



**Digital Assignment for:
Parallel and Distributed Computing (CSE4001)**

Lab -1 Upload

Submitted by:

Name	Registration Number
Prerna Sultania	18BCE0869

**Under the guidance of
Prof. M. Narayana Moorthi**

Slot:L11+L12

Question 1

Study of Computer System Organization

A computer system is made up of various components. The components can be hardware or software. Because these systems are so massively complex, the components are organized in layers.

Layer	People	Domain
Application Programs	Application Programmers	Software
System Utility Programs	System Programmers	
Operating System		
I/O System (BIOS)		
Computer System	Computer Engineers	Hardware
CPU	Computer Architects	
Memories, Logic Circuits, Flip-Flops, Gates	Logic Designers	
Transistors, Diodes, Resistors, Power Supplies	Materials Scientists	

Computer organization consist of following parts

- CPU – central processing unit
 - It is alternatively referred to as the **brain of the computer, processor, central processor, or microprocessor**, the CPU (pronounced as C-P-U) was first developed at Intel with the help of Ted Hoff in the early 1970's and is short for **Central Processing Unit**. The computer CPU is responsible for handling all instructions it receives from hardware and software running on the computer.
 - CPU is considered as the brain of the computer. CPU performs all types of data processing operations. It stores data, intermediate results and instructions (program). It controls the operation of all parts of computer.
 - **CPU itself has following three components**
 - **ALU (Arithmetic Logic Unit)** All arithmetic calculations and logical operation are performed using the Arithmetic/Logical Unit or ALU
 - **Memory Unit** A memory is just like a human brain. It is used to store data and instruction. Computer memory is use to Stores information being processed by the CPU
 - **Control Unit** Control unit help to perform operations of input unit, output unit, Memory unit and ALU in a sequence.
- Memory
 - Computer **memory** is any physical device capable of storing information temporarily or permanently. For example, Random Access Memory **RAM** is a type of volatile memory that is stores information on an integrated circuit, and that is used by the operating system, software, hardware, or the user.
 - **Computer memory divide into two parts**

- **Volatile memory:** Volatile memory is a temporary memory that loses its contents when the computer or hardware device loses power.eg. RAM
 - **Non-volatile memory:** Non-volatile memory keeps its contents even if the power is lost. Example: ROM or EPROM is a good example of a non-volatile memory
-
- Input devices
- A device that can be used to insert data into a computer system is called as input device. It allows people to supply information to computers. An **input device** is any hardware device that sends data to the computer, without any input devices, a computer would only be a display device and not allow users to interact with it, much like a TV. The most fundamental pieces of information are keystrokes on a *keyboard* and clicks with a *mouse*. These two input devices are essential for you to interact with your computer. Input devices represent one type of computer **peripheral**. Examples of input devices include keyboards, **mouse**, scanners, digital cameras and joysticks.
- Output devices

A device which is used to display result from a computer is called as output device. It Allows people to receive information from computers. An **output device** is any peripheral that receives or displays output from a computer. The picture shows an inkjet printer, an output device that can make a hard copy of anything being displayed on a monitor. Output device is electronic equipment connected to a computer and used to transfer data out of the computer in the form of text, images, sounds or print.

Examples of output devices include Printer, Scanner, Monitor, etc.

Question 2

Study of parallel programming languages and introduction to Open MP

INTRODUCTION TO PARALLEL PROGRAMMING LANGUAGES

In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed

Some Parallel Programming Languages:

- Adaptor
- CC++
- Dino
- Force
- Fortran 90
- Fortran D
- Fortran M
- Grids
- HPF
- Jade
- Mentat
- O-O Fortran
- P-D Linda
- P- Languages

Let's talk about a few of them:

- **GRIDS: (FORTRAN Based language)**

Functions:

- A computing environment for grid-based numerically intensive computation
 - Declarative language for overall control of solution method
 - Topology description separated from computational algorithms
 - Grid constructs as extensions to Fortran for computational algorithms
- A runtime system which exploits knowledge of the parallelism inherent in the problem and the grid-based solution methods (Explicit parallel programming not needed)
- Support for regular and irregular grids
- A pre-processor to translate a Grids code (topology, declarative part, and extended Fortran procedures) to standard Fortran
- Costs around \$400 and requires specific platforms like networks of IBM RS6000.

- **JADE: (C- Based language)**

Functions:

- A declarative, data-oriented language for coarse-grain parallel programming
- Preservation of the abstractions of serial semantics and a single address space
- Constructs for specifying how a program written in a standard sequential, imperative programming language accesses data.
- Translation of a C program with Jade constructs into a C program with calls to the Jade implementation

Strong Points:

- Jade's abstraction of serial semantics eliminates nondeterministic, timing-dependent bugs: all parallel executions of a Jade program deterministically generate the same result as the serial execution.
- Jade's abstraction of a single address space eliminates the need for programmers to manage the distribution of data across the parallel machine.
- Programmers can effectively use Jade to develop coarse-grain parallel programs that execute efficiently on a range of parallel architectures.

Weak Points:

- No support for writing nondeterministic programs
- No user control for the low-level execution of the program for maximal efficiency

- **PC++ (Parallel C++) (Based on Object Oriented programming Languages)**

Functions:

- A data-parallel extension to C++
- Collection class for concurrent aggregates (structured sets of objects that are distributed over the processors and memories in a parallel system)
- Concurrent application of arbitrary functions to the elements of arbitrary distributed, aggregate data structures
- Collection library to provide a set of primitive algebraic structures that may be used in scientific and engineering computations
- A pre-processor that translates a pC++ code into C++ code
- Works on any platform

INTRODUCTION TO OPENMP

OpenMP

- An Application Program Interface (API) that may be used to explicitly direct *multi-threaded, shared memory* parallelism.
- Comprised of three primary API components:
 - Compiler Directives
 - Runtime Library Routines
 - Environment Variables
- An abbreviation for: Open Multi-Processing

Things Achieved with OpenMP

- Standardization:
 - Provide a standard among a variety of shared memory architectures/platforms
 - Jointly defined and endorsed by a group of major computer hardware and software vendors
- Lean and Mean:
 - Establish a simple and limited set of directives for programming shared memory machines.
 - Significant parallelism can be implemented by using just 3 or 4 directives.
 - This goal is becoming less meaningful with each new release, apparently.
- Ease of Use:
 - Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
 - Provide the capability to implement both coarse-grain and fine-grain parallelism
- Portability:
 - The API is specified for C/C++ and Fortran
 - Public forum for API and membership
 - Most major platforms have been implemented including Unix/Linux platforms and Windows

OpenMP includes some of the runtime routines like:

Routine	Purpose
OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_MAX_THREADS	Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function
OMP_GET_THREAD_NUM	Returns the thread number of the thread, within the team, making this call
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_NUM_THREADS	Returns the number of processors that are available to the program
OMP_IS_IN_PARALLEL	Used to determine if the section of code which is executing is parallel or not
OMP_SET_DYNAMIC	Enables or disables dynamic adjustment (by the run-time system) of the number of threads available for execution of parallel regions
OMP_GET_DYNAMIC	Used to determine if dynamic thread adjustment is enabled or not
OMP_SET_NESTED	Used to enable or disable nested parallelism
OMP_GET_NESTED	Used to determine if nested parallelism is enabled or not
OMP_SET_SCHEDULE	Sets the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_GET_SCHEDULE	Returns the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_SET_MAX_ACTIVE_LEVELS	Sets the maximum number of nested parallel regions
OMP_GET_MAX_ACTIVE_LEVELS	Returns the maximum number of nested parallel regions
OMP_GET_LEVEL	Returns the current level of nested parallel regions
OMP_GET_ANCESTOR_THREAD_NUM	Returns, for a given nested level of the current thread, the thread number of ancestor thread
OMP_GET_TEAM_SIZE	Returns, for a given nested level of the current thread, the size of the thread team
OMP_GET_ACTIVE_LEVEL	Returns the number of nested, active parallel regions enclosing the task that contains the call
OMP_IN_FINAL	Returns true if this routine is executed in the final task region, otherwise it returns false
OMP_INIT_LOCK	Initializes a lock associated with the lock variable
OMP_DESTROY_LOCK	Deassociates the given lock variable from any locks
OMP_SET_LOCK	Acquires ownership of a lock
OMP_UNSET_LOCK	Releases a lock
OMP_TEST_LOCK	Attempts to set a lock, but does not block if the lock is unavailable
OMP_INIT_NEST_LOCK	Initializes a nested lock associated with the lock variable
OMP_DESTROY_NEST_LOCK	Deassociates the given nested lock variable from any locks

Question 3

Write a c-program using open MP to print hello world

Code:

```
#include<omp.h>
#include<stdio.h>
void main (void)
{
#pragma omp parallel
{
printf ("hello world \n");
}
}
```

Screenshots:

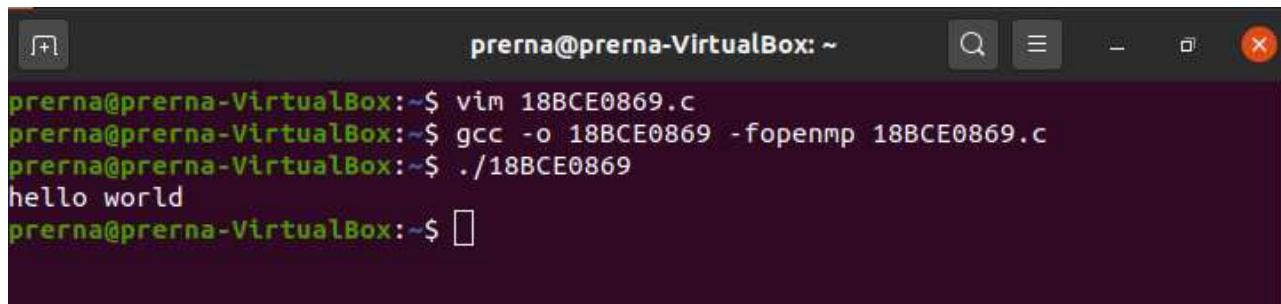
Inserting the code:

The screenshot shows a terminal window titled "prerna@prerna-VirtualBox: ~". The window contains the following C code:

```
#include<omp.h>
#include<stdio.h>
void main(void)
{
#pragma omp parallel
{
printf("hello world\n");
}
}
```

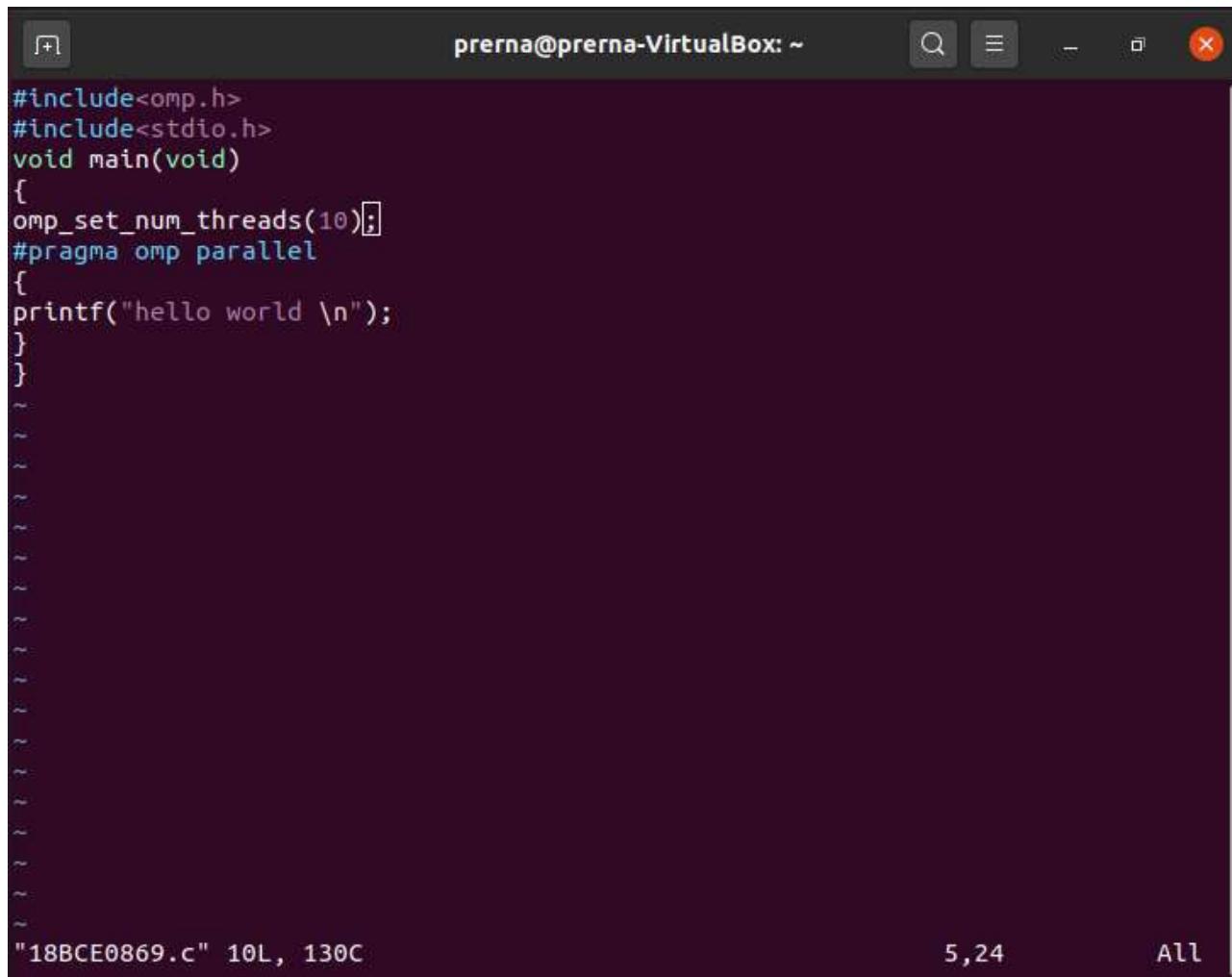
The code is being typed into the terminal. The cursor is at the end of the final closing brace "}" on the second-to-last line. The terminal window has a dark background with light-colored text. The bottom status bar shows the file name "18BCE0869.c", line count "9L", character count "105C", page number "2,17", and a "All" button.

Compiling and running the code:



```
prerna@prerna-VirtualBox:~$ vim 18BCE0869.c
prerna@prerna-VirtualBox:~$ gcc -o 18BCE0869 -fopenmp 18BCE0869.c
prerna@prerna-VirtualBox:~$ ./18BCE0869
hello world
prerna@prerna-VirtualBox:~$
```

Code to create 10 different threads:



```
#include<omp.h>
#include<stdio.h>
void main(void)
{
omp_set_num_threads(10);
#pragma omp parallel
{
printf("hello world \n");
}
}
```

"18BCE0869.c" 10L, 130C 5,24 All

Compiling and running the code:



**Digital Assignment for:
Parallel and Distributed Computing(CSE4001)**

**Vtop Upload
Lab 2**

Submitted by:

Name	Registration Number
Prerna Sultania	18BCE0869

**Under the guidance of
Prof. Narayananmoorthy**

**Slot:
L11 +L12**

Question 1:**Code:**

```
#include<stdio.h>
#include<omp.h>
void main(void)
{
    int a,b,c,d,e;
    double time = omp_get_wtime();

    a = omp_get_num_procs();
    printf("The number of processors = %d\n",a);
    b = omp_get_max_threads();
    printf("The maximum thread number = %d\n",b);
    c = omp_get_num_threads();
    printf("The number of threads = %d\n",c);

    printf("Now setting the number of threads = 6\n");
    omp_set_num_threads(6);

    printf("-----Starting the execution of the parallel Section -----");
    #pragma omp parallel
    {
        e = omp_get_thread_num();
        printf("To get the thread number = %d\n",e);
    }

    printf("-----End of the parallel Section -----");
    printf("The time for the given computation = %f\n",time);
}
```

Screenshots:

```
prerna@prerna-VirtualBox:~$ vim ex2a.c
prerna@prerna-VirtualBox:~$ gcc -fopenmp ex2a.c
prerna@prerna-VirtualBox:~$ ./a.out
The number of processors = 1
The maximum thread number = 1
The number of threads = 1
Now setting the number of threads = 6
-----Starting the execution of the parallel Section-----
To get the thread number = 0
To get the thread number = 5
To get the thread number = 4
To get the thread number = 3
To get the thread number = 2
To get the thread number = 1
-----End of the parallel Section-----
The time for the given computation = 389.284517
```

```
prerna@prerna-VirtualBox: ~
#include<stdio.h>
#include<omp.h>
void main(void)
{
    int a,b,c,d,e;
    double time = omp_get_wtime();
    a = omp_get_num_procs();
    printf("The number of processors = %d\n",a);
    b = omp_get_max_threads();
    printf("The maximum thread number = %d\n",b);
    c = omp_get_num_threads();
    printf("The number of threads = %d\n",c);
    printf("Now setting the number of threads = 6\n");
    omp_set_num_threads(6);
    printf("-----Starting the execution of the parallel Section-----\n");
    #pragma omp parallel
    {
        e = omp_get_thread_num();
        printf("To get the thread number = %d\n",e);
    }
    printf("-----End of the parallel Section-----\n");
    printf("The time for the given computation = %f\n",time);
}
```

Question 2

Code:

```
#include<stdio.h>
#include<omp.h>
#include<time.h>
void main(void)
{
    int a,b,c,a1,b1,c1;
    double d,d1;
    int x=10,y=20;

    printf("Now setting the number of threads = 4\n");
    omp_set_num_threads(4);
    double time = omp_get_wtime();
    printf("-----Starting the execution of the parallel Section-----\n");
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            a = x+y;
            printf("Addition of x,y = %d\n",a);
        }
        #pragma omp section
        {
            b = y-x;
            printf("Subtraction of x,y = %d\n",b);
        }
        #pragma omp section
        {
            c = x*y;
            printf("Multiplication of x,y = %d\n",c);
        }
        #pragma omp section
        {
            d = y/x;
            printf("Division of x,y = %f\n",d);
        }
    }

    printf("-----End of the parallel Section ----- \n");
    printf("The time for the given computation = %f\n",time);
}
```

Screenshots

```

prerna@prerna-VirtualBox:~$ vim ex2b.c
prerna@prerna-VirtualBox:~$ gcc -fopenmp ex2b.c
prerna@prerna-VirtualBox:~$ ./a.out
Now setting the number of threads = 4
-----Starting the execution of the parallel Section-----
Addition of x,y = 30
Subtraction of x,y = 10
Multiplication of x,y = 200
Division of x,y = 2.000000
-----End of the parallel Section-----
The time for the given computation = 710.120171
prerna@prerna-VirtualBox:~$ # 18BCE0869

```

```

#include<stdio.h>
#include<omp.h>
#include<time.h>
void main(void)
{
int a,b,c,a1,b1,c1;
double d,d1;
int x=10,y=20;
printf("Now setting the number of threads = 4\n");
omp_set_num_threads(4);
double time = omp_get_wtime();
printf("-----Starting the execution of the parallel Section-----\n");
#pragma omp parallel sections
{
#pragma omp section
{
a = x+y;
printf("Addition of x,y = %d\n",a);
}
#pragma omp section
{
b = y-x;
printf("Subtraction of x,y = %d\n",b);
}
#pragma omp section
{
c = x*y;
printf("Multiplication of x,y = %d\n",c);
}
#pragma omp section
{
d = y/x;
printf("Division of x,y = %f\n",d);
}
printf("-----End of the parallel Section-----\n");
printf("The time for the given computation = %f\n",time);
}

```

Question 3**Code:**

```

#include<stdio.h>
#include<omp.h>
void main(void)
{
int max,min;
int a[]={10,3,7};

printf("Now setting the number of threads = 2\n");
omp_set_num_threads(2);
double time = omp_get_wtime();
printf("-----Starting the execution of the parallel Section----- \n");
#pragma omp parallel sections
{
    //CALC MAXIMUM
    #pragma omp section
    {
        if(a[0]>a[1] && a[0]>a[2]) max=a[0];
        else if(a[1]>a[0] && a[1]>a[2])
            max=a[1];
        else
            max=a[2];
        printf("Maximum of 3 numbers (i.e. a[]={10,3,7}) is = %d\n",max);
    }

    //CALC MINIMUM
    #pragma omp section
    {
        if(a[0]<a[1] && a[0]<a[2]) min=a[0];
        else if(a[1]<a[0] && a[1]<a[2])
            min=a[1];
        else
            min=a[2];
        printf("Minimum of 3 numbers (i.e. a[]={10,3,7}) is = %d\n",min);
    }
}

printf("-----End of the parallel Section----- \n");
printf("The time for the given computation = %f\n",time);
}

```

Screenshots:

```
prerna@prerna-VirtualBox:~$ vim ex2c.c
prerna@prerna-VirtualBox:~$ gcc -fopenmp ex2c.c
prerna@prerna-VirtualBox:~$ ./a.out
Now setting the number of threads = 2
-----Starting the execution of the parallel Section-----
Maximum of 3 numbers (i.e. a[]={10,3,7}) is = 10
Minimum of 3 numbers (i.e. a[]={10,3,7}) is = 3
-----End of the parallel Section-----
The time for the given computation = 1209.483603
prerna@prerna-VirtualBox:~$ #18BCE0869
```

```
#include<stdio.h>
#include<omp.h>
void main(void)
{
int max,min;
int a[]={10,3,7};
printf("Now setting the number of threads = 2\n");
omp_set_num_threads(2);
double time = omp_get_wtime();
printf("-----Starting the execution of the parallel Section-----\n");
#pragma omp parallel sections
{
//CALC MAXIMUM
#pragma omp section
{
if(a[0]>a[1] && a[0]>a[2])
max=a[0];
else if(a[1]>a[0] && a[1]>a[2])
max=a[1];
else
max=a[2];
printf("Maximum of 3 numbers (i.e. a[]={10,3,7}) is = %d\n",max);
}
//CALC MINIMUM
#pragma omp section
{
if(a[0]<a[1] && a[0]<a[2])
min=a[0];
else if(a[1]<a[0] && a[1]<a[2])
min=a[1];
else
min=a[2];
printf("Minimum of 3 numbers (i.e. a[]={10,3,7}) is = %d\n",min);
}
printf("-----End of the parallel Section-----\n");
printf("The time for the given computation = %f\n",time);
}
```

1,1 Top

```
else if(a[1]<a[0] && a[1]<a[2])
min=a[1];
else
min=a[2];
printf("Minimum of 3 numbers (i.e. a[]={10,3,7}) is = %d\n",min);
}
}
printf("-----End of the parallel Section-----\n");
printf("The time for the given computation = %f\n",time);
}
```



**Parallel and Distributed Computing
(CSE4001)**

**Vtop Upload
Lab 3**

Submitted by:

Name	Registration Number
Prerna Sultania	18BCE0869

**Under the guidance of
Prof. Narayananmoorthy**

**Slot:
L11 +L12**

Write an open MP program using C for the following illustrations:

Code:

Private Data

```
int x = 5;  
#pragma omp parallel  
{  
    int x;  
    x = 3;  
    printf("local: x is %d\n", x);  
}
```

First Private

```
int t = 2;  
#pragma omp parallel firstprivate(t)  
{  
    t += f(omp_get_thread_num());  
    g(t);  
}
```

Shared

```
int x = 5;  
#pragma omp parallel  
{  
    x = x + 1;  
    printf("shared: x is %d\n", x);  
}
```

Last Private

```
#pragma omp parallel for lastprivate(tmp)  
for (i = 0; i < N; i +)  
{  
    x[i] = temp * x[i];  
}
```

- **Critical / Atomic / Master / Single constructs**

Critical Construct

```
int dequeue(float *a);
void work(int i, float *a);
void critical_example(float *x, float *y)
{
    int ix_next, iy_next;
#pragma omp parallel shared(x, y) private(ix_next, iy_next) {
#pragma omp critical(xaxis)
    ix_next = dequeue(x);
    work(ix_next, x);
#pragma omp critical(yaxis)
    iy_next = dequeue(y);
    work(iy_next, y);
}
}
```

Master Construct

```
#include <stdio.h>
extern float average(float, float, float);
void master_example(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);
}

```

Atomic construct

```

float work1(int i)
{
    return 1.0 * i;
}

float work2(int i)
{

```

```

    return 2.0 * i;
}

void atomic_example(float *x, float *y, int *index, int n)
{
    int i;

#pragma omp parallel for shared(x, y, index, n)
    for (i = 0; i < n; i++)
    {
        #pragma omp atomic update
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
}

int main()
{
    float x[1000];
    float y[10000];
    int index[10000];
    int i;
    for (i = 0; i < 10000; i++)
    {
        index[i] = i % 1000;
        y[i] = 0.0;
    }
    for (i = 0; i < 1000; i++)
        x[i] = 0.0;
    atomic_example(x, y, index, 10000);
    return 0;
}

```

Single construct

```

#include <stdio.h>
void work1() {}
void work2() {}
void single_example()
{

```

```

#pragma omp parallel
{
#pragma omp single
    printf("Beginning work1.\n");
    work1();
#pragma omp single
    printf("Finishing work1.\n");
#pragma omp single nowait
    printf("Finished work1 and beginning work2.\n");
    work2();
}
}

```

- **work sharing constructs using for loop and sections directives**

Loop directives

```

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 10
int main(int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];
    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

#pragma omp parallel shared(a, b, c, nthreads, chunk) private(i, tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    for (i = tid * chunk; i < (tid + 1) * chunk; i++)
        c[i] = a[i] + b[i];
}

```

```

    }

    printf("Thread %d starting...\n", tid);

#pragma omp for schedule(dynamic, chunk)
    for (i = 0; i < N; i++)
    {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
    }
}
}

```

Screenshot:

```

Thread 1 starting...
Thread 3 starting...
Number of threads = 4
Thread 0 starting...
Thread 2 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000

```

Section derivatives

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 10

int main(int argc, char *argv[])
{
    int i, nthreads, tid;
    float a[N], b[N], c[N], d[N];
    for (i = 0; i < N; i++)
    {
        a[i] = i * 1.5;
    }
}

```

```

b[i] = i + 22.35;
c[i] = d[i] = 0.0;
}

#pragma omp parallel shared(a, b, c, d, nthreads) private(i, tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);

#pragma omp sections nowait
{
    #pragma omp section
    {
        printf("Thread %d doing section 1\n", tid);
        for (i = 0; i < N; i++)
        {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
        }
    }

    #pragma omp section
    {
        printf("Thread %d doing section 2\n", tid);
        for (i = 0; i < N; i++)
        {
            d[i] = a[i] * b[i];
            printf("Thread %d: d[%d]= %f\n", tid, i, d[i]);
        }
    }

    printf("Thread %d done.\n", tid);
}
}

```

Screenshot:

```
Thread 2 starting...
Number of threads = 4
Thread 0 starting...
Thread 0 doing section 2
Thread 0: d[0]= 0.000000
Thread 0: d[1]= 35.025002
Thread 0: d[2]= 73.050003
Thread 0: d[3]= 114.075005
Thread 0: d[4]= 158.100006
Thread 0: d[5]= 205.125000
Thread 0: d[6]= 255.150009
Thread 0: d[7]= 308.175018
Thread 0: d[8]= 364.200012
Thread 0: d[9]= 423.225006
Thread 0 done.
Thread 3 starting...
Thread 3 done.
Thread 1 starting...
Thread 1 done.
Thread 2 doing section 1
Thread 2: c[0]= 22.350000
Thread 2: c[1]= 24.850000
Thread 2: c[2]= 27.350000
Thread 2: c[3]= 29.850000
Thread 2: c[4]= 32.349998
Thread 2: c[5]= 34.849998
Thread 2: c[6]= 37.349998
Thread 2: c[7]= 39.849998
Thread 2: c[8]= 42.349998
Thread 2: c[9]= 44.849998
Thread 2 done.
```

Write a C program using open MP to perform the following matrix operations. Matrix addition, subtraction and multiplication.

Code:

```
#include<stdio.h>
#include<omp.h>
#include<math.h>
void main(void)
{
    int i,a[10];
    for(i=0;i<10;i++)
        a[i]=(i+1);

    double time = omp_get_wtime();
    printf("-----Starting the execution of the parallel Section-----\n");

    //PART1
    double time1 = omp_get_wtime();
    printf("Printing the array elements:\n");
    #pragma omp parallel for
    for(i=0;i<10;i++)
        printf("%d ",a[i]);
    printf("\nTime taken = %f\n\n",time1);

    //PART2
    double time2 = omp_get_wtime();
    printf("Printing the array elements after adding 10 and Multiplying
5 to each element:\n");

    int x = 10;
    #pragma omp parallel for
    for(i=0;i<10;i++)
    {
        a[i]=(a[i]+x)*5;
        printf("%d ",a[i]);
    }
    printf("\nTime taken = %f\n\n",time2);

    for(i=0;i<10;i++)
        a[i]=(i+1);

    //PART3
    double time3 = omp_get_wtime();
    printf("Printing the sum / odd sum / even sum of elements:\n");
    int sum=0,osum=0,esum=0;
    #pragma omp parallel for
    for(i=0;i<10;i++){
        sum=sum+a[i];
        if(a[i]%2!=0)
            osum=osum+a[i];
    }
}
```

```

        else
            esum=esum+a[i];
    }
printf("Total Sum = %d\n",sum);
printf("Odd Sum = %d\n",osum);
printf("Even Sum = %d\n",esum);
printf("Time taken = %f\n\n",time3);

//PART4
double time4 = omp_get_wtime();
printf("Printing the number of odd and even number in the
array:\n");
int odd=0,even=0;
#pragma omp parallel for
for(i=0;i<10;i++)
{
    if(a[i]%2==0)
        even=even+1;
    else
        odd=odd+1;
}
printf("There are %d Odd Numbers in the Array.\n",odd);
printf("There are %d Even Numbers in the Array.\n",even);
printf("Time taken = %f\n\n",time4);

//PART5
double time5 = omp_get_wtime();
printf("Printing Squares and Cubes of array elements:\n");

#pragma omp parallel for
for(i=0;i<10;i++)
{
    printf("Square of %d is %.1f and cube of %d is
%.1f\n",a[i],pow(a[i],2),a[i],pow(a[i],3));
}
printf("\nTime taken = %f\n\n",time5);

//PART6
double time6 = omp_get_wtime();
printf("Printing the maximum and minimum element:\n");
int max=a[0],min=a[0];
#pragma omp parallel for
for(i=0;i<10;i++)
{
    if(a[i]>max)
        max=a[i];
    if(a[i]<min)
        min=a[i];
}
printf("The maximum number in the array is = %d\n",max);
printf("The minimum number in the array is = %d\n",min);
printf("Time taken = %f\n\n",time6);

```

```

//PART7
double time7 = omp_get_wtime();
printf("Printing the number of prime number and their sum in the
array:\n");
printf("Prime Numbers:- ");
int j,psum=0,prime=0,flag;
for(i=0;i<10;i++)
    a[i]=i+1;
#pragma omp parallel for
for(i=0;i<10;i++)
{
    flag = 0;
    for(j=2;j<a[i];j++)
    {
        if(a[i]%j==0)
        {
            flag = 1;
            break;
        }
    }
    if(flag==0 && a[i]!=1)
    {
        printf("%d ",a[i]);
        prime = prime+1;
        psum = psum + a[i];
    }
    else
        continue;
}
printf("These are the %d Prime numbers in the Array\n",prime);
printf("The sum of the prime numbers is = %d\n",psum);
printf("Time taken = %f\n\n",time7);

printf("-----End of the parallel Section-----\n");
printf("The time for the given computation = %f\n",time);

}

```

Screenshots:

```
-----Starting the execution of the parallel Section-----  
Printing the array elements:  
5 1 2 3 4 7 9 10 8 6  
Time taken = 1600767954.630000  
  
Printing the array elements after adding 10 and Multiplying 5 to each element:  
75 95 100 80 90 65 70 85 55 60  
Time taken = 1600767954.634000  
  
Printing the sum / odd sum / even sum of elements:  
Total Sum = 55  
Odd Sum = 25  
Even Sum = 30  
Time taken = 1600767954.635000  
  
Printing the number of odd and even number in the array:  
There are 5 Odd Numbers in the Array.  
There are 5 Even Numbers in the Array.  
Time taken = 1600767954.635000  
  
Printing Squares and Cubes of array elements:  
Square of 7 is 49.0 and cube of 7 is 343.0  
Square of 5 is 25.0 and cube of 5 is 125.0  
Square of 3 is 9.0 and cube of 3 is 27.0  
Square of 4 is 16.0 and cube of 4 is 64.0  
Square of 10 is 100.0 and cube of 10 is 1000.0  
Square of 9 is 81.0 and cube of 9 is 729.0  
Square of 6 is 36.0 and cube of 6 is 216.0  
Square of 1 is 1.0 and cube of 1 is 1.0  
Square of 2 is 4.0 and cube of 2 is 8.0
```

```
Time taken = 1600767954.635000  
  
Printing Squares and Cubes of array elements:  
Square of 7 is 49.0 and cube of 7 is 343.0  
Square of 5 is 25.0 and cube of 5 is 125.0  
Square of 3 is 9.0 and cube of 3 is 27.0  
Square of 4 is 16.0 and cube of 4 is 64.0  
Square of 10 is 100.0 and cube of 10 is 1000.0  
Square of 9 is 81.0 and cube of 9 is 729.0  
Square of 6 is 36.0 and cube of 6 is 216.0  
Square of 1 is 1.0 and cube of 1 is 1.0  
Square of 2 is 4.0 and cube of 2 is 8.0  
Square of 8 is 64.0 and cube of 8 is 512.0  
  
Time taken = 1600767954.635000  
  
Printing the maximum and minimum element:  
The maximum number in the array is = 10  
The minimum number in the array is = 1  
Time taken = 1600767954.637000  
  
Printing the number of prime number and their sum in the array:  
Prime Numbers:- 5 7 3 2 These are the 4 Prime numbers in the Array  
The sum of the prime numbers is = 17  
Time taken = 1600767954.647000  
  
-----End of the parallel Section-----  
The time for the given computation = 1600767954.630000  
  
Process returned 55 (0x37) execution time : 0.066 s
```

Write a C program using open MP to perform the following matrix operations. Matrix addition, subtraction and multiplication, sum of rows and sum of columns.

Code:

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    omp_set_num_threads(2);
    int n;
    printf("FOR NUMBER OF THREADS = 2\n\n");
    printf("Enter the dimension of the matrix : ");
    scanf("%d", &n);
    int **mul1=NULL, **mul2=NULL, **mul3=NULL, **mul4=NULL;
    printf("=====FOR MULTIPLICATION=====\\n\\n");
    mul1 = (int **)malloc(n * sizeof(int *));
    mul2 = (int **)malloc(n * sizeof(int *));
    mul3 = (int **)malloc(n * sizeof(int *));
    mul4 = (int **)malloc(n * sizeof(int *));
    for(int i=0;i<n;i++)
    {
        mul1[i] = (int *)malloc(n * sizeof(int));
        mul2[i] = (int *)malloc(n * sizeof(int));
        mul3[i] = (int *)malloc(n * sizeof(int));
        mul4[i] = (int *)malloc(n * sizeof(int));
    }
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            mul1[i][j]=rand()%100;
            mul2[i][j]=rand()%100;
        }
    }
    double stamp1 = omp_get_wtime();
    int i, j, k;
    #pragma omp parallel for private (j, k)
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            mul3[i][j]=0;
            for(k=0;k<n;k++)
            {
                mul3[i][j] += mul1[i][k]*mul2[k][j];
            }
        }
    }
    double stamp2 = omp_get_wtime();
```

```

printf("PARALLEL TIME IS: %f\n", (stamp2-stamp1));

double stamp3 = omp_get_wtime();
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul4[i][j] = 0;
        for(int k=0;k<n;k++)
        {
            mul4[i][j] += mul1[i][k]*mul2[k][j];
        }
    }
}
double stamp4 = omp_get_wtime();
printf("SERIAL TIME IS: %f\n", stamp4-stamp3);
printf("\n\n");

printf("====================FOR ADDITION=====================\n\n");

for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul1[i][j] = rand()%100;
    }
}
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul2[i][j] = rand()%100;
    }
}

double stamp5 = omp_get_wtime();
#pragma omp parallel for private (i,j)
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        mul3[i][j] = mul1[i][j]+mul2[i][j];
    }
}
double stamp6 = omp_get_wtime();
printf("PARALLEL TIME IS: %f\n", stamp6-stamp5);

double stamp7 = omp_get_wtime();
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        mul3[i][j] = mul1[i][j]+mul2[i][j];
    }
}
double stamp8 = omp_get_wtime();
printf("SERIAL TIME IS: %f\n", stamp8-stamp7);

printf("\n\n");
printf("====================FOR SUBTRACTION=====================\n\n");

```

```

for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul1[i][j] = rand()%100;
    }
}
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul2[i][j] = rand()%100;
    }
}
double stamp9 = omp_get_wtime();
#pragma omp parallel for private (i,j)
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        mul3[i][j] = mul1[i][j]-mul2[i][j];
    }
}
double stamp10 = omp_get_wtime();
printf("PARALLEL TIME IS: %f\n",stamp10-stamp9);
double stamp11 = omp_get_wtime();
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        mul3[i][j] = mul1[i][j]-mul2[i][j];
    }
}
double stamp12 = omp_get_wtime();
printf("SERIAL TIME IS: %f\n", stamp12-stamp11);
printf("\n\n");

printf("=====SUM OF ROWS=====\\n\\n");
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul1[i][j] = rand()%100;
    }
}
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul2[i][j] = rand()%100;
    }
}

double stamp13 = omp_get_wtime();
int sum=0;
#pragma omp parallel for private (i,j)
//Sum of row elements of matrix 1
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {

```

```

        sum += mul1[i][j];
    }
}
sum=0;
//Sum of row elements of matrix 2
#pragma omp parallel for private (i,j)
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        sum += mul2[i][j];
    }
}
double stamp14 = omp_get_wtime();
printf("PARALLEL TIME IS: %f\n",stamp14-stamp13);

double stamp15 = omp_get_wtime();

sum=0;

//Sum of row elements of matrix 1
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        sum += mul1[i][j];
    }
}
sum=0;
//Sum of row elements of matrix 2

for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        sum += mul2[i][j];
    }
}

double stamp16 = omp_get_wtime();
printf("SERIAL TIME IS: %f\n", stamp16-stamp15);
printf("\n\n");

printf("=====SUM OF COLUMNS=====\\n\\n");
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul1[i][j] = rand()%100;
    }
}
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul2[i][j] = rand()%100;
    }
}

double stamp17 = omp_get_wtime();
sum=0;

```

```

#pragma omp parallel for private (i,j)
//Sum of row elements of matrix 1
for(j=0;j<n;j++)
{
    for(i=0;i<n;i++)
    {
        sum += mul1[i][j];
    }
}
sum=0;
//Sum of row elements of matrix 2
#pragma omp parallel for private (i,j)
for(j=0;j<n;j++)
{
    for(i=0;i<n;i++)
    {
        sum += mul2[i][j];
    }
}
double stamp18 = omp_get_wtime();
printf("PARALLEL TIME IS: %f\n",stamp18-stamp17);

double stamp19 = omp_get_wtime();

sum=0;

//Sum of column elements of matrix 1
for(j=0;j<n;j++)
{
    for(i=0;i<n;i++)
    {
        sum += mul1[i][j];
    }
}
sum=0;
//Sum of column elements of matrix 2

for(j=0;j<n;j++)
{
    for(i=0;i<n;i++)
    {
        sum += mul2[i][j];
    }
}

double stamp20 = omp_get_wtime();
printf("SERIAL TIME IS: %f\n", stamp20-stamp19);
printf("\n\n");
return 0;
}

```

Screenshots:

```
prerna@prerna-VirtualBox:~$ vim matrix2.c
prerna@prerna-VirtualBox:~$ gcc -fopenmp matrix2.c
prerna@prerna-VirtualBox:~$ ./a.out
FOR NUMBER OF THREADS = 4

Enter the dimension of the matrix : 919
=====FOR MULTIPLICATION=====

PARALLEL TIME IS: 1.580504
SERIAL TIME IS: 5.358661

=====FOR ADDITION=====

PARALLEL TIME IS: 0.001186
SERIAL TIME IS: 0.003213

=====FOR SUBTRACTION=====

PARALLEL TIME IS: 0.001300
SERIAL TIME IS: 0.003356

=====SUM OF ROWS=====

PARALLEL TIME IS: 0.016676
SERIAL TIME IS: 0.004720
```

```
=====SUM OF COLUMNS=====

PARALLEL TIME IS: 0.017185
SERIAL TIME IS: 0.005742

prerna@prerna-VirtualBox:~$
```

Write an open MP program using c to find the value of PI

Code:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 8

static long steps = 1000000000;
double step;

int main (int argc, const char *argv[]) {

    int i,j;
    double x;
    double pi, sum = 0.0;
    double start, delta;

    step = 1.0/(double) steps;

    for (j=1; j<= MAX_THREADS; j++) {

        printf(" running on %d threads: ", j);

        omp_set_num_threads(j);

        sum = 0.0;
        double start = omp_get_wtime();

        #pragma omp parallel for reduction(+:sum) private(x)
        for (i=0; i < steps; i++) {
            x = (i+0.5)*step;
            sum += 4.0 / (1.0+x*x);
        }
        pi = step * sum;
        delta = omp_get_wtime() - start;
        printf("PI = %.16g computed in %.4g seconds\n", pi, delta);
    }
}
```

Screenshots:

```
prerna@prerna-VirtualBox:~$ ./a.out
running on 1 threads: PI = 3.141592653589971 computed in 6.98 seconds
running on 2 threads: PI = 3.141592653589901 computed in 3.821 seconds
running on 3 threads: PI = 3.141592653589962 computed in 2.843 seconds
running on 4 threads: PI = 3.141592653589821 computed in 2.28 seconds
running on 5 threads: PI = 3.141592653589595 computed in 2.29 seconds
running on 6 threads: PI = 3.141592653589682 computed in 2.386 seconds
running on 7 threads: PI = 3.141592653589631 computed in 2.287 seconds
running on 8 threads: PI = 3.141592653589769 computed in 2.287 seconds
prerna@prerna-VirtualBox:~$
```

Compare and contrast OpenMP and MPI?

1. MPI stands for *Message Passing Interface*. These are available as API(Application programming interface) or in library form for C,C++ and FORTRAN.
 - A. Different MPI's API are available in market i.e OpenMPI, MPICH, HP-MPI, Intel MPI, etc. Whereas many are freely available like OpenMPI, MPICH etc , other like Intel MPI comes with license i.e you need to pay for it .
 - B. One can use any one of above to parallelize programs . MPI standards maintain that all of these APIs provided by different vendors or groups follow similar standards, so all functions or subroutines in all different MPI API follow similar functionality as well arguments.
 - C. The difference lies in implementation that can make some MPIs API to be more efficient than other. Many commercial CFD-Packages gives user option to select between different MPI API. However HP-MPI as well Intel MPIs are considered to be more efficient in performance.
 - D. When MPI was developed, it was aimed at distributed memory system but now focus is both on distributed as well shared memory system. However it does not mean that with MPI , one cannot run program on shared memory system, it just that earlier, we could not take advantage of shared memory but now we can with latest MPI 3
2. OpenMP stand for *Open Multiprocessing* . OpenMP is basically an add on in compiler. It is available in gcc (gnu compiler) , Intel compiler and with other compilers.
 - A. OpenMP target shared memory systems i.e where processor shared the main memory.
 - B. OpenMP is based on thread approach . It launches a single process which in turn can create n number of thread as desired. It is based on what is called "fork and join method" i.e depending on particular task it can launch desired number of thread as directed by user.
 - C. Programming in OpenMP is relatively easy and involve adding *pragma* directive . User need to tell number of thread it need to use. (Note that launching more thread than number of processing unit available can actually slow down the whole program)

Difference between MPI and openMP:

MPI	OpenMP
Available from different vendor and can be compiled in desired platform with desired compiler. One can use any of MPI API i.e MPICH, OpenMPI or other	OpenMP are hooked with compiler so with gnu compiler and with Intel compiler one have specific implementation. User is at liberty with changing compiler but not with openmp implementation.
MPI support C,C++ and FORTRAN	OpenMP support C,C++ and FORTRAN
MPI target both distributed as well shared memory system	OpenMP target only shared memory system
Based on both process and thread based approach .(Earlier it was mainly process based parallelism but now with MPI 2 and 3 thread based parallelism is there too. Usually a process can contain more than 1 thread and call MPI subroutine as desired	Only thread based parallelism.
Overhead for creating process is one time	Depending on implementation threads can be created and joined for particular task which add overhead
Compilation of MPI program require 1. Adding header file : #include "mpi.h" 2. compiler as:(in linux) mpic++ mpi.cxx -o mpiExe (User need to set environment variable PATH and LD_LIBRARY_PATH to MPI as OpenMPI installed folder or binaries) (For Linux)	Need to add omp.h and then can directly compile code with -fopenmp in Linux environment g++ -fopenmp openmp.cxx -o openmpExe
Data racing is not there if not using any thread in process .	Data racing is inherent in OpenMP model
There are overheads associated with transferring message from one process to another	No such overheads, as thread can share variables

10 MPI function calls

MPI function calls are:

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Send
- MPI_Recv
- MPI_Get_processor_name
- MPI_Init_thread
- MPI_THREAD_SINGLE
- MPI_FLOAT

Write a hello world MPI program?

Code:

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
        processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Screenshots:

```
prerna@prerna-VirtualBox:~$ vim hello1.c
prerna@prerna-VirtualBox:~$ mpicc hello1.c -o hello1
prerna@prerna-VirtualBox:~$ mpirun -np 4 ./hello1
Hello world from processor prerna-VirtualBox, rank 1 out of 4 processors
Hello world from processor prerna-VirtualBox, rank 3 out of 4 processors
Hello world from processor prerna-VirtualBox, rank 2 out of 4 processors
Hello world from processor prerna-VirtualBox, rank 0 out of 4 processors
prerna@prerna-VirtualBox:~$
```

Write an MPI program to perform various tasks (arithmetic operations) using process rank id?

Code:

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int a=10,b=5;
    int c;
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    if(world_rank==0){
        c=a+b;
        printf("Addition operation from processor %s, rank %d out of %d processors\n%d
+ %d = %d\n=====\\n",processor_name, world_rank,
world_size,a,b,c);

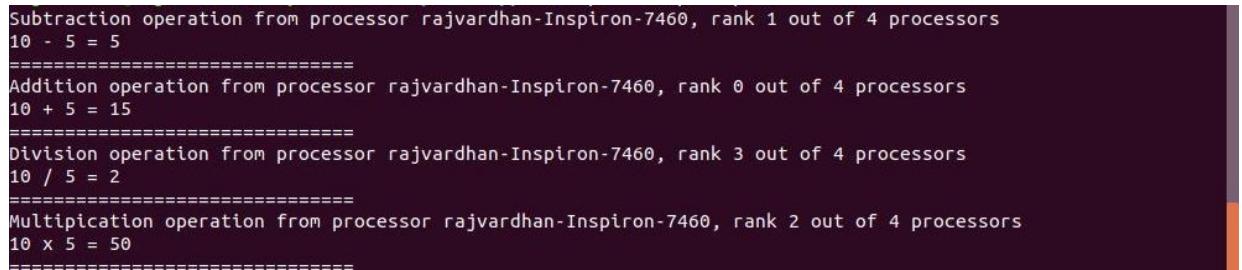
    }
    if(world_rank==1){
        c=a-b;
        printf("Subtraction operation from processor %s, rank %d out of %d
processors\\n%d - %d = %d\\n=====\\n",processor_name,
world_rank, world_size,a,b,c);
    }
    if(world_rank==2){
        c=a*b;
```

```

        printf("Multipication operation from processor %s, rank %d out of %d
processors\n%d x %d = %d\n=====\n",processor_name,
world_rank, world_size,a,b,c);
    }if(world_rank==3){
        c=a/b;
        printf("Division operation from processor %s, rank %d out of %d processors\n%d
/ %d = %d\n=====\n",processor_name, world_rank,
world_size,a,b,c);
    }
    // Finalize the MPI environment.
    MPI_Finalize();
}

```

Screenshots:



A terminal window displaying the output of MPI operations. The output is as follows:

```

Subtraction operation from processor rajvardhan-Inspiron-7460, rank 1 out of 4 processors
10 - 5 = 5
=====
Addition operation from processor rajvardhan-Inspiron-7460, rank 0 out of 4 processors
10 + 5 = 15
=====
Division operation from processor rajvardhan-Inspiron-7460, rank 3 out of 4 processors
10 / 5 = 2
=====
Multipication operation from processor rajvardhan-Inspiron-7460, rank 2 out of 4 processors
10 x 5 = 50
=====
```

Write an MPI program for Message Communication?

Code: (Passing an Array)

```
#include<mpi.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    int rank, size, i, provided;
    float A[10];
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE,&provided);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        for(i=0;i<10;i++)
            A[i] = i;
        for(i=1;i<size; i++)
            MPI_Send(A, 10, MPI_FLOAT, i, 0,MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(A, 10, MPI_FLOAT, 0, 0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    printf("My rank %d of %d\n", rank, size );
    printf("Here are my values for A\n");
    for(i=0; i<10; i++)
        printf("%f", A[i]);
    printf("\n");
    MPI_Finalize();
}
```

Screenshots:

```
prerna@prerna-VirtualBox:~$ vim q3.c
prerna@prerna-VirtualBox:~$ mpicc q3.c -o q3
prerna@prerna-VirtualBox:~$ mpirun -np 4 ./q3
My rank 3 of 4
Here are my values for A
0.0000001.0000002.0000003.0000004.0000005.0000006.0000007.0000008.0000009.00000
0
My rank 0 of 4
Here are my values for A
My rank 1 of 4
Here are my values for A
0.0000001.0000002.0000003.0000004.0000005.0000006.0000007.0000008.0000009.00000
0
My rank 2 of 4
Here are my values for A
0.0000001.0000002.0000003.0000004.0000005.0000006.0000007.0000008.0000009.00000
0
0.0000001.0000002.0000003.0000004.0000005.0000006.0000007.0000008.0000009.00000
0
```

Name : Prerna Sultania
Reg no.: 18BCE0869

LAB ASSESSMENT - 6
Parallel Distribution and computing
(CSE4001)

- (1) Write short notes on performance metrics and bench mark programs of parallel computing.
- (2) Write an open MP program using C for the following and discuss your findings.
 - (a) Fibonacci series
 - (b) Factorial of n elements in an array
 - (c) Number base conversion of the following: decimal to binary, decimal to octal and decimal to hexadecimal and vice-versa
- (3) Write a program using open MP or MPI to compute the following series: Sin(x) and Cos(x)

Answers:

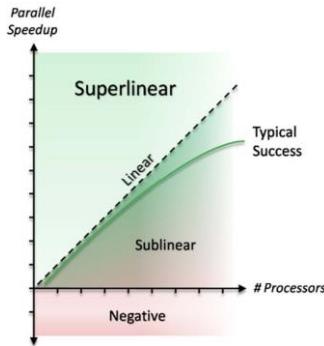
1.

In performance metrice, we mostly measure the performance of parallel applications. Speedup is a measure of performance. It measures the ratio between the sequential execution time and the parallel execution time. Efficiency is a measure of the usage of the computational capacity.

There are 2 distinct classes of performance metrics:

- Performance metrics for processors/cores – assess the performance of a processing unit, normally done by measuring the speed or the number of operations that it does in a certain period of time
- Performance metrics for parallel applications – assess the performance of a parallel application, normally done by comparing the execution time with multiple processing units against the execution time with just one processing R.
- Some of the best known metrics are:
 - MIPS – Millions of Instructions Per Second
 - MFLOPS – Millions of FLoating point Operations Per Second
 - SPECint – SPEC (Standard Performance Evaluation Corporation) benchmarks that evaluate processor performance on integer arithmetic (first release in 1992)
- Run Time: The parallel run time is defined as the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution. • Notation: Serial run time , parallel run time
- The speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel

algorithm to solve the same problem on p processors.



- The cost of solving a problem on a parallel system is defined as the product of run time and the number of processors.
- A cost-optimal parallel system solves a problem with a cost proportional to the execution time of the fastest known sequential algorithm on a single processor.

In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.

Benchmarks are designed to mimic a particular type of workload on a component or system. Synthetic benchmarks do this by specially created programs that impose the workload on the component. Application benchmarks run real-world programs on the system. While application benchmarks usually give a much better measure of real-world performance on a given system, synthetic benchmarks are useful for testing individual components, like a hard disk or networking device.

Benchmarks are particularly important in CPU design, giving processor architects the ability to measure and make tradeoffs in microarchitectural decisions.

CPUs that have many execution units such as a superscalar CPU, a VLIW CPU, or

a reconfigurable computing CPU typically have slower clock rates than a sequential CPU with one or two execution units when built from transistors that are just as fast. Nevertheless, CPUs with many execution units often complete real-world and benchmark tasks in less time than the supposedly faster high-clock-rate CPU.

Given the large number of benchmarks available, a manufacturer can usually find at least one benchmark that shows its system will outperform another system; the other systems can be shown to excel with a different benchmark.

Benchmarking is not easy and often involves several iterative rounds in order to arrive at predictable, useful conclusions. Interpretation of benchmarking data is also extraordinarily difficult. Here is a partial list of common challenges:

- Vendors tend to tune their products specifically for industry-standard benchmarks. Norton SysInfo (SI) is particularly easy to tune for, since it mainly biased toward the speed of multiple operations.

- Some vendors have been accused of "cheating" at benchmarks doing things that give much higher benchmark numbers, but make things worse on the actual likely workload.
- Many benchmarks focus entirely on the speed of computational performance, neglecting other important features of a computer system, such as:
 - Qualities of service, aside from raw performance. Examples of unmeasured qualities of service include security, availability, reliability, execution integrity, serviceability, scalability.

Types of Bench Mark:

1. Real program
 - word processing software
 - tool software of CAD
 - user's application software (i.e.: MIS)
2. Component Benchmark / Microbenchmark
 - core routine consists of a relatively small and specific piece of code.
 - measure performance of a computer's basic components.
3. Kernel
 - contains key codes
 - popular kernel: Livermore loop.
 - results are represented in Mflop/s.

2.(a) factorial

```
#include
<stdio.h>
#include
<omp.h> int
fib(int n)
{
int i, j, r;
omp_lock_t *lock[n];
omp_init_lock(&lock[n]);
omp_set_lock( &(lock[n]) ); if (n<2)
return n; else
{
```

```
#pragma omp task shared(j) firstprivate(n)
```

```
j=fib(n-2);
```

```
#pragma omp taskwait

r = i+j;
}

omp_unset_lock( &(lock[n]) );

return r;
}

int main() {

int n; printf("18BCE2402\n\n\n");

printf("Enter a number: ");

scanf("%d", &n); printf("\n\n");

omp_set_dynamic(0);

omp_set_num_threads(4);

#pragma omp parallel shared(n)

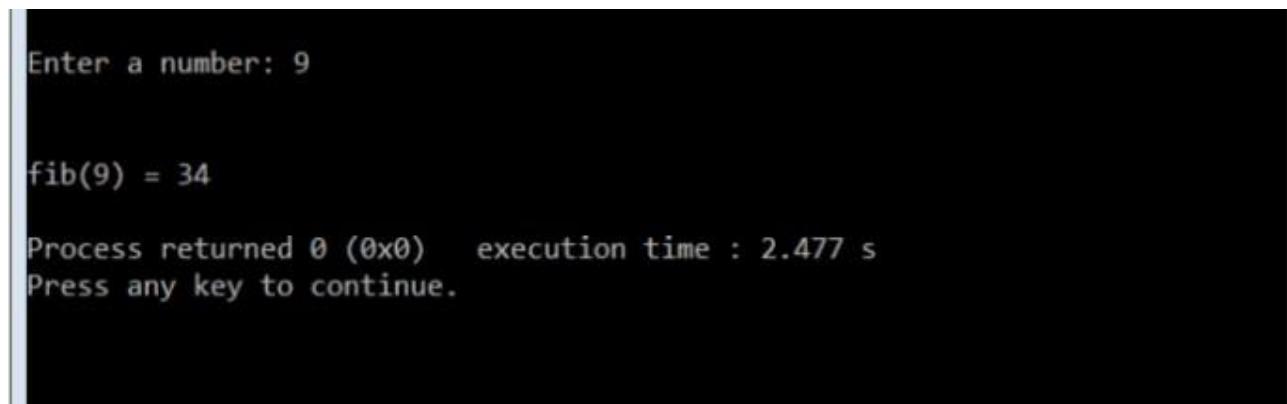
{

#pragma omp single

printf ("fib(%d) = %d\n", n, fib(n));

}

}
```



```
Enter a number: 9

fib(9) = 34

Process returned 0 (0x0)  execution time : 2.477 s
Press any key to continue.
```

(b) natural number

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void) {
    int n = 10;
    int factorial[n];
    factorial[1] = 1;
    int *proda;
    #pragma omp parallel
    {
        int ithread = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        #pragma omp single
        {
            proda = malloc(nthreads * sizeof *proda);
            proda[0] = 1;
        }
        int prod = 1;
        #pragma omp for schedule(static) nowait
        for (int i=2; i<n; i++) {
            prod *= i;
            factorial[i] = prod;
        }
        proda[ithread+1] = prod;
        #pragma omp barrier
        int offset = 1;
        for(int i=0; i<(ithread+1); i++) offset *= proda[i];
    }
```

```
#pragma omp for schedule(static)
for(int i=1; i<n; i++) factorial[i] *= offset;
}
free(proda);
for(int i=1; i<n; i++) printf("%d\n", factorial[i]); putchar('\n');
}
```

Output:

```
1
4
18
96
600
4320
35280
322560
3265920

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

(c) conversion

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#include <math.h>
#include <string.h>
```

```
void f1 (void);
```

```
void f2 (void);
```

```
void f3 (void);
```

```
void f4 (void);
```

```
int convert1(long long n);
long long convert2(int n);
int convertDecimalToOctal(int decimalNumber);
long long convertOctalToDecimal(int octalNumber);
int main()
{
    int i,tid;
#pragma omp parallel sections
    {
        #pragma omp section
        {
            f1();
        }
        #pragma omp section
        {
            f2();
        }
        #pragma omp section
        {
            f3();
        }
        #pragma omp section
        {
            f4();
        }
    }
    return 0;
}
```

```
}
```

```
void f1 (void)
```

```
{
```

```
long long n=111;
```

```
    printf("\n\nBinary number:111 ");
```

```
    printf("%lld in binary = %d in decimal\n", n, convert1(n));
```

```
    return 0;
```

```
}
```

```
void f2 (void)
```

```
{
```

```
    int n=10;
```

```
    printf("\n\nDecimal number:10 ");
```

```
    printf("%d in decimal = %lld in binary", n, convert2(n));
```

```
    return 0;
```

```
}
```

```
void f3 (void)
```

```
{
```

```
    int decimalNumber=10;
```

```
    printf("\n\nEnter a decimal number:10 ");
```

```
        printf("% d in decimal = % d in octal", decimalNumber, convertDecimalToOctal(decimalNumber));
```

```
    return 0;
```

```
}
```

```
void f4 (void)
{
    int octalNumber=111;

    printf("\n\nEnter an octal number:111 ");

    printf("%d in octal = %lld in decimal", octalNumber,
convertOctalToDecimal(octalNumber));

    return 0;
}

int convert1(long long n) {
    int dec = 0, i = 0, rem;
    while (n != 0) {
        rem = n % 10;
        n /= 10;
        dec += rem * pow(2, i);
        ++i;
    }
    return dec;
}

long long convert2(int n) {
    long long bin = 0;
    int rem, i = 1, step = 1;
    while (n != 0) {
        rem = n % 2;
        printf("Step %d: %d/2, Remainder = %d, Quotient = %d\n", step++, n, rem, n /
2);
        n /= 2;
    }
}
```

```
n /= 2;
```

```
bin += rem * i;
```

```
i *= 10;
```

```
}return bin;
```

```
}
```

```
int convertDecimalToOctal(int decimalNumber)
```

```
{
```

```
int octalNumber = 0, i = 1;
```

```
while (decimalNumber != 0)
```

```
{
```

```
octalNumber += (decimalNumber % 8) * i;
```

```
decimalNumber /= 8;
```

```
i *= 10;
```

```
}
```

```
return octalNumber;
```

```
}
```

```
long long convertOctalToDecimal(int octalNumber)
```

```
{
```

```
int decimalNumber = 0, i = 0;
```

```
while(octalNumber != 0)
```

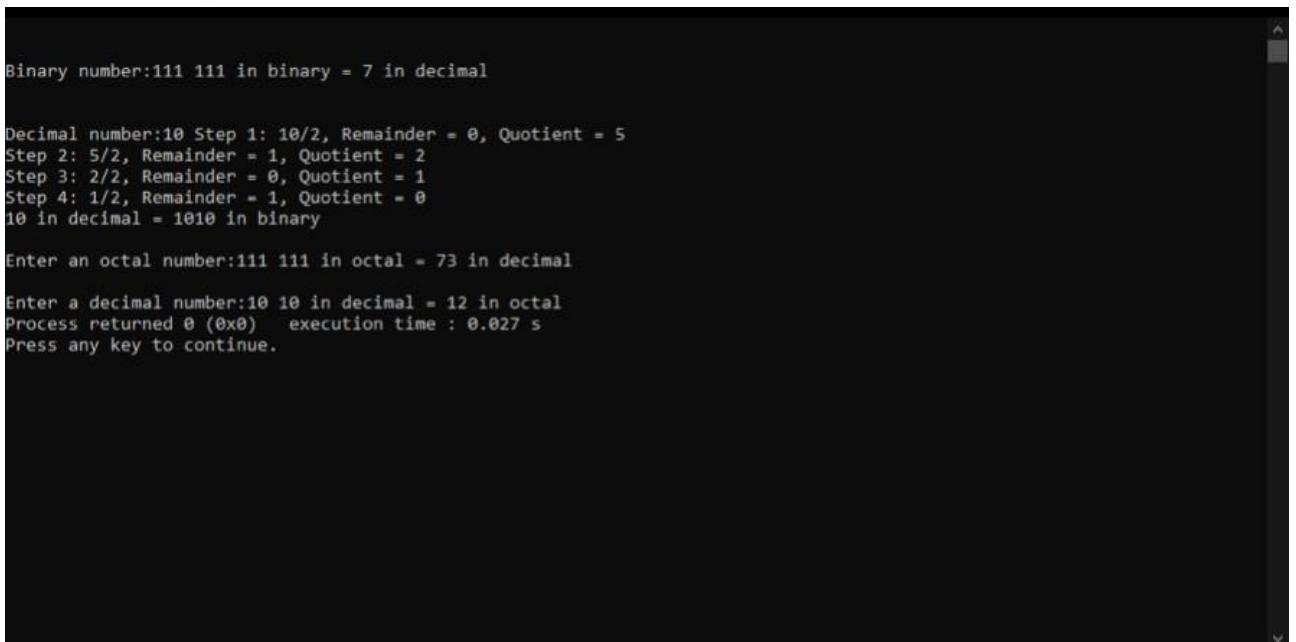
```
{
```

```
decimalNumber += (octalNumber% 10) * pow(8,i);
```

```
++i;
```

```
    octalNumber/=10;  
}  
  
i = 1;  
  
return decimalNumber;  
}
```

Output:



```
Binary number:111 111 in binary = 7 in decimal  
  
Decimal number:10 Step 1: 10/2, Remainder = 0, Quotient = 5  
Step 2: 5/2, Remainder = 1, Quotient = 2  
Step 3: 2/2, Remainder = 0, Quotient = 1  
Step 4: 1/2, Remainder = 1, Quotient = 0  
10 in decimal = 1010 in binary  
  
Enter an octal number:111 111 in octal = 73 in decimal  
  
Enter a decimal number:10 10 in decimal = 12 in octal  
Process returned 0 (0x0)   execution time : 0.027 s  
Press any key to continue.
```

3. sin(x) and cos(x)

Code:(sine)

```
#include<stdio.h>  
  
#include<conio.h>  
  
#include<omp.h>
```

```
int main()
{
    int i, n1=32,n2=35;
    float x1=33,x2=21 ,sum=0, t;
    #pragma omp parallel
    {
        printf("\n-----Sin(x)----- ");
        x1 = x1 * 3.14 / 180;
        t = x1;
        for (i = 1; i <= n1; i++)
        {
            t=(t*(-1)*x1*x1)/(2*i*(2*i+1));
            sum = sum + t;
        }
        printf(" \nThe value of Sin(%f) = %.4f\n", x1, sum);
    }
    getch();
    return 0;
}
```

Code:(cosine)

```
#include<stdio.h>
#include<conio.h>
#include<omp.h>
int main()
{
    int i, n1=32,n2=35;
    float x1=33,x2=21 ,sum=0, t;
    #pragma omp parallel
    {
        {
            printf("\n\n-----Cos(x) -----");
            x2 = x2 * 3.14159 / 180;
            t = x2;
            sum = x2;
            for (i = 1; i <= n2; i++)
            {
                t = t*(-1)*x2*x2/(2*i*(2*i-1));
                sum = sum + t;
            }
            printf("\n The value of Cos(%f) is : %.4f\n", x2, sum);
        }
        getch();
    }
    return 0;
}
```

Code:

-----Cos(x)-----
The value of Cos(0.366519) is : 0.3422

-----Cos(x)-----
The value of Cos(0.006397) is : 0.0064

-----Cos(x)-----
The value of Cos(0.000112) is : 0.0001

-----Cos(x)-----
The value of Cos(0.000002) is : 0.0000

-----Cos(x)-----
The value of Cos(0.000000) is : 0.0000

-----Cos(x)-----
The value of Cos(0.000000) is : 0.0000

Parallel and Distributed Computing

CSE4001

Lab Assignment 2

Shreya Maheshwari

18BCE0167

Question-

- a) Using OpenMP, Design, develop and run a multi-threaded program to perform Loop work Sharing.
- b) Using OpenMP, Design, develop and run a multi-threaded program to perform Section work sharing

a) Loop work sharing

Code:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N    10

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];

    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
    tid = omp_get_thread_num();
```

```

if (tid == 0)
{
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}
printf("Thread %d starting...\n", tid);

#pragma omp for schedule(dynamic,chunk)
for (i=0; i<N; i++)
{
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
}

}

}

```

Output:

```

cisco@ubuntu:~$ gedit loop.c
cisco@ubuntu:~$ export OMP_NUM_THREADS=4
cisco@ubuntu:~$ gcc loop.c -fopenmp -o loop
cisco@ubuntu:~$ ./loop
Thread 1 starting...
Thread 1: c[0]= 0.000000
Thread 1: c[1]= 2.000000
Thread 1: c[2]= 4.000000
Thread 1: c[3]= 6.000000
Thread 1: c[4]= 8.000000
Thread 1: c[5]= 10.000000
Thread 1: c[6]= 12.000000
Thread 1: c[7]= 14.000000
Thread 1: c[8]= 16.000000
Thread 1: c[9]= 18.000000
Number of threads = 4
Thread 0 starting...
Thread 3 starting...
Thread 2 starting...
cisco@ubuntu:~$ //Shreya Maheshwari 18BCE0167

```

b) Section work sharing

Code:

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 10

int main (int argc, char *argv[])
{
int i, nthreads, tid;
float a[N], b[N], c[N], d[N];

for (i=0; i<N; i++) {
a[i] = i * 1.5;
b[i] = i + 22.35;
c[i] = d[i] = 0.0;
}

#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
tid = omp_get_thread_num();
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
printf("Thread %d starting...\n",tid);

#pragma omp sections nowait
{
#pragma omp section
{
printf("Thread %d doing section 1\n",tid);
for (i=0; i<N; i++)
{
c[i] = a[i] + b[i];
printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
}

#pragma omp section
{

```

```
printf("Thread %d doing section 2\n",tid);
for (i=0; i<N; i++)
{
    d[i] = a[i] * b[i];
    printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
}
}

printf("Thread %d done.\n",tid);

}
```

Output:

```
cisco@ubuntu:~$ gedit section.c
cisco@ubuntu:~$ export OMP_NUM_THREADS=4
cisco@ubuntu:~$ gcc section.c -fopenmp -o section
cisco@ubuntu:~$ ./section
Thread 1 starting...
Thread 1 doing section 1
Thread 1: c[0]= 22.350000
Thread 1: c[1]= 24.850000
Thread 1: c[2]= 27.350000
Thread 1: c[3]= 29.850000
Thread 1: c[4]= 32.349998
Thread 1: c[5]= 34.849998
Thread 1: c[6]= 37.349998
Thread 1: c[7]= 39.849998
Thread 1: c[8]= 42.349998
Thread 1: c[9]= 44.849998
Thread 1 doing section 2
Thread 1: d[0]= 0.000000
Thread 1: d[1]= 35.025002
Thread 1: d[2]= 73.050003
Thread 1: d[3]= 114.075005
Thread 1: d[4]= 158.100006
Thread 1: d[5]= 205.125000
Thread 1: d[6]= 255.150009
Thread 1: d[7]= 308.175018
Thread 1: d[8]= 364.200012
Thread 1: d[9]= 423.225006
Thread 1 done.
Number of threads = 4
Thread 0 starting...
Thread 0 done.
Thread 3 starting...
Thread 3 done.
Thread 2 starting...
Thread 2 done.
cisco@ubuntu:~$ //Shreya Maheshwari 18BCE0167
```

Parallel and Distributed Computing

CSE4001

Lab Assignment 3

Shreya Maheshwari

18BCE0167

Question-

- a) Using OpenMP, Design, develop and run a multi-threaded program to perform Combined parallel loop reduction.
- b) Using OpenMP, Design, develop and run a multi-threaded program to perform and Orphaned parallel loop reduction.

a) Combined parallel loop reduction

Code:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int i, n;
    float a[100], b[100], sum;

    n = 100;
    for (i=0; i < n; i++)
        a[i] = b[i] = i * 1.0;
    sum = 0.0;

    #pragma omp parallel for reduction(+:sum)
    for (i=0; i < n; i++)
        sum = sum + (a[i] * b[i]);

    printf(" Sum = %f\n",sum);
```

```
}
```

Output:

```
cisco@ubuntu:~$ gedit reduction.c
cisco@ubuntu:~$ gcc reduction.c -fopenmp -o reduction
cisco@ubuntu:~$ ./reduction
Sum = 328350.000000
cisco@ubuntu:~$ //Shreya Maheshwari 18BCE0167
```

b) Orphaned parallel loop reduction

Code:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define VECLEN 10

float a[VECLEN], b[VECLEN], sum;

float dotprod ()
{
int i,tid;

tid = omp_get_thread_num();
#pragma omp for reduction(+:sum)
for (i=0; i < VECLEN; i++)
{
sum = sum + (a[i]*b[i]);
printf(" tid= %d i=%d\n",tid,i);
}
}

int main (int argc, char *argv[]) {
int i;

for (i=0; i < VECLEN; i++)
a[i] = b[i] = 1.0 * i;
```

```
sum = 0.0;

#pragma omp parallel
dotprod();

printf("Sum = %f\n",sum);

}
```

Output:

```
cisco@ubuntu:~$ gedit orphaned.c
cisco@ubuntu:~$ export OMP_NUM_THREADS=4
cisco@ubuntu:~$ gcc orphaned.c -fopenmp -o orphaned
cisco@ubuntu:~$ ./orphaned
tid= 1 i=3
tid= 1 i=4
tid= 1 i=5
tid= 0 i=0
tid= 0 i=1
tid= 0 i=2
tid= 3 i=8
tid= 3 i=9
tid= 2 i=6
tid= 2 i=7
Sum = 285.000000
cisco@ubuntu:~$ //Shreya Maheshwari 18BCE0167
```

Parallel and Distributed Computing

CSE4001

Lab Assignment 4

Shreya Maheshwari

18BCE0167

Question-

- a) Using MPI, Design, develop and run Broadcast communication (MPI_Bcast) using MPI_Send and MPI_Recv.
- b) Using MPI, Design, develop and run Reduce communication for dot product operation.

a)

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
MPI_Comm communicator)

{
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);
    if (world_rank == root) {
        int i;
```

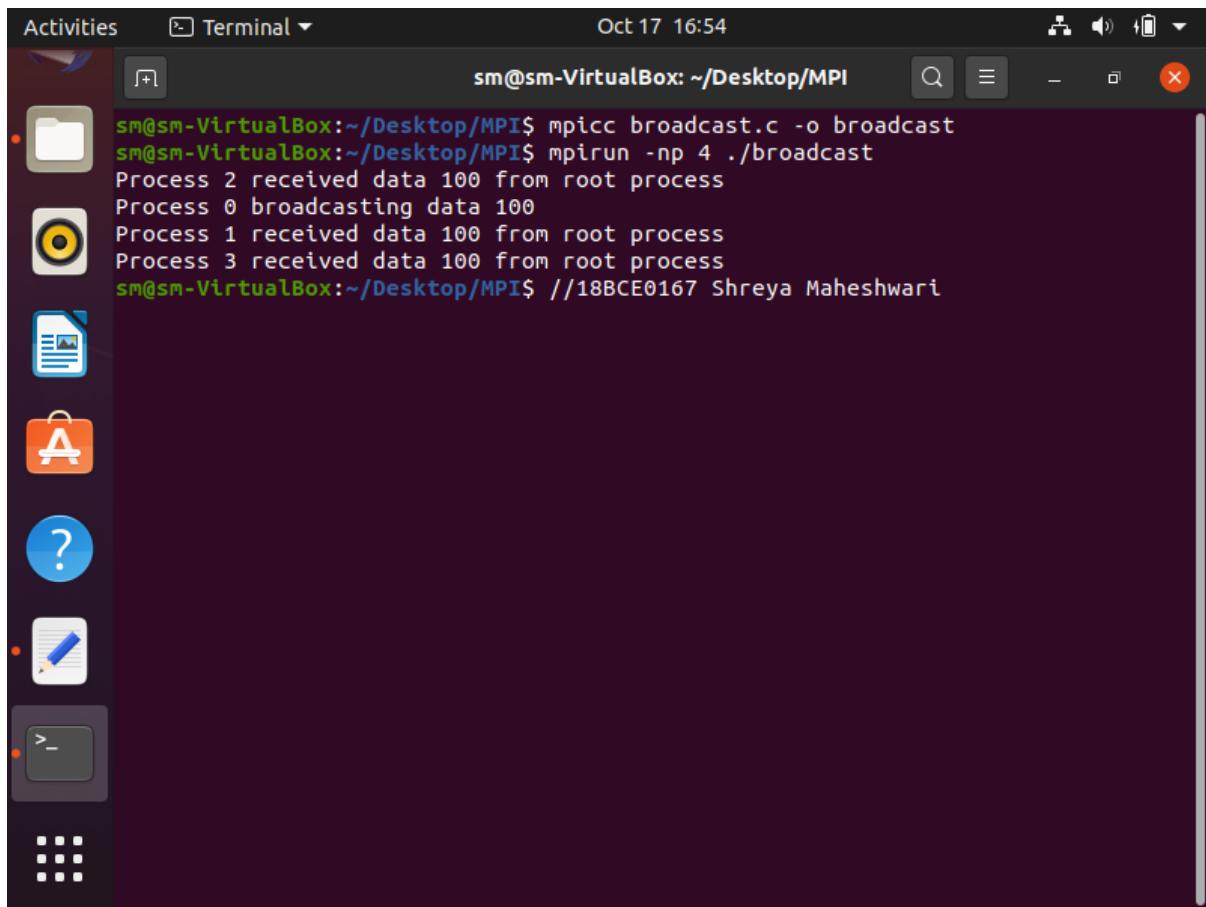
```

for (i = 0; i < world_size; i++) {
    if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
    }
}
} else {
    MPI_Recv(data, count, datatype, root, 0, communicator,
    MPI_STATUS_IGNORE);
}

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int data;
    if (world_rank == 0) {
        data = 100;
        printf("Process 0 broadcasting data %d\n", data);
        my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } else {
        my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf("Process %d received data %d from root process\n", world_rank, data);
    }
    MPI_Finalize();
}

```

Output:

A screenshot of an Ubuntu desktop environment. On the left is a vertical dock with icons for file manager, terminal, browser, and other applications. The main area shows a terminal window titled "Terminal" with the command "sm@sm-VirtualBox: ~/Desktop/MPI". The terminal output is:

```
sm@sm-VirtualBox:~/Desktop/MPI$ mpicc broadcast.c -o broadcast
sm@sm-VirtualBox:~/Desktop/MPI$ mpirun -np 4 ./broadcast
Process 2 received data 100 from root process
Process 0 broadcasting data 100
Process 1 received data 100 from root process
Process 3 received data 100 from root process
sm@sm-VirtualBox:~/Desktop/MPI$ //18BCE0167 Shreya Maheshwari
```

b)

Code:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
/* Define length of dot product vectors */
#define VECLEN 10
int main (int argc, char* argv[])
{
    int i,myid, numprocs, len=VECLEN;
    double *a, *b;
    double mysum, allsum;
    /* MPI Initialization */
    MPI_Init (&argc, &argv);
}
```

```
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &myid);
/*
```

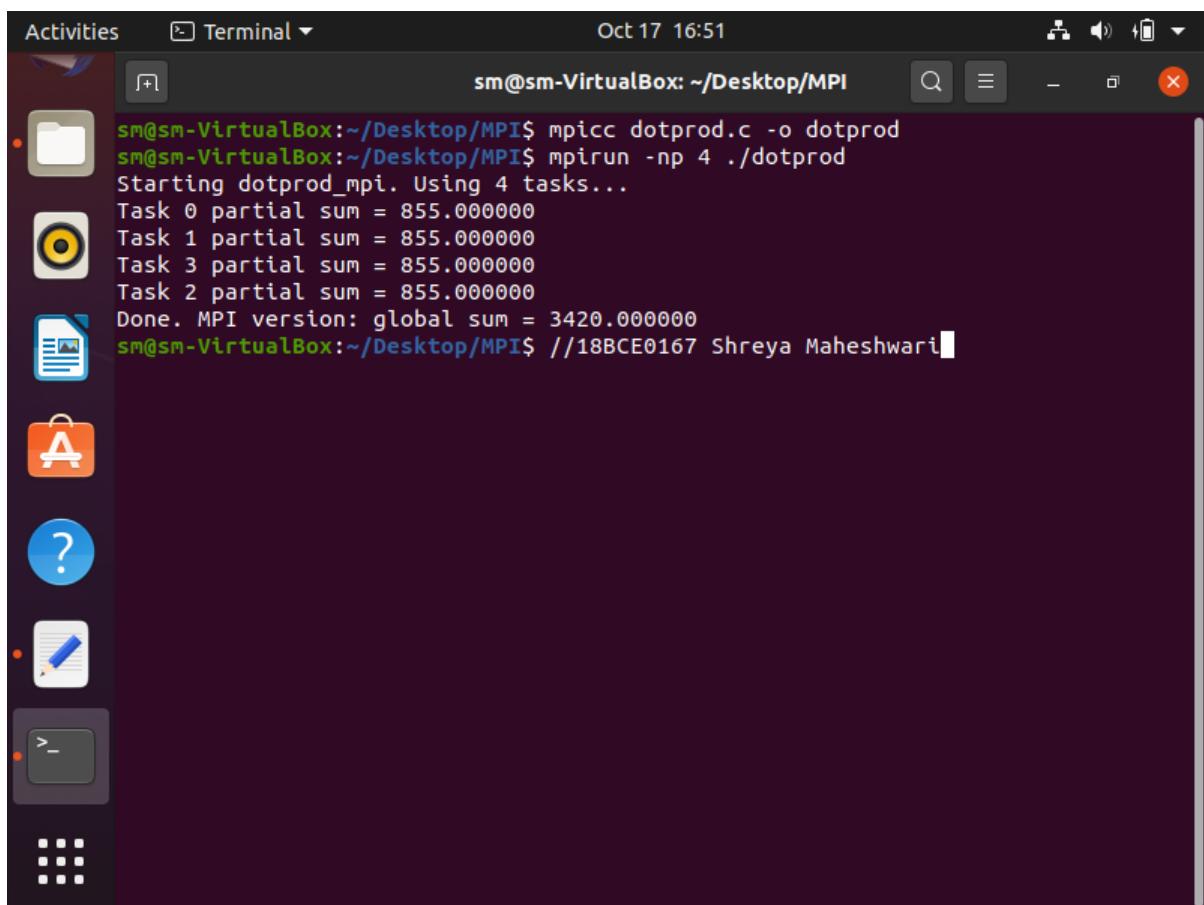
Each MPI task performs the dot product, obtains its partial sum, and then calls MPI_Reduce to obtain the global sum.

```
*/
```

```
if (myid == 0)
printf("Starting dotprod_mpi. Using %d tasks...\n",numprocs);
/* Assign storage for dot product vectors */
a = (double*) malloc (len*sizeof(double));
b = (double*) malloc (len*sizeof(double));
/* Initialize dot product vectors */
for (i=0; i<len; i++) {
    a[i]=i;
    b[i]=a[i]+2*i;
}
/* Perform the dot product */
mysum = 0.0;
for (i=0; i<len; i++)
{
    mysum += a[i] * b[i];
}
printf("Task %d partial sum = %f\n",myid, mysum);
/* After the dot product, perform a summation of results on each node */
MPI_Reduce (&mysum, &allsum, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
if (myid == 0)
printf ("Done. MPI version: global sum = %f \n", allsum);
```

```
free (a);  
free (b);  
MPI_Finalize();  
}
```

Output:



The screenshot shows a terminal window titled "Terminal" in the Activities overview. The terminal window has a dark background and displays the following command-line session:

```
sm@sm-VirtualBox:~/Desktop/MPI$ mpicc dotprod.c -o dotprod  
sm@sm-VirtualBox:~/Desktop/MPI$ mpirun -np 4 ./dotprod  
Starting dotprod_mpi. Using 4 tasks...  
Task 0 partial sum = 855.000000  
Task 1 partial sum = 855.000000  
Task 3 partial sum = 855.000000  
Task 2 partial sum = 855.000000  
Done. MPI version: global sum = 3420.000000  
sm@sm-VirtualBox:~/Desktop/MPI$ //18BCE0167 Shreya Maheshwari
```

Parallel and Distributed Computing

CSE4001

Lab Assignment 5

Shreya Maheshwari

18BCE0167

Question-

- a) Using MPI, Design, develop and run matrix multiplication using MPI_Send and MPI_Recv. In this code, the master task distributes a matrix multiply operation to numtasks-1 worker tasks.
- b) Using MPI, Design, develop and compute pi value using MPI_Send and MPI_Recv.

a)

Code:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define MATSIZE 6
#define NRA MATSIZE
#define NCA MATSIZE
#define NCB MATSIZE
#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2

int main (int argc, char *argv[])
{
    int    numtasks,
          taskid,
          numworkers,
          source,
```

```

dest,
mtype,
rows,
averow, extra, offset,
i, j, k, rc;
double      a[NRA][NCA],
            b[NCA][NCB],
            c[NRA][NCB];
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
if (numtasks < 2 ) {
    printf("Need at least two MPI tasks. Quitting...\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
    exit(1);
}
numworkers = numtasks-1;

/**************** master task *****/
if (taskid == MASTER)
{
    printf("mpi_mm has started with %d tasks.\n",numtasks);
    //printf("Initializing arrays...\n");
    for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
            a[i][j]= i+j;
    for (i=0; i<NCA; i++)
        for (j=0; j<NCB; j++)
            b[i][j]= i*j;

    /* Measure start time */
    double start = MPI_Wtime();

    /* Send matrix data to the worker tasks */
    averow = NRA/numworkers;
    extra = NRA%numworkers;
    offset = 0;
    mtype = FROM_MASTER;
    for (dest=1; dest<=numworkers; dest++)
    {

```

```

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

/* Receive results from worker tasks */
mtype = FROM_WORKER;
for (i=1; i<=numworkers; i++)
{
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
    MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
              MPI_COMM_WORLD, &status);
    printf("Received results from task %d\n",source);
}

/* Print results */

printf("*****\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
{
    printf("\n");
    for (j=0; j<NCB; j++)
        printf("%6.2f ", c[i][j]);
}
printf("\n*****\n");

/* Measure finish time */
double finish = MPI_Wtime();
printf("Done in %f seconds.\n", finish - start);
}

```

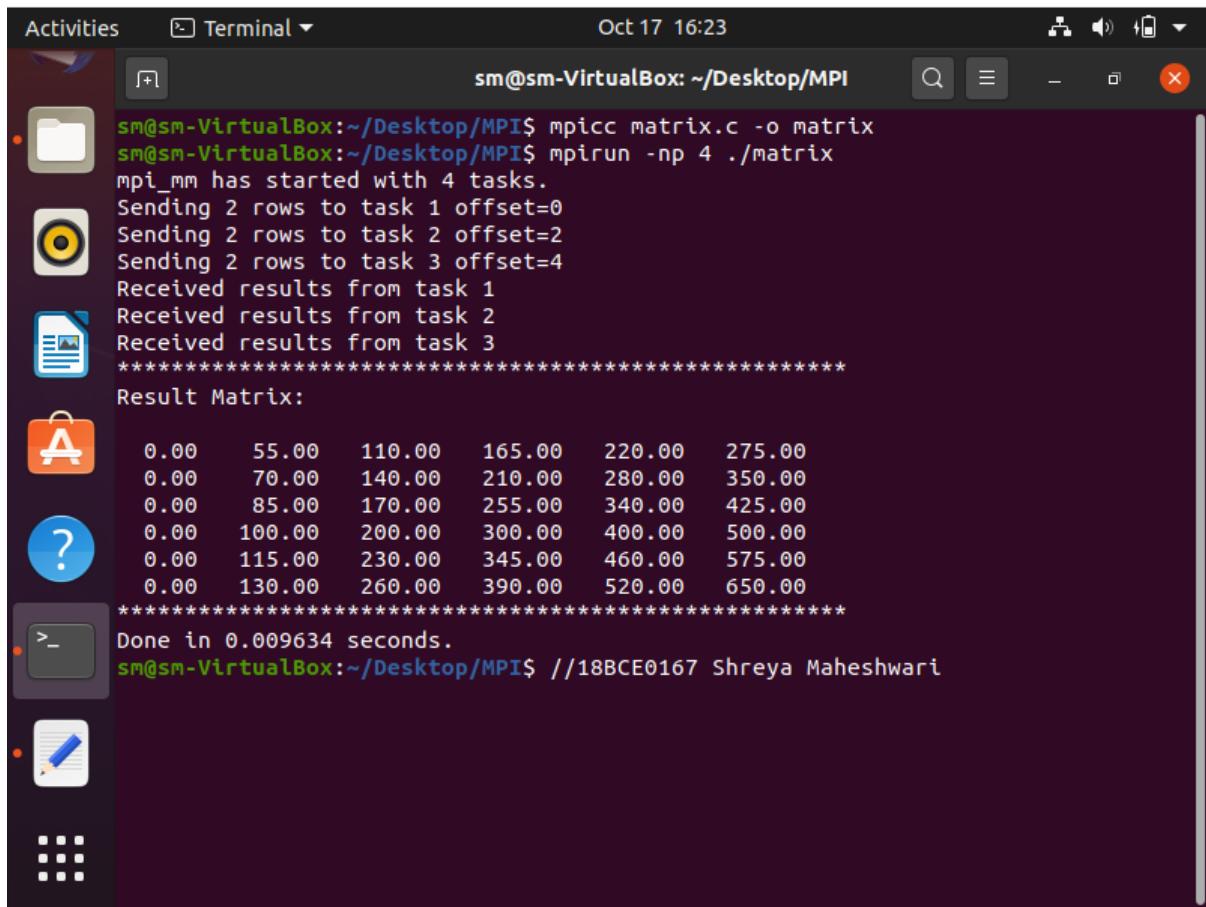
```

***** worker task *****/
if (taskid > MASTER)
{
    mtype = FROM_MASTER;
    MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype,
MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype,
MPI_COMM_WORLD, &status);
    MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD, &status);
    MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD, &status);

    for (k=0; k<NCB; k++)
        for (i=0; i<rows; i++)
        {
            c[i][k] = 0.0;
            for (j=0; j<NCA; j++)
                c[i][k] = c[i][k] + a[i][j] * b[j][k];
        }
    mtype = FROM_WORKER;
    MPI_Send(&offset, 1, MPI_INT, MASTER, mtype,
MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, MASTER, mtype,
MPI_COMM_WORLD);
    MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD);
}
MPI_Finalize();
}

```

Output:



The screenshot shows a terminal window titled "Terminal" with the command line "sm@sm-VirtualBox: ~/Desktop/MPI". The output of the command "mpirun -np 4 ./matrix" is displayed, showing the process of sending and receiving data between four tasks. The resulting matrix is printed as follows:

```
sm@sm-VirtualBox:~/Desktop/MPI$ mpicc matrix.c -o matrix
sm@sm-VirtualBox:~/Desktop/MPI$ mpirun -np 4 ./matrix
mpi_mm has started with 4 tasks.
Sending 2 rows to task 1 offset=0
Sending 2 rows to task 2 offset=2
Sending 2 rows to task 3 offset=4
Received results from task 1
Received results from task 2
Received results from task 3
*****
Result Matrix:
*****
0.00    55.00   110.00   165.00   220.00   275.00
0.00    70.00   140.00   210.00   280.00   350.00
0.00    85.00   170.00   255.00   340.00   425.00
0.00   100.00   200.00   300.00   400.00   500.00
0.00   115.00   230.00   345.00   460.00   575.00
0.00   130.00   260.00   390.00   520.00   650.00
*****
Done in 0.009634 seconds.
sm@sm-VirtualBox:~/Desktop/MPI$ //18BCE0167 Shreya Maheshwari
```

b)

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <math.h>
#define SEED 35791246

int main(int argc, char* argv[])
{
    long niter = 1000000;
    int myid;
    double x,y;
    int i, count=0;
    double z;
    double pi;
    int nodenum;
```

```

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &nodenumber);
int received[nodenumber];
long recvniter[nodenumber];
srand(SEED+myid);

if(myid != 0)
{
    for (i=0; i<niter; ++i)
    {
        x= ((double)rand())/RAND_MAX;
        y =((double)rand())/RAND_MAX;
        z = sqrt(x*x+y*y);
        if (z<=1)
        {
            count++;
        }
    }
    for(i=0; i<nodenumber; ++i)
    {
        MPI_Send(&count,
                 1,
                 MPI_INT,
                 0,
                 1,
                 MPI_COMM_WORLD);

        MPI_Send(&niter,
                 1,
                 MPI_LONG,
                 0,
                 2,
                 MPI_COMM_WORLD);
    }
}
else if (myid == 0)
{
    for(i=0; i<nodenumber; ++i)
    {
        MPI_Recv(&received[i],
                 nodenum,
                 MPI_INT,

```

```

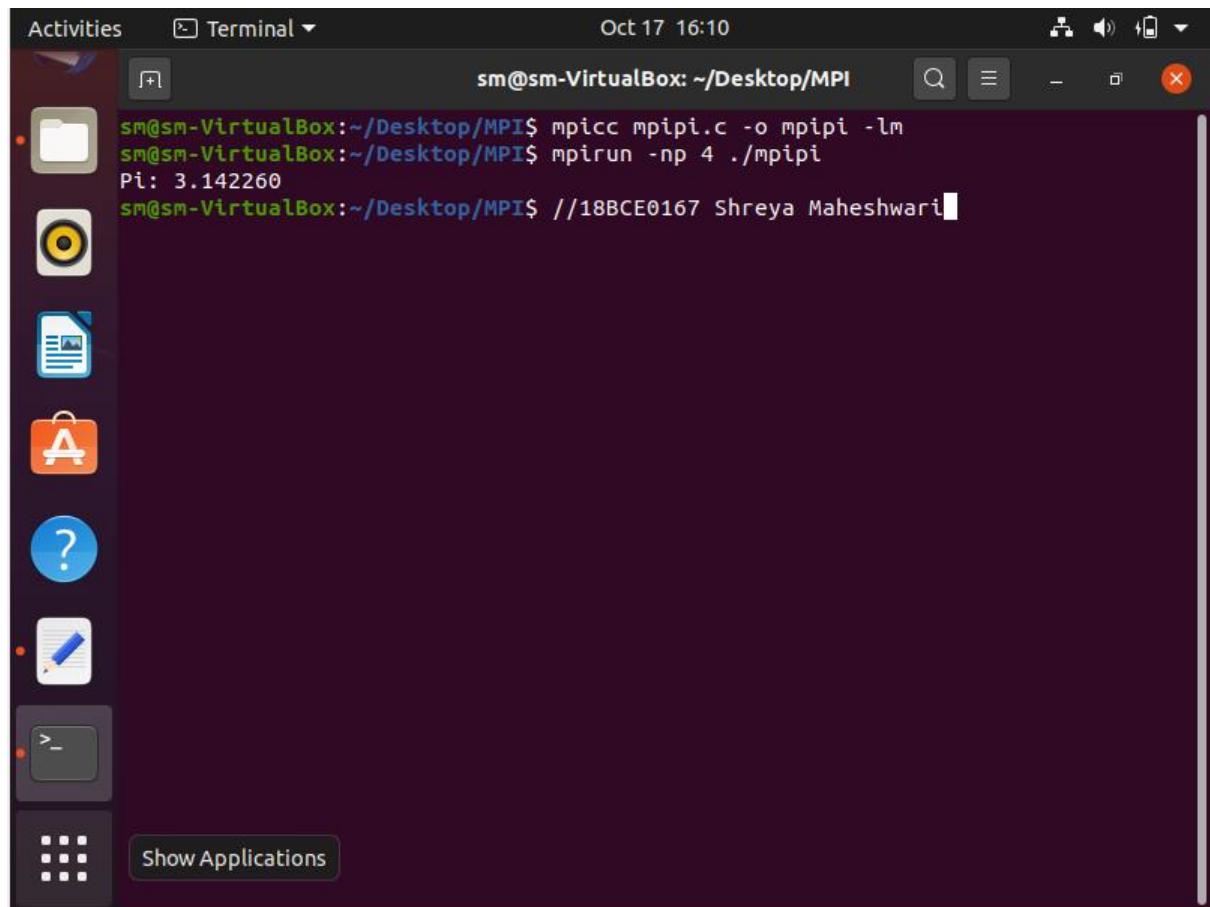
        MPI_ANY_SOURCE,
        1,
        MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Recv(&recvniter[i],
            nodenum,
            MPI_LONG,
            MPI_ANY_SOURCE,
            2,
            MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
}
}

if (myid == 0)
{
    int finalcount = 0;
    long finalniter = 0;
    for(i = 0; i<nodenum; ++i)
    {
        finalcount += recieveed[i];
        finalniter += recvniter[i];
    }
    pi = ((double)finalcount/(double)finalniter)*4.0;
    printf("Pi: %f\n", pi);
}

MPI_Finalize();
return 0;
}

```

Output:



Parallel and Distributed Computing

CSE4001

Lab Assignment 6

Shreya Maheshwari

18BCE0167

Question-

- a) Using CUDA, Develop and run a multi-threaded program to perform and print dot product operation.

Link to Google colab

<https://colab.research.google.com/drive/1NgMlyJhPWss8jq4X8vI9XT4dxfl9gRKKB?usp=sharing>

Code:

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#define N (3)
#define THREADS_PER_BLOCK (1)
#define N_BLOCKS (N/THREADS_PER_BLOCK)
/* Function to generate a random integer between 1-10 */
void random_ints (int *a, int n)
{
    int i;
    srand(time(NULL)); //Seed rand() with current time
    for(i=0; i<n; i++)
    {
        a[i] = rand()%10 + 1;
    }
    return;
}

/* Kernel that adds two integers a & b, stores result in c */
__global__ void add(int *a, int *b, int *c) {
//global indicates function that runs on
```

```

//device (GPU) and is called from host (CPU) code
int index = threadIdx.x + blockIdx.x * blockDim.x;
c[index] = a[index] + b[index];
}

/* Kernel for dot product */
__global__ void dot(int *a, int *b, int *c)
{
    __shared__ int product[THREADS_PER_BLOCK]; //All threads in a block must be able
    int index = threadIdx.x + blockIdx.x * blockDim.x; //index
    product[threadIdx.x] = a[index] * b[index]; //result of elementwise
    //multiplication goes into product
    if(index==0) *c = 0;
    __syncthreads();
    //Sum the elements serially to obtain dot product
    if( 0 == threadIdx.x ) //Pick one thread to sum, otherwise all will execute
    {
        int sum = 0;
        for(int j=0; j < THREADS_PER_BLOCK; j++) sum += product[j];
        atomicAdd(c, sum);
    }
}

int main(void)
{
    int *a, *b, *c, *dotProduct; //host copies of a,b,c etc
    int *d_a, *d_b, *d_c, *d_dotProduct; //device copies of a,b,c etc
    int size = N * sizeof(int); //size of memory that needs to be allocated
    int i=0;
    //Allocate space for device copies of a,b,c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    a = (int *)malloc(size); random_ints(a,N);
    b = (int *)malloc(size); random_ints(b,N);
    c = (int *)malloc(size);

    //Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    //Launch add() kernel on GPU
    add<<<N_BLOCKS,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
}

```

```

//Output
printf("a = { ");
for (i=0; i<N; i++) printf(" %d",a[i]);
printf(" }\n");
printf("b = { ");
for (i=0; i<N; i++) printf(" %d",b[i]);
printf(" }\n");

//Calculate dot product of a & b
dotProduct = (int *)malloc(sizeof(int)); //Allocate host memory to dot
Product
*dotProduct = 0;
cudaMalloc((void **) &d_dotProduct, sizeof(int)); //Allocate device mem
ory to d_dotProduct
dot<<<N_BLOCKS,THREADS_PER_BLOCK>>(d_a, d_b, d_dotProduct);
cudaMemcpy(dotProduct, d_dotProduct, sizeof(int), cudaMemcpyDeviceToHo
st); //Copy result into dotProduct

//Output
printf("\ndot(a,b) = %d\n", *dotProduct);

//Cleanup
free(a);
free(b);
free(c);
free(dotProduct);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
cudaFree(d_dotProduct);
return 0;
}

```

Output:

a = { 10 4 5 }

b = { 10 4 5 }

dot(a,b) = 141

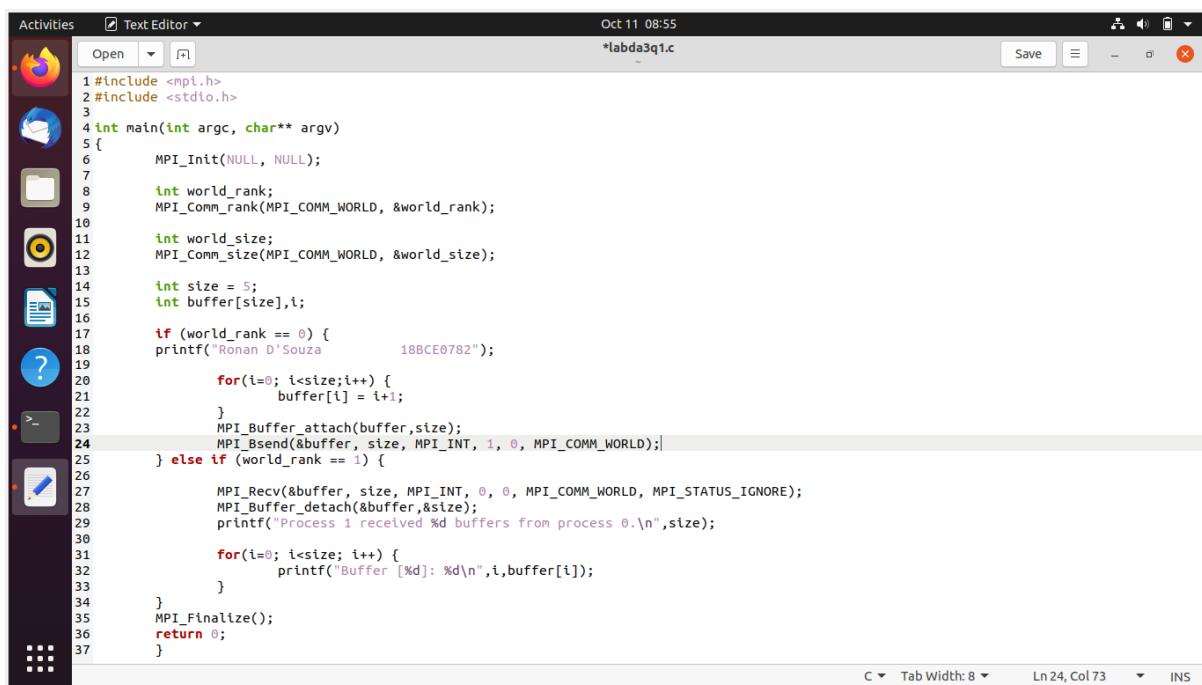
LAB ASSIGNMENT – 3

Name: Ronan D'Souza

Reg. No.:18BCE0782

1. MPI_Bsend is the asynchronous blocking send (with user provided Buffering), it will block until a copy of the buffer is passed. Write a program to establish a point to point communication between two process using the above function.

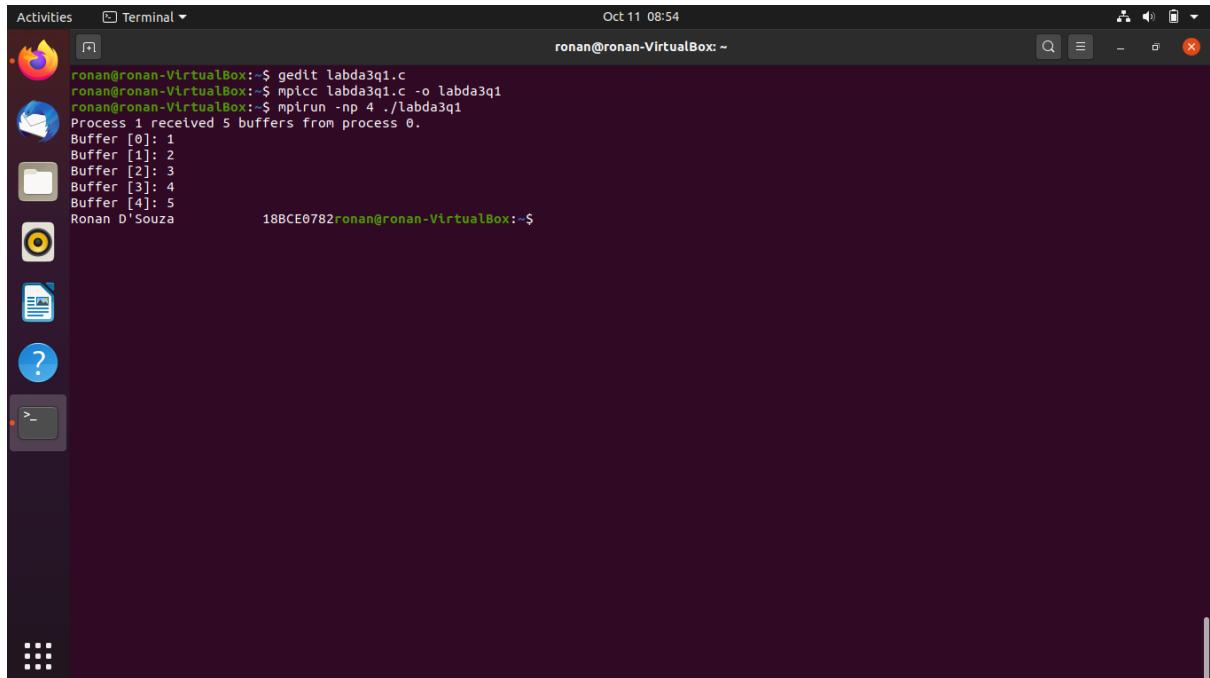
Code:



A screenshot of a Linux desktop environment showing a terminal window titled "Text Editor". The window contains C code for MPI communication. The code initializes MPI, sets up a world communicator, defines a buffer size of 5, and performs a MPI_Bsend operation from rank 0 to rank 1. It also includes MPI_Recv and MPI_Finalize calls. The terminal window has a standard Linux interface with icons for file operations and a status bar at the bottom.

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char** argv)
5 {
6     MPI_Init(NULL, NULL);
7
8     int world_rank;
9     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
10
11    int world_size;
12    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
13
14    int size = 5;
15    int buffer[size], i;
16
17    if (world_rank == 0) {
18        printf("Ronan D'Souza          18BCE0782");
19
20        for(i=0; i<size;i++) {
21            buffer[i] = i+1;
22        }
23        MPI_Buffer_attach(buffer, size);
24        MPI_Bsend(&buffer, size, MPI_INT, 1, 0, MPI_COMM_WORLD);
25    } else if (world_rank == 1) {
26
27        MPI_Recv(&buffer, size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
28        MPI_Buffer_detach(&buffer,&size);
29        printf("Process 1 received %d buffers from process 0.\n",size);
30
31        for(i=0; i<size; i++) {
32            printf("Buffer [%d]: %d\n",i,buffer[i]);
33        }
34    }
35    MPI_Finalize();
36    return 0;
37 }
```

Output:

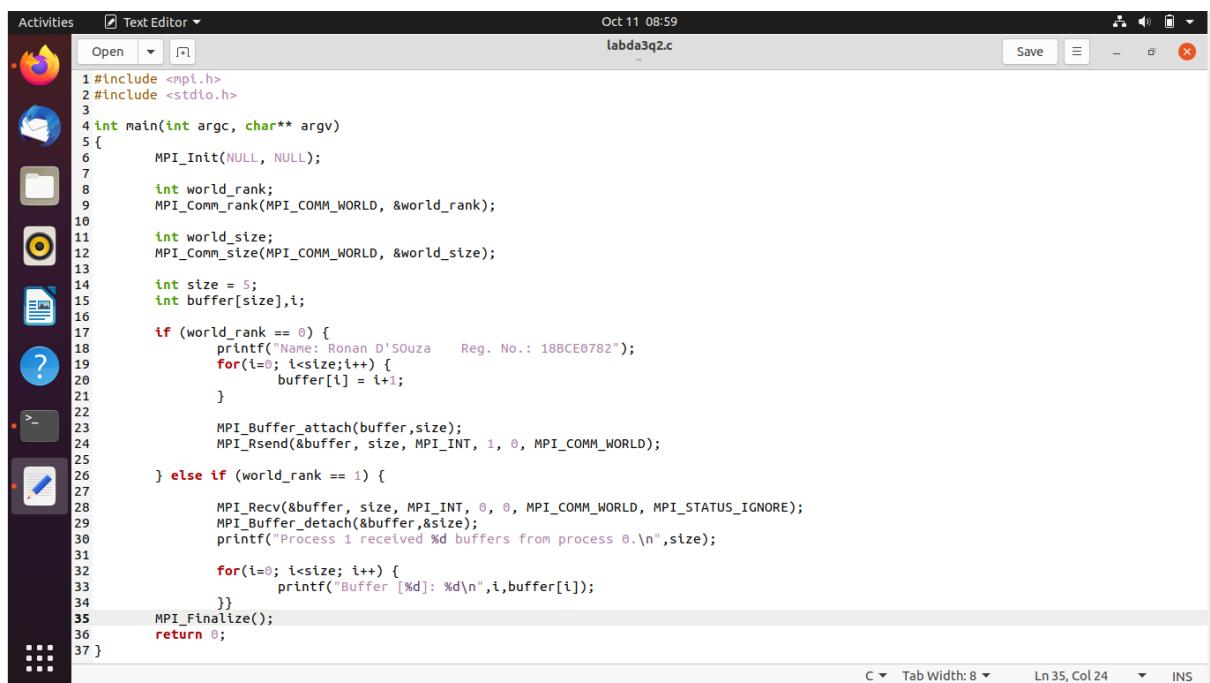


A screenshot of a Linux desktop environment showing a terminal window. The terminal window title is "Terminal" and the date and time are "Oct 11 08:54". The command line shows the execution of an MPI program:

```
ronan@ronan-VirtualBox:~$ gedit labda3q1.c
ronan@ronan-VirtualBox:~$ mpicc labda3q1.c -o labda3q1
ronan@ronan-VirtualBox:~$ mpirun -np 4 ./labda3q1
Process 1 received 5 buffers from process 0.
Buffer [0]: 1
Buffer [1]: 2
Buffer [2]: 3
Buffer [3]: 4
Buffer [4]: 5
Ronan D'Souza
```

2. Develop a MPI program which initiate a process to complete the send operation immediately.

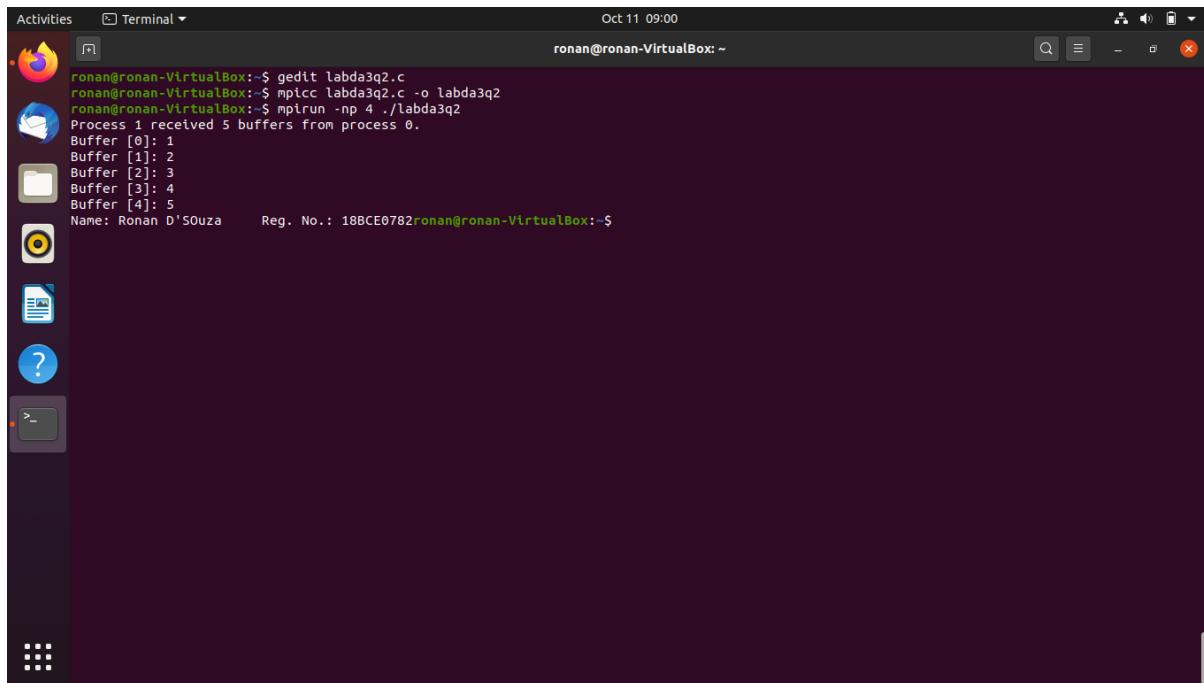
Code:



A screenshot of a text editor window titled "labda3q2.c". The code implements MPI operations to complete a send operation immediately:

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char** argv)
5 {
6     MPI_Init(NULL, NULL);
7
8     int world_rank;
9     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
10
11    int world_size;
12    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
13
14    int size = 5;
15    int buffer[size];
16
17    if (world_rank == 0) {
18        printf("Name: Ronan D'SOUZA    Reg. No.: 18BCE0782");
19        for(i=0; i<size;i++) {
20            buffer[i] = i+1;
21        }
22
23        MPI_Buffer_attach(buffer,size);
24        MPI_Rsend(&buffer, size, MPI_INT, 1, 0, MPI_COMM_WORLD);
25
26    } else if (world_rank == 1) {
27
28        MPI_Recv(&buffer, size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
29        MPI_Buffer_detach(&buffer,ssize);
30        printf("Process 1 received %d buffers from process 0.\n",size);
31
32        for(i=0; i<size; i++) {
33            printf("Buffer [%d]: %d\n",i,buffer[i]);
34        }
35    MPI_Finalize();
36    return 0;
37 }
```

Output:



A screenshot of a Linux desktop environment showing a terminal window. The terminal window title bar says "Activities Terminal" and the date and time "Oct 11 09:00". The terminal itself shows the following command-line session:

```
ronan@ronan-VirtualBox:~$ gedit labda3q2.c
ronan@ronan-VirtualBox:~$ mpicc labda3q2.c -o labda3q2
ronan@ronan-VirtualBox:~$ mpirun -np 4 ./labda3q2
Process 1 received 5 buffers from process 0.
Buffer [0]: 1
Buffer [1]: 2
Buffer [2]: 3
Buffer [3]: 4
Buffer [4]: 5
Name: Ronan D'Souza      Reg. No.: 18BCE0782ronan@ronan-VirtualBox:~$
```

The terminal window has a dark background and light-colored text. The left edge of the screen shows the Unity interface with various application icons.

3. A communicator has 3 process (ID: 0, 1 and 2) such that process 0 sends a binary value to 001 to process 1, in-turn process 1 update the value 010 and send the same to process 2, in-turn process 2 update the value as 011 and sent the same to process 0. Develop a MPI program to implement the above scenario.

Code:

Activities Text Editor

Oct 11 09:04 labda3q3.c

```
1 #include <mpi.h>
2 #include <stdio.h>
3 int main(int argc, char** argv)
4 {
5     MPI_Init(NULL, NULL);
6     int world_rank;
7     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
8     int world_size;
9     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
10    int size = 3;
11    int buffer[size];
12    if(world_rank == 0) {
13        printf("Name: Ronan D'Souza      Reg. No.:18BCE0782      \n");
14        buffer[0] = 1;
15        buffer[1] = 0;
16        buffer[2] = 0;
17        MPI_Send(&buffer, size, MPI_INT, 1, 0, MPI_COMM_WORLD);
18        MPI_Recv(&buffer, size, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
19        printf("Process 0 received %d%d%d from Process 2.\n", buffer[2], buffer[1], buffer[0]);
20    } else if (world_rank == 1) {
21        MPI_Recv(&buffer, size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
22        printf("Process 1 received %d%d%d from Process 0.\n", buffer[2], buffer[1], buffer[0]);
23        buffer[0] = 0;
24        buffer[1] = 1;
25        buffer[2] = 0;
26        MPI_Send(&buffer, size, MPI_INT, 2, 0, MPI_COMM_WORLD);
27    } else if (world_rank == 2) {
28        MPI_Recv(&buffer, size, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
29        printf("Process 2 received %d%d%d from Process 1.\n", buffer[2], buffer[1], buffer[0]);
30        buffer[0] = 1;
31        buffer[1] = 0;
32        buffer[2] = 0;
33        MPI_Send(&buffer, size, MPI_INT, 0, 0, MPI_COMM_WORLD);
34    }
35    MPI_Finalize();
36    return 0;
37 }
```

C Tab Width: 8 Ln 35, Col 24 INS

Output:

Activities Terminal

Oct 11 09:06 ronan@ronan-VirtualBox: ~

```
ronan@ronan-VirtualBox:~$ gedit labda3q3.c
ronan@ronan-VirtualBox:~$ mpicc labda3q3.c -o labda3q3
ronan@ronan-VirtualBox:~$ mpirun -np 4 ./labda3q3
Name: Ronan D'Souza      Reg. No.:18BCE0782
Process 1 received 001 from Process 0.
Process 2 received 010 from Process 1.
Process 0 received 011 from Process 2.
ronan@ronan-VirtualBox:~$
```

Parallel and Distributed Computing

CSE4001

Lab Assignment 1

Shreya Maheshwari

18BCE0167

- 1) Using OpenMP, Develop and run a multi-threaded program to perform and print vector addition.

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#define ARRAY_SIZE 8
#define NUM_THREADS 4

int main (int argc, char *argv[])
{
    int * a;
    int * b;
    int * c;

    int n = ARRAY_SIZE;
    int n_per_thread;
    int total_threads = NUM_THREADS;
    int i;

    a = (int *) malloc(sizeof(int)*n);
    b = (int *) malloc(sizeof(int)*n);
```

Output:

Ubuntu_CyberEss_1 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

cisco@ubuntu: ~/Desktop

```
cisco@ubuntu:~$ cd Desktop
cisco@ubuntu:~/Desktop$ gcc -fopenmp vector-addition.c
cisco@ubuntu:~/Desktop$ ./a.out
Thread 1 works on element2
Thread 1 works on element3
Thread 3 works on element6
Thread 3 works on element7
Thread 2 works on element4
Thread 2 works on element5
Thread 0 works on element0
Thread 0 works on element1
i      a[i]      +      b[i]      =      c[i]
0          0          0          0
1          1          1          2
2          2          2          4
3          3          3          6
4          4          4          8
5          5          5         10
6          6          6         12
7          7          7         14
cisco@ubuntu:~/Desktop$ //Shreya Maheshwari 18BCE0167
```

2) Using OpenMP, Develop and run a multi-threaded program to perform and print dot product operation.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#define SIZE 10

int main (int argc, char *argv[]) {

    float u[SIZE], v[SIZE], dp,dpp;
    int i, j, tid;
```

```

dp=0.0;
for(i=0;i<SIZE;i++){
    u[i]=1.0*(i+1);
    v[i]=1.0*(i+2);
}
printf("\n values of u and v:\n");

for (i=0;i<SIZE;i++){
    printf(" u[%d]= %.1f\t v[%d]= %.1f\n",i,u[i],i,v[i]);
}
#pragma omp parallel shared(u,v,dp,dpp) private (tid,i)
{
    tid=omp_get_thread_num();

    #pragma omp for private (i)
    for(i=0;i<SIZE;i++){
        dpp+=u[i]*v[i];
        printf("thread: %d\n", tid);
    }
    #pragma omp critical
    {
        dp=dpp;
        printf("thread %d\n",tid);
    }
}

printf("\n dot product is %f\n",dp);
}

```

Output:

Ubuntu_CyberEss_1 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal

```
cisco@ubuntu:~/Desktop
cisco@ubuntu:~$ cd Desktop
cisco@ubuntu:~/Desktop$ gcc -fopenmp dot-product.c
cisco@ubuntu:~/Desktop$ ./a.out

values of u and v:
u[0]= 1.0      v[0]= 2.0
u[1]= 2.0      v[1]= 3.0
u[2]= 3.0      v[2]= 4.0
u[3]= 4.0      v[3]= 5.0
u[4]= 5.0      v[4]= 6.0
u[5]= 6.0      v[5]= 7.0
u[6]= 7.0      v[6]= 8.0
u[7]= 8.0      v[7]= 9.0
u[8]= 9.0      v[8]= 10.0
u[9]= 10.0     v[9]= 11.0
thread: 0
dot product is 440.000000
cisco@ubuntu:~/Desktop$ Shreya Maheshwari 18BCE0167
```

3) OpenMP- Get Environment Information using the following library function call and print all one by one at a single time.

```
omp_get_num_procs();
omp_get_num_threads();
omp_get_max_threads();
omp_in_parallel();
omp_get_dynamic();
omp_get_nested();
```

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(){

    #pragma omp parallel
```

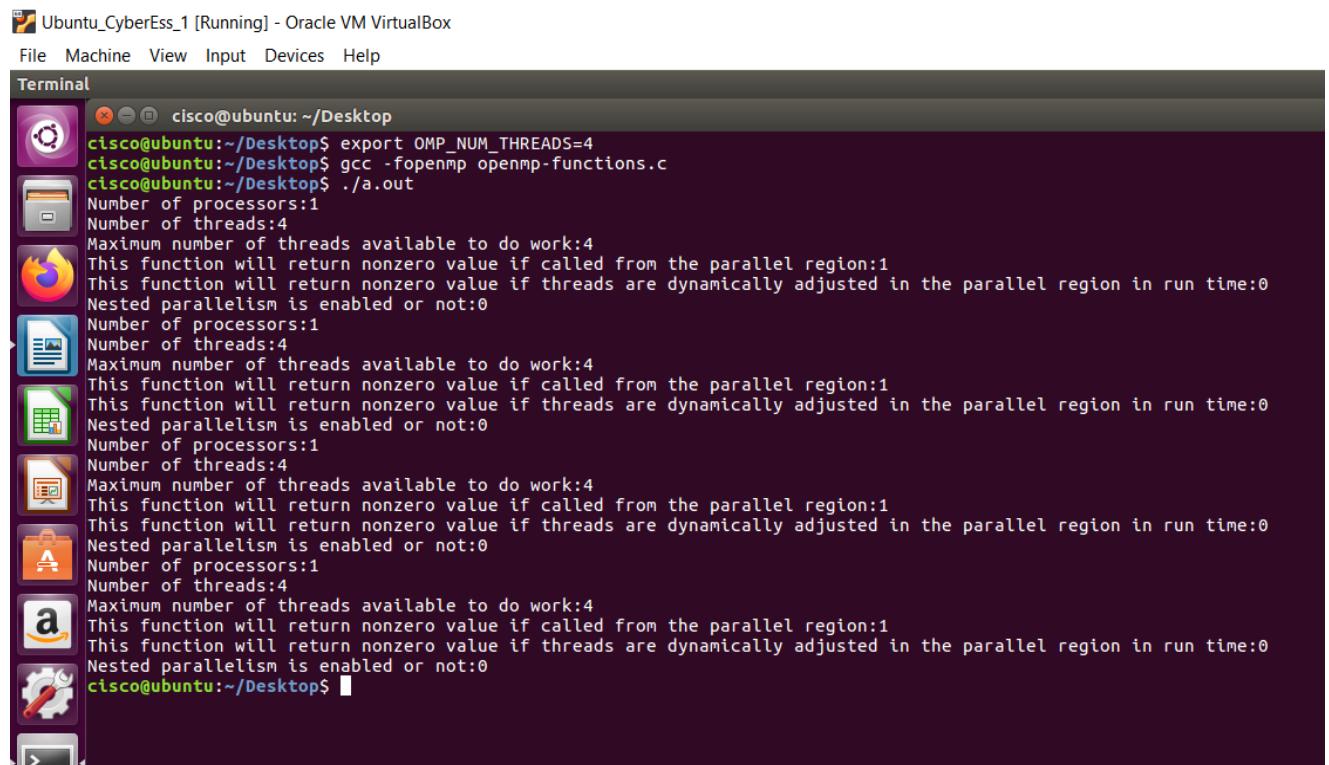
```

{
    printf("Number of processors:");
    printf("%d\n", omp_get_num_procs());
    printf("Number of threads:");
    printf("%d\n", omp_get_num_threads());
    printf("Maximum number of threads available to do work:");
    printf("%d\n", omp_get_max_threads());
    printf("This function will return nonzero value if called from the
parallel region:");
    printf("%d\n", omp_in_parallel());
    printf("This function will return nonzero value if threads are
dynamically adjusted in the parallel region in run time:");
    printf("%d\n", omp_get_dynamic());
    printf("Nested parallelism is enabled or not:");
    printf("%d\n", omp_get_nested());
}

return 0;
}

```

Output:



The screenshot shows a terminal window titled "Terminal" on a dark-themed desktop. The window contains four identical outputs of a C program. Each output shows the program's logic being executed by four different threads. The outputs are as follows:

```

cisco@ubuntu:~/Desktop$ export OMP_NUM_THREADS=4
cisco@ubuntu:~/Desktop$ gcc -fopenmp openmp-functions.c
cisco@ubuntu:~/Desktop$ ./a.out
Number of processors:1
Number of threads:4
Maximum number of threads available to do work:4
This function will return nonzero value if called from the parallel region:1
This function will return nonzero value if threads are dynamically adjusted in the parallel region in run time:0
Nested parallelism is enabled or not:0
Number of processors:1
Number of threads:4
Maximum number of threads available to do work:4
This function will return nonzero value if called from the parallel region:1
This function will return nonzero value if threads are dynamically adjusted in the parallel region in run time:0
Nested parallelism is enabled or not:0
Number of processors:1
Number of threads:4
Maximum number of threads available to do work:4
This function will return nonzero value if called from the parallel region:1
This function will return nonzero value if threads are dynamically adjusted in the parallel region in run time:0
Nested parallelism is enabled or not:0
cisco@ubuntu:~/Desktop$ 

```

The output is printed 4 times because 4 threads are running.

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

Exercise 1 (OpenMP-I)

SCENARIO – I

Write a simple OpenMP program for vector addition (sum). It is helpful to understand how threads are created in OpenMP.

To examine the above scenario, we are going to take two one-dimensional arrays, each of size of 5. We will then create 5 threads. Each thread will be responsible for one addition operation.

Note: In OpenMP, pre-defined preprocessor directive `#pragma omp parallel` is used to create threads. We can mention the number of threads to be created as a parameter to `num_threads()`. If you are not mentioning `num_threads()`, then the number of threads to be created is equal to number of cores in processor. Thread id can be obtained by using predefined function `omp_get_thread_num()`.

ALGORITHM:

```
START
Set ARRAY_SIZE =8
Set NUM_THREADS = 4
Set n=ARRAY_SIZE
Set total_threads=NUM_THREADS
For i=0;i<n;i++
    Set a[i]=i
For i=0;i<n;i++
    Set b[i]=i
Set n_
per_threads=n/total_threads
For i=0;i<n;i++
    Set c[i]=a[i]+b[i]
    Show Thread (thread_num) works on element (i)
}
For i=0;i<n;i++
    Show i,a[i],b[i],c[i]
}
STOP
```

SOURCE CODE:

Reg No : 18BCE0164

Name : JAHNVI MISHRA

Slot : L11+L12

```
#include <stdlib.h> //malloc and free
#include <stdio.h> //printf
#include <omp.h> //OpenMP
```

```
#define ARRAY_SIZE 8 //Size of arrays whose elements will be added together.
#define NUM_THREADS 4 //Number of threads to use for vector addition.
```

```
int main (int argc, char *argv[])
{
    int * a;
    int * b;
    int * c;

    int n = ARRAY_SIZE;           // number of array elements
    int n_per_thread;           // elements per thread
    int total_threads = NUM_THREADS; // number of threads to use
    int i;                      // loop index

    // allocate space for the arrays
    a = (int *) malloc(sizeof(int)*n);
    b = (int *) malloc(sizeof(int)*n);
    c = (int *) malloc(sizeof(int)*n);

    // initialize arrays a and b with consecutive integer values
    // as a simple example
    for(i=0; i<n; i++) {
        a[i] = i;
    }
    for(i=0; i<n; i++) {
        b[i] = i;
    }

    omp_set_num_threads(total_threads);

    // determine how many elements each process will work on
    n_per_thread = n/total_threads;

    #pragma omp parallel for shared(a, b, c) private(i) schedule(static, n_per_thread)
    for(i=0; i<n; i++) {
        c[i] = a[i]+b[i];
        // Which thread am I? Show who works on what for this small example
        printf("Thread %d works on element%d\n", omp_get_thread_num(), i);
    }
}
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

```
    printf("i\ta[i]\t+\tb[i]\t=\tc[i]\n");
    for(i=0; i<n; i++) {
        printf("%d\t%d\t%d\t%d\n", i, a[i], b[i], c[i]);
    }

    // clean up memory
    free(a); free(b); free(c);

    return 0;
}
```

OUTPUT SCREEN SHOT:

```
jahnvi@18bce0164:~$ gedit vector.c
jahnvi@18bce0164:~$ gcc vector.c -fopenmp -o vector
jahnvi@18bce0164:~$ ./vector
Thread 0 works on element0
Thread 0 works on element1
Thread 3 works on element6
Thread 3 works on element7
Thread 2 works on element4
Thread 2 works on element5
Thread 1 works on element2
Thread 1 works on element3
i      a[i]      +      b[i]      =      c[i]
0          0          0          0
1          1          1          2
2          2          2          4
3          3          3          6
4          4          4          8
5          5          5         10
6          6          6         12
7          7          7         14
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

RESULTS:

Thread 0 works on element0
Thread 0 works on element1
Thread 3 works on element6
Thread 3 works on element7
Thread 2 works on element4
Thread 2 works on element5
Thread 1 works on element2
Thread 1 works on element3

i	a[i]	+	b[i]	=	c[i]
0	0		0		0
1	1		1		2
2	2		2		4
3	3		3		6
4	4		4		8
5	5		5		10
6	6		6		12
7	7		7		14

SCENARIO – II

Write a simple OpenMP program for performing dot product It is helpful to understand how threads can be executed in parallel to sequentially check each element of the list for the target value until a match is found or until all the elements have been searched.

Note: In OpenMP, to parallelize the for loop, the openMP directive is: #pragma omp parallel for.

BRIEF ABOUT YOUR APPROACH:

START
Set SIZE 10
Set dp=0.0
For i=0;i<SIZE;i++{
 Set u[i]=1.0*(i+1)
 Set v[i]=1.0*(i+2)
}
Show values of u and v
For i=0;i<SIZE;i++{

Reg No : 18BCE0164
 Name : JAHNVI MISHRA
 Slot : L11+L12
 Show u[i] and v[i]
 }
 For i=0;i<SIZE;i++{
 Set dpp+=u[i]+v[i]
 Show thread(thread_num)
 }
 Set dp=dpp
 Show Thread(thread_num)
 Show dot product
 STOP

SOURCE CODE:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#define SIZE 10

int main (int argc, char *argv[]) {

    float u[SIZE], v[SIZE], dp,dpp;
    int i, j, tid;

    dp=0.0;
    for(i=0;i<SIZE;i++){
        u[i]=1.0*(i+1);
        v[i]=1.0*(i+2);
    }
    printf("\n Values of u and v:\n");

    for (i=0;i<SIZE;i++){
        printf(" u[%d]= %.1f\t v[%d]= %.1f\n",i,u[i],i,v[i]);
    }
    #pragma omp parallel shared(u,v,dp,dpp) private (tid,i)
    {
  
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

```
tid=omp_get_thread_num();

#pragma omp for private (i)
for(i=0;i<SIZE;i++){
    dpp+=u[i]*v[i];
    printf("thread: %d\n", tid);
}
#pragma omp critical
{
    dp=dpp;
    printf("thread %d\n",tid);
}

printf("\n DOT Product is %f\n",dp);

}
```

OUTPUT :

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

```
jahnvi@18bce0164:~$ gedit dotproduct.c
jahnvi@18bce0164:~$ gcc dotproduct.c -fopenmp -o dotproduct
jahnvi@18bce0164:~$ ./dotproduct

Values of u and v:
u[0]= 1.0      v[0]= 2.0
u[1]= 2.0      v[1]= 3.0
u[2]= 3.0      v[2]= 4.0
u[3]= 4.0      v[3]= 5.0
u[4]= 5.0      v[4]= 6.0
u[5]= 6.0      v[5]= 7.0
u[6]= 7.0      v[6]= 8.0
u[7]= 8.0      v[7]= 9.0
u[8]= 9.0      v[8]= 10.0
u[9]= 10.0     v[9]= 11.0
thread: 0
thread: 0
thread: 3
thread: 6
thread: 2
thread: 4
thread: 7
thread: 1
thread: 1
thread: 5
thread: 2
thread: 6
thread: 3
thread: 1
thread: 7
thread: 0
thread: 5
thread: 4

DOT Product is 312.000000
```

RESULTS:

Values of u and v:

u[0]= 1.0	v[0]= 2.0
u[1]= 2.0	v[1]= 3.0
u[2]= 3.0	v[2]= 4.0
u[3]= 4.0	v[3]= 5.0
u[4]= 5.0	v[4]= 6.0
u[5]= 6.0	v[5]= 7.0
u[6]= 7.0	v[6]= 8.0
u[7]= 8.0	v[7]= 9.0
u[8]= 9.0	v[8]= 10.0
u[9]= 10.0	v[9]= 11.0

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
u[9]= 10.0 v[9]= 11.0
thread: 0
thread: 0
thread: 3
thread: 6
thread: 2
thread: 4
thread: 7
thread: 1
thread: 1
thread: 5
thread 2
thread 6
thread 3
thread 1
thread 7
thread 0
thread 5
thread 4

DOT Product is 312.000000

SCENARIO – III

Write a simple openMP program to demonstrate the sharing of a loop iteration by number of threads . You can have a chunk size of 10 .

ALGORITHM

START
Set CHUNKSIZE =10
Set N=10
For i=0;i<N;i++-{
 Set a[i]=b[i]=i*1.0
 Set tid = thread_num
 If tid==0{
 Set n_threads = thread_num
 Show n_threads }
 Show thread starting...
 For i=0;i<N;i++ {

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
Set c[i]=a[i]+b[i]
Show Thread (tid): c[i]
STOP

SOURCE CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 10

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n",tid);

        #pragma omp for schedule(dynamic,chunk)
        for (i=0; i<N; i++)
        {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
    } /* end of parallel section */
}
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

OUTPUT:

```
jahnvi@18bce0164:~$ gedit loop.c
jahnvi@18bce0164:~$ gcc loop.c -fopenmp -o loop
jahnvi@18bce0164:~$ ./loop
Number of threads = 8
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 2 starting...
Thread 1 starting...
Thread 4 starting...
Thread 3 starting...
Thread 6 starting...
Thread 7 starting...
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
Thread 5 starting...
```

RESULTS:

Number of threads = 8
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 2 starting...
Thread 1 starting...
Thread 4 starting...
Thread 3 starting...
Thread 6 starting...
Thread 7 starting...

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
Thread 5 starting...

SCENARIO – IV

Write a open MP program to demonstrate the sharing of works of a section using threads. You can perform arithmetic operations on the one dimensional array and this section load can be shared by the threads.

ALGORITHM

START
Set N=10
For i=0;i<N;i++{
 Set a[i]=i*1.5
 Set b[i]=i+22.35
 Set c[i]=d[i]=0.0 }
Set tid=thread_num
If tid==0 {
 Set nthreads=thread_num
 Show Number of threads (nthreads) }
Show Thread (tid) starting...
Show Thread (tid) doing section 1
For i=0;i<N;i++{
 Set c[i]=a[i]+b[i]
 Show Thread (tid): c[i] }
For i=0;i<N;i++{
 Set d[i]=a[i]*b[i]
 Show Thread (tid): d[i] }
Show Thread (tid) done
STOP

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

SOURCE CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 10

int main (int argc, char *argv[])
{
    int i, nthreads, tid;
    float a[N], b[N], c[N], d[N];

    /* Some initializations */
    for (i=0; i<N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
        c[i] = d[i] = 0.0;
    }

    #pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n",tid);

        #pragma omp sections nowait
        {
            #pragma omp section
            {
                printf("Thread %d doing section 1\n",tid);
                for (i=0; i<N; i++)
                {
                    c[i] = a[i] + b[i];
                    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
                }
            }
            #pragma omp section
            {
                printf("Thread %d doing section 2\n",tid);
                for (i=0; i<N; i++)
            }
        }
    }
}
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

```
{  
d[i] = a[i] * b[i];  
printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);  
}  
}  
  
} /* end of sections */  
  
printf("Thread %d done.\n",tid);  
  
} /* end of parallel section */  
}
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA

Slot : L11+ L12

OUTPUT:

```
jahnvi@18bce0164:~$ gedit section.c
jahnvi@18bce0164:~$ gcc section.c -fopenmp -o section
jahnvi@18bce0164:~$ ./section
Number of threads = 8
Thread 0 starting...
Thread 0 doing section 1
Thread 0: c[0]= 22.350000
Thread 0: c[1]= 24.850000
Thread 0: c[2]= 27.350000
Thread 0: c[3]= 29.850000
Thread 0: c[4]= 32.349998
Thread 0: c[5]= 34.849998
Thread 0: c[6]= 37.349998
Thread 0: c[7]= 39.849998
Thread 0: c[8]= 42.349998
Thread 0: c[9]= 44.849998
Thread 0 doing section 2
Thread 0: d[0]= 0.000000
Thread 0: d[1]= 35.025002
Thread 0: d[2]= 73.050003
Thread 0: d[3]= 114.075005
Thread 0: d[4]= 158.100006
Thread 0: d[5]= 205.125000
Thread 0: d[6]= 255.150009
Thread 0: d[7]= 308.175018
Thread 0: d[8]= 364.200012
Thread 0: d[9]= 423.225006
Thread 0 done.
Thread 3 starting...
Thread 3 done.
Thread 7 starting...
Thread 7 done.
Thread 1 starting...
Thread 1 done.
Thread 2 starting...
Thread 5 starting...
Thread 5 done.
Thread 4 starting...
Thread 4 done.
Thread 6 starting...
Thread 2 done.
Thread 6 done.
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA

SLOT: L11+L12

RESULTS:

Number of threads = 8
Thread 0 starting...
Thread 0 doing section 1
Thread 0: c[0]= 22.350000
Thread 0: c[1]= 24.850000
Thread 0: c[2]= 27.350000
Thread 0: c[3]= 29.850000
Thread 0: c[4]= 32.349998
Thread 0: c[5]= 34.849998
Thread 0: c[6]= 37.349998
Thread 0: c[7]= 39.849998
Thread 0: c[8]= 42.349998
Thread 0: c[9]= 44.849998
Thread 0 doing section 2
Thread 0: d[0]= 0.000000
Thread 0: d[1]= 35.025002
Thread 0: d[2]= 73.050003
Thread 0: d[3]= 114.075005
Thread 0: d[4]= 158.100006
Thread 0: d[5]= 205.125000
Thread 0: d[6]= 255.150009
Thread 0: d[7]= 308.175018
Thread 0: d[8]= 364.200012
Thread 0: d[9]= 423.225006
Thread 0 done.
Thread 3 starting...
Thread 3 done.
Thread 7 starting...
Thread 7 done.
Thread 1 starting...
Thread 1 done.
Thread 2 starting...
Thread 5 starting...
Thread 5 done.
Thread 4 starting...
Thread 4 done.
Thread 6 starting...
Thread 2 done.
Thread 6 done.

Reg No : **18BCE0164**
Name : **JAHNVI MISHRA**
Slot : **L11+L12**

Exercise 2 (OpenMP-II)

SCENARIO – I

Write a simple OpenMP program to employ a '*reduction*' clause to express the reduction of a for loop. In order to specify the reduction in OpenMP, we must provide

1. An operation (+ / * / o)
2. A reduction variable (sum / product / reduction). This variable holds the result of the computation.

ALGORITHM:

```
float a[100], b[100], sum;  
for (i=0; i < n; i++)  
a[i] = b[i] = i * 1.0;  
sum = 0.0;  
  
#pragma omp for reduction(+:sum)  
for (i=0; i < n; i++)  
{  
sum = sum + (a[i] * b[i]);  
} // end of for loop  
}
```

SOURCE CODE:

```
#include <omp.h>  
#include <stdio.h>  
#include <stdlib.h>
```

Reg No : **18BCE0164**
Name : **JAHNVI MISHRA**
Slot : **L11+L12**

```
int main (int argc, char *argv[])
{
int i, n;
float a[100], b[100], sum;

/* Some initializations */
n = 100;
for (i=0; i < n; i++)
a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)
for (i=0; i < n; i++)
sum = sum + (a[i] * b[i]);

printf(" Sum = %f\n",sum);

}
```

OUTPUT SCREEN SHOT:

```
jahnvi@18bce0164:~$ gedit reduction.c
jahnvi@18bce0164:~$ gcc reduction.c -fopenmp -o reduction
jahnvi@18bce0164:~$ ./reduction
Sum = 328350.000000
```

RESULTS:

Sum = 328350.000000

Reg No : **18BCE0164**
Name : **JAHNVI MISHRA**
Slot : **L11+L12**

SCENARIO – II

Write a simple OpenMP program for performing the reduction of orphaned parallel loops.
Perform this operation for a dot product by printing the thread IDs and sum.

ALGORITHM:

If shared scoping is not inherited:

- Orphaned task variables are firstprivate by default!
- Non-Orphaned task variables inherit the shared attribute!

→ Variables are firstprivate unless shared in the enclosing context

```
#pragma omp for reduction(+:sum)
for (i=0; i < VECLEN; i++)
{
    sum = sum + (a[i]*b[i]);
    printf(" tid= %d i=%d\n",tid,i);
}
#pragma omp parallel
dotprod();
printf("Sum = %f\n",sum);
```

Reg No : **18BCE0164**
Name : **JAHNVI MISHRA**
Slot : **L11+L12**

SOURCE CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define VECLEN 10

float a[VECLEN], b[VECLEN], sum;

float dotprod ()
{
int i,tid;

tid = omp_get_thread_num();
#pragma omp for reduction(+:sum)
for (i=0; i < VECLEN; i++)
{
sum = sum + (a[i]*b[i]);
printf(" tid= %d i=%d\n",tid,i);
}
}

int main (int argc, char *argv[])
{
int i;

for (i=0; i < VECLEN; i++)
a[i] = b[i] = 1.0 * i;
sum = 0.0;

#pragma omp parallel
dotprod();

printf("Sum = %f\n",sum);

}
```

Reg No : **18BCE0164**
Name : **JAHNVI MISHRA**
Slot : **L11+L12**

OUTPUT:

```
jahnvi@18bce0164:~$ gedit orphaned.c
jahnvi@18bce0164:~$ gcc orphaned.c -fopenmp -o orphaned
jahnvi@18bce0164:~$ ./orphaned
tid= 0 i=0
tid= 0 i=1
tid= 2 i=4
tid= 5 i=7
tid= 4 i=6
tid= 3 i=5
tid= 1 i=2
tid= 1 i=3
tid= 6 i=8
tid= 7 i=9
Sum = 285.000000
```

Reg No : **18BCE0164**
Name : **JAHNVI MISHRA**
Slot : **L11+L12**

RESULTS:

tid= 0 i=0
tid= 0 i=1
tid= 2 i=4
tid= 5 i=7
tid= 4 i=6
tid= 3 i=5
tid= 1 i=2
tid= 1 i=3
tid= 6 i=8
tid= 7 i=9

Sum = 285.0000

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

EX 3 (OPENMP –III)

SCENARIO – I

Write a simple OpenMP program for Matrix Multiplication. It is helpful to understand how threads make use of the cores to enable course-grain parallelism.

Note that different threads will write different parts of the result in the array **a**, so we don't get any problems during the parallel execution. Note that accesses to matrices **b** and **c** are read-only and do not introduce any problems either.

Code Snippet

```
int alg_matmul2D(int m, int n, int p, float** a, float** b, float** c)
{
    int i,j,k;
#pragma omp parallel shared(a,b,c) private(i,j,k)
    {
#pragma omp for
        schedule(static)    for (i=0; i<m;
        i=i+1){      for (j=0; j<n;
        j=j+1){
            a[i][j]=0.;
            for (k=0; k<p; k=k+1){
                a[i][j]=(a[i][j])+(b[i][k]*(c[k][j]));
            }
        }
    }
    return 0;
}
```

ALGORITHM:

- 1: Start the Program.
- 2: Enter the row and column of the first (a) matrix.
- 3: Enter the row and column of the second (b) matrix.
- 4: Enter the elements of the first (a) matrix.
- 5: Enter the elements of the second (b) matrix.
- 6: Print the elements of the first (a) matrix in matrix form.
- 7: Print the elements of the second (b) matrix in matrix form.

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
8: Set a loop up to row.

9: Set an inner loop up to the column.

10: Set another inner loop up to the column.

11: Multiply the first (a) and second (b) matrix and store the element in the third matrix (c)
12: Print the final matrix.

13: Stop the Program.

SOURCE CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define NRA 3          /* number of rows in matrix A */
#define NCA 3          /* number of columns in matrix A */
#define NCB 3          /* number of columns in matrix B */

int main (int argc, char *argv[])
{
    int tid, nthreads, i, j, k, chunk;
    double a[NRA][NCA],      /* matrix A to be multiplied */
           b[NCA][NCB],      /* matrix B to be multiplied */
           c[NRA][NCB];      /* result matrix C */

    chunk = 10;             /* set loop iteration chunk size */

    /*** Spawn a parallel region explicitly scoping all variables ***/
    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Starting matrix multiple example with %d threads\n",nthreads);
            printf("Initializing matrices...\n");
        }
        /*** Initialize matrices ***/
        #pragma omp for schedule (static, chunk)
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

```
for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
        a[i][j]= i+j;
#pragma omp for schedule (static, chunk)
for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
        b[i][j]= i*j;
#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
        c[i][j]= 0;

/** Do matrix multiply sharing iterations on outer loop **/
/** Display who does which iterations for demonstration purposes **/
printf("Thread %d starting matrix multiply...\n",tid);
#pragma omp for schedule (static, chunk)
for (i=0; i<NRA; i++)
{
    printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
            c[i][j] += a[i][k] * b[k][j];
}
/** End of parallel region **/

/** Print results **/
printf("*****\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
{
    for (j=0; j<NCB; j++)
        printf("%6.2f ", c[i][j]);
    printf("\n");
}
printf("*****\n");
printf ("Done.\n");

}
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
OUTPUT SCREEN SHOT:

```
jahnvi@18bce0164:~$ gedit matrix.c
jahnvi@18bce0164:~$ gcc matrix.c -fopenmp -o matrix
jahnvi@18bce0164:~$ ./matrix
Starting matrix multiple example with 8 threads
Initializing matrices...
Thread 7 starting matrix multiply...
Thread 2 starting matrix multiply...
Thread 6 starting matrix multiply...
Thread 3 starting matrix multiply...
Thread 1 starting matrix multiply...
Thread 5 starting matrix multiply...
Thread 4 starting matrix multiply...
Thread 0 starting matrix multiply...
Thread=0 did row=0
Thread=0 did row=1
Thread=0 did row=2
*****
Result Matrix:
 0.00    5.00   10.00
 0.00    8.00   16.00
 0.00   11.00   22.00
*****
Done.
```

RESULTS:

Starting matrix multiple example with 8 threads
Initializing matrices...
Thread 7 starting matrix multiply...
Thread 2 starting matrix multiply...
Thread 6 starting matrix multiply...
Thread 3 starting matrix multiply...
Thread 1 starting matrix multiply...
Thread 5 starting matrix multiply...
Thread 4 starting matrix multiply...
Thread 0 starting matrix multiply...
Thread=0 did row=0
Thread=0 did row=1
Thread=0 did row=2

Result Matrix:

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
0.00 5.00 10.00
0.00 8.00 16.00
0.00 11.00 22.00

Done.

SCENARIO – II

Write a OpenMP program to calculate the value of PI by evaluating the integral $4/(1 + x^2)$. You can use three pragma block directives to handle the critical section , the sum and the product display.

ALGORITHM:

1. Let x=0
2. Let sum=0.0
3. Let step=1.0/number_of_steps
4. For(i=0;i<number_of_steps;i++){
5. X=(i+0.5)*step
6. Sum+=4.0/(1.0+x*x)}
7. Pi=step*sum
8. SHOW pi

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
/* Main Program */

```
int main()
{
    int      Noofintervals, i;
    float     sum, x, totalsum, h, partialsum, sumthread;

    printf("Enter number of intervals\n");
    scanf("%d", &Noofintervals);

    if (Noofintervals <= 0) {
        printf("Number of intervals should be positive integer\n");
        exit(1);
    }

    sum = 0.0;
    h = 1.0 / Noofintervals;

/*
 * OpenMP Parallel Directive With Private Shared Clauses And Critical
 * Section
 */

#pragma omp parallel for private(x) shared(sumthread)
for (i = 1; i < Noofintervals + 1; i = i + 1) {
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
 $x = h * (i - 0.5);$

/* OPENMP Critical Section Directive */

```
#pragma omp critical
sumthread = sumthread + 4.0 / (1 + x * x);
}
partialsum = sumthread * h;
```

/* OpenMP Critical Section Directive */

```
#pragma omp critical
sum = sum + partialsum;
printf("The value of PI is %f\n", sum);
return 0;
}
```

OUTPUT :

```
jahnvi@18bce0164:~$ gedit pi.c
jahnvi@18bce0164:~$ gcc pi.c -fopenmp -o pi
jahnvi@18bce0164:~$ ./pi
Enter number of intervals
4
The value of PI is 3.146801
```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

RESULTS:

Enter number of intervals
4
The value of PI is 3.146801

SCENARIO – III

Write a simple OpenMP program that uses some OpenMP API functions to extract information about the environment. It should be helpful to understand the language / compiler features of OpenMP runtime library.

To examine the above scenario, the functions such as **omp_get_num_procs()**, **omp_set_num_threads()**, **omp_get_num_threads()**, **omp_in_parallel()**, **omp_get_dynamic()** and **omp_get_nested()** are listed and the explanation is given below to explore the concept practically.

- **omp_set_num_threads()** takes an integer argument and requests that the Operating System provide that number of threads in subsequent parallel regions.
- **omp_get_num_threads()** (integer function) returns the actual number of threads in the current team of threads. ○ **omp_get_thread_num()** (integer function) returns the ID of a thread, where the ID ranges from 0 to the number of threads minus 1. The thread with the ID of 0 is the master thread.
- **omp_get_num_procs()** returns the number of processors that are available when the function is called.
- **omp_get_dynamic()** returns a value that indicates if the number of threads available in subsequent parallel region can be adjusted by the run time.
- **omp_get_nested()** returns a value that indicates if nested parallelism is enabled.

ALGORITHM:

1. START
2. OBTAIN THREAD INFORMATION
3. GET ENVIRONMENT INFORMATION

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

4. PRINT ENVIRONMENT INFORMATION

5. STOP

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main (int argc, char *argv[])
{
    int nthreads, tid, procs, maxt, inpar, dynamic, nested;
/* Start parallel region */
#pragma omp parallel private(nthreads, tid)
{
/* Obtain thread number */
tid = omp_get_thread_num(); /* Only master thread does this */
if (tid == 0)
{
printf("Thread %d getting environment info...\n", tid);
/* Get environment information */
procs = omp_get_num_procs();
nthreads = omp_get_num_threads();
maxt = omp_get_max_threads();
```

```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12
inpar = omp_in_parallel();

dynamic = omp_get_dynamic();

nested = omp_get_nested();

/* Print environment information */

printf("Number of processors = %d\n", procs);

printf("Number of threads = %d\n", nthreads);

printf("Max threads = %d\n", maxt);

printf("In parallel? = %d\n", inpar);

printf("Dynamic threads enabled? = %d\n", dynamic);

printf("Nested parallelism supported? = %d\n", nested);

}

}

return 0;

}

```

EXECUTION:

```

jahnvi@18bce0164:~$ gedit environment.c
jahnvi@18bce0164:~$ gcc environment.c -fopenmp -o environment
jahnvi@18bce0164:~$ ./environment
Thread 0 getting environment info...
Number of processors = 8
Number of threads = 8
Max threads = 8
In parallel? = 1
Dynamic threads enabled? = 0
Nested parallelism supported? = 0

```

Reg No : 18BCE0164
Name : JAHNVI MISHRA
Slot : L11+L12

RESULTS:

Thread 0 getting environment info...

Number of processors = 8

Number of threads = 8

Max threads = 8

In parallel? = 1

Dynamic threads enabled? = 0

Nested parallelism supported? = 0

Reg. No: 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2

Exercise 4 (MPI -I)

SCENARIO – I

Construct a MPI program to print a Hello World message in the terminal.

MPI is a message passing interface system which facilitates the processors which have distributed memory architecture to communicate and send & receive messages.

You are asked to use MPI_init() and MPI_Finalize() methods to implement the program.

ALGORITHM:

`MPI_Init(&argc, &argv);`: initiates MPI

`MPI_Comm_rank(MPI_COMM_WORLD, &node);`: gets current process ID

`printf("Hello World from Node %d\n", node);`: prints hello world along with the node number

`MPI_Finalize();`: exits MPI

SOURCE CODE:

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int node;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &node);

    printf("Hello World from Node %d\n",node);
```

Reg. No: 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2

```
MPI_Finalize();  
return 0;  
}
```

OUTPUT SCREEN SHOT:

```
jahnvi@18bce0164:~$ gedit hello.c  
jahnvi@18bce0164:~$ mpicc hello.c -o hello  
jahnvi@18bce0164:~$ mpirun -np 4 ./hello  
Hello World from Node 0  
Hello World from Node 1  
Hello World from Node 2  
Hello World from Node 3
```

RESULTS:

```
jahnvi@18bce0164:~$ gedit hello.c  
jahnvi@18bce0164:~$ mpicc hello.c -o hello  
jahnvi@18bce0164:~$ mpirun -np 4 ./hello  
Hello World from Node 0  
Hello World from Node 1  
Hello World from Node 2  
Hello World from Node 3
```

SCENARIO – II

- Construct a MPI program to print rank of the processor, size of the domain(group) and name of the processor/domain in the terminal.
- MPI is a message passing interface system which facilitates the processors which have distributed memory architecture to communicate and send & receive messages.
- You are asked to use MPI_Comm_rank(), MPI_Comm_size(),

Reg. No: 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2

MPI_Get_processor_name() methods

BRIEF ABOUT YOUR APPROACH:

I am going to use three functions for the task and to use those three functions we need 2 more functions. The functions are as follows:

MPI_Init(&argc, &argv);: initiates MPI

MPI_Comm_rank(MPI_COMM_WORLD, &rank);: gets current process ID

MPI_Comm_size(MPI_COMM_WORLD, &size);: get the number of processes

MPI_Get_processor_name(name, &length);: it gets the ID of the processor which is working

MPI_Finalize();: terminates MPI

SOURCE CODE:

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char ** argv) {
    int rank, size;
    char name[80];
    int length;

    MPI_Init(&argc, &argv); // note that argc and argv are passed
                           // by address

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Get_processor_name(name,&length);

    printf("Hello MPI: processor %d of %d on %s\n", rank,size,name);
    MPI_Finalize();
}
```

Reg. No: 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2
OUTPUT :

```
jahnvi@18bce0164:~$ gedit rank.c
jahnvi@18bce0164:~$ mpicc rank.c -o rank
jahnvi@18bce0164:~$ mpirun -np 4 ./rank
Hello MPI: processor 0 of 4 on 18bce0164
Hello MPI: processor 1 of 4 on 18bce0164
Hello MPI: processor 2 of 4 on 18bce0164
Hello MPI: processor 3 of 4 on 18bce0164
```

RESULTS:

```
jahnvi@18bce0164:~$ gedit rank.c
jahnvi@18bce0164:~$ mpicc rank.c -o rank
jahnvi@18bce0164:~$ mpirun -np 4 ./rank
Hello MPI: processor 0 of 4 on 18bce0164
Hello MPI: processor 1 of 4 on 18bce0164
Hello MPI: processor 2 of 4 on 18bce0164
Hello MPI: processor 3 of 4 on 18bce0164
```

Reg No : 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2

EX 5 (MPI-II)

SCENARIO – I

Implement a MPI program to demonstrate a simple **MPI broadcast**.

MPI is a message passing interface system which facilitates the processors which have distributed memory architecture to communicate and send & receive messages.

Use `MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD)` method .

ALGORITHM:

- Including the MPI header files using `#include`
- Initialize the MPI environment using `MPI_init()`. Here it takes two arguments.
- We then use `MPI_Comm_rank()` to return the rank of a process and `MPI_Comm_size()` to return the size of a communicator.
- Broadcast the message using `MPI_Bcast` from the process with rank "root" to all other processes of the communicator.
- Lastly `MPI_Finalize()` is used to clean up the MPI environment. No more MPI calls can be made after this one.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int i,myid, numprocs;
    int source,count;
    int buffer[4];
```

Reg No : 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2

```
MPI_Status status;
MPI_Request request;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
source=0;
count=4;
if(myid == source){
    for(i=0;i<count;i++)
        buffer[i]=i;
}
MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD);
for(i=0;i<count;i++)
    printf("%d ",buffer[i]);
printf("\n");
MPI_Finalize();
}
```

OUTPUT SCREEN SHOT:

```
jahnvi@18bce0164:~$ gedit broadcast.c
jahnvi@18bce0164:~$ mpicc broadcast.c -o broadcast
jahnvi@18bce0164:~$ mpirun -np 4 ./broadcast
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

RESULTS:

```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

Reg No : 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2
SCENARIO – II

Implement a MPI program to demonstrate a simple **Send and Receive**.

MPI is a message passing interface system which facilitates the processors which have distributed memory architecture to communicate and send & receive messages.

Use the following methods:

```
MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
```

ALGORITHM:

- Including the MPI header files using #include
- Initialize the MPI environment using MPI_init(). Here it takes two arguments.
- We then use MPI_Comm_rank() to return the rank of a process and MPI_Comm_size() to return the size of the communicator.
- A message (buffer) is send and received using MPI_Send and MPI_Recv.
- Lastly MPI_Finalize() is used to clean up the MPI environment. No more MPI calls can be made after this one.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
```

Reg No : 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2

```
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
tag=1234;
source=0;
destination=1;
count=1;
if(myid == source){
    buffer=5678;
    MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
    printf("processor %d sent %d\n",myid,buffer);
}
if(myid == destination){

MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    printf("processor %d got %d\n",myid,buffer);
}
MPI_Finalize();
}
```

OUTPUT:

```
jahnvi@18bce0164:~$ gedit send.c
jahnvi@18bce0164:~$ mpicc send.c -o send
jahnvi@18bce0164:~$ mpirun -np 4 ./send
processor 0 sent 5678
processor 1 got 5678
```

RESULTS:

processor 0 sent 5678
processor 1 got 5678

Reg No : 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2
SCENARIO – III

Implement a MPI program to calculate the size of the incoming message.

MPI is a message passing interface system which facilitates the processors which have distributed memory architecture to communicate and send & receive messages.

ALGORITHM:

- Including the MPI header files using #include
- Initialize the MPI environment using MPI_init(). Here it takes two arguments.
- We then use MPI_Comm_rank() to return the rank of a process and MPI_Comm_size() to return the size of the communicator.
- A message is sent using MPI_send.
- MPI_probe is then used to block testing for the message.
- The message is received using MPI_recv.
- We then use MPI_Get_count to get the no. of entries received.
- Lastly MPI_Finalize() is used to clean up the MPI environment. No more MPI calls can be made after this one.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
int main(int argc,char **argv)
{
    int myid, numprocs;
    MPI_Status status;
    int mytag,ierr,icount,j,*i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Reg No : 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2

```
printf(" Hello from c process: %d  Numprocs is
%d\n",myid,numprocs);

mytag=123;
if(myid == 0) {
    j=200;
    icount=1;
    ierr=MPI_Send(&j,icount,MPI_INT,1,mytag,MPI_COMM_WORLD);
}
if(myid == 1){
    ierr=MPI_Probe(0,mytag,MPI_COMM_WORLD,&status);
    ierr=MPI_Get_count(&status,MPI_INT,&icount);
    i=(int*)malloc(icount*sizeof(int));
    printf("getting %d\n",icount);
    ierr =
MPI_Recv(i,icount,MPI_INT,0,mytag,MPI_COMM_WORLD,&status);
    printf("i= ");
    for(j=0;j<icount;j++)
        printf("%d ",i[j]);
    printf("\n");
}
MPI_Finalize();
}
```

EXECUTION:

```
jahnvi@18bce0164:~$ gedit size.c
jahnvi@18bce0164:~$ mpicc size.c -o size
jahnvi@18bce0164:~$ mpirun -np 4 ./size
Hello from c process: 2  Numprocs is 4
Hello from c process: 1  Numprocs is 4
Hello from c process: 0  Numprocs is 4
Hello from c process: 3  Numprocs is 4
getting 1
i= 200
```

RESULTS:

- . Hello from c process: 2 Numprocs is 4
- Hello from c process: 1 Numprocs is 4
- Hello from c process: 0 Numprocs is 4
- Hello from c process: 3 Numprocs is 4
- getting 1

Reg No : 18BCE0164
NAME: JAHNVI MISHRA
SLOT: C2
i= 200



**Digital Assignment for:
Parallel and Distributive Computing (CSE4001)**

**Vtop Upload
Lab 3**

Submitted by:

Name	Registration Number
Rajvardhan Deshmukh	17BCE0919

**Under the guidance of
Prof. Narayananmoorthy**

**Slot:
L13 + L14**

Question 1

Code:

```
#include<stdio.h>
#include<omp.h>
void main(void)
{
    int max,min;
    int a[]={10,3,7};

    printf("Now setting the number of threads = 2\n");
    omp_set_num_threads(2);
    double time = omp_get_wtime();
    printf("-----Starting the execution of the parallel Section-----\n");
    #pragma omp parallel sections
    {
        //CALC MAXIMUM
        #pragma omp section
        {
            if(a[0]>a[1] && a[0]>a[2])
                max=a[0];
            else if(a[1]>a[0] && a[1]>a[2])
                max=a[1];
            else
                max=a[2];
            printf("Maximum of 3 numbers (i.e. a[]={ 10,3,7 }) is = %d\n",max);
        }

        //CALC MINIMUM
        #pragma omp section
        {
            if(a[0]<a[1] && a[0]<a[2])
                min=a[0];
            else if(a[1]<a[0] && a[1]<a[2])
                min=a[1];
            else
                min=a[2];
            printf("Minimum of 3 numbers (i.e. a[]={ 10,3,7 }) is = %d\n",min);
        }
    }

    printf("-----End of the parallel Section-----\n");
    printf("The time for the given computation = %f\n",time);
}
```

Screenshot:

```
rajvardhan@rajvardhan-Inspiron-7460:~$ gedit ex3a.c
^C
rajvardhan@rajvardhan-Inspiron-7460:~$ gcc -fopenmp ex3a.c
rajvardhan@rajvardhan-Inspiron-7460:~$ ./a.out
Now setting the number of threads = 2
-----Starting the execution of the parallel Section-----
Maximum of 3 numbers (i.e. a[]={10,3,7}) is = 10
Minimum of 3 numbers (i.e. a[]={10,3,7}) is = 3
-----End of the parallel Section-----
The time for the given computation = 10794.210426
rajvardhan@rajvardhan-Inspiron-7460:~$
```



```
Open ▾ Save ex3a.c ~/
#include<stdio.h>
#include<omp.h>
void main(void)
{
    int max,min;
    int a[]={10,3,7};

    printf("Now setting the number of threads = 2\n");
    omp_set_num_threads(2);
    double time = omp_get_wtime();
    printf("-----Starting the execution of the parallel Section-----\n");
    #pragma omp parallel sections
    {
        //CALC MAXIMUM
        #pragma omp section
        {
            if(a[0]>a[1] && a[0]>a[2])
                max=a[0];
            else if(a[1]>a[0] && a[1]>a[2])
                max=a[1];
            else
                max=a[2];
            printf("Maximum of 3 numbers (i.e. a[]={10,3,7}) is = %d\n",max);
        }

        //CALC MINIMUM
        #pragma omp section
        {
            if(a[0]<a[1] && a[0]<a[2])
                min=a[0];
            else if(a[1]<a[0] && a[1]<a[2])
                min=a[1];
            else
                min=a[2];
            printf("Minimum of 3 numbers (i.e. a[]={10,3,7}) is = %d\n",min);
        }
    }

    printf("-----End of the parallel Section-----\n");
    printf("The time for the given computation = %f\n",time);
}
```

Question 2

Code

```
#include<stdio.h>
#include<omp.h>
#include<math.h>
void main(void)
{
    int i,a[10];
    for(i=0;i<10;i++)
        a[i]=i+1;

    double time = omp_get_wtime();
    printf("-----Starting the execution of the parallel Section-----\n");

    //PART1
    double time1 = omp_get_wtime();
    printf("Printing the array elements:\n");
    #pragma omp parallel for
    for(i=0;i<10;i++)
        printf("%d , ",a[i]);
    printf("\nTime taken = %f\n\n",time1);

    //PART2
    double time2 = omp_get_wtime();
    printf("Printing the array elements after adding 10 and Multiplying 3 to each element:\n");

    int x = 10;
    #pragma omp parallel for
    for(i=0;i<10;i++)
    {
        a[i]=(a[i]+x)*3;
        printf("%d , ",a[i]);
    }
    printf("\nTime taken = %f\n\n",time2);

    //PART3
    double time3 = omp_get_wtime();
    printf("Printing the sum / odd sum / even sum of elements:\n");
    int sum=0,osum=0,esum=0;
    #pragma omp parallel for
    for(i=0;i<10;i++){
        sum=sum+a[i];
        if(a[i]%2!=0)
            osum=osum+a[i];
        else
            esum=esum+a[i];
    }
}
```

```

printf("Total Sum = %d\n",sum);
printf("Odd Sum = %d\n",osum);
printf("Even Sum = %d\n",esum);
printf("Time taken = %f\n\n",time3);

//PART4
double time4 = omp_get_wtime();
printf("Printing the number of odd and even number in the array:\n");
int odd=0,even=0;
#pragma omp parallel for
for(i=0;i<10;i++)
{
    if(a[i]%2==0)
        even=even+1;
    else
        odd=odd+1;
}
printf("There are %d Odd Numbers in the Array.\n",odd);
printf("There are %d Even Numbers in the Array.\n",even);
printf("Time taken = %f\n\n",time4);

//PART5
double time5 = omp_get_wtime();
printf("Printing the maximum and minimum element:\n");
int max=a[0],min=a[0];
#pragma omp parallel for
for(i=0;i<10;i++)
{
    if(a[i]>max)
        max=a[i];
    if(a[i]<min)
        min=a[i];
}
printf("The maximum number in the array is = %d\n",max);
printf("The minimum number in the array is = %d\n",min);
printf("Time taken = %f\n\n",time5);

//PART6
double time6 = omp_get_wtime();
printf("Printing the number of prime number and their sum in the array:\n");
int j,psum=0,prime=0,flag;
for(i=0;i<10;i++)
    a[i]=i+1;
#pragma omp parallel for
for(i=0;i<10;i++)
{
    flag = 0;
    for(j=2;j<a[i];j++)
    {
        if(a[i]%j==0)
        {
            flag = 1;
        }
    }
    if(flag == 0)
        prime++;
    psum += a[i];
}
printf("The number of prime numbers in the array is = %d\n",prime);
printf("The sum of prime numbers in the array is = %d\n",psum);
printf("Time taken = %f\n\n",time6);

```

```

        break;
    }
}
if(flag==0)
{
    prime = prime+1;
    psum = psum + a[i];
}
else
    continue;
}
printf("There are %d Prime numbers in the Array (1,2,3,4,5,6,7,8,9,10)\n",prime);
printf("The sum of the prime numbers is = %d\n",psum);
printf("Time taken = %f\n\n",time6);

printf("-----End of the parallel Section-----\n");
printf("The time for the given computation = %f\n",time);
}

```

Screenshots

```

rajvardhan@rajvardhan-Inspiron-7460:~$ gedit ex3b.c
^C
rajvardhan@rajvardhan-Inspiron-7460:~$ gcc -fopenmp ex3b.c
rajvardhan@rajvardhan-Inspiron-7460:~$ ./a.out
-----Starting the execution of the parallel Section-----
Printing the array elements:
1 , 2 , 9 , 3 , 10 , 4 , 7 , 5 , 8 , 6 ,
Time taken = 14006.317654

Printing the array elements after adding 10 and Multiplying 3 to each element:
57 , 33 , 36 , 39 , 51 , 60 , 54 , 42 , 45 , 48 ,
Time taken = 14006.320223

Printing the sum / odd sum / even sum of elements:
Total Sum = 348
Odd Sum = 225
Even Sum = 240
Time taken = 14006.320346

Printing the number of odd and even number in the array:
There are 5 Odd Numbers in the Array.
There are 5 Even Numbers in the Array.
Time taken = 14006.320507

Printing the maximum and minimum element:
The maximum number in the array is = 60
The minimum number in the array is = 33
Time taken = 14006.320544

Printing the number of prime number and their sum in the array:
There are 5 Prime numbers in the Array (1,2,3,4,5,6,7,8,9,10)
The sum of the prime numbers is = 18
Time taken = 14006.320581

-----End of the parallel Section-----
The time for the given computation = 14006.317624
rajvardhan@rajvardhan-Inspiron-7460:~$ █

```

Write a C program using open MP to perform the following matrix operations. Matrix addition, subtraction and multiplication.

Code:

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    omp_set_num_threads(4);
    int n;
    printf("FOR NUMBER OF THREADS = 4\n\n");
    printf("Enter the dimension of the matrix : ");
    scanf("%d", &n);
    int **mul1=NULL, **mul2=NULL, **mul3=NULL, **mul4=NULL;
    printf("=====FOR
MULTIPLICATION=====\\n\\n");
    mul1 = (int **)malloc(n * sizeof(int *));
    mul2 = (int **)malloc(n * sizeof(int *));
    mul3 = (int **)malloc(n * sizeof(int *));
    mul4 = (int **)malloc(n * sizeof(int *));
    for(int i=0;i<n;i++)
    {
        mul1[i] = (int *)malloc(n * sizeof(int));
        mul2[i] = (int *)malloc(n * sizeof(int));
        mul3[i] = (int *)malloc(n * sizeof(int));
        mul4[i] = (int *)malloc(n * sizeof(int));
    }
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            mul1[i][j]=rand()%100;
            mul2[i][j]=rand()%100;
        }
    }
    float stamp1 = omp_get_wtime();
```

```

int i, j, k;
#pragma omp parallel for private (j, k)
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            mul3[i][j]=0;
            for(k=0;k<n;k++)
            {
                mul3[i][j] += mul1[i][k]*mul2[k][j];
            }
        }
    }
float stamp2 = omp_get_wtime();
printf("PARALLEL TIME IS: %f\n", (stamp2-stamp1));

float stamp3 = omp_get_wtime();
for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul4[i][j] = 0;
        for(int k=0;k<n;k++)
        {
            mul4[i][j] += mul1[i][k]*mul2[k][j];
        }
    }
}
float stamp4 = omp_get_wtime();
printf("SERIAL TIME IS: %f\n", stamp4-stamp3);
printf("\n\n");
printf("=====FOR
ADDITION=====\\n\\n");

for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul1[i][j] = rand()%100;
    }
}
for(int i=0;i<n;i++)

```

```

{
    for(int j=0;j<n;j++)
    {
        mul2[i][j] = rand()%100;
    }
}

float stamp5 = omp_get_wtime();
#pragma omp parallel for private (i,j)
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        mul3[i][j] = mul1[i][j]+mul2[i][j];
    }
}

float stamp6 = omp_get_wtime();
printf("PARALLEL TIME IS: %f\n",stamp6-stamp5);
float stamp7 = omp_get_wtime();
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        mul3[i][j] = mul1[i][j]+mul2[i][j];
    }
}

float stamp8 = omp_get_wtime();
printf("SERIAL TIME IS: %f\n", stamp8-stamp7);
printf("\n\n");
printf("=====FOR
SUBTRACTION=====\\n\\n");

for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul1[i][j] = rand()%100;
    }
}

for(int i=0;i<n;i++)
{
    for(int j=0;j<n;j++)
    {
        mul2[i][j] = rand()%100;
    }
}

```

```

        }
    }

float stamp9 = omp_get_wtime();
#pragma omp parallel for private (i,j)
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        mul3[i][j] = mul1[i][j]-mul2[i][j];
    }
}

float stamp10 = omp_get_wtime();
printf("PARALLEL TIME IS: %f\n",stamp10-stamp9);
float stamp11 = omp_get_wtime();
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        mul3[i][j] = mul1[i][j]-mul2[i][j];
    }
}

float stamp12 = omp_get_wtime();
printf("SERIAL TIME IS: %f\n", stamp12-stamp11);
printf("\n\n");
return 0;
}

```

Screenshots:

```

rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gcc -fopenmp matrix.c
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ ./a.out
FOR NUMBER OF THREADS = 4

Enter the dimension of the matrix : 919
=====FOR MULTIPLICATION=====

PARALLEL TIME IS: 3.032227
SERIAL TIME IS: 5.012695

=====FOR ADDITION=====

PARALLEL TIME IS: 0.001953
SERIAL TIME IS: 0.003906

=====FOR SUBTRACTION=====

PARALLEL TIME IS: 0.000977
SERIAL TIME IS: 0.004883

```

Write an open MP program using C for the following illustrations:

(i) Shared/private variable

Code:

```
#include<stdio.h>
#include<omp.h>
void main()
{
    int a, i;
    int n = 10;
    int b[n];
    omp_set_num_threads(4);
    #pragma omp parallel shared(a,b) private(i)
    {
        #pragma omp single
        {
            a = 10;
            printf("Single construct executed by thread %d\n",
omp_get_thread_num());
        }
        #pragma omp for
        for(i=0;i<n;i++)
        {
            b[i] = a;
        }
    }
    printf("After the parallel region:\n");
    for(i=0;i<n;i++)
        printf("b[%d] = %d\n", i, b[i]);
}
```

Screenshots:

```
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gedit shared.c
^[[A^C
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gcc -fopenmp shared.c
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ ./a.out
Single construct executed by thread 2
After the parallel region:
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
b[5] = 10
b[6] = 10
b[7] = 10
b[8] = 10
b[9] = 10
```

(ii) Reduction operation

Code:

```
#include<stdio.h>
#include<math.h>
#include<omp.h>
int main()
{
    int *lis = NULL;
    printf("\nEnter number of elements: ");
    int num;
    long sum = 0;
    scanf("%d",&num);
    lis = (int*)malloc(num*sizeof(int));
    for(int i=0;i<num;i++)
        lis[i] = rand();
    float stamp1 = omp_get_wtime();
    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<num; i++)
    {
        sum+=lis[i];
    }
    printf("Total sum is %d\n",sum);
    float stamp2 = omp_get_wtime();
    printf("Total time taken in parallel is %f\n",stamp2-stamp1);
    return 0;
}
```

Screenshots:

```
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gedit red.c
^C
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gcc -fopenmp red.c
red.c: In function ‘main’:
red.c:11:14: warning: implicit declaration of function ‘malloc’ [-Wimplicit-function-declaration]
  lis = (int*)malloc(num*sizeof(int));
             ^
red.c:11:14: warning: incompatible implicit declaration of built-in function ‘malloc’
red.c:11:14: note: include ‘<stdlib.h>’ or provide a declaration of ‘malloc’
red.c:13:12: warning: implicit declaration of function ‘rand’; did you mean ‘nanl’? [-Wimplicit-function-declaration]
  lis[i] = rand();
             ^
               nanl
red.c:20:24: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘long int’ [-Wformat]
  printf("Total sum is %d\n",sum);
             ~^
               %ld
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ ./a.out

Enter number of elements: 444
Total sum is -1819338716
Total time taken in parallel is 0.000977
```

(iii) Critical operation

Code:

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int a=0, b=0;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp single
        a++;
        #pragma omp critical
        b++;
    }
    printf("single: %d -- critical: %d\n", a, b);
    return 0;
}
```

Screenshots:

```
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gedit critical.c
^C
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gcc -fopenmp critical.c
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ ./a.out
single: 1 -- critical: 4
```

Write an open MP program using c to find the value of PI

Code:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 8

static long steps = 10000000000;
double step;

int main (int argc, const char *argv[]) {

    int i,j;
    double x;
    double pi, sum = 0.0;
    double start, delta;

    step = 1.0/(double) steps;

    for (j=1; j<= MAX_THREADS; j++) {

        printf(" running on %d threads: ", j);

        omp_set_num_threads(j);

        sum = 0.0;
        double start = omp_get_wtime();

        #pragma omp parallel for reduction(+:sum) private(x)
        for (i=0; i < steps; i++) {
            x = (i+0.5)*step;
            sum += 4.0 / (1.0+x*x);
        }
        pi = step * sum;
        delta = omp_get_wtime() - start;
        printf("PI = %.16g computed in %.4g seconds\n", pi, delta);
    }
}
```

Screenshots:

```
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ ./a.out
running on 1 threads: PI = 3.141592653589971 computed in 13.94 seconds
running on 2 threads: PI = 3.141592653589901 computed in 6.97 seconds
running on 3 threads: PI = 3.141592653589961 computed in 4.837 seconds
running on 4 threads: PI = 3.141592653589821 computed in 3.673 seconds
running on 5 threads: PI = 3.141592653589595 computed in 3.969 seconds
running on 6 threads: PI = 3.141592653589682 computed in 3.751 seconds
running on 7 threads: PI = 3.141592653589631 computed in 3.73 seconds
running on 8 threads: PI = 3.141592653589769 computed in 4.089 seconds
```

Compare and contrast OpenMP and MPI?

1. MPI stands for *Message Passing Interface*. These are available as API(Application programming interface) or in library form for C,C++ and FORTRAN.
 - A. Different MPI's API are available in market i.e OpenMPI, MPICH, HP-MPI, Intel MPI, etc. Whereas many are freely available like OpenMPI, MPICH etc , other like Intel MPI comes with license i.e you need to pay for it .
 - B. One can use any one of above to parallelize programs . MPI standards maintain that all of these APIs provided by different vendors or groups follow similar standards, so all functions or subroutines in all different MPI API follow similar functionality as well arguments.
 - C. The difference lies in implementation that can make some MPIs API to be more efficient than other. Many commercial CFD-Packages gives user option to select between different MPI API. However HP-MPI as well Intel MPIs are considered to be more efficient in performance.
 - D. When MPI was developed, it was aimed at distributed memory system but now focus is both on distributed as well shared memory system. However it does not mean that with MPI , one cannot run program on shared memory system, it just that earlier, we could not take advantage of shared memory but now we can with latest MPI 3
2. OpenMP stand for *Open Multiprocessing* . OpenMP is basically an add on in compiler. It is available in gcc (gnu compiler) , Intel compiler and with other compilers.
 - A. OpenMP target shared memory systems i.e where processor shared the main memory.
 - B. OpenMP is based on thread approach . It launches a single process which in turn can create n number of thread as desired. It is based on what is called "fork and join method" i.e depending on particular task it can launch desired number of thread as directed by user.
 - C. Programming in OpenMP is relatively easy and involve adding *pragma* directive . User need to tell number of thread it need to use. (Note that launching more thread than number of processing unit available can actually slow down the whole program)

Difference between MPI and openMP:

MPI	OpenMP
Available from different vendor and can be compiled in desired platform with desired compiler. One can use any of MPI API i.e MPICH, OpenMPI or other	OpenMP are hooked with compiler so with gnu compiler and with Intel compiler one have specific implementation. User is at liberty with changing compiler but not with openmp implementation.
MPI support C,C++ and FORTRAN	OpenMP support C,C++ and FORTRAN
MPI target both distributed as well shared memory system	OpenMP target only shared memory system
Based on both process and thread based approach .(Earlier it was mainly process based parallelism but now with MPI 2 and 3 thread based parallelism is there too. Usually a process can contain more than 1 thread and call MPI subroutine as desired	Only thread based parallelism.
Overhead for creating process is one time	Depending on implementation threads can be created and joined for particular task which add overhead
Compilation of MPI program require 1. Adding header file : #include "mpi.h" 2. compiler as:(in linux) mpic++ mpi.cxx -o mpiExe (User need to set environment variable PATH and LD_LIBRARY_PATH to MPI as OpenMPI installed folder or binaries) (For Linux)	Need to add omp.h and then can directly compile code with -fopenmp in Linux environment g++ -fopenmp openmp.cxx -o openmpExe
Data racing is not there if not using any thread in process .	Data racing is inherent in OpenMP model
There are overheads associated with transferring message from one process to another	No such overheads, as thread can share variables

Write a hello world MPI program?

Code:

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Screenshots:



A terminal window showing the execution of a MPI hello world program. The user first edits the source code 'hello.c' using gedit. Then, they compile it with mpicc and run it with mpirun -np 4. The output shows four instances of the program running on different processors, each printing its rank and name.

```
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gedit hello.c
^C
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ mpicc hello.c -o hello
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ mpirun -np 4 ./hello
Hello world from processor rajvardhan-Inspiron-7460, rank 0 out of 4 processors
Hello world from processor rajvardhan-Inspiron-7460, rank 3 out of 4 processors
Hello world from processor rajvardhan-Inspiron-7460, rank 2 out of 4 processors
Hello world from processor rajvardhan-Inspiron-7460, rank 1 out of 4 processors
```

Write an MPI program to perform various tasks (arithmetic operations) using process rank id?

Code:

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int a=10,b=5;
    int c;
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    if(world_rank==0){
        c=a+b;
        printf("Addition operation from processor %s, rank %d out of %d processors\n%d
+ %d = %d\n=====\\n",processor_name, world_rank,
world_size,a,b,c);

    }
    if(world_rank==1){
        c=a-b;
        printf("Subtraction operation from processor %s, rank %d out of %d
processors\\n%d - %d = %d\\n=====\\n",processor_name,
world_rank, world_size,a,b,c);
    }
    if(world_rank==2){
        c=a*b;
    }
}
```

```

        printf("Multipication operation from processor %s, rank %d out of %d
processors\n%d x %d = %d\n=====\\n",processor_name,
world_rank, world_size,a,b,c);
    }if(world_rank==3){
        c=a/b;
        printf("Division operation from processor %s, rank %d out of %d processors\n%d
/ %d = %d\\n=====\\n",processor_name, world_rank,
world_size,a,b,c);
    }
// Finalize the MPI environment.
MPI_Finalize();
}

```

Screenshots:



A terminal window showing the execution of a C program (q3.c) using MPI. The program performs subtraction, addition, division, and multiplication operations across four processors.

```

rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gedit q3.c
^[[A^[[A^C
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ mpicc q3.c -o o3
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ mpirun -np 4 ./o3
Subtraction operation from processor rajvardhan-Inspiron-7460, rank 1 out of 4 processors
10 - 5 = 5
=====
Addition operation from processor rajvardhan-Inspiron-7460, rank 0 out of 4 processors
10 + 5 = 15
=====
Division operation from processor rajvardhan-Inspiron-7460, rank 3 out of 4 processors
10 / 5 = 2
=====
Multipication operation from processor rajvardhan-Inspiron-7460, rank 2 out of 4 processors
10 x 5 = 50
=====
```

Write an MPI program to simulate send and receive operations?

Code: (Passing an Array)

```
#include<mpi.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
    int rank, size, i, provided;
    float A[10];
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE,&provided);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        for(i=0;i<10;i++)
            A[i] = i;
        for(i=1;i<size; i++)
            MPI_Send(A, 10, MPI_FLOAT, i, 0,MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(A, 10, MPI_FLOAT, 0, 0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    printf("My rank %d of %d\n", rank, size );
    printf("Here are my values for A\n");
    for(i=0; i<10; i++)
        printf("%f", A[i]);
    printf("\n");
    MPI_Finalize();
}
```

Screenshots:

```
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ gedit q4.c
^C
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ mpicc q4.c -o o4
rajvardhan@rajvardhan-Inspiron-7460:~/Desktop/PDC$ mpirun -np 2 ./o4
My rank 0 of 2
Here are my values for A
0.0000001.0000002.0000003.0000004.0000005.0000006.0000007.0000008.0000009.000000
My rank 1 of 2
Here are my values for A
0.0000001.0000002.0000003.0000004.0000005.0000006.0000007.0000008.0000009.000000
```

PDC LAB5

Name: Umang Raval

Regno: 18BCE0170

Assignment 5

Point to point communication

code:

```
#include<stdio.h>
#include<mpi.h>

int main(int argc, char *argv[]){
    int rank;
    int size;
    char name[80];
    int length;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &length);

    // pack these values into a string
    int buffer_len=150;
    char buffer[buffer_len];
    sprintf(buffer, "Rank: %d Size: %d Machine: %s", rank, size, name);

    // synchronization so we can remove interleaved output
    if(rank == 0){
        // always print from rank 0
        printf("%s\n", buffer);
        for(int i=1;i<size;i++){
            MPI_Recv(buffer, buffer_len, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);

            // print out received msg
            printf("%s\n", buffer);
        }
    }
    else{
        MPI_Send(buffer, buffer_len, MPI_CHAR, 0, rank, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Result:

```
umang@umang-13:~/Documents/VIT/LABS/PDC/Lab5$ mpic++ 18BCE0170_P2P.c -o 18BCE0170_P2P
umang@umang-13:~/Documents/VIT/LABS/PDC/Lab5$ mpirun -np 4 ./18BCE0170_P2P
Rank: 0 Size: 4 Machine: umang-13
Rank: 3 Size: 4 Machine: umang-13
Rank: 1 Size: 4 Machine: umang-13
Rank: 2 Size: 4 Machine: umang-13
```

PDC LAB5

Name: Umang Raval

Regno: 18BCE0170

Group communication

code:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Check that 4 processes are used
    int comm_size;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    if(comm_size != 4)
    {
        printf("This application is meant to be run with 4 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    // Get the group or processes of the default communicator
    MPI_Group world_group;
    MPI_Comm_group(MPI_COMM_WORLD, &world_group);

    // Get my rank in the world group
    int my_world_group_rank;
    MPI_Group_rank(world_group, &my_world_group_rank);

    // Create the small group by including only processes 1 and 3 from the world group
    int small_group_ranks[2] = {1, 3};
    MPI_Group small_group;
    MPI_Group_incl(world_group, 2, small_group_ranks, &small_group);

    // Get my rank in the small group
    int my_small_group_rank;
    MPI_Group_rank(small_group, &my_small_group_rank);

    // Continue only if we are part of the small group
    if(my_small_group_rank != MPI_UNDEFINED)
    {
        printf("I am process %d in world group and %d in small group.\n", my_world_group_rank, my_small_group_rank);
    }
    else
    {
        printf("I am process %d in world group but I am not part of the small group.\n", my_world_group_rank);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

PDC LAB5

Name: Umang Raval

Regno: 18BCE0170

Result:

```
umang@umang-13:~/Documents/VIT/LABS/PDC/lab5$ mpic++ 18BCE0170_group.cpp -o 18BCE0170_group
umang@umang-13:~/Documents/VIT/LABS/PDC/lab5$ mpirun -n 4 ./18BCE0170_group
I am process 0 in world group but I am not part of the small group.
I am process 3 in world group and 1 in small group.
I am process 1 in world group and 0 in small group.
I am process 2 in world group but I am not part of the small group.
```

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 4th September 2020

Assessment No: 6

AIM:

Consider a suitable instance that has MPI routines to assign different tasks to different processors.

For example, parts of an input data set might be divided and processed by different processors, or a finite difference grid might be divided among the processors available. This means that the code needs to identify processors. In this example, processors are identified by rank - an integer from 0 to total number of processors.

- 1. Implement the logic using C**
- 2. Build the code**
- 3. Show the screenshots with proper justification**

SOURCE CODE:

```
#include<stdio.h>
#include<mpi.h>

int main(int argc, char **argv)
{
    int my_rank; int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello world! I'm rank (processor number) %d of size %d\n", my_rank, size);
    MPI_Finalize();
}
```

EXECUTION:

```

arpit@LAPTOP-D1TGC16E: ~
arpit@LAPTOP-D1TGC16E:~$ nano sudo mpi.c
arpit@LAPTOP-D1TGC16E:~$ nano sudo m1.c
arpit@LAPTOP-D1TGC16E:~$ mpicc m1.c -o H
arpit@LAPTOP-D1TGC16E:~$ mpirun -np 3 ./H
-----
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

Local host: LAPTOP-D1TGC16E
-----
Hello world! I'm rank (processor number) 0 of size 3
Hello world! I'm rank (processor number) 1 of size 3
Hello world! I'm rank (processor number) 2 of size 3
[LAPTOP-D1TGC16E:0156] 2 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[LAPTOP-D1TGC16E:0156] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
arpit@LAPTOP-D1TGC16E:~$ 

```

```

arpit@LAPTOP-D1TGC16E: ~
[1/1] m1.c
#include<stdio.