UNIT 1 PARALLEL ALGORITHMS

Structure **Page Nos.**

1.0 INTRODUCTION

An algorithm is defined as a sequence of computational steps required to accomplish a specific task. The algorithm works for a given input and will terminate in a well defined state. The basic conditions of an algorithm are: input, output, definiteness, effectiveness and finiteness. The purpose of the development an algorithm is to solve a general, well specified problem.

A concern while designing an algorithm also pertains to the kind of computer on which the algorithm would be exectued. The two forms of architectures of computers are: sequential computer and parallel computer. Therefore, depending upon the architecture of the computers, we have sequential as well as parallel algorithms.

The algorithms which are executed on the sequential computers simply perform according to sequence of steps for solving a given problem. Such algorithms are known as sequential algorithms.

However, a problem can be solved after dividing it into sub-problems and those in turn are executed in parallel. Later on, the results of the solutions of these subproblems can be combined together and the final solution can be achieved. In such situations, the number of processors required would be more than one and they would be communicating with each other for producing the final output. This environment operates on the parallel computer and the special kind of algorithms called parallel algorithms are designed for these computers. The parallel algorithms depend on the kind of parallel computer they are desinged for. Hence, for a given problem, there would be a need to design the different kinds of parallel algorithms depending upon the kind of parallel architecture.

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations, multiple-processor workstations, and embedded systems. The parallel computers can be represented with the help of various kinds of models such as random access machine (RAM), parallel random access machine (PRAM), Interconnection Networks etc. While designing a parallel algorithm, the computational power of various models can be analysed and compared, parallelism can be involved for a given problem on a specific model after understanding the characteriscitics of a model. The analysis of parallel algorithm on different models assist in determining the best model for a problem after receiving the results in terms of the time and space complexity.

In this unit, we have first discussed the various parameters for analysis of an algorithm. Thereafter, the various kinds of computational models such as combinational circuits etc. have been presented. Subsequently, a few problems have been taken up, e.g., sorting, matrix multiplication etc. and solved using parallel algorithms with the help of various parallel compuational models.

1.1 OBJECTIVES

After studying this unit the learner will be able to understand about the following:

- Analysis of Parallel Algorithms;
- Different Models of Computation;
  - Combinational Circuits
  - Interconnection Networks
  - PRAM
- Sorting Computation, and
- Matrix Computation.

1.2 ANALYSIS OF PARALLEL ALGORITHMS

A generic algorithm is mainly analysed on the basis of the following parameters: the time complexity (execution time) and the space complexity (amount of space required). Usually we give much more importance to time complexity in comparison with space complexity. The subsequent section highlights the criteria of analysing the complexity of parallel algorithms. The fundamental parameters required for the analysis of parallel algorithms are as follow:

- Time Complexity
- The Total Number of Processors Required
- The Cost Involved.

1.2.1 Time Complexity

As it happens, most people who implement algorithms want to know how much of a particular resource (such as time or storage) is required for a given algorithm. The parallel architectures have been designed for improving the computation power of the various algorithms. Thus, the major concern of evaluating an algorithm is the determination of the amount of time required to execute. Usually, the time complexity is calculated on the basis of the total number of steps executed to accomplish the desired output.

The Parallel algorithms usually divide the problem into more symmetrical or asymmetrical subproblems and pass them to many processors and put the results back together at one end. The resource consumption in parallel algorithms is both processor cycles on each processor and also the communication overhead between the processors.

Thus, first in the computation step, the local processor performs an arthmetic and logic operation. Thereafter, the various processors communicate with each other for exchanging messages and/or data. Hence, the time complexity can be calculated on the basis of computational cost and communication cost invloved.

The time complexity of an algorithm varies depending upon the instance of the input for a given problem. For example, the already sorted list (10,17, 19, 21, 22, 33) will consume less amout of time than the reverse order of list (33, 22, 21,19,17,10). The time complexity of an algorithm has been categorised into three forms, viz:

i) Best Case Complexity;
ii) Average Case Complexity; and
iii) Worst Case Complexity.

The best case complexity is the least amount of time required by the algorithm for a given input. The average case complexity is the average running time required by the algorithm for a given input. Similarly, the worst case complexity can be defined as the maximum amount of time required by the algorithm for a given input.

Therefore, the main factors involved for analysing the time complexity depends upon the algorightm, parallel computer model and specific set of inputs. Mostly the size of the input is a function of time complexity of the algorithm. The generic notation for describing the time-complexity of any algorithm is discussed in the subsequent sections.

1.2.1.1 Asymptotic Notations

These notations are used for analysing functions. Suppose we have two functions f(n) and g(n) defined on real numbers,

i) *Theta Θ Notation*: The set Θ(g(n)) consists of all functions f(n), for which there exist positive constants c1,c2 such that f(n) is sandwiched between c1*g(n) and c2*g(n), for sufficiently large values of n. In other words,
$$\Theta(g(n)) = \{ 0 <= c1*g(n) <= f(n) <= c2*g(n) \text{ for all } n >= n_o \}$$

ii) *Big O Notation*: The set O(g(n)) consists of all functions f(n), for which there exists positive constants c such that for sufficiently large values of n, we have 0<= f(n) <= c*g(n). In other words,
$$O(g(n)) = \{ 0 <= f(n) <= c*g(n) \text{ for all } n >= n_o \}$$

iii) *Ω Notation*: The function f(n) belongs to the set Ω (g(n)) if there exists positive constants c such that for sufficiently large values of n,
we have 0<= c*g(n) <=f(n). In other words,
$$O(g(n)) = \{ 0 <= c*g(n) <= f(n) \text{ for all } n >= n_o \}.$$

Suppose we have a function $f(n)= 4n^2 + n$, then the order of function is $O(n^2)$. The asymptotic notations provide information about the lower and upper bounds on complexity of an algorithm with the help of Ω and O notations. For example, in the sorting algorithm the lower bound is Ω (n ln n) and upper bound is O (n ln n). However, problems like matrix multiplication have complexities like $O(n^3)$ to $O(n^{2.38})$. Algorithms

which have similar upper and lower bounds are known as optimal algorithms. Therefore, few sorting algorithms are optimal while matrix multiplication based algorithms are not.

Another method of determining the performance of a parallel algorithm can be carried out after calculating a parameter called "speedup". Speedup can be defined as the ratio of the worst case time complexity of the fastest known sequential algorithm and the worst case running time of the parallel algorithm. Basically, speedup determines the performance improvement of parallel algorithm in comparison to sequential algorithm.

$$Speedup = \frac{Worst\ case\ running\ time\ of\ Sequential\ Algorithm}{Worst\ case\ running\ time\ of\ Parallel\ Algorithm}$$

1.2.2 Number of Processors

One of the other factors that assist in analysis of parallel algorithms is the total number of processors required to deliver a solution to a given problem. Thus, for a given input of size say n, the number of processors required by the parallel algorithm is a function of n, usually denoted by TP (n).

1.2.3 Overall Cost

Finally, the total cost of the algorithm is a product of time complexity of the parallel algorithm and the total number of processors required for computation.

*Cost = Time Complexity * Total Number of Processors*

The other form of defining the cost is that it specifies the total number of steps executed collectively by the n number of processors, i.e., *summation of steps*. Another term related with the analysis of the parallel algorithms is *efficiency* of the algorithm. It is defined as the ratio of the worst case running time of the best sequential algorithm and the cost of the parallel algorithm. The efficiency would be mostly less than or equal to 1. In a situation, if efficiency is greater than 1 then it means that the sequential algorithm is faster than the parallel algorithm.

$$Efficiency = \frac{Worst\ case\ running\ time\ of\ Sequential\ Algorithm}{Cost\ of\ Parallel\ Algorithm}$$

1.3 DIFFERENT MODELS OF COMPUTATION

There are various computational models for representing the parallel computers. In this section, we discuss various models. These models would provide a platform for the designing as well as the analysis of the parallel algorithms.

1.3.1 Combinational Circuits

Combinational Circuit is one of the models for parallel computers. In interconnection networks, various processors communicate with each other directly and do not require a shared memory in between. Basically, combinational circuit (cc) is a connected arrangement of logic gates with a set of m input lines and a set of n output lines as shown in *Figure 1*. The combinational circuits are mainly made up of various interconnected components arranged in the form known as *stages* as shown in *Figure 2*.

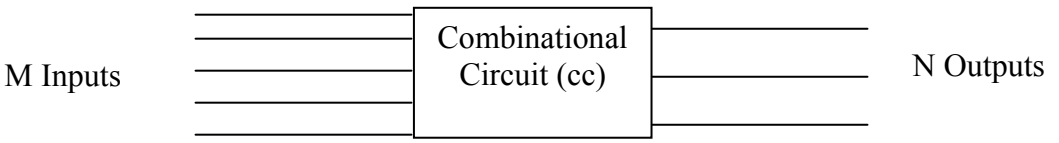


Figure 1: Combinational circuit

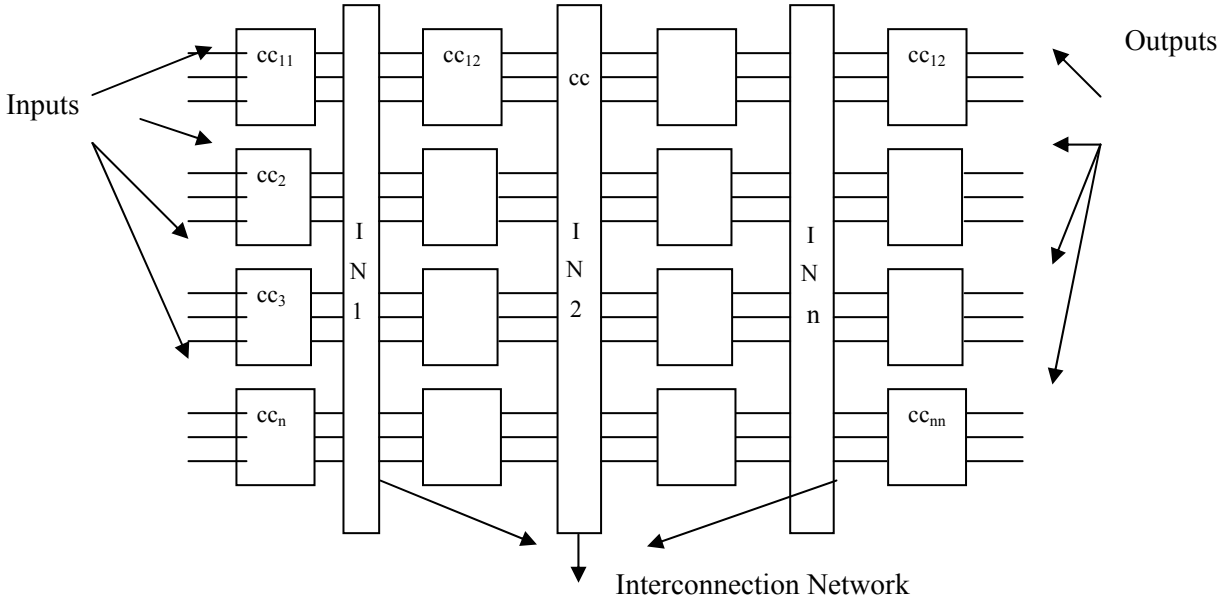


Figure 2: Detailed combinational circuit

It may be noted here that there is no feedback control employed in combinational circuits. There are few terminologies followed in the context of combinational circuits such as fan in and fan out. Fan in signifies the number of input lines attached to each device and fan out signifies the number of output lines. In *Figure 2*, the fan in is 3 and fan out is also 3. The following parameters are used for analysing a combinational circuit:

1) *Depth:* It means that the total number of stages used in the combinational circuit starting from the input lines to the output lines. For example, in the depth is 4, as there are four different stages attached to a interconnection network. The other form of interpretation of depth can be that it represents the worst case time complexity of solving a problem as input is given at the initial input lines and data is transferred between various stages through the interconnection network and at the end reaches the output lines.

2) *Width:* It represents the total number of devices attached for a particular stage. For example in Figure 2, there are 4 components attached to the interconnection network. It means that the width is 4.

3) *Size:* It represents the total count of devices used in the complete combinational circuit. For example, in Figure 2, the size of combinational circuit is 16 i.e. (width * depth).

1.4 PARALLEL RANDOM ACCESS MACHINES (PRAM)

PRAM is one of the models used for designing the parallel algorithm as shown in *Figure 3*. The PRAM model contains the following components:

i) A set of identical type of processors say P_1, P_2, P_3 …P_n.

ii) It contains a single shared memory module being shared by all the N processors. As the processors cannot communicate with each other directly, shared memory acts as a communication medium for the processors.

iii) In order to connect the N processor with the single shared memory, a component called Memory Access Unit (MAU) is used for accessing the shared memory.

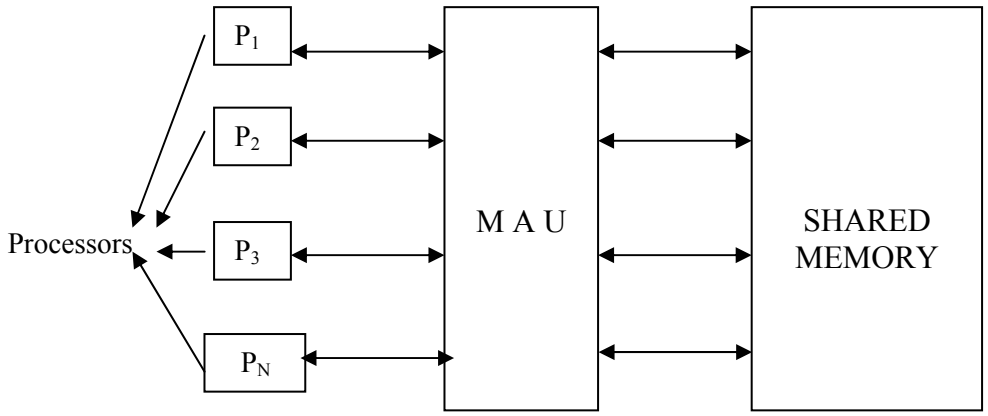


Figure 3: PRAM Model

Following steps are followed by a PRAM model while executing an algorithm:

i) *Read phase:* First, the N processors simultaneously read data from N different memory locations of the shared memory and subsequently store the read data into its local registers.

ii) *Compute phase:* Thereafter, these N processors perform the arithmetic or logical operation on the data stored in their local registers.

iii) *Write phase:* Finally, the N processors parallel write the computed values from their local registers into the N memory locations of the shared memory.

1.5 INTERCONNECTION NETWORKS

As in PRAM, there was no direct communication medium between the processors, thus another model known as interconnection networks have been designed. In the interconnection networks, the N processors can communicate with each other through direct links. In the interconnection networks, each processor has an independent local memory.

☞ **Check Your Progress 1**

1) Which of the following model of computation requires a shared memory?
   1) PRAM
   2) RAM
   3) Interconnection Networks
   4) Combinational Circuits

2) Which of the following model of computation has direct link between processors?
   1) PRAM
   2) RAM
   3) Interconnection Networks
   4) Combinational Circuits

3) What does the term width depth in combinational circuits mean?
   1) Cost
   2) Running Time
   3) Maximum number of components in a given stage
   4) Total Number of stages

4) Explain the concept of analysis of parallel algorithms.

……
……
……
……

1.6 SORTING

The term sorting means arranging elements of a given set of elements, in a specific order i.e., ascending order / descending order / alphabetic order etc. Therefore, sorting is one of the interesting problems encountered in computations for a given data. In the current section, we would be discussing the various kinds of sorting algorithms for different computational models of parallel computers.

The formal representation of sorting problem is as explained: Given a string of m numbers, say X= x_1, x_2, x_3, x_4 ..... x_m and the order of the elements of the string X is initially arbitrary. The solution of the problem is to rearrange the elements of the string X such that the resultant sequence is in an ascending order.

Let us use the combinational circuits for sorting the string.

1.7 COMBINATIONAL CIRCUIT FOR SORTING THE STRING

Each input line of the combinational circuit represents an individual element of the string say x_i and each output line results in the form of a sorted list. In order to achieve the above mentioned task, a comparator is employed for the processing.

Each comparator has two input lines, say a and b, and similarly two output lines, say c and d. Each comparator provides two outputs i.e., c provides maximum of a and b **(max**

(a, b)) and d provides minimum of a and b **(min (a, b))** in comparator InC and DeC it is opposite, as shown in *Figure 4 and 5*.

In general, there are two types of comparators, often known as increasing comparators and decreasing comparators denoted by + BM(n) and – BM(n) where n denotes the number of input lines and output lines of the comparator. The depth of + BM(n) and – BM(n) is log n. These comparators are employed for constructing the circuit of sorting.

