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*.

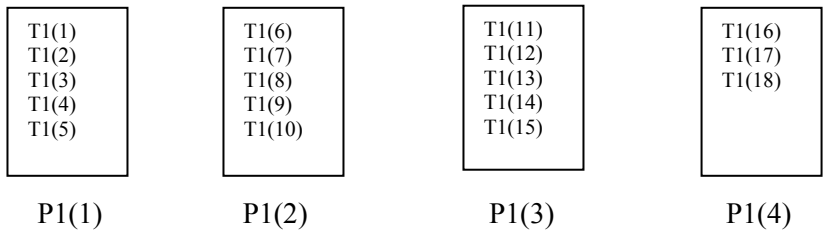

Figure 1: Block Distribution of Data

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*.

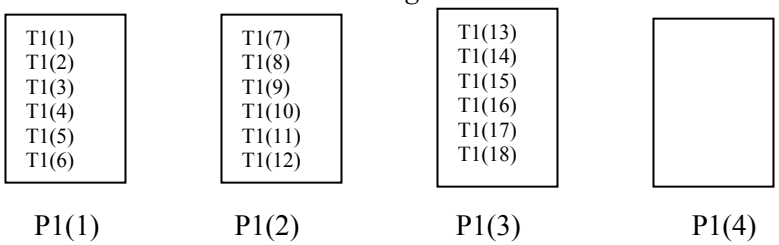

Figure 2: Variation of Block Distribution

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*.

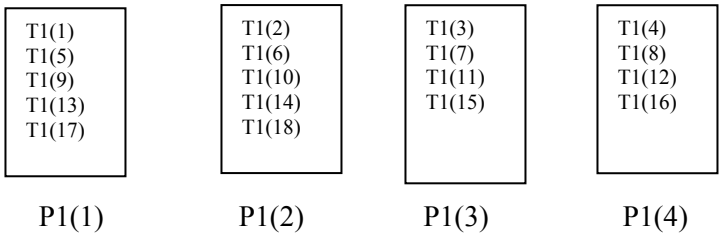

Figure 3: Cyclic Distribution

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

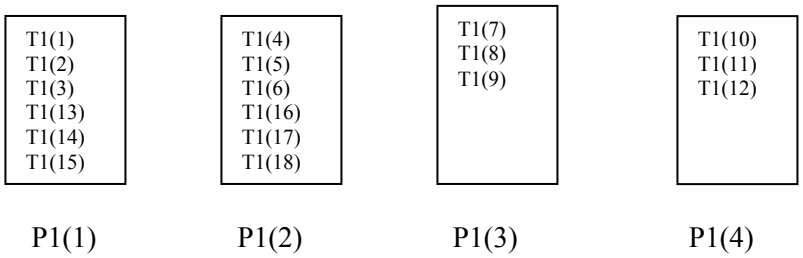

Figure 4: Variation of Cyclic Distribution

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*.

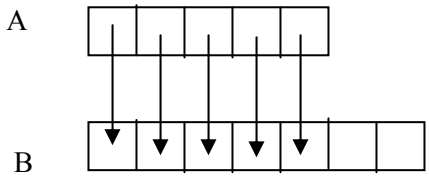

Figure 5: ALIGN A[i] WITH B[i]

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*.

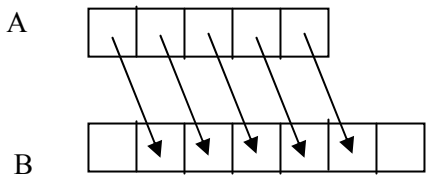

Figure 6: ALIGN A[i] WITH B[i+1]

* 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*.

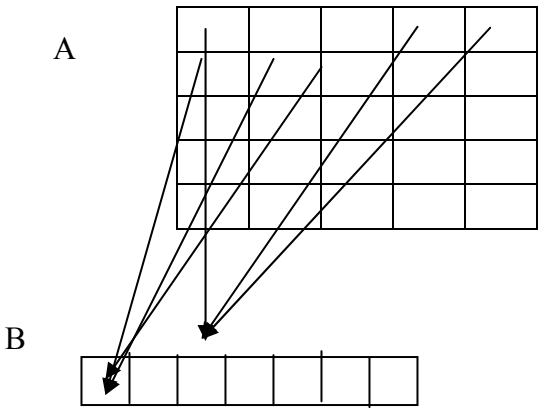

Figure 7: * alignment

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:

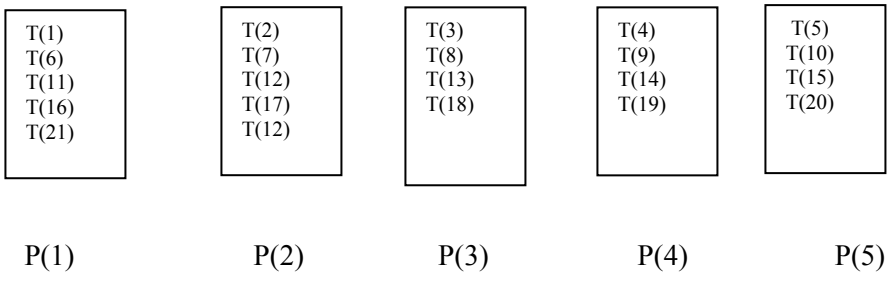

Figure 8

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.