts of data in a matter of minutes.
- Parallel computing can also be used to speed up the prediction process in AI and ML. For example, in image classification, parallel computing can be used to process multiple images simultaneously, allowing for faster predictions. In recommendation systems, parallel computing can be used to process user data in real-time, making it possible to make recommendations to users quickly.
- In conclusion, parallel computing is a key technology for improving the performance of AI and ML applications. By using parallel computing, computations can be performed faster, large amounts of data can be processed more efficiently, and predictions can be made more quickly. The use of parallel computing in AI and ML is leading to significant advancements in these fields, making it possible to develop and deploy more complex and sophisticated models.

Review Questions

Q. 1 Explain the issue in Sorting of Parallel Algorithm. **(8 Marks)**

Q. 2 Write a pseudo code for parallel Quick Sort. **(7 Marks)**

Q. 3 Write Pseudo code for Single Source Shortest Path. **(8 Marks)**

Q. 4 Describe Parallel DFS or BFS in details? **(7 Marks)**

Q. 5 Write a short note on Parallel-Best-First-Search. **(7 Marks)**

Q. 6 Write a short note on Parallel-Depth-First-Search. **(7 Marks)**