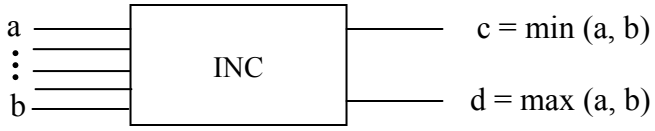


Figure 4 (a) Increasing Comparator, for 2 inputs

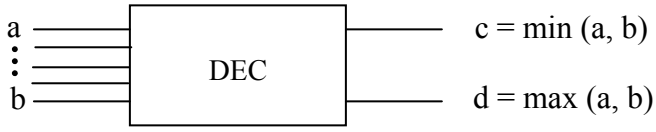


Figure 4 (b) Decreasing Comparator, for 2 inputs

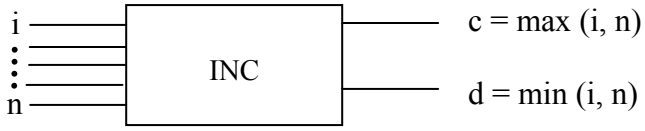


Figure 5 (a): Increasing Comparator, for n inputs

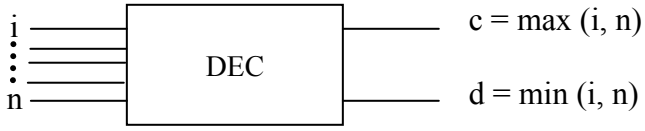


Figure 5 (b): Decreasing Comparator, for n inputs

Now, let us assume a famous sequence known as bitonic sequence and sort out the elements using a combinational circuit consisting of a set of comparators. The property of bitonic sequence is as follows:

Consider a sequence X= $x_0, x_1, x_2, x_3, x_4 \ldots\ldots\ldots\ldots x_{n-1}$ such that condition 1:either $x_0, x_1, x_2, x_3, x_4 \ldots\ldots\ldots\ldots x_i$ is a monotonically increasing sequence and $x_{i+1}, x_{i+2}, \ldots\ldots\ldots\ldots x_{n-1}$ is a monotonically decreasing sequence or condition 2: there exists a cyclic shift of the sequence $x_0, x_1, x_2, x_3, x_4 \ldots\ldots\ldots\ldots x_{n-1}$ such that the resulting sequence satisfies the condition 1.

Let us take a few examples of bitonic sequence:

B1= 4,7,8,9,11,6,3,2,1 is bitonic sequence
B2= 12,15,17,18,19,11,8,7,6,4,5 is bitonic sequence

In order to provide a solution to such a bitonic sequence, various stages of comparators are required. The basic approach followed for solving such a problem is as given:

Let us say we have a bitonic sequence X= x_0, x_1, x_2, x_3, x_4 ..... x_{n-1} with the property that first n/2 elements are monotonically increasing elements are monotonically increasing like $x_0 < x_1 < x_2 < x_3 < x_4 \ldots < x_{n/2-1}$ and other numbers are monotonically decreasing as $x_{n/2} > x_{n/2+1} > \ldots > x_{n-1}$. Thereafter, these patterns are compared with the help of a comparator as follows:

$$Y = \min(x_0, x_{n/2}), \min(x_1, x_{n/2+1}), \min(x_2, x_{n/2+2}), \ldots\ldots\ldots\ldots \text{Min}\ (x_{n/2-1}, x_{n-1})$$
$$Z = \max(x_0, x_{n/2}), \max\ (x_1, x_{n/2+1}), \max\ (x_2, x_{n/2+2}), \ldots\ldots\ldots\ldots \max\ (x_{n/2-1}, x_{n-1})$$

After the above discussed iteration, the two new bitonic sequences are generated and the method is known as bitonic split. Thereafter, a recursive operation on these two bitonic sequences is processed until we are able to achieve the sorted list of elements. The exact algorithm for sorting the bitonic sequence is as follows:

Sort_Bitonic (X)

// Input: N Numbers following the bitonic property
// Output: Sorted List of numbers

1) *The Sequence, i.e. X is transferred on the input lines of the combinational circuit which consists of various set of comparators.*
2) *The sequence X is splitted into two sub-bitonic sequences say, Y and Z, with the help of a method called bitonic split.*
3) *Recursively execute the bitonic split on the sub sequences, i.e. Y and Z, until the size of subsequence reaches to a level of 1.*
4) *The sorted sequence is achieved after this stage on the output lines.*

With the help of a diagram to illustrate the concept of sorting using the comparators.
Example 1: Consider a unsorted list having the element values as
{3,5,8,9,10,12,14,20,95,90,60,40,35,23,18,0}

This list is to be sorted in ascending order. To sort this list, in the first stage comparators of order 2 (i.e. having 2 input and 2 output) will be used. Similarly, 2nd stage will consist of 4, input comparators, 3rd stage 8 input comparator and 4th stage a 16 input comparator.

Let us take an example with the help of a diagram to illustrate the concept of sorting using the comparators (see *Figure 6*).

<table>
<tr><td>3
5</td><td>+BM(2)</td><td>3
5</td><td rowspan="2">+BM(4)</td><td>3
5</td><td rowspan="8"></td><td>3
5</td><td rowspan="16"></td><td>0
3</td></tr>
<tr><td>8
9</td><td>-BM(2)</td><td>9
8</td><td>8
9</td><td>8
9</td><td>5
8</td></tr>
<tr><td>10
12</td><td>+BM(2)</td><td>10
12</td><td rowspan="3">-BM(4)</td><td>20
14</td><td>10
12</td><td>9
10</td></tr>
<tr><td>14
20</td><td>-BM(2)</td><td>20
14</td><td>12
10</td><td>14
20</td><td>12
14</td></tr>
<tr><td>95
90</td><td>+BM(2)</td><td>90
95</td><td rowspan="3">+BM(4)</td><td>40
60</td><td rowspan="7">-BM(8)</td><td>95
90</td><td>18
20</td></tr>
<tr><td>60
40</td><td>-BM(2)</td><td>60
40</td><td>90
95</td><td>60
40</td><td>23
35</td></tr>
<tr><td>35
23</td><td>+BM(2)</td><td>23
35</td><td rowspan="3">-BM(4)</td><td>35
23</td><td>35
23</td><td>40
60</td></tr>
<tr><td>18
0</td><td>-BM(2)</td><td>18
0</td><td>18
0</td><td>18
0</td><td>90
95</td></tr>
</table>

Figure 6: Sorting using Combinational Circuit

Analysis of Sort_Bitonic(X)

The bitonic sorting network requires log n number of stages for performing the task of sorting the numbers. The first n-1 stages of the circuit are able to sort two n/2 numbers and the final stage uses a +BM (n) comparator having the depth of log n. As running time of the sorting is dependent upon the total depth of the circuit, therefore it can be depicted as:

$$D(n) = D(n/2) + \log n$$

Solving the above mentioned recurrence relation

$$D(n) = (\log^2 n + \log n)/2 = O(\log^2 n)$$

Thus, the complexity of solving a sorting algorithm using a combinational circuit is $O(\log^2 n)$.

Another famous sorting algorithm known as merge sort based algorithm can also be depicted / solved with the help of combinational circuit. The basic working of merge sort algorithm is discussed in the next section

1.8 MERGE SORT CIRCUIT

First, divide the given sequence of n numbers into two parts, each consisting of n/2 numbers. Thereafter, recursively split the sequence into two parts until each number acts as an independent sequence. Consequently, the independent numbers are first sorted and recursively merged until a sorted sequence of n numbers is not achieved.

In order to perform the above-mentioned task, there will be two kinds of circuits which would be used in the following manner: the first one for sorting and another one for merging the sorted list of numbers.

Let us discuss the sorting circuit for merge sort algorithm. The sorting Circuit.

Odd-Even Merging Circuit

Let us firstly illustrate the concept of merging two sorted sequences using a odd-even merging circuit. The working of a merging circuit is as follows:

1) Let there be two sorted sequences $A=(a_1, a_2, a_3, a_4 \ldots\ldots a_m)$ and $B=(b_1, b_2, b_3, b_4 \ldots\ldots b_m)$ which are required to be merged.
2) With the help of a merging circuit (m/2,m/2), merge the odd indexed numbers of the two sub sequences i.e. $(a_1, a_3, a_5 \ldots\ldots a_{m-1})$ and $(b_1, b_3, b_5 \ldots\ldots b_{m-1})$ and thus resulting in sorted sequence $(c_1, c_2, c_3 \ldots\ldots c_{m)}$.
3) Thereafter, with the help of a merging circuit (m/2,m/2), merge the even indexed numbers of the two sub sequences i.e. $(a_2, a_4, a_6 \ldots\ldots a_m)$ and $(b_2, b_4, b_6 \ldots\ldots b_m)$ and thus resulting in sorted sequence $(d_1, d_2, d_3 \ldots\ldots d_{m)}$.
4) The final output sequence $O=(o_1, o_2, o_3 \ldots\ldots o_{2m})$ is achieved in the following manner: $o_1 = a_1$ and $o_{2m} = b_m$ .In general the formula is as given below: $o_{2i} = \min(a_{i+1}, b_i)$ and $o_{2I+1} = \max(a_{i+1}, b_i)$ for i=1,2,3,4…….m-1.

Now, let us take an example for merging the two sorted sequences of length 4, i.e., $A=(a_1, a_2, a_3, a_4)$ and $B=(b_1, b_2, b_3, b_4)$. Suppose the numbers of the sequence are A=(4,6,9,10) and B=(2,7,8,12). The circuit of merging the two given sequences is illustrated in *Figure 7*.

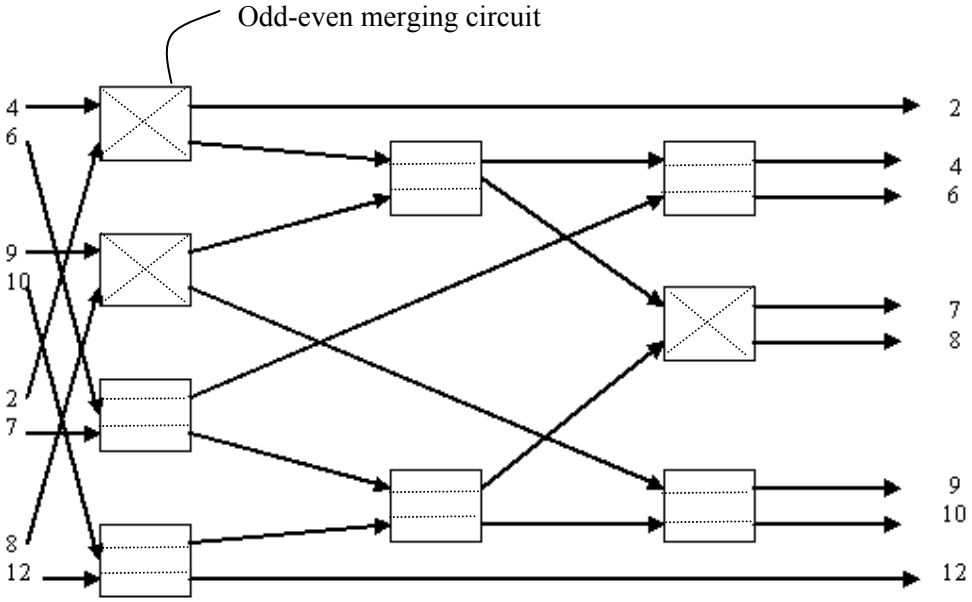


Figure 7: Merging Circuit

Sorting Circuit along with Odd-Even Merging Circuit

As we already know, the merge sort algorithm requires two circuits, i.e. one for merging and another for sorting the sequences. Therefore, the sorting circuit has been derived from the above-discussed merging circuit. The basic steps followed by the circuit are highlighted below:

i) The given input sequence of length n is divided into two sub-sequences of length n/2 each.
ii) The two sub sequences are recursively sorted.
iii) The two sorted sub sequences are merged (n/2,n/2) using a merging circuit in order to finally get the sorted sequence of length n.

Now, let us take an example for sorting the n numbers say 4,2,10,12 8,7,6,9. The circuit of sorting + merging given sequence is illustrated in *Figure 8*.

Analysis of Merge Sort

i) The width of the sorting + merging circuit is equal to the maximum number of devices required in a stage is O(n/2). As in the above figure the maximum number of devices for a given stage is 4 which is 8/2 or n/2.
ii) The circuit contains two sorting circuits for sorting sequences of length n/2 and thereafter one merging circuit for merging of the two sorted sub sequences (see stage 4^{th} in the above figure). Let the functions Ts and Tm denote the time complexity of sorting and merging in terms of its depth.

The Ts can be calculated as follows:

$$Ts(n) = Ts(n/2) + Tm(n/2)$$

$$Ts(n) = Ts(n/2) + \log(n/2) ,$$

Therefore, Ts (n) is equal to $O(\log^2 n)$.

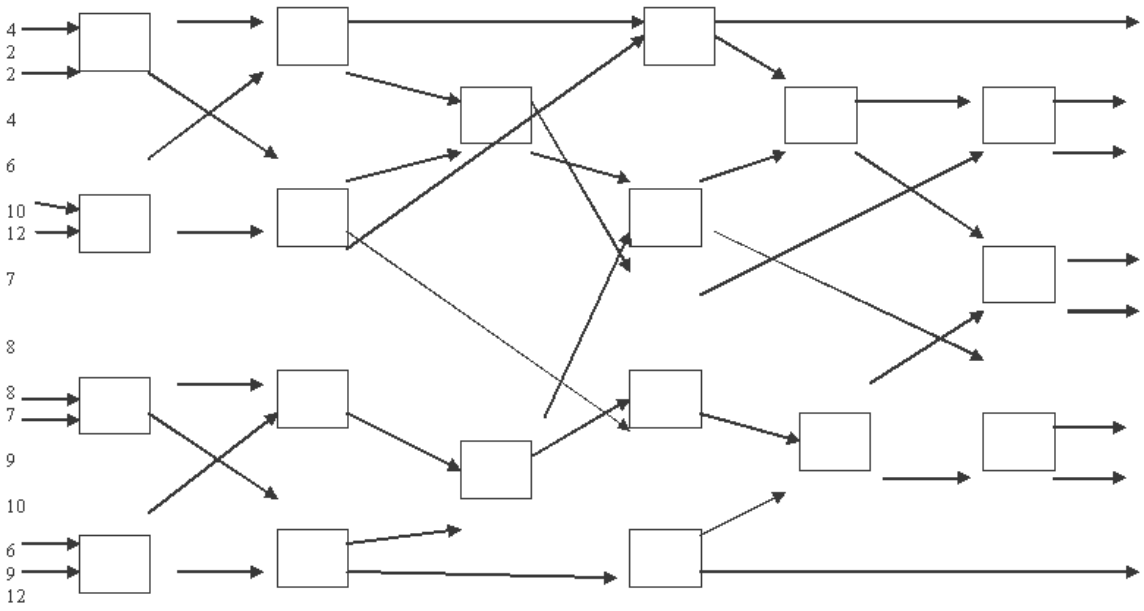


Figure 8: Sorting + Merging Circuit

1.9 SORTING USING INTERCONNECTION NETWORKS

The combinational circuits use the comparators for comparing the numbers and storing them on the basis of minimum and maximum functions. Similarly, in the interconnection networks the two processors perform the computation of minimum and maximum functions in the following way:

Let us consider there are two processors p_i and p_j. Each of these processors has been given as input an element of the sequence, say e_i and e_j. Now, the processor p_i sends the element e_i to p_j and consequently processor p_j sends e_j to p_i. Thereafter, processor p_i calculates the minimum of e_i and e_j i.e., min (e_i,e_j) and processor p_j calculates the maximum of e_i and e_j, i.e. max (e_i,e_j). The above method is known as compare-exchange and it has been depicted in the Figure 9.

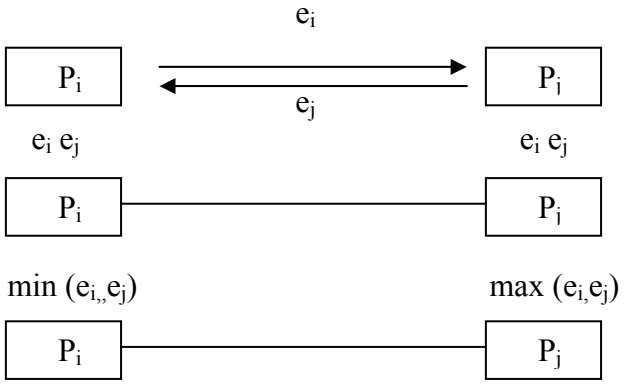


Figure 9: Illustration of Exchange-cum-Comparison in interconnection networks

The sorting problem selected is bubble sort and the interconnection network can be depicted as n processors interconnected with each other in the form of a linear array as

shown in *Figure 10*. The technique adopted for solving the bubble sort is known as odd-even transposition. Assume an input sequence is B=($b_1, b_2, b_3, b_4 \ldots\ldots b_n$) and each number is assigned to a specific processor. In the odd-even transposition, the sorting is performed with the help of two phases called odd phase and even phase. In the odd phase, the elements stored in (p_1, p_2), (p_3, p_4), (p_5, p_6)…… (p_{n-1}, p_n) are compared according to the Figure and subsequently exchanged if required i.e. if they are out of order. In the even phase, the elements stored in (p_2, p_3), (p_4, p_5), (p_6, p_7)…… (p_{n-2}, p_{n-1}) are compared according to the Figure and subsequently exchanged if required, i.e. if they are out of order. Remember, in the even phase the elements stored in p_1 and p_n are not compared and exchanged. The total number of phases required for sorting the numbers is n i.e. n/2 odd phases and n/2 even phases. The algorithmic representation of the above discussed odd-even transposition is given below:

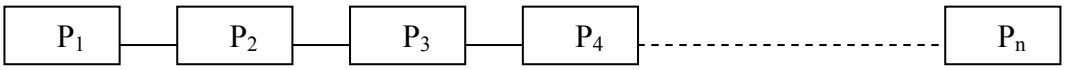


Figure 10: Interconnection network in the form of a Linear Array

Algorithm: Odd-Even Transposition

```
//Input: N numbers that are in the unsorted form
//Assume that element bi is assigned to pi

for I=1 to N
   {
   If (I%2 != 0) //i.e Odd phase
        {
        For j=1,3,5,7,……2*n/2-1
              {
                  Apply compare-exchange(Pj, Pj+1)  //Operation is performed in parallel
              }
        }
   else // Even phase
        {
        For j=2,4,6,8……2*(n-1)/2-1
              {
                  Apply compare-exchange(Pj, Pj+1)  //Operation is performed in parallel
              }
        }
   I++
  }
```

Let us take an example and illustrate the odd-even transposition algorithm (see Figure 11).

Analysis

The above algorithm requires one ‘for loop’ starting from I=1 to N, i.e. N times and for each value of I, one ‘for loop’ of J is executed in parallel. Therefore, the time complexity of the algorithm is O(n) as there are total n phases and each phase performs either odd or even transposition in O(1) time.

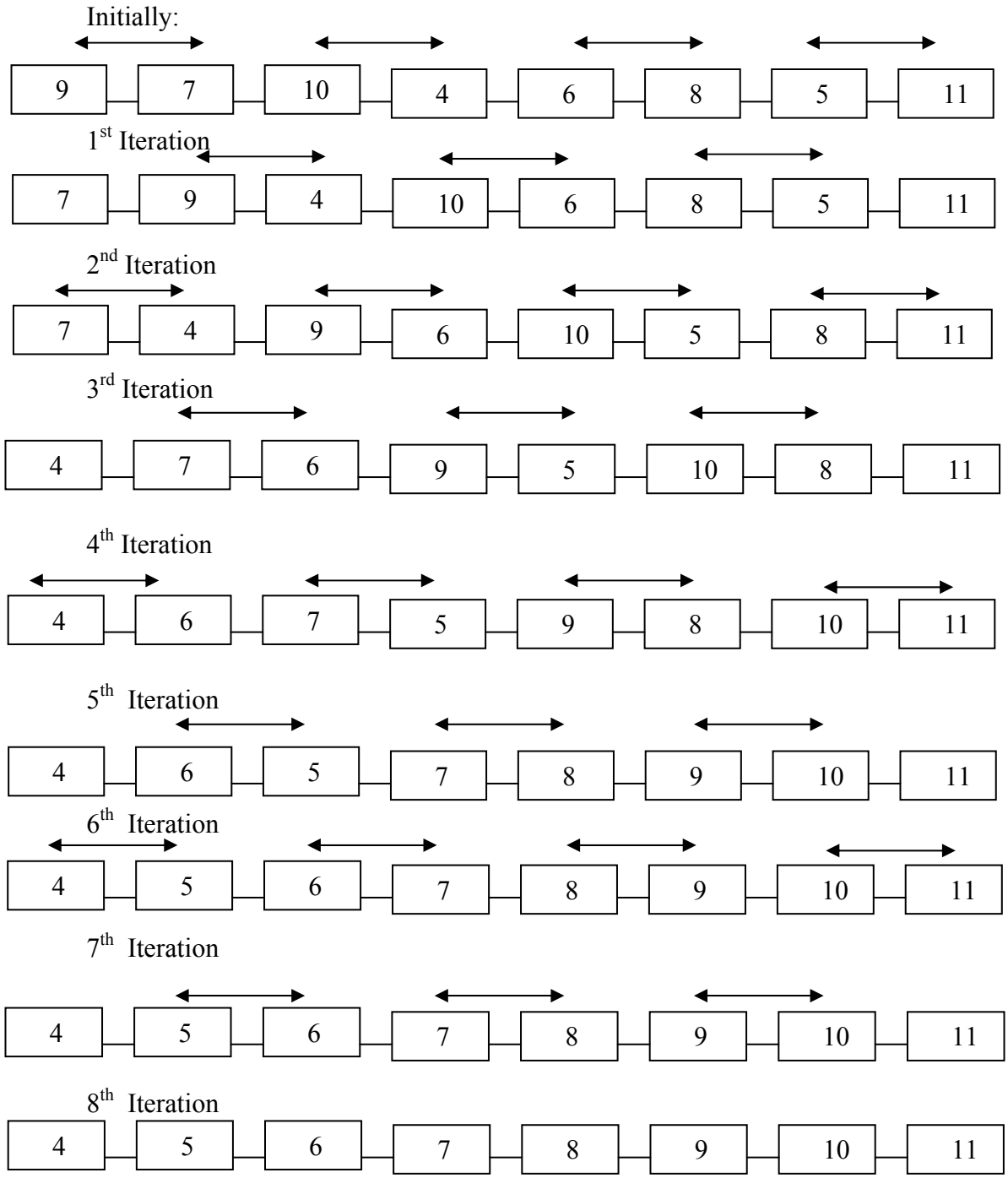


Figure 11: Example

☞ Check Your Progress 2

1) Which circuit has a time complexity of O(n) for sorting the n numbers?

1) Sort-Merge Circuit
2) Interconnection Networks
3) Combinational Circuits
4) None of the above

2) Which circuit has a time complexity of $O(\log n^2)$ for sorting the n numbers?

1) PRAM
2) Interconnection Networks
3) Combinational Circuits
4) Both 1 and 3.

3) Explain the concept of sorting in the combinational circuits

……
……
……
……

1.10 MATRIX COMPUTATION

In the subsequent section, we shall discuss the algorithms for solving the matrix multiplication problem using the parallel models.

Matrix Multiplication Problem

Let there be two matrices, M1 and M2 of sizes a x b and b x c respectively. The product of M1 x M2 is a matrix O of size a x c.
The values of elements stored in the matrix O are according to the following formulae:

$$O_{ij} = \text{Summation x of } (M1_{ix} * M2_{xj}) \text{ x=1 to b, where } 1<i<a \text{ and } 1<j<c.$$

Remember, for multiplying a matrix M1 with another matrix M2, the number of columns in M1 must equal number of rows in M2. The well known matrix multiplication algorithm on sequential computers takes $O(n^3)$ running time. For multiplication of two 2x2, matrices, the algorithm requires 8 multiplication operations and 4 addition operations. Another algorithm called Strassen algorithm has been devised which requires 7 multiplication operations and 14 addition and subtraction operations. The time complexity of Strassen's algorithm is $O(n^{2.81})$. The basic sequential algorithm is discussed below:

Algorithm: Matrix Multiplication

Input// Two Matrices M1 and M2

```
For I=1 to n
    For j=1 to n
     {
          Oij = 0;
          For k=1 to n
              Oij= Oij + M1ik * M2kj
          End For
     }
    End For
End For
```

Now, let us modify the above discussed matrix multiplication algorithm according to parallel computation models.

1.11 CONCURRENTLY READ CONCURRENTLY WRITE (CRCW)

It is one of the models based on PRAM. In this model, the processors access the memory locations concurrently for reading as well as writing operations. In the algorithm, which uses CRCW model of computation, n^3 number of processors are employed. Whenever a concurrent write operation is performed on a specific memory location, say m, than there are chances of occurrence of a conflict. Therefore, the write conflicts i.e. (WR, RW, WW) have been resolved in the following manner. In a situation when more than one processor tries to write on the same memory location, the value stored in the memory location is always the sum of the values computed by the various processors.

Algorithm Matrix Multiplication using CRCW

```
Input// Two Matrices M1 and M2
For I=1 to n          //Operation performed in PARALLEL
    For j=1 to n      //Operation performed in PARALLEL
      For k=1 to n  //Operation performed in PARALLEL
          Oij = 0;
          Oij = M1ik * M2kj
          End For
     End For
End For
```

The complexity of CRCW based algorithm is O(1).

1.12 CONCURRENTLY READ EXCLUSIVELY WRITE (CREW)

It is one of the models based on PRAM. In this model, the processors access the memory location concurrently for reading while exclusively for writing operations. In the algorithm which uses CREW model of computation, n^2 number of processors have been attached in the form of a two dimensional array of size n x n.

Algorithm Matrix Multiplication using CREW

Input// Two Matrices M1 and M2

```
For I=1 to n          //Operation performed in PARALLEL
    For j=1 to n      //Operation performed in PARALLEL
     {
          Oij = 0;
          For k=1 to n
              Oij = Oij + M1ik * M2kj
          End For
    }
    End For
End For
```

The complexity of CREW based algorithm is O(n).

☞ **Check Your Progress 3**

1) Which of the models has a complexity of O(n) for matrix multiplication?
   1) RAM
   2) Interconnection Networks
   3) CRCW
   4) CREW
2) Which of the models has a complexity of O(1) for matrix multiplication?
   1) RAM
   2) Interconnection Networks
   3) CRCW
   4) CREW
3) Explain the algorithm for matrix multiplication in sequential circuits.

……
……
……
…… …… ..

1.13 SUMMARY

In this chapter, we have discussed the various topics pertaining to the art of writing parallel algorithms for various parallel computation models in order to improve the efficiency of a number of numerical as well as non-numerical problem types. In order to evaluate the complexity of parallel algorithms there are mainly three important parameters, which are involved i.e., 1) Time Complexity, 2) Total Number of Processors Required, and 3) Total Cost Involved. Consequently, we have discussed the various computation models for parallel computers, e.g. combinational circuits, interconnection networks, PRAM etc. A combinational circuit can be defined as an arrangement of logic gates with a set of m input lines and a set of n output lines. In the interconnection networks, the N processors can communicate with each other through direct links. In the interconnection networks, each processor has an independent local memory. In the PRAM, it contains n processors, a single shared memory module being shared by all the N processors and which also acts as a communication medium for the processors. In order to connect the N processors with the single shared memory, a component called Memory Access Unit (MAU) is used for accessing the shared memory. Subsequently, we have discussed and applied these models on few numerical problems like sorting and matrix multiplication. In case of sorting, initially a combinational circuit was used for sorting. A bitonic sequence was given as an input to a combinational circuit consisting of a set of comparators interconnected with each other. The complexity of sorting using Combinational Circuit is O ($\log^2 n$). Another famous sorting algorithm known as merge sort based algorithm can also be depicted / solved with the help of the combinational circuit. The complexity of merge-sort using Combinational Circuit is O ($\log^2 n$). The interconnection network can be used for solving the sorting problem known as bubble sort. The interconnection network can be depicted as n processors interconnected with each other in the form of a linear array. The complexity of bubble sort using interconnection network is O (n). The well known matrix multiplication algorithm on sequential computers take O (n^3) running time and strassen algorithm take O($n^{2.81}$). In the present study, we have discussed two models based on PRAM for solving the matrix multiplication problem. In CRCW model, the processors access the memory location concurrently for reading as well as for writing operation. In the algorithm which uses CRCW model of computation, n^3 number of processors are employed. The complexity of CRCW based algorithm is O(1). In CREW model, the processors access the memory

location concurrently for reading while exclusively for writing operation. In the algorithm which uses CREW model of computation, n^2 number of processors have been attached in the form of a two dimensional array of size n x n. The complexity of CREW based algorithm is O(n).

1.14 SOLUTIONS/ANSWERS

☞ **Check Your Progress 1**

1: 1
2: 3
3: 3
4: The fundamental parameters required for the analysis of parallel algorithms are as under:

1. Time Complexity;
2. The Total Number of Processors Required; and
3. The Cost Involved.

☞ **Check Your Progress 2**

1: 2
2: 4
3: Each input line of the combinational circuit represents an individual element of the string say, X_i, and each output line results in the form of a sorted list. In order to achieve the above-mentioned task, a comparator is employed for processing.

☞ **Check Your Progress 3**

1: 4
2: 3
3: The values of elements stored in matrix O are according to the following formulae:

O_{ij} = Summation of ($M1_{ix}$ * $M2_{xj}$) x=1 to b, where 1<i<a and 1<j<c

1.15 REFERENCES/FURTHER READINGS

1) Cormen T. H., *Introduction to Algorithms*, Second Edition, Prentice Hall of India, 2002.

2) Rajaraman V. and Siva Ram Murthy C. *Parallel Computers - Architecture and Programming*, Second Edition, Prentice Hall of India, 2002.

3) Xavier C. and Iyengar S. S. *Introduction to Parallel Algorithm.*

UNIT 2 PRAM ALGORITHMS

Structure **Page Nos.**

2.0 INTRODUCTION

PRAM (Parallel Random Access Machine) model is one of the most popular models for designing parallel algorithms. A PRAM consists of unbounded number of processors interacting with each other through shared memory and a common communication network. There are many ways to implement the PRAM model. We shall discuss three of them in this unit: message passing, shared memory and data parallel. We shall also cover these models and associated data structures here.

A number of languages and routine libraries have been invented to support these models. Some of them are architecture independent and some are specific to particular platforms. We shall introduce two of the widely accepted routine libraries in this unit. These are Message Passing Interface (MPI) and Parallel Virtual Machine (PVM).

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the concepts of message passing programming;
- list the various communication modes for communication among processors;
- explain the concepts of shared programming model;
- describe and use the functions defined in MPI;
- understand and use functions defined in PVM;
- explain the concepts of data parallel programming, and
- learn about different data structures like array, linked list and hypercube.

2.2 MESSAGE PASSING PROGRAMMING

Message passing is probably the most widely used parallel programming paradigm today. It is the most natural, portable and efficient approach for distributed memory systems. It provides natural synchronisation among the processes so that explicit synchronisation of memory access is redundant. The programmer is responsible for determining all parallelism. In this programming model, multiple processes across the arbitrary number

of machines, each with its own local memory, exchange data through *send and receive* communication between processes. This model can be best understood through the diagram shown in *Figure 1*:

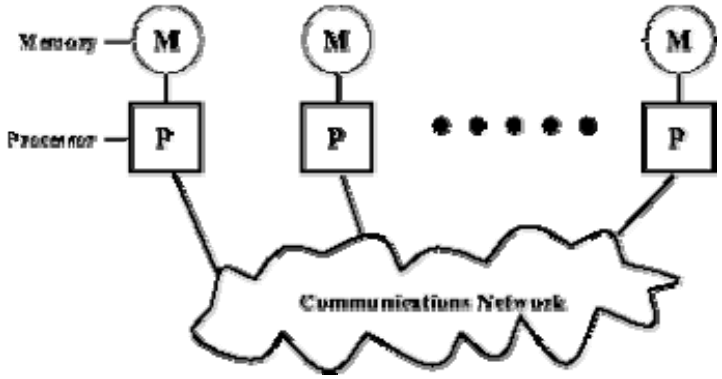


Figure 1: Message passage model

As the diagram indicates, each processor has its own local memory. Processors perform computations with the data in their own memories and interact with the other processors, as and when required, through communication network using message-passing libraries. The message contains the data being sent. But data is not the only constituent of the message. The other components in the message are:

- The identity/address of the processor that sending the message;
- Starting address of the data on the sending processor;
- The type of data being sent;
- The size of data;
- The identity/address of processor(s) are receiving the message, and
- Starting address of storage for the data on the receiving processor.

Once the message has been created it is sent through the communication network. The communication may be in the following two forms:

i) Point-to-point Communication

The simplest form of message is a point-to- point communication. A message is sent from the sending processor to a receiving processor. Message passing in point-to-point communication can be in two modes: synchronous and asynchronous. In synchronous transfer mode, the next message is sent only after the acknowledgement of delivery of the last message. In this mode the sequence of the messages is maintained. In asynchronous transfer mode, no acknowledgement for delivery is required.

ii) Collective Communications

Some message-passing systems allow communication involving more than two processors. Such type of communication may be called collective communication. Collective communication can be in one of these modes:

Barrier: In this mode no actual transfer of data takes place unless all the processors involved in the communication execute a particular block, called barrier block, in their message passing program.

Broadcast: Broadcasting may be one-to-all or all-to-all. In one-to-all broadcasting, one processor sends the same message to several destinations with a single operation whereas

in all-to-all broadcasting, communication takes place in many-to-many fashion. The messages may be personalised or non-personalized. In a personalized broadcasting, unique messages are being sent to every destination processor.

Reduction: In reductions, one member of the group collects data from the other members, reduces them to a single data item which is usually made available to all of the participating processors.

Merits of Message Passage Programming

- Provides excellent low-level control of parallelism;
- Portable;
- Minimal overhead in parallel synchronisation and data distribution; and
- It is less error prone.

Drawbacks

- Message-passing code generally requires more software overhead than parallel shared-memory code.

2.1.1 Shared Memory

In shared memory approach, more focus is on the control parallelism instead of data parallelism. In this model, multiple processes run independently on different processors, but they share a common address space accessible to all as shown in *Figure 2*.

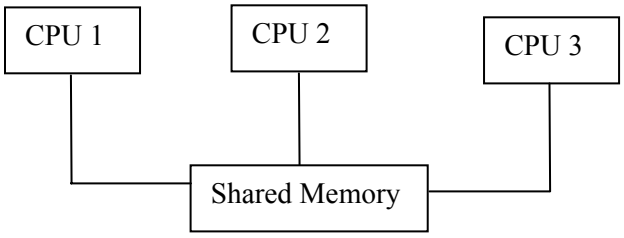


Figure 2: Shared memory

The processors communicate with one another by one processor writing data into a location in memory and another processor reading the data. Any change in a memory location effected by one processor is visible to all other processors. As shared data can be accessed by more than one processes at the same time, some control mechanism such as locks/ semaphores should be devised to ensure mutual exclusion. This model is often referred to as SMP (Symmetric Multi Processors), named so because a common symmetric implementation is used for several processors of the same type to access the same shared memory. A number of multi-processor systems implement a shared-memory programming model; examples include: NEC SX-5, SGI Power Onyx/ Origin 2000; Hewlett-Packard V2600/HyperPlex; SUN HPC 10000 400 MHz ;DELL PowerEdge 8450.

Shared memory programming has been implemented in various ways. We are introducing some of them here.

Thread libraries

The most typical representatives of shared memory programming models are thread libraries present in most of the modern operating systems. Examples for thread libraries

are, *POSIX threads* as implemented in Linux, *SolarisTM threads* for solaris , *Win32 threads* available in Windows NT and Windows 2000 , and *JavaTM threads* as part of the standard JavaTM Development Kit (JDK).

Distributed Shared Memory Systems

Distributed Shared Memory (DSM) systems emulate a shared memory abstraction on loosely coupled architectures in order to enable shared memory programming despite missing hardware support. They are mostly implemented in the form of standard libraries and exploit the advanced user–level memory management features present in modern operating systems. Examples include Tread Marks System, IVY, Munin, Brazos, Shasta, and Cashmere.

Program Annotation Packages

A quite renowned approach in this area is OpenMP, a newly developed industry standard for shared memory programming on architectures with uniform memory access characteristics. OpenMP is based on functional parallelism and focuses mostly on the parallelisation of loops. OpenMP implementations use a special compiler to evaluate the annotations in the application's source code and to transform the code into an explicitly parallel code, which can then be executed. We shall have a detailed discussion on OpenMP in the next unit.

Shared memory approach provides low-level control of shared memory system, but they tend to be tedious and error prone. They are more suitable for system programming than to application programming.

Merits of Shared Memory Programming

- Global address space provides a user-friendly programming perspective to memory.
- Data sharing between processes is both fast and uniform due to the proximity of memory to CPUs.
- No need to specify explicitly the communication of data between processes.
- Negligible process-communication overhead.
- More intuitive and easier to learn.

Drawbacks

- Not portable.
- Difficult to manage data locality.
- Scalability is limited by the number of access pathways to memory.
- User is responsible for specifying synchronization, e.g., locks.

2.1.2 Message Passing Libraries

In this section, we shall discuss about message passing libraries. Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications. We shall discuss only two worldwide accepted message passing libraries namely; MPI and PVM.

Message Passing Interface (MPI)

The **Message Passing Interface** (**MPI**) is a universal standard for providing communication among the multiple concurrent processes on a distributed memory system. Most, if not all, of the popular parallel computing platforms offer at least one implementation of MPI. It was developed by the MPI forum consisting of several experts

from industry and academics. MPI has been implemented as the library of routines that can be called from languages like, Fortran, C, C++ and Ada programs. MPI was developed in two stages, MPI-1 and MPI-2. MPI-1 was published in 1994.

Features of MPI-1

- Point-to-point communication,
- Collective communication,
- Process groups and communication domains,
- Virtual process topologies, and
- Binding for Fortran and C.

Features added in MPI-2

- Dynamic process management,
- Input/output,
- One-sided operations for remote memory access, and
- Binding for C++.

MPI's advantage over older message passing libraries is that it is both portable (because MPI has been implemented for almost every distributed memory architecture) and fast (because each implementation is optimized for the hardware it runs on).

Building and Running MPI Programs

MPI parallel programs are written using conventional languages like, Fortran and C .One or more header files such as "mpi.h" may be required to provide the necessary definitions and declarations. Like any other serial program, programs using MPI need to be compiled first before running the program. The command to compile the program may vary according to the compiler being used. If we are using *mpcc* compiler, then we can compile a C program named "program.c" using the following command:

mpcc program.c -o program.o

Most implementations provide command, typically named *mpirun* for spawning MPI processes. It provides facilities for the user to select number of processes and which processors they will run on. For example to run the object file "program" as n processes on n processors we use the following command:

mpirun program -np n

MPI functions

MPI includes hundreds of functions, a small subset of which is sufficient for most practical purposes. We shall discuss some of them in this unit.

Functions for MPI Environment:

int MPI_Init (int *argc, char ** argv)

It initializes the MPI environment. No MPI function can be called before MPI_Init.

int MPI_Finalize (void)

It terminates the MPI environment. No MPI function can be called after MPI_Finalize.

Every MPI process belongs to one or more groups (also called communicator). Each process is identified by its rank (0 to group size –1) within the given group. Initially, all

processes belong to a default group called MPI_COMM_WORLD group. Additional groups can be created by the user as and when required. Now, we shall learn some functions related to communicators.

int MPI_Comm_size (MPI_Comm comm, int *size)

returns variable *size* that contains number of processes in group *comm*.

int MPI_Comm_rank (MPI_Comm comm, int *rank)

returns **rank** of calling process in group *comm*.

Functions for Message Passing:

MPI processes do not share memory space and one process cannot directly access other process's variables. Hence, they need some form of communication among themselves. In MPI environment this communication is in the form of message passing. A message in MPI contains the following fields:

msgaddr: It can be any address in the sender's address space and refers to location in memory where message data begins.

count: Number of occurrences of data items of message datatype contained in message.

datatype: Type of data in message. This field is important in the sense that MPI supports heterogeneous computing and the different nodes may interpret *count* field differently. For example, if the message contains a strings of 2n characters (count =2n), some machines may interpret it having 2n characters and some having n characters depending upon the storage allocated per character (1 or 2). The basic datatype in MPI include all basic types in Fortran and C with two additional types namely MPI_BYTE and MPI_PACKED. MPI_BYTE indicates a byte of 8 bits .

source or *dest* : Rank of sending or receiving process in communicator.

tag: Identifier for specific message or type of message. It allows the programmer to deal with the arrival of message in an orderly way, even if the arrival of the message is not orderly.

comm.: Communicator. It is an object wrapping context and group of a process .It is allocated by system instead of user.

The functions used for messaging passing are:

int MPI_Send(void *msgaddr, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm.)
on return, *msg* can be reused immediately.

int MPI_Recv(void *msgaddr, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm.)
on return, *msg* contains requested message.

MPI message passing may be either point-to-point or collective.

Point-to-point Message Passing

In point-to-point message passing, one process sends/receives message to/from another process. There are four communication modes for sending a message:

i) **Buffered mode:** Send can be initiated whether or not matching receive has been initiated, and send may complete before matching receive is initiated.

ii) **Synchronous mode:** Send can be initiated whether or not matching receive has been initiated, but send will complete only after matching receive has been initiated.

iii) **Ready mode:** Send can be initiated only if matching receive has already been initiated.

iv) **Standard mode:** May behave like either buffered mode or synchronous mode, depending on specific implementation of MPI and availability of memory for buffer space.

MPI provides both blocking and non-blocking send and receive operations for all modes.

Functions for various communication modes

Mode	Blocking	Non-Blocking
Standard	*MPI_Send*	*MPI_Isend*
Buffered	*MPI_Bsend*	*MPI_Ibsend*
Synchronus	*MPI_Ssend*	*MPI_Issend*
Ready	*MPI_Rsend*	*MPI_Irsend*

MPI_Recv and *MPI_Irecv* are blocking and nonblocking functions for receiving messages, regardless of mode.

Besides send and receive functions, MPI provides some more useful functions for communications. Some of them are being introduced here.

MPI_Buffer_attach used to provide buffer space for buffered mode. Nonblocking functions include *request* argument used subsequently to determine whether requested operation has completed.

MPI_Wait and *MPI_Test* wait or test for completion of nonblocking communication.

MPI_Probe and *MPI_Iprobe* probe for incoming message without actually receiving it. Information about message determined by probing can be used to decide how to receive it.

MPI_Cancel cancels outstanding message request, useful for cleanup at the end of a program or after major phase of computation.

Collective Message Passing

In collective message passing, all the processes of a group participate in communication. MPI provides a number of functions to implement the collective message passing. Some of them are being discussed here.

MPI_Bcast(msgaddr, count, datatype, rank, comm):

This function is used by a process ranked *rank* in group *comm* to broadcast the message to all the members (including self) in the group.

MPI_Allreduce

***MPI_Scatter*(Sendaddr, Scount, Sdatatype, Receiveaddr, Rcount, Rdatatype, Rank, Comm):**

Using this function process with rank rank *in group comm sends personalized message to all the processes (including self) and sorted message (according to the rank of sending processes) are stored in the send buffer of the process. First three parameters define buffer of sending process and next three define buffer of receiving process.*

MPI_Gather (Sendaddr, Scount, Sdatatype, Receiveaddr, Rcount, Rdatatype,Rank, Comm):
Using this function process with rank rank in group comm receives personalized message from all the processes (including self) and sorted message (according to the rank of sending processes) are stored in the receive buffer of the process. First three parameters define buffer of sending process and next three define buffer of receiving process.

MPI_Alltoall()
Each process sends a personalized message to every other process in the group.

MPI_Reduce (Sendaddr , Receiveaddr , count, datatype, op, rank, comm):

This function reduces the partial values stored in Sendaddr of each process into a final result and stores it in Receiveaddr of the process with rank rank. op specifies the reduction operator**.**

***MPI_Scan** (Sendaddr,, Receiveaddr , count, datatype, op, comm):*
It combines the partial values into p final results which are received into the Receiveaddr of all p processes in the group comm.

MPI_Barrier(cmm):
This function synchronises all processes in the group comm.

Timing in MPI program

MPI_Wtime ( ) returns elapsed wall-clock time in seconds since some arbitrary point in past. Elapsed time for program segment is given by the difference between *MPI_Wtime* values at beginning and end of process. Process clocks are not necessarily synchronised, so clock values are not necessarily comparable across the processes, and care must be taken in determining overall running time for parallel program. Even if clocks are explicitly synchronised, variation across clocks still cannot be expected to be significantly less than round-trip time for zero-length message between the processes.

Now, we shall illustrate use of these functions with an example.

Example 1:

```
#include <mpi.h>
int main(int argc, char **argv) {
int i, tmp, sum, s, r, N, x[100];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &s);
MPI_Comm_rank(MPI_COMM_WORLD, &r);
If(r==0)
 {
  printf( “Enter N:”);
  scanf (“%d”, &N);
  for (i=1; i<s; i++)
     MPI_Send(&N, 1, MPI_INT,i, i, MPI_COMM_WORLD);
  for (i=r, i<N; i=i+s)
     sum+= x[i];
```

```
 for (i=r, i<s; i++)
 {
   MPI_Recv(&tmp, 1, MPI_INT,i, i,  MPI_COMM_WORLD, &status);
   Sum+=tmp;
}
printf( "%d", sum);
}
else {
        MPI_Recv(&N, 1, MPI_INT,0, i, MPI_COMM_WORLD, &status);
        for (i=r, i<N; i=i+s)
             sum+= x[i];
        MPI_Send(&sum, 1, MPI_INT, 0, i, MPI_COMM_WORLD);
}
MPI_Finalize( );
}
```

Merits of MPI

- Runs on either shared or distributed memory architectures;
- Can be used on a wider range of problems than OpenMP;
- Each process has its own local variables; and
- Distributed memory computers are less expensive than large shared memory computers.

Drawbacks of MPI

- Requires more programming changes to go from serial to parallel version.
- Can be harder to debug, and
- Performance is limited by the communication network between the nodes.

Parallel Virtual Machine (PVM)

PVM (Parallel Virtual Machine) is a portable message-passing programming system, designed to link separate heterogeneous host machines to form a "virtual machine" which is a single, manageable parallel computing resource. Large computational problems such as molecular dynamics simulations, superconductivity studies, distributed fractal computations, matrix algorithms, can thus be solved more cost effectively by using the aggregate power and memory of many computers.

PVM was developed by the University of Tennessee, The Oak Ridge National Laboratory and Emory University. The first version was released in 1989, version 2 was released in 1991 and finally version 3 was released in 1993. The PVM software enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. It transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures. The programming interface of PVM is very simple .The user writes his application as a collection of cooperating tasks. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronisation between the tasks. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast and barrier synchronization.

Features of PVM:

- Easy to install;

- Easy to configure;
- Multiple users can each use PVM simultaneously;
- Multiple applications from one user can execute;
- C, C++, and Fortran supported;
- Package is small;
- Users can select the set of machines for a given run of a PVM program;
- Process-based computation;
- Explicit message-passing model, and
- Heterogeneity support.

When the PVM is starts it examines the virtual machine in which it is to operate, and creates a process called the PVM demon, or simply pvmd on each machine. An example of a daemon program is the mail program that runs in the background and handles all the incoming and outgoing electronic mail on a computer, pvmd provides inter-host point of contact, authenticates task and executes processes on machines. It also provides the fault detection, routes messages not from or intended for its host, transmits messages from its application to a destination, receives messages from the other pvmd's, and buffers it until the destination application can handle it.

PVM provides a library of functions, libpvm3.a, that the application programmer calls. Each function has some particular effect in the PVM. However, all this library really provides is a convenient way of asking the local pvmd to perform some work. The pvmd then acts as the virtual machine. Both pvmd and PVM library constitute the PVM system.

The PVM system supports functional as well as data decomposition model of parallel programming. It binds with C, C++, and Fortran . The C and C++ language bindings for the PVM user interface library are implemented as functions (subroutines in case of FORTRAN) . User programs written in C and C++ can access PVM library by linking the library libpvm3.a (libfpvm3.a in case of FORTRAN).

All PVM tasks are uniquely identified by an integer called *task identifier* (TID) assigned by local pvmd. Messages are sent to and received from tids. PVM contains several routines that return TID values so that the user application can identify other tasks in the system. PVM also supports grouping of tasks. A task may belong to more than one group and one task from one group can communicate with the task in the other groups. To use any of the group functions, a program must be linked with `libgpvm3.a.`

Set up to Use PVM

PVM uses two environment variables when starting and running. Each PVM user needs to set these two variables to use PVM. The first variable is PVM_ROOT, which is set to the location of the installed `pvm3` directory. The second variable is PVM_ARCH , which tells PVM the architecture of this host. The easiest method is to set these two variables in your `.`cshrc file. Here is an example for setting PVM_ROOT:

```
setenv PVM_ROOT $HOME/pvm3
```

The user can set PVM_ARCH by concatenating to the file `.cshrc`, the content of file `$PVM_ROOT/lib/cshrc.stub`.

Starting PVM

To start PVM, on any host on which PVM has been installed we can type

```
% pvm
```

The PVM console, called `pvm`, is a stand-alone PVM task that allows the user to interactively start, query, and modify the virtual machine. Then we can add hosts to virtual machine by typing at the console prompt (got after last command)

```
pvm> add hostname
```

To delete hosts (except the one we are using) from virtual machine we can type

```
pvm> delete hostname
```

We can see the configuration of the present virtual machine, we can type

```
pvm> conf
```

To see what PVM tasks are running on the virtual machine, we should type

```
pvm> ps -a
```

To close the virtual machine environment, we should type

```
pvm> halt
```

Multiple hosts can be added simultaneously by typing the hostnames in a file one per line and then type

```
% pvm hostfile
```

PVM will then add all the listed hosts simultaneously before the console prompt appears.

Compiling and Running the PVM Program

Now, we shall learn how to compile and run PVM programs. To compile the program , change to the directory pvm/lib/archname where archname is the architecture name of your computer. Then the following command:

```
cc  program.c -lpvm3 -o prgram
```

will compile a program called program.c. After compiling, we must put the executable file in the directory pvm3/bin/ARCH. Also, we need to compile the program separately for every architecture in virtual machine. In case we use dynamic groups, we should also add -lgpvm3 to the compile command. The executable file can then be run. To do this, first run PVM. After PVM is running, executable file may be run from the unix command line, like any other program.

PVM supplies an architecture-independent make, `aimk`, that automatically determines PVM_ARCH and links any operating system specific libraries to your application. To compile the C example, type

```
% aimk master.c
```

Now, from one window, start PVM and configure some hosts. In another window change directory to `$HOME/pvm3/bin/PVM_ARCH` and type

```
% master
```

It will ask for a number of tasks to be executed. Then type the number of tasks.

Programming with PVM

The general method for writing a program with PVM is as follows:
A user writes one or more sequential programs in C, C++, or Fortran 77 containing embedded PVM function (or subroutine) calls. Each program corresponds to a task making up the application. These programs are compiled for each architecture in the host pool, and the resulting object files are placed at a location accessible from machines in the host pool. To execute an application, a user typically starts one copy of one task (usually the "master" or "initiating" task) by hand from a machine within the host pool. This process subsequently starts other PVM tasks, eventually resulting in a collection of active tasks that then compute locally and exchange messages with each other to solve the problem.

PVM library routines

In this section we shall give a brief description of the routines in the PVM 3 user library. Every PVM program should include the PVM header file "`pvm3.h`" (in a C program) or "fpvm3.h" (in a Fortran program).

In PVM 3, all PVM tasks are identified by an integer supplied by the local pvmd. In the following descriptions this task identifier is called TID. Now we are introducing some commonly used functions in PVM programming (as in C. For Fortran, we use prefix pvmf against pvm in C).

Process Management

- int **pvm_mytid(void)**

Returns the *tid* of the calling process. **tid** values less than zero indicate an error.

- **int pvm_exit(void)**

Tells the local pvmd that this process is leaving PVM. **info** Integer status code returned by the routine. Values less than zero indicate an error.

- **pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)**

start new PVM processes. **task**, a character string is the executable file name of the PVM process to be started. The executable must already reside on the host on which it is to be started. **Argv is a** pointer to an array of arguments to **task**. If the executable needs no arguments, then the second argument to pvm_spawn is NULL. **flag** Integer specifies spawn options. **where** , a character string specifying where to start the PVM process. If *flag* is 0, then *where* is ignored and PVM will select the most appropriate host. **ntask** ,an integer, specifies the number of copies of the executable to start. **tids** ,Integer array of length *ntask* returns the tids of the PVM processes started by this pvm_spawn call. The function returns the actual number of processes returned. Negative values indicate error.

- **int pvm_kill(int tid)**

Terminates a specified PVM process. **tid** Integer task identifier of the PVM process to be killed (not itself). Return values less than zero indicate an error.

- **int pvm_catchout(FILE *ff)**

Catch output from child tasks. **ff** is file descriptor on which we write the collected output. The default is to have the PVM write the *stderr* and *stdout* of spawned tasks.

Information

- **int pvm_parent(void)**

Returns the tid of the process that spawned the calling process.

- **int pvm_tidtohost(tid)**

Returns the host of the specified PVM process. Error if negative value is returned.

- **int pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp)**

```
struct pvmhostinfo {
int hi_tid;
char *hi_name;
char *hi_arch;
int hi_speed;
};
```

Returns information about the present virtual machine configuration. **nhost** is the number of hosts (pvmds) in the virtual machine. **narch** is the number of different data formats being used. **hostp** is pointer to an array of structures which contains the information about each host including its pvmd task ID, name, architecture, and relative speed(default is 1000).

- **int info = pvm_tasks(int where, int *ntask, struct pvmtaskinfo **taskp)**

```
struct pvmtaskinfo {
    int ti_tid; int ti_ptid;
    int ti_host;
    int ti_flag; char *ti_a_out; } taskp;
```

Returns the information about the tasks running on the virtual machine. **where** specifies what tasks to return the information about. The options are:

0
    for all the tasks on the virtual machine

pvmd tid
    for all tasks on a given host

tid
    for a specific task

ntask returns the number of tasks being reported on.

taskp is a pointer to an array of structures which contains the information about each task including its task ID, parent tid, pvmd task ID, status flag, and the name of this task's executable file. The status flag values are: waiting for a message, waiting for the pvmd, and running.

Dynamic Configuration

- **int pvm_addhosts(char **hosts, int nhost, int *infos)**
  Add hosts to the virtual machine. **hosts** is an array of strings naming the hosts to be added. **nhost** specifies the length of array hosts. **infos** is an array of length *nhost* which returns the status for each host. Values less than zero indicate an error, while positive values are TIDs of the new hosts.
  Signaling

- **int pvm_sendsig(int tid, int signum**)

Sends a signal to another PVM process. **tid** is task identifier of PVM process to receive the signal. **signum** is the signal number.

- **int info = pvm_notify(int what, int msgtag, int cnt, int *tids)**

Request notification of PVM event such as host failure. **What** specifies the type of event to trigger the notification. Some of them are:

PvmTaskExit
: Task exits or is killed.

PvmHostDelete
: Host is deleted or crashes.

PvmHostAdd
: New host is added.

msgtag is message tag to be used in notification. **cnt** For *PvmTaskExit* and *PvmHostDelete*, specifies the length of the *tids* array. For *PvmHostAdd* specifies the number of times to notify.

tids for *PvmTaskExit* and *PvmHostDelete* is an array of length *cnt* of task or pvmd TIDs to be notified about. The array is not used with the *PvmHostAdd* option.

Message Passing

The PVM communication model provides asynchronous blocking send, asynchronous blocking receive, and nonblocking receive functions. In our terminology, a blocking send returns as soon as the send buffer is free for reuse, and an asynchronous send does not depend on the receiver calling a matching receive before the send can return. There are options in PVM 3 that request that data be transferred directly from task to task. In this case, if the message is large, the sender may block until the receiver has called a matching receive.

A nonblocking receive immediately returns with either the data or a flag that the data has not arrived, while a blocking receive returns only when the data is in the receive buffer. In addition to these point-to-point communication functions, the model supports the multicast to a set of tasks and the broadcast to a user-defined group of tasks. There are also functions to perform global max, global sum, etc. across a user-defined group of tasks. Wildcards can be specified in the receive for the source and the label, allowing either or both of these contexts to be ignored. A routine can be called to return the information about the received messages.

The PVM model guarantees that the message order is preserved. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both the messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

- **int bufid = pvm_mkbuf(int encoding)**

creates a new message buffer. **encoding** specifying the buffer's encoding scheme.

Encoding data options

PVM uses SUN's XDR library to create a machine independent data format if you request it. Settings for the encoding option are:

PvmDataDefault: Use XDR by default, as the local library cannot know in advance where you are going to send the data.

PvmDataRaw: No encoding, so make sure you are sending to a similar machine.

PvmDataInPlace: Not only is there no encoding, but the data is not even going to be physically copied into the buffer.

- **int info = pvm_freebuf(int bufid)**

Disposes of a message buffer. **bufid** message buffer identifier.

- **int pvm_getsbuf(void)**

returns the active send buffer identifier.

- **int pvm_getrbuf(void)**

returns the active receive buffer identifier.

- **int pvm_setsbuf(int bufid)**

sets the active send buffer to bufid, saves the state of the previous buffer, and returns the previous active buffer identifier.

- **int pvm_setrbuf(int bufid)**

sets the active receive buffer to bufid, saves the state of the previous buffer, and returns the previous active buffer identifier.

- **intpvm_initsend(int encoding)**

Clear default sends buffer and specifies the message encoding. **Encoding** specifies the next message's encoding scheme.

- **int pvm_send(int tid, int msgtag)**

Immediately sends the data in the active message buffer. **tid** Integer task identifier of destination process. **msgtag** Integer message tag supplied by the user. msgtag should be nonnegative.

- **int pvm_recv(int tid, int msgtag)**

Receive a message. **tid** is integer task identifier of sending process supplied by the user and **msgtag** is the message tag supplied by the user(should be non negative integer). The process returns the value of the new active receive buffer identifier. Values less than zero indicate an error. It blocks the process until a message with label *msgtag* has arrived from *tid*. pvm_recv then places the message in a new *active* receive buffer, which also clears the current receive buffer.

Packing and Unpacking Data

- pvm_packs - Pack the active message buffer with arrays of prescribed data type:

- **int info = pvm_packf(const char *fmt, ...)**
- **int info = pvm_pkbyte(char *xp, int nitem, int stride)**
- **int info = pvm_pkcplx(float *cp, int nitem, int stride)**
- **int info = pvm_pkdcplx(double *zp, int nitem, int stride)**
- **int info = pvm_pkdouble(double *dp, int nitem, int stride)**
- **int info = pvm_pkfloat(float *fp, int nitem, int stride)**
- **int info = pvm_pkint(int *ip, int nitem, int stride)**
- **int info = pvm_pkuint(unsigned int *ip, int nitem, int stride)**
- **int info = pvm_pkushort(unsigned short *ip, int nitem, int stride)**
- **int info = pvm_pkulong(unsigned long *ip, int nitem, int stride)**
- **int info = pvm_pklong(long *ip, int nitem, int stride)**
- **int info = pvm_pkshort(short *jp, int nitem, int stride)**
- **int info = pvm_pkstr(char *sp)**

fmt Printf-like format expression specifying what to pack. **nitem** is the total number of *items* to be packed (not the number of bytes). **stride** is the stride to be used when packing the items.

- pvm_unpack - Unpacks the active message buffer into arrays of prescribed data type. It has been implemented for different data types:

- **int info = pvm_unpackf(const char *fmt, ...)**
- **int info = pvm_upkbyte(char *xp, int nitem, int stride)**
- **int info = pvm_upkcplx(float *cp, int nitem, int stride)**
- **int info = pvm_upkdcplx(double *zp, int nitem, int stride)**
- **int info = pvm_upkdouble(double *dp, int nitem, int stride)**
- **int info = pvm_upkfloat(float *fp, int nitem, int stride)**
- **int info = pvm_upkint(int *ip, int nitem, int stride)**
- **int info = pvm_upkuint(unsigned int *ip, int nitem, int stride)**
- **int info = pvm_upkushort(unsigned short *ip, int nitem, int stride)**
- **int info = pvm_upkulong(unsigned long *ip, int nitem, int stride)**
- **int info = pvm_upklong(long *ip, int nitem, int stride)**
- **int info = pvm_upkshort(short *jp, int nitem, int stride)**
- **int info = pvm_upkstr(char *sp)**

Each of the pvm_upk* routines unpacks an array of the given data type from the active receive buffer. The arguments for each of the routines are a pointer to the array to be unpacked into, *nitem* which is the total number of items to unpack, and *stride* which is the stride to use when unpacking. An exception is pvm_upkstr() which by definition unpacks a NULL terminated character string and thus does not need *nitem* or *stride* arguments.

Dynamic Process Groups

To create and manage dynamic groups, a separate library `libgpvm3.a` must be linked with the user programs that make use of any of the group functions. Group management work is handled by a group server that is automatically started when the first group function is invoked. Any PVM task can join or leave any group dynamically at any time without having to inform any other task in the affected groups. Tasks can broadcast messages to groups of which they are not members. Now we are giving some routines that handle dynamic processes:

- **int pvm_joingroup(char *group)**

Enrolls the calling process in a named group. **group** is a group name of an existing group. Returns instance number.Instance numbers run from 0 to the number of group members minus 1. In PVM 3, a task can join multiple groups. If a process leaves a group and then rejoins it, that process may receive a different instance number.

- **int info = pvm_lvgroup(char *group)**

Unenrolls the calling process from a named group.

- **int pvm_gettid(char *group, int inum)**

Returns the tid of the process identified by a group name and instance number.

- **int pvm_getinst(char *group, int tid)**

Returns the instance number in a group of a PVM process.

- **int size = pvm_gsize(char *group)**

Returns the number of members presently in the named group.

- **int pvm_barrier(char *group, int count)**

Blocks the calling process until all the processes in a group have called it. **count** species the number of group members that must call pvm_barrier before they are all released.

- **int pvm_bcast(char *group, int msgtag)**

Broadcasts the data in the active message buffer to a group of processes. **msgtag** is a message tag supplied by the user. It allows the user's program to distinguish between different kinds of messages .It should be a nonnegative integer.

- **int info = pvm_reduce(void (*func)(), void *data, int count, int datatype, int msgtag, char *group, int rootginst)**

Performs a reduce operation over members of the specified group. **func** is function defining the operation performed on the global data. Predefined are PvmMax, PvmMin, PvmSum and PvmProduct. Users can define their own function. **data** is pointer to the starting address of an array of local values. **count** species the number of elements of *datatype* in the data array. **Datatype** is the type of the entries in the data array. **msgtag** is the message tag supplied by the user. msgtag should be greater than zero. It allows the user's program to distinguish between different kinds of messages. **group** is the group name of an existing group. **rootginst** is the instance number of group member who gets the result.

We are writing here a program that illustrates the use of these functions in the parallel programming:

Example 2: Hello.c

```
#include "pvm3.h"
main()
{
   int cc, tid, msgtag;
   char buf[100];

   printf("%x\n", pvm_mytid());

   cc = pvm_spawn("hello_other", (char**)0, 0, "", 1,
&tid);

   if (cc == 1) {
        msgtag = 1;
      pvm_recv(tid, msgtag);
      pvm_upkstr(buf);
      printf("from t%x: %s\n", tid, buf);
   } else
      printf("can't start hello_other\n");

   pvm_exit();
}
```

In this program, pvm_mytid( ), returns the TID of the running program (In this case, task id of the program hello.c). This program is intended to be invoked manually; after

printing its task id (obtained with `pvm_mytid()`), it initiates a copy of another program called *hello_other* using the `pvm_spawn()` function. A successful spawn causes the program to execute a blocking receive using `pvm_recv`. After receiving the message, the program prints the message sent by its counterpart, as well its task id; the buffer is extracted from the message using `pvm_upkstr`. The final `pvm_exit` call dissociates the program from the PVM system.

hello_other.c

```
#include "pvm3.h"

main()
{
   int ptid, msgtag;
   char buf[100];

   ptid = pvm_parent();

   strcpy(buf, "hello, world from ");
   gethostname(buf + strlen(buf), 64);
    msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(ptid, msgtag);

   pvm_exit();
}
```

Program is a listing of the ``slave'' or spawned program; its first PVM action is to obtain the task id of the ``master'' using the `pvm_parent` call. This program then obtains its hostname and transmits it to the master using the three-call sequence - `pvm_initsend` to initialize the send buffer; `pvm_pkstr` to place a string, in a strongly typed and architecture-independent manner, into the send buffer; and `pvm_send` to transmit it to the destination process specified by *ptid*, ``tagging'' the message with the number1.

☞ **Check Your Progress 1**

1) Write a program to give a listing of the “slave” or spawned program.

……
……
……
……
……

2.2.3 Data Parallel Programming

In the data parallel programming model, major focus is on performing simultaneous operations on a data set. The data set is typically organized into a common structure, such as an array or hypercube. Processors work collectively on the same data structure. However, each task works on a different partition of the same data structure. It is *more restrictive* because not all algorithms can be specified in the data-parallel terms. For these reasons, data parallelism, although important, is not a universal parallel programming paradigm.

Programming with the data parallel model is usually accomplished by writing a program with the data parallel constructs. The constructs can be called to a data parallel subroutine

library or compiler directives. Data parallel languages provide facilities to specify the data decomposition and mapping to the processors. The languages include data distribution statements, which allow the programmer to control which data goes on what processor to minimize the amount of communication between the processors. Directives indicate how arrays are to be aligned and distributed over the processors and hence specify agglomeration and mapping. Communication operations are not specified explicitly by the programmer, but are instead inferred by the compiler from the program. Data parallel languages are more suitable for SIMD architecture though some languages for MIMD structure have also been implemented. Data parallel approach is more effective for highly regular problems, but are not very effective for irregular problems.

The main languages used for this are Fortran 90, High Performance Fortran (HPF) and HPC++. We shall discuss HPF in detail in the next unit. Now, we shall give a brief overview of some of the early data parallel languages:

- ***Computational Fluid Dynamics:*** CFD was a FORTRAN-like language developed in the early 70s at the Computational Fluid Dynamics Branch of Ames Research Center for ILLIAC IV machines, a SIMD computer for array processing. The language design was extremely pragmatic. No attempt was made to hide the hardware peculiarities from the user; in fact, every attempt was made to give the programmers the access and control of all of the hardware to help constructing efficient programs. This language made the architectural features of the ILLIAC IV very apparent to the programmer, but it also contained the seeds of some practical programming language abstractions for data-parallel programming. In spite of its simplicity and ad hoc machine-dependencies, CFD allowed the researchers at Ames to develop a range of application programs that efficiently used the ILLIAC IV.

- ***Connection Machine Fortran:*** Connection Machine Fortran was a later SIMD language developed by Thinking Machines Corporation. Connection Machine Fortran included all of FORTRAN 77, together with the new array syntax of Fortran 90. It added various machine specific features, but unlike CFD or DAP FORTRAN these appeared as *compiler directives* rather than special syntax in Fortran declarations or executable statements. A major improvement over the previous languages was that, distributed array dimensions were no longer constrained to exactly fit in the size of the processing element array; the compiler could transparently map dimensions of arbitrary extent across the available processor grid dimensions. Finally the language added an explicitly parallel looping construct called FORALL. Although CM Fortran looked syntactically like standard Fortran, the programmer had to be aware of many nuances. Like the ILLIAC IV, the Connection

Machine allowed the Fortran arrays to either be distributed across the processing nodes (called *CM arrays*, or distributed arrays), or allocated in the memory of the front-end computer (called *front-end arrays*, or sequential arrays). Unlike the control unit of the ILLIAC, the Connection Machine front-end was a conventional, general-purpose computer--typically a VAX or Sun. But there were still significant restrictions on how arrays could be manipulated, reflecting the two possible homes.

Glypnir ,IVTRAN and *LISP are some of the other early data parallel languages.

Let us conclude this unit with the introduction of a typical data parallel programming style called **SPMD.**

Single Program Multiple Date

A common style of writing data parallel programs for MIMD computers is SPMD (single program, multiple data): all the processors execute the same program, but each operates

on a different portion of problem data. It is easier to program than true MIMD, but more flexible than SIMD. Although most parallel computers today are MIMD architecturally, they are usually programmed in SPMD style. In this style, although there is no central controller, the worker nodes carry on doing *essentially* the same thing at *essentially* the same time. Instead of central copies of control variables stored on the control processor of a SIMD computer, control variables (iteration counts and so on) are usually stored in a replicated fashion across MIMD nodes. Each node has its own local copy of these global control variables, but every node updates them in an identical way. There are no centrally issued parallel instructions, but communications usually happen in the well-defined collective phases. These data exchanges occur in a prefixed manner that explicitly or implicitly synchronize the peer nodes. The situation is something like an orchestra without a conductor. There is no central control, but each individual plays from the same script. The group as a whole stays in lockstep. This loosely synchronous style has some similarities to the Bulk Synchronous Parallel (BSP) model of computing introduced by the theorist Les Valiant in the early 1990s. The restricted pattern of the collective synchronization is easier to deal with than the complex synchronisation problems of a general concurrent programming.

A natural assumption was that it should be possible and not too difficult to capture the SPMD model for programming MIMD computers in data-parallel languages, along lines similar to the successful SIMD languages. Various research prototype languages attempted to do this, with some success. By the 90s the value of portable, standarised programming languages was universally recognized, and there seemed to be some consensus about what a standard language for SPMD programming ought to look like. Then the High Performance Fortran (HPF) standard was introduced.

2.3 DATA STRUCTURES FOR PARALLEL ALGORITHMS

To implement any algorithm, selection of a proper data structure is very important. A particular operation may be performed with a data structure in a smaller time but it may require a very large time in some other data structure. For example, to access i^{th} element in a set may need constant time if we are using arrays but the required time becomes a polynomial in case of a linked list. So, the selection of data structure must be done keeping in mind the type of operation to be performed and the architecture available. In this section, we shall introduce some data structures commonly used in a parallel programming.

2.3.1 Linked List

A linked list is a data structure composed of zero or more nodes linked by pointers. Each node consists of two parts, as shown in *Figure 3*: *info* field containing specific information and *next* field containing address of next node. First node is pointed by an external pointer called *head*. Last node called tail node does not contain address of any node. Hence, its next field points to null. Linked list with zero nodes is called null linked list.

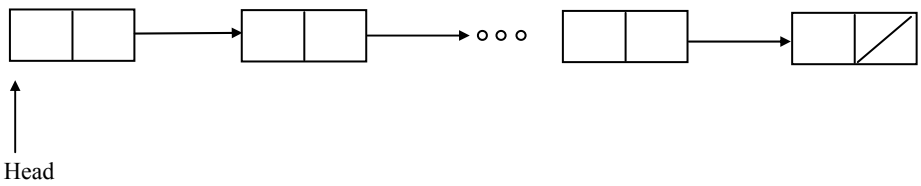


Figure 3: Linked List

A large number of operations can be performed using the linked list. For some of the operations like insertion or deletion of the new data, linked list takes constant time, but it is time consuming for some other operations like searching a data. We are giving here an example where linked list is used:

Example 3:
Given a linear linked list, rank the list elements in terms of the distance from each to the last element.

A parallel algorithm for this problem is given here. The algorithm assumes there are p number of processors.

Algorithm:

```
Processor j, 0≤ j<p, do
  if next[j]=j then
    rank[j]=0
   else rank[j] =1
 endif
 while rank[next[first]]≠0 Processor j, 0≤ j<p, do
      rank[j]=rank[j]+rank[next[j]]
      next[j]=next[next[j]]
 endwhile
```

The working of this algorithm is illustrated by the following diagram:

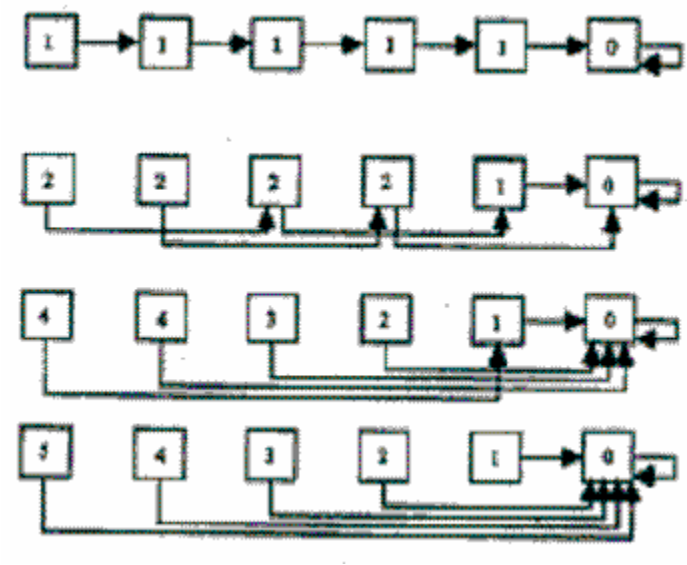


Figure 4 : Finding rank of elements

2.3.2 Arrays Pointers

An array is a collection of the similar type of data. Arrays are very popular data structures in parallel programming due to their easiness of declaration and use. At the one hand, arrays can be used as a common memory resource for the shared memory programming, on the other hand they can be easily partitioned into sub-arrays for data parallel programming. This is the flexibility of the arrays that makes them most

frequently used data structure in parallel programming. We shall study arrays in the context of two languages Fortran 90 and C.

Consider the array shown below. The size of the array is 10.

5	10	15	20	25	30	35	40	45	50

Index of the first element in Fortran 90 is 1 but that in C is 0 and consequently the index of the last element in Fortran 90 is 10 and that in C is 9. If we assign the name of array as A, then i^{th} element in Fortran 90 is A(i) but in C it is A[i-1]. Arrays may be one-dimensional or they may be multi-dimensional.

General form of declaration of array in Fortran 90 is

```
type, DIMENSION(bound) [,attr] :: name
```

for example the declaration

```
INTEGER, DIMENSION(5): A
```

declare an array A of size 5.

General form of declaration of array in C is

type array_name [size]

For example the declaration A

int A[10]

declares an array of size 10.

Fortran 90 allows one to use particular sections of an array. To access a section of an array, you need the name of the array followed by the two integer values separated by a colon enclosed in the parentheses. The integer values represent the indices of the section required.
For example, a(3:5) refers to elements 3, 4, 5 of the array, a(1:5:2) refers to elements 1, 3, 5 of the array , and b(1:3, 2:4) refers to the elements from rows 1 to 3 and columns 2 to 4. In C there is only one kind of array whose size is determined statically, though there are provisions for dynamic allocation of storage through pointers and dynamic memory allocation functions like *calloc* and *malloc* functions. In Fortran 90, there are 3 possible types of arrays depending on the binding of an array to an amount of storage : *Static arrays* with fixed size at the time of declaration and cannot be altered during execution ; *Semi-dynamic arrays* or *automatic arrays*: the size is determined after entering a subroutine and arrays can be created to match the exact size required, but local to a subroutine ; and *Dynamic arrays* or *allocatable arrays* : the size can be altered during execution.

In these languages, array operations are written in a compact form that often makes programs more readable.
Consider the loop:

```
s=0
do i=1,n
   a(i)=b(i)+c(i)
    s=s+a(i)
 end do
```

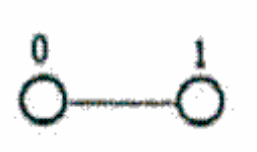
It can be written (in Fortran 90 notation) as follows:

```
a(1:n) = b(1:n) +c(1:n)
s=sum(a(1:n))
```

In addition to Fortran 90 , there are many languages that provide succinct operations on arrays. Some of the most popular are APL, and MATLAB. Although these languages were not developed for parallel computing, rather for expressiveness, they can be used to express parallelism since array operations can be easily executed in parallel. Thus, all the arithmetic operations (+, -, * /, **) involved in a vector expression can be performed in parallel. Intrinsic reduction functions, such as the sum above, also can be performed in a parallel .

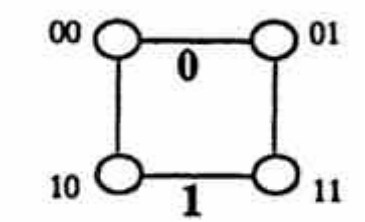
2.3.3 Hypercube Network

The hypercube architecture has played an important role in the development of parallel processing and is still quite popular and influential. The highly symmetric recursive structure of the hypercube supports a variety of elegant and efficient parallel algorithms. Hypercubes are also called n-cubes, where n indicates the number of dimensions. An-cube can be defined recursively as depicted below:



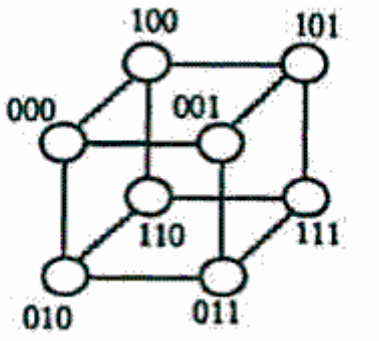
1-cube built of 2 0-cubes

Figure 5(a): 1-cube



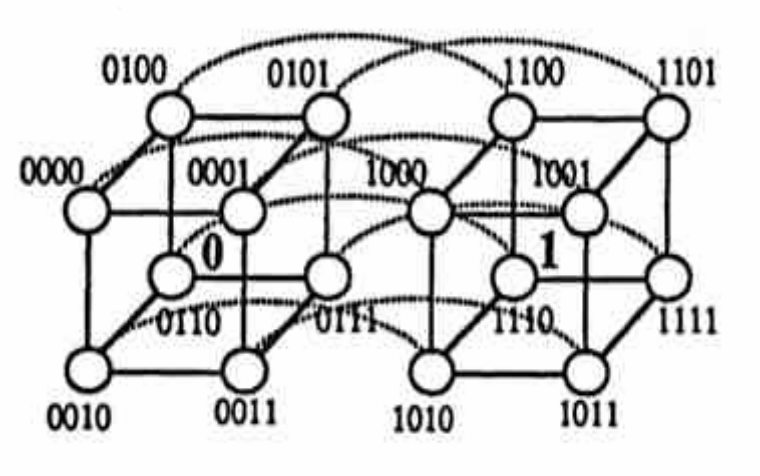
2-cube built of 2 1-cubes

Figure 5(b): 2-cube



3-cube built of 2 2-cubes

Figure 5(c): 3-cube



4-cube built of 2 3-cubes

Figure 5(d): 4-cube

Properties of Hypercube:

- A node p in a n-cube has a unique label, its binary ID, that is a n-bit binary number.
- The labels of any two neighboring nodes differ in exactly 1 bit.
- Two nodes whose labels differ in *k* bits are connected by a shortest path of length *k*.
- Hypercube is both node- and edge- symmetric.

Hypercube structure can be used to implement many parallel algorithms requiring all-to-all communication, that is, algorithms in which each task must communicate with every other task. This structure allows a computation requiring all-to-all communication among P tasks to be performed in just log P steps compared t polynomial time using other data structures like arrays and linked lists.

2.4 SUMMARY

In this unit, a number of concepts have been introduced in context designing of algorithms for PRAM model of parallel computation. The concepts introduced include message passing programming, data parallel programming, message passing interface (MPI), and parallel virtual machine. Also, topics relating to modes of communication between processors, the functions defined in MPI and PVM are discussed in sufficient details.

2.5 SOLUTIONS/ANSWERS

☞ **Check Your Progress 1**

1) ***hello_other.c***

```
#include "pvm3.h"

main()
{
   int ptid, msgtag;
   char buf[100];

   ptid = pvm_parent();

   strcpy(buf, "hello, world from ");
   gethostname(buf + strlen(buf), 64);
    msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(ptid, msgtag);

   pvm_exit();
}
```

Program is a listing of the “slave” or spawned program; its first PVM action is to obtain the task id of the ``master'' using the `pvm_parent` call. This program then obtains its hostname and transmits it to the master using the three-call sequence - `pvm_initsend` to initialize the send buffer; `pvm_pkstr` to place a string, in a strongly typed and

architecture-independent manner, into the send buffer; and `pvm_send` to transmit it to the destination process specified by *ptid*, ``tagging'' the message with the number1.

2.6 REFERENCES/FURTHER READINGS

1) Kai Hwang: *Advanced Computer Architecture*: Parallelism, Scalability, Programmability (2001), Tata McGraw Hill, 2001.

2) Henessy J. L. and Patterson D. A. *Computer Architecture*: A Qualititative Approach, Morgan Kaufman (1990)

3) Rajaraman V. and Shive Ram Murthy C. *Parallel Computer: Architecture and Programming*: Prentice Hall of India

4) Salim G. *Parallel Computation, Models and Methods*: Akl Prentice Hall of India

UNIT 3 PARALLEL PROGRAMMING

3.0 INTRODUCTION

After getting a great breakthrough in the serial programming and figuring out its limitations, computer professionals and academicians are focusing now on parallel programming. Parallel programming is a popular choice today for multi-processor architectures to solve the complex problems. If developments in the last decade give any indications, then the future belongs to of parallel computing. Parallel programming is intended to take advantages the of non-local resources to save time and cost and overcome the memory constraints.

In this section, we shall introduce parallel programming and its classifications. We shall discuss some of the high level programs used for parallel programming. There are certain compiler-directive based packages, which can be used along with some high level languages. We shall also have a detailed look upon them.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the basics of parallel programming;
- describe the parallel programming based on message passing;
- learn programming using High Performance Fortran, and
- learn compiler directives of OpenMP.

3.2 INTRODUCTION TO PARALLEL PROGRAMMING

Traditionally, a software has been written for serial computation in which programs are written for computers having a single Central Processing Unit (CPU). Here, the problems are solved by a series of instructions, executed one after the other, one at a time, by the CPU. However, many complex, interrelated events happening at the same time like planetary and galactic orbital events, weather and ocean patterns and tectonic plate drift may require super high complexity serial software. To solve these large problems and save the computational time, a new programming paradigm called parallel programming was introduced.

To develop a parallel program, we must first determine whether the problem has some part which can be parallelised. There are some problems like generating the Fibonacci Sequence in the case of which there is a little scope for parallelization. Once it has been determined that the problem has some segment that can be parallelized, we break the problem into discrete chunks of work that can be distributed to multiple tasks. This partition of the problem may be data-centric or function-centric. In the former case, different functions work with different subsets of data while in the latter each function performs a portion of the overall work. Depending upon the type of partition approach, we require communication among the processes. Accordingly, we have to design the mode of communication and mechanisms for process synchronization.

3.3 TYPES OF PARALLEL PROGRAMMING

There are several parallel programming models in common use. Some of these are:

- Message Passing;
- Data Parallel programming;
- Shared Memory; and
- Hybrid.

In the last unit, we had a brief introduction about these programming models. In this unit we shall discuss the programming aspects of these models.

3.3.1 Programming Based on Message Passing

As we know, the programming model based on message passing uses high level programming languages like C/C++ along with some message passing libraries like MPI and PVM. We had discussed MPI and PVM at great length in unit 2 on PRAM algorithms. Hence we are restricting ourselves to an example program of message passing.

Example 1: Addition of array elements using two processors.

In this problem, we have to find the sum of all the elements of an array A of size n. We shall divide n elements into two groups of roughly equal size. The first [n/2] elements are added by the first processor, P_0, and last [n/2] elements the by second processor, P_1. The two sums then are added to get the final result. The program is given below:

Program for P_0

```
#include <mpi.h>
#define n 100
int main(int argc, char **argv) {
int A[n];
int sum0 =0, sum1=0,sum;
MPI_Init(&argc, &argv);
for( int i=0;i<n;i++)
scanf("%d", &A[i]);
MPI_Send( &n/2, n/2, MPI_INT,1, 0, MPI_COMM_WORLD);
for(i=1; i<n/2; i++)
 sum0+=A[i];
sum1=MPI_Recv(&n/2, n/2, MPI_INT,1, 0, MPI_COMM_WORLD);
sum=sum0+sum1;
printf("%d",sum);
MPI_Finalize();
}
```

Program for P_1,

```
int func( int B[int n])
{
MPI_Recv(&n/2, n/2, MPI_INT,0, 0, MPI_COMM_WORLD);
 int sum1=0 ;
for (i=0; i</2; i++)
  sum1+=B[i];
MPI_Send( 0, n/2, MPI_INT,0, 0, MPI_COMM_WORLD);
}
```

☞ **Check Your Progress 1**

1) Enumerate at least five applications of parallel programming.
……
……
……
……

2) What are the different steps to write a general parallel program?
……
……
……
……

3) Write a program to find the sum of the elements of an array using k processors.
……
……
……
……

3.3.2 Programming Based on Data Parallelism

In a data parallel programming model, the focus is on data distribution. Each processor works with a portion of data. In the last unit we introduced some data parallel languages. Now, we shall discuss one of the most popular languages for parallel programming based on data parallel model.

High Performance FORTRAN

In 1993 the *High Performance FORTRAN Forum*, a group of many leading hardware and software vendors and academicians in the field of parallel processing, established an informal language standard called *High Performance Fortran* (HPF). It was based on Fortran 90, then it extended the set of parallel features, and provided extensive support for computation on *distributed memory* parallel computers. The standard was supported by a majority of vendors of parallel hardware.

HPF is a highly suitable language for data parallel programming models on MIMD and SIMD architecture. It allows programmers to add a number of compiler directives that minimize inter-process communication overhead and utilize the load-balancing techniques.

We shall not discuss the complete HPF here, rather we shall focus only on augmenting features like:

- Processor Arrangements,
- Data Distribution,
- Data Alignment,
- FORALL Statement,
- INDEPENDENT loops, and
- Intrinsic Functions.

3.3.2.1 Processor Arrangements

It is a very frequent event in data parallel programming to group a number of processors to perform specific tasks. To achieve this goal, HPF provides a directive called *PROCESSORS* directive. This directive declares a conceptual processor grid. In other words, the PROCESSORS directive is used to specify the shape and size of an array of abstract processors. These do not have to be of the same shape as the underlying hardware.The syntax of a PROCESSORS directive is:

!HPF$ PROCESSORS array_name (dim1, dim 2, ….dim n)

where array_name is collective name of abstract processors. dim i specifies the size of i^{th} dimension of array_name.

Example 2:

!HPF$ PROCESSORS P (10)
This introduces a set of 10 abstract processors, assigning them the collective name `P`.

!HPF$ PROCESSORS Q (4, 4)
It introduces 16 abstract processors in a 4 by 4 array.

3.3.2.2 Data Distribution

Data distribution directives tell the compiler how the program data is to be distributed amongst the memory areas associated with a set of processors. The logic used for data distribution is that if a set of data has independent sub-blocks, then computation on them can be carried out in parallel. They do not allow the programmer to state directly which processor will perform a particular computation. But it is expected that if the operands of

a particular sub-computation are all found on the same processor, the compiler will allocate that part of the computation to the processor holding the operands, whereupon no remote memory accesses will be involved.

Having seen how to define one or more target processor arrangements, we need to introduce mechanisms for distributing the data arrays over those arrangements. The DISTRIBUTE directive is used to distribute a data object) onto an abstract processor array.

The syntax of a DISTRIBUTE directive is:

!HPF$ DISTRIBUTE array_lists [ONTO arrayp]

where array_list is the list of array to be distributed and arrayp is abstract processor array.

The ONTO specifier can be used to perform a distribution across a particular processor array. If no processor array is specified, one is chosen by the compiler.

HPF allows arrays to be distributed over the processors directly, but it is often more convenient to go through the intermediary of an explicit *template*. A template can be declared in much the same way as a processor arrangement.

```
!HPF$ TEMPLATE T(50, 50, 50)
```

declares a 50 by 50 by 50 three-dimensional template called T. Having declared it, we can establish a relation between a template and some processor arrangement by using DISTRIBUTE directive. There are three ways in which a template may be distributed over Processors: *Block, cyclic and **.

(a) Block Distribution

Simple block distribution is specified by

!HPF$ DISTRIBUTE T1(BLOCK) ONTO P1
where T1 is some template and P1 is some processor arrangement.
In this case, each processor gets a contiguous block of template elements. All processors get the same sized block. The last processor may get lesser sized block.

Example 3:

```
!HPF$ PROCESSORS P1(4)
!HPF$ TEMPLATE T1(18)
!HPF$ DISTRIBUTE T1(BLOCK) ONTO P1
```

As a result of these instructions, distribution of data will be as shown in *Figure 1*.

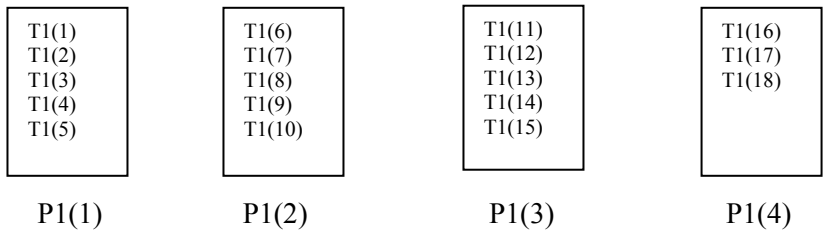


Figure 1: Block Distribution of Data

In a variant of the block distribution, the number of template elements allocated to each processor can be explicitly specified, as in
!HPF$ DISTRIBUTE T1 (BLOCK (6)) ONTO P1

Distribution of data will be as shown in *Figure 2*.

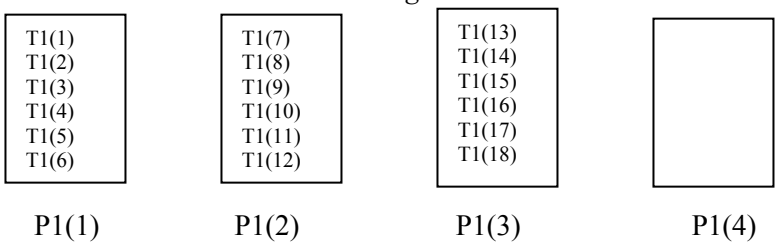


Figure 2: Variation of Block Distribution

It means that we allocate all template elements before exhausting processors, some processors are left empty.

(b) Cyclic Distribution

Simple cyclic distribution is specified by

!HPF$ DISTRIBUTE T1(CYCLIC) ONTO P1

The first processor gets the first template element, the second gets the second, and so on. When the set of processors is exhausted, go back to the first processor, and continue allocating the template elements from there.

Example 4

!HPF$ PROCESSORS P1(4)
!HPF$ TEMPLATE T1(18)
!HPF$ DISTRIBUTE T1(CYCLIC) ONTO P1

The result of these instructions is shown in *Figure 3*.

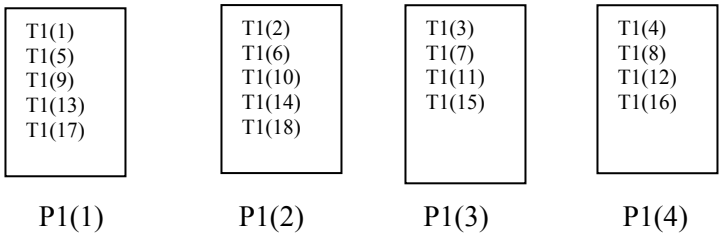


Figure 3: Cyclic Distribution

But in an analogous variant of the cyclic distribution
!HPF$ DISTRIBUTE T1 (BLOCK (3)) ONTO P1

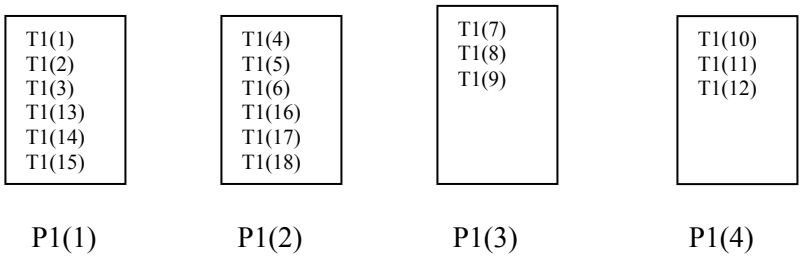


Figure 4: Variation of Cyclic Distribution

That covers the case where both the template and the processor are one dimensional. When the and processor have (the same) higher dimension, each dimension can be distributed independently, mixing any of the four distribution formats. The correspondence between the template and the processor dimension is the obvious one. In
!HPF$ PROCESSORS P2 (4, 3)
!HPF$ TEMPLATE T2 (17, 20)
!HPF$ DISTRIBUTE T2 (CYCLIC, BLOCK) ONTO P2
the first dimension of T2 is distributed cyclically over the first dimension of P2; the second dimension is distributed blockwise over the second dimension of P2.

*(c) * Distribution*

Some dimensions of a template may have ``collapsed distributions'', allowing a template to be distributed onto a processor arrangement with fewer dimensions than the template.

Example 5

```
!HPF$ PROCESSORS P2 (4, 3)
!HPF$ TEMPLATE T2 (17, 20)
!HPF$ DISTRIBUTE T2 (BLOCK, *) ONTO P1
```

means that the first dimension of `T2` will be distributed over `P1` in blockwise order but for a fixed value of the first index of `T2`, all values of the second subscript are mapped to the same processor.

3.3.2.3 Data Alignment

Arrays are aligned to templates through the ALIGN directive. The `ALIGN` directive is used to align elements of different arrays with each other, indicating that they should be distributed in the same manner. The syntax of an ALIGN derivative is:

!HPF$ ALIGN array1 WITH array2

where array1 is the name of array to be aligned and array2 is the array to be aligned to.

Example 6

Consider the statement
ALIGN A[i] WITH B[i]

This statement aligns each A[i] with B[i] as shown in *Figure 5*.

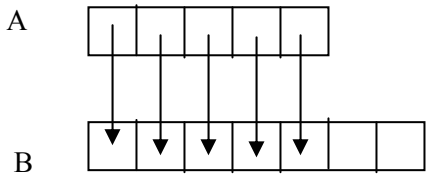


Figure 5: ALIGN A[i] WITH B[i]

Consider the statement

ALLIGN A[i] WITH B[i+1]

This statement aligns the each A[i] with B[i+1] as shown in *Figure 6*.

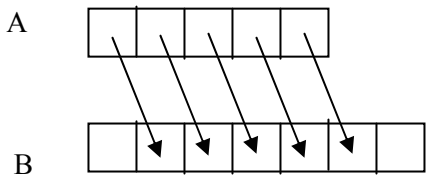


Figure 6: ALIGN A[i] WITH B[i+1]

* can be used to collapse dimensions. Consider the statement

ALLIGN A[:, *] WITH B[:]

This statement aligns two dimensional array with one dimensional array by collapsing as shown in *Figure 7*.

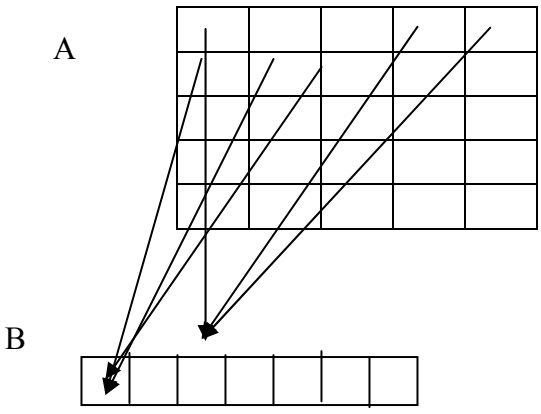


Figure 7: * alignment

3.3.2.4 The FORALL Statement

The FORALL statement allows for more general assignments to sections of an array. A FORALL statement has the general form.

FORALL (*triplet*, ..., *triplet*, *mask*)
statement

where *triplet* has the general form

subscript = *lower*: *upper* : *step-size*

and specifies a set of indices. step-size is optional. *statement* is an arithmetic or pointer assignment and the assignment statement is evaluated for those index values specified by the list of triplets that are not rejected by the optional *mask*.

Example 7 The following statements set each element of matrix `X` to the sum of its indices.

FORALL (i=1:m, j=1:n) X(i,j) = i+j

and the following statement sets the upper right triangle of matrix `Y to` zero .

FORALL (i=1:n, j=1:n, i<j) Y(i,j) = 0.0

Multi-statement FORALL construct:
Multi-statement FORALL is shorthand for the series of single statement FORALLs. The syntax for FORALL is

FORALL (index-spec-list [,mask])
Body
END FORALL

Nesting of FORALL is allowed.

Example 8
Let a= [2,4,6,8,10], b=[1,3,5,7,9], c=[0,0,0,0,0]
Consider the following program segment

```
FORALL (i = 2:4)
  a(i) = a(i-1)+a(i+1)
  c(i) = b(i) *a(i+1).
END FORALL
```

The computation will be

```
  a[2] =a[1]+a[3] =2+6=8
  a[3] =a[2]+a[4] =4+8=12
  a[4] =a[3]+a[5] = 6+10=16
  c[2] = b[2] *a[3] = 3*12=36
  c[3] = b[3] *a[4] = 5*16=80
  c[4] = b[4] *a[5] =7*10=70
```

Thus output is
a= [2,8,12,16,10], b=[1,3,5,7,9], c=[0,36,80,70,0]

3.3.2.5 INDEPENDENT Loops

HPF provides additional opportunities for parallel execution by using the INDEPENDENT directive to assert that the iterations of a do-loop can be performed independently---that is, in any order or concurrently---without affecting the result . In effect, this directive changes a do-loop from an implicitly parallel construct to an explicitly parallel construct.

The INDEPENDENT directive must immediately precede the do-loop to which it applies. In its simplest form, it has no additional argument and asserts simply that no iteration of the do-loop can affect any other iteration.

Example 9

In the following code fragment, the directives indicate that the outer two loops are independent. The inner loop assigns the elements of `A` repeatedly and hence it is not independent.

```
!HPF$   INDEPENDENT

do i=1,n1  ! Loop over  i independent

  !HPF$   INDEPENDENT

  do j=1,n2      ! Loop over  j independent

    do k=1,n3          ! Inner loop not independent

      A(i,j) = A(i,j) + B(i,j,k)*C(i,j)

          enddo

  enddo

enddo
```

3.3.2.6 Intrinsic Functions

HPF introduces some new intrinsic functions in addition to those defined in F90. The two most frequently used in parallel programming are the system inquiry functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE. These functions provide the information about the *number* of physical processors on which the running program executes and processor configuration. General syntax of is

NUMBER_OF_PROCESSORS is

NUMBER_OF_PROCESSORS (dim)

where `dim` is an optional argument. It returns the number of processors in the underlying array or, if the optional argument is present, the size of this array along a specified dimension.

General syntax of PROCESSORS_SHAPE is

PROCESSORS_SHAPE()

It returns an one-dimensional array, whose i^{th} element gives the size of the underlying processor array in its i^{th} dimension.

Example 10

Consider the call of the two intrinsic functions discussed above for a 32-Processor (4×8) Multicomputer:

The function call NUMBER_OF_PROCESORS () will return 32.
The function call NUMBER_OF_PROCESORS (1) will return 4.
The function call NUMBER_OF_PROCESORS (2) will return 8.
The function call PROCESSORS_SHAPE () will return an array with two elements 4 and 8.

We can use these intrinsic functions in tandem with array declarations and HPF directives, to provide flexibility to the programmer to declare abstract processor arrays that match available physical resources. For example, the following statement !HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS()) declares an abstract processor array `P` with size equal to the number of physical processors.

☞ Check Your Progress 2

1) Give the output of the following instructions in HPF:
   (a) !HPF$ PROCESSORS Q (s, r)
   (b) !HPF$ PROCESSORS P1(5)
   !HPF$ TEMPLATE T1(22)
   !HPF$ DISTRIBUTE T1(CYCLIC) ONTO P1.

2) Write a FORALL statement to set lower triangle of a matrix X to zero.
   ……
   ……
   ……
   ……

3) What are intrinsic functions? Name any two of them.
   ……
   ……
   ……
   ……

3.3.3 Shared Memory Programming

As discussed in unit 2, we know that all processors share a common memory in shared memory model. Each processor, however, can be assigned a different part of the program stored in the memory to execute with the data stored in specified locations. Each processor does computation independently with the data allocated to them by the controlling program, called the main program. After finishing their computations, all of these processors join the main program. The main program finishes only after all child processes have been terminated completely. There are many alternatives to implement these ideas through high level programming. Some of them are:

i) Using heavy weight processes.
ii) Using threads.(e.g. Pthreads).
iii) Using a completely new programming language for parallel programming (e.g. Ada).
iv) Using library routines with an existing sequential programming language.
v) Modifying the syntax of an existing sequential programming language to create a parallel programming language (e.g. UPC).

(vi) Using an existing sequential programming language supplemented with the compiler directives for specifying parallelism (e.g. OpenMP).

We shall adopt the last alternative. We shall also give an introduction to the shared programming using library routines with an existing sequential programming language.

3.3.3.1 OpenMP

OpenMP is a compiler directive based standard developed in the late 1990s jointly by a group of major computer hardware and software vendors. It is portable across many popular platforms including Unix and Windows NT platforms. The OpenMP Fortran API was released on October 28, 1997 and the C/C++ API was released in late 1998. We shall discuss only about C/C++ API.

The OpenMP API uses the fork-join model of parallel execution. As soon as an OpenMP program starts executing it creates a single thread of execution, called the initial thread. The initial thread executes sequentially. As soon as it gets a *parallel* construct, the thread creates additional threads and works as the master thread for all threads. All of the new threads execute the code inside the *parallel* construct. Only the master thread continues execution of the user code beyond the end of the *parallel* construct. There is no restriction on the number of *parallel* constructs in a single program. When a thread with its child threads encounters a work-sharing construct, the work inside the construct is divided among the members of the team and executed co-operatively instead of being executed by every thread. Execution of the code by every thread in the team resumes after the end of the work-sharing construct. Synchronization constructs and the library routines are available in OpenMP to co-ordinate threads and data in *parallel* and work-sharing constructs.

Each OpenMP directive starts with *#pragma omp*. The general syntax is

#pragma omp directive-name [Set of clauses]

where *omp* is an OpenMP keyword. There may be additional clauses (parameters) after the directive name for different options.

Now, we shall discuss about some compiler directives in OpenMP.

(i) Parallel Construct

The syntax of the *parallel* construct is as follows:

#pragma omp parallel [set of clauses]
where *clause* is one of the following:
structured-block
if(scalar-expression)
private(list)
firstprivate(list)
default(shared | none)
shared(list)
copyin(list)

When a thread encounters a *parallel* construct, a set of new threads is created to execute the *parallel* region. Within a parallel region each thread has a unique thread number. Thread number of the master thread is zero. Thread number of a thread can be obtained by

the call of library function *omp_get_thread_num***.** Now, we are giving the description of the clauses used in a parallel construct.

(a) Private Clause:

This clause declares one or more list items to be private to a thread. The syntax of the *private* clause is

private(list).

(b) Firstprivate Clause:

The *firstprivate* clause declares one or more list items to be private to a thread, and initializes each of them with the value that the corresponding original item has when the construct is encountered. The syntax of the *firstprivate* clause is as follows:

*firstprivate***(***list***).**

(c) Shared Clause:

The *shared* clause declares one or more list items to be shared among all the threads in a team. The syntax of the *shared* clause is :

*shared***(***list***)**

(d) Copyin Clause:

The *copyin* clause provides a mechanism to copy the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region. The syntax of the *copyin* clause is :

*copyin***(***list***)**

(ii) Work-Sharing Constructs

A work-sharing construct distributes the execution of the associated region among the members of the team that encounters it. A work-sharing construct does not launch new threads.

OpenMP defines three work-sharing constructs: *sections, for, and single*.
In all of these constructs, there is an implicit barrier at the end of the construct unless a *nowait* clause is included.

(a) Sections

The *sections* construct is a no iterative work-sharing construct that causes the structured blocks to be shared among the threads in team. Each structured block is executed once by one of the threads in the team. The syntax of the *sections* construct is:

```
#pragma omp sections [set of clauses.]
{
 #pragma omp section
  structured-bloc
#pragma omp section
structured-block
.
.
.
}
```

The *clause* is one of the following:
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait

(i) Lastprivate Clause

The *lastprivate* clause declares one or more list items to be private to a thread, and causes the corresponding original list item to be updated after the end of the region. The syntax of the *lastprivate* clause is:
Lastprivate (*list*)

(ii) Reduction Clause

The *reduction* clause specifies an operator and one or more list items. For each list item, a private copy is created on each thread, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator. The syntax of the *reduction* clause is :

reduction (*operator***:***list*)

(b) For Loop Construct

The loop construct causes the for loop to be divided into parts and parts shared among threads in the team. The syntax of the loop construct is :
#*pragma omp for [set of clauses.]*
for-loop

The *clause* is one of the following:
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)

(c) Single Construct

The *single* construct specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The other threads in the team do not execute the block, and wait at an implicit barrier at the end of the *single* construct, unless a *nowait* clause is specified.The syntax of the *single* construct is as follows:

#pragma omp single [set of clauses]
structured-block

The *clause* is one of the following:
private(list)
firstprivate(list)
copyprivate(list)
nowait

(iii) Combined Parallel Work-sharing Constructs

Combined parallel work-sharing constructs are shortcuts for specifying a work-sharing construct nested immediately inside a *parallel* construct. The combined parallel work-sharing constructs allow certain clauses which are permitted on both *parallel* constructs and on work-sharing constructs. OpenMP specifies the two combined parallel work-sharing constructs: *parallel loop* construct, and *parallel sections* construct.

(a) Parallel Loop Construct

The parallel loop construct is a shortcut for specifying a *parallel* construct containing one loop construct and no other statements. The syntax of the parallel loop construct is :
#pragma omp parallel for [set of clauses]
for-loop

(a) Parallel Sections Construct

The *parallel sections* construct is a shortcut for specifying a *parallel* construct containing one *sections* construct and no other statements. The syntax of the *parallel sections* construct is:
#pragma omp parallel sections [set of clauses]
{
[#pragma omp section]
structured-block
[#pragma omp section
structured-block]
...
}
In the following example, routines *xaxis*, *yaxis*, and *zaxis* can be executed concurrently. The first *section* directive is optional. Note that all the *section* directives need to appear in the *parallel sections* construct.

(iv) Master Construct

The master directive has the following general form:

#pragma omp master
structured_block

It causes the master thread to execute the structured block. Other threads encountering this directive will ignore it and the associated structured block, and will move on. In the example, the master keeps track of how many iterations have been executed and prints out a progress report. The other threads skip the master region without waiting.

(v) Critical Directive

The *critical* directive allows one thread execute the associated structured block. When one or more threads reach the critical directive, they will wait until no other thread is executing the same critical section (one with the same name), and then one thread will proceed to execute the structured block. The syntax of the critical directive is

#pragma omp critical [name]
structured_block

name is optional. All critical sections with no name are considered to be one undefined name.

(vi) Barrier Directive

The syntax of the barrier directive is
#pragma omp barrier
When a thread reaches the barrier it waits until all threads have reached the barrier and then they all proceed together. There are restrictions on the placement of barrier directive in a program. The *barrier* directive may only be placed in the program at a position where ignoring or deleting the directive would result in a program with correct syntax.

(vii) Atomic Directive

The *atomic* directive ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The syntax of atomic directive is:

#pragma omp atomic
expression_statement

The atomic directive implements a critical section efficiently when the critical section simply updates a variable by arithmetic operation defined by expression_statement.

(viii) Ordered directive

This directive is used in conjunction with *for* and *parallel for* directives to cause an iteration to be executed in the order that it would have occurred if written as a sequential loop. The syntax of the *ordered* construct is as follows:
#pragma omp ordered new-line
structured-block

3.3.3.2 Shared Programming Using Library Routines

The most popular of them is the use of combo function called fork() and join(). Fork() function is used to create a new child process. By calling join() function parent process waits the terminations of the child process to get the desired result.

Example 11: Consider the following set of statements

```
Process A                    Process B

  :                            :

fork B ;                       :

 :                             :

join B;                      end B;
```

In the above set of statements process A creates a child process B by the statement fork B. Then A and B continue their computations independently until A reaches the join statement, At this stage, if B is already finished, then A continues executing the next statement otherwise it waits for B to finish.

In the shared memory model, a common problem is to synchronize the processes. It may be possible that more than one process are trying to simultaneously modify the same variable. To solve this problem many synchronization mechanism like test_and_set, semaphores and monitors have been used. We shall not go into the details of these mechanisms. Rather, we shall represent them by a pair of two processes called lock and unlock. Whenever a process P locks a common variable, then only P can use that variable. Other concurrent processes have to wait for the common variable until P calls the unlock on that variable. Let us see the effect of locking on the output of a program when we do not use lock and when we use lock.

Example 12

Let us write a pseudocode to find sum of the two functions f(A) + f(B). In the first algorithm we shall not use locking.

```
Process A                    Process B

 sum = 0                        :

   :                            :

  fork B                    sum = sum+ f(B)

   :                            :

 sum = sum + f(A)              end B

   :

   join B

   :

 end A
```

If process A executes the statement sum = sum + f(A) and writes the results into main memory followed by the computation of sum by process B, then we get the correct result. But consider the case when B executes the statement sum = sum + f(B) before process A could write result into the main memory. Then the sum contains only f(B) which is incorrect. To avoid such inconsistencies, we use locking.

```
Process A                    Process B

 sum = 0                        :

   :                            :

   :                          lock sum

  fork B                    sum = sum + f(B)

   :                          unlock sum
```

```
lock sum                        :

 sum = sum + f(A)               end B

unlock sum

  :

   join B

  :

 end A
```

In this case whenever a process acquires the sum variable, it locks it so that no other process can access that variable which ensures the consistency in results.

3.3.4 Example Programmes for Parallel Systems

Now we shall finish this unit with the examples on shared memory programming.

Example 13: Adding elements of an array using two processor

```
 int sum, A[ n] ;  //shared variables
void main ( ){

                int i ;

        for (i=0; i<n; i++)
        scanf ("%d",&A[i] );
        sum=0;
        // now create process to be executed by processor P1
        fork(1) add (A,n/2,n-1, sum); // process to add elements from index n/2 to -
        1.sum is output variable        // now create process to be executed by processor
        P0                                                      add (A,0,n/2-1,
        sum);
        join 1 ;
        printf ("%d", sum);

        }

 add (int A[ ], int lower, int upper, int sum) {

                        int sum1=0, i;
        for (i=lower; i<=upper; i++)
        sum1=sum1+A[i];
        lock sum;
        sum=sum+sum1;
        unlock  sum ;
        }
```

In this program, the last half of the array is passed to processor P_1 which adds them. Meanwhile processor P_0 adds the fist half of the array. The variable sum is locked to avoid inconsistency.

Example 14: In this example we will see the use of parallel construct with private and firstprivate clauses. At the end of the program i and j remain undefined as these are private to thread in parallel construct.

```
#include <stdio.h>
int main()
{
int i, j;
i = 1;
j = 2;
#pragma omp parallel private(i) firstprivate(j)
{
i = 3;
j = j + 2;
}
printf("%d %d\n", i, j); /* i and j are undefined */
return 0;
}
```

In the following example, each thread in the **parallel** region decides what part of the global array *x* to work on, based on the thread number:

Example 15

```
#include <omp.h>
void subdomain(float x[ ], int istart, int ipoints)
{
int i;
for (i = 0; i < ipoints; i++)
x[istart+i] = 123.456;
}
void sub(float x[ 10000], int npoints)
{
int t_num, num_t, ipoints, istart;
#pragma omp parallel default(shared) private(t_num , num_t, ipoints, istart)
{
t_num = omp_get_thread_num(); //thread number of current thread
num_t = omp_get_num_threads(); //number of threads
ipoints = npoints / num_t; /* size of partition */
istart = t_num * ipoints; /* starting array index */
if (t_num == num_t-1) /* last thread may do more */
ipoints = npoints - istart;
subdomain(x, istart, ipoints);
}
}

int main()
{
float array[10000];
sub(array, 10000);
```

```
return 0;
}
```

In this example we used two library methods : *omp_get_num_threads()* and *omp_get_thread_num()*.
omp_get_num_threads() returns number of threads that are currently being used in parallel directive.
omp_get_thread_num() returns thread number (an integer from 0 to omp_get_num_threads() - 1 where thread 0 is the master thread).

Example 16

This example illustrate the use of lastprivate clause

```
void for_loop (int n, float *a, float *b)
{
int i;
#pragma omp parallel
{
#pragma omp for lastprivate(i)
for (i=0; i<n-1; i++)
a[i] = b[i] + b[i+1];
}
a[i]=b[i];   /* i == n-1 here */
}
```

Example 17

This example demonstrates the use of parallel sections construct. The three functions, fun1, fun2, and fun3, all can be executed concurrently. Note that all the section directives need to appear in the parallel sections construct.

```
void fun1();
void fun2();
void fun3();
void  parallel_sec()
{
#pragma omp parallel sections
{
#pragma omp section
fun1();
#pragma omp section
fun2();
#pragma omp section
fun3();
}
}
```

☞ **Check Your Progress 3**

1) Write the syntax of the following compiler directives in OpenMP:
   (a) Parallel
   (b) Sections
   (c) Master

2) What do you understand by synchronization of processes? Explain at least one mechanism for process synchronisation.
   ……
   ……
   ……
   ……

3) Write a shared memory program to process marks of the students. Your program should take the roll number as input and the marks of students in 4 different subjects and find the grade of the student, class average and standard deviation.
   ……
   ……
   ……
   ……

3.4 SUMMARY

In this unit, the following four types of models of parallel computation are discussed in detail:

- Message Passing;
- Data Parallel programming;
- Shared Memory; and
- Hybrid.

Programming based on message passing has already been discussed in Unit 2. In context of data parallel programming, various issues related to High Performance Fortran, e.g., data distribution, block distribution, cyclic distribution, data alignment etc are discussed in sufficient details. Next, in respect of the third model of parallel computation, viz Shared Memory, various constructs and features provided by the standard OpenMP are discussed. Next, in respect of this model, some issues relating to programming using library routines are discussed.

3.5 SOLUTIONS/ANSWERS

☞ **Check Your Progress 1**

1) Application of parallel programming:

   i) In geology and metrology, to solve problems like planetary and galactic orbits, weather and ocean patterns and tectonic plate drift;
   ii) In manufacturing to solve problems like automobile assembly line;
   iii) In science and technology for problems like chemical and nuclear reactions, biological, and human genome;
   iv) Daily operations within a business; and
   v) Advanced graphics and virtual reality, particularly in the entertainment industry.

2) Steps to write a parallel program:

i) Understand the problem thoroughly and analyze that portion of the program that can be parallelized;
ii) Partition the problem either in data centric way or in function centric way depending upon the nature of the problem;
iii) Decision of communication model among processes:
iv) Decision of mechanism for synchronization of processes,
v) Removal of data dependencies (if any),
vi) Load balancing among processors,
vii) Performance analysis of program.

3) Program for P_0

```
#include <mpi.h>
#define n 100
int main(int argc, char **argv) {
int A[n];
int sum0 =0, sum1[ ],sum, incr, last_incr;
MPI_Init(&argc, &argv);
for( int i=0;i<n;i++)   //taking input array
scanf("%d", &A[i]);
incr = n/k;     //finding number of data for each processor
last_incr = incr n + n%k;  // last processor may get lesser number of data
for(i=1; i<= k-2; i++)
MPI_Send ( (i-1)*incr, incr,  MPI_INT,i, i, MPI_COMM_WORLD); // P0 sends
data to other processors
MPI_Send( (i-1)*incr, last_incr,  MPI_INT,i, i, MPI_COMM_WORLD); // P0
sends data to last processor
for(i=1; i<=incr; i++)    // P0 sums its own elements
 sum0+=A[i];
for(i=1; i<= k-1; i++)          // P0 receives results from other processors
sum1[i] =MPI_Recv(i, 1, MPI_INT,0, 0, MPI_COMM_WORLD);
for(i=1; i<= k-1; i++)                    //results are added to get final results
sum=sum0+sum1[i];
printf("%d",sum);
MPI_Finalize();
}
```

// Program for P_r for r=1 2 …k-2,

```
int func( int B[int n])
{
MPI_Recv(1, incr, MPI_INT,0, 0, MPI_COMM_WORLD);
 int sum1=0 ;
for (i=0; i<incr; i++)
  sum1+=B[i];
MPI_Send( 0, incr, MPI_INT,0, 0, MPI_COMM_WORLD);
}
```

Program for the last processor Pk_{-1},

```
int func( int B[int n])
{
```

```
MPI_Recv(1, last_incr, MPI_INT,0, 0, MPI_COMM_WORLD);
 int sum1=0 ;
for (i=0; i<last_incr; i++)
  sum1+=B[i];
MPI_Send( 0, last_incr, MPI_INT,0, 0, MPI_COMM_WORLD);
}
```

☞ Check Your Progress 2

1) a) !HPF$ PROCESSORS Q (s, r)
It maps s × r processors along a two dimensional array and gives them collective name Q.

b) !HPF$ PROCESSORS P(5)
!HPF$ TEMPLATE T(22)
!HPF$ DISTRIBUTE T(CYCLIC) ONTO P.
Data is distributed n 5 processors as shown below:

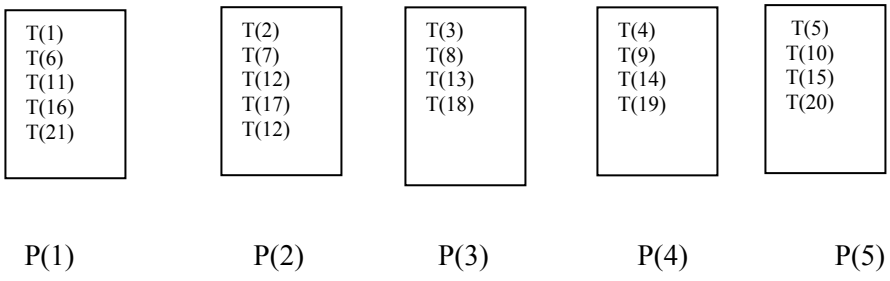


Figure 8

2) FORALL (i=1:n, j=1:n, i >j) Y(i,j) = 0.0

3) Intrinsic functions are library-defined functions in a programming languages to support various constructs in the language. The two most frequently used in parallel programming are the system inquiry functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE. These functions provide information about the number of physical processors on which the running program executes and processor configuration. General syntax of NUMBER_OF_PROCESSORS is

NUMBER_OF_PROCESSORS(dim)

where dim is an optional argument. It returns the number of processors in the underlying array or, if the optional argument is present, the size of this array along a specified dimension.

General syntax of PROCESSORS_SHAPE is

PROCESSORS_SHAPE()

It returns an one-dimensional array, whose i^{th} element gives the size of the underlying processor array in its i^{th} dimension.

☞ Check Your Progress 3

1) (a) syntax for parallel directive :
#pragma omp parallel [set of clauses]
where *clause* is one of the following:
structured-block
if(scalar-expression)
private(list)
firstprivate(list)
default(shared | none)
shared(list)
copyin(list)

(b) syntax for sections directive :

#pragma omp sections [set of clauses.]
{
#pragma omp section
structured-bloc
#pragma omp section
structured-block
.
.
.
}

The *clause* is one of the following:
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait

(c) syntax for master directive :

#pragma omp master
structured_block

2) In parallel programming, shared memory programming in particular, processes very often compete with each other for common resources. To ensure the consistency, we require some mechanisms called process synchronization. There are numerous techniques to enforce synchronization among processes. We discussed one of them in unit 3.2.2 using lock and unlock process. Now we are introducing semaphores. Semaphores were devised by Dijkstra in 1968. It consists of two operations P and V operating on a positive integer s (including zero).P waits until *s* is greater than zero and then decrements *s* by one and allows the process to continue. V increments *s* by one and releases one of the waiting processes (if any). **P** and **V** operations are performed atomically. Mechanism for activating waiting processes is also implicit in **P** and **V** operations. Processes delayed by **P**(s) are kept in abeyance until released by a **V**(s) on the same semaphore. Mutual exclusion of critical sections can be achieved with one semaphore having the value 0 or 1 (a binary semaphore), which acts as a lock variable.

3)
```
#include <math.h>

int total_m, num_stud=0,sum_marks, sum_square;

void main ( )

{
int roll_no, marks[ ][4],i ;
char grade;
float class_av, std_dev;
while (!EOF) {
scanf (“%d”, &roll_no);
for (i=0; i<4; i++)          //taking marks of individual student
scanf (“%d”,&marks[num_stud][i] );
total_m =0;
num_stud++;
for (i=0; i<4; i++)
total_m = total_m+marks[num_stud][i];   //sum of marks
fork(1) grade = find_grade();    //create new parallel process to find grade
fork(2) find_stat();                  // create  new parallel process to  find sum and
squares
join 1;             //wait until process find_grade terminates
join 2;             // wait until process find_stat terminates
printf (“roll number %d”, roll_no);
for (i=0; i<4; i++)
printf (“%d”,marks[num_stud][i] );
printf (“%d”,total_m );
printf (“%c”,grade );
}
class_av= sum_marks/num_stud;
std_dev=sqrt ((sum_square/std_num) –(class_av*class_av)) ;
printf (“%f %f”,class_av,std_dev );
}

char find_grade( ) {
char g;
if  (total_m >80)    g=’E’;
else if   (total_m >70)    g=’A’;
else if   (total_m >60)    g=’B’;
else if   (total_m >50)    g=’C’;
else if   (total_m >40)    g=’D’;
else   g=’F’;
return g;
}

find_stat( ) {
sum_marks =sum_marks+total_m;
sum_square =sum_square+total_m*total_m;
}
```

Example 18: master construct

```
#include <stdio.h>
extern float average(float,float,float);
void master_construct ( float* x, float* xold, int n, float tol )
```

```
{
int c, i, toobig;
float error, y;
c = 0;
#pragma omp parallel
{
do{
#pragma omp for private(i)
for( i = 1; i < n-1; ++i ){
xold[i] = x[i];
}
#pragma omp single
{
toobig = 0;
}
#pragma omp for private(i,y,error) reduction(+:toobig)
for( i = 1; i < n-1; ++i ){
y = x[i];
x[i] = average( xold[i-1], x[i], xold[i+1] );
error = y - x[i];
if( error > tol || error < -tol ) ++toobig;
}
#pragma omp master
{
++c;
printf( "iteration %d, toobig=%d\n", c, toobig );
}
}while( toobig > 0 );
}
}
```

3.6 REFERENCES/FURTHER READINGS

1) Quinn Michael J. *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Education (2004).

2) Jorg Keller, Kesler Christoph W. and Jesper Larsson *Practical PRAM Programming* (wiley series in Parallel Computation).

3) Ragsdale & Susan, (ed) "*Parallel Programming*" McGraw-Hill.

4) Chady, K. Mani & Taylor slephen "*An Introduction to Parallel Programming*, Jones and Bartleh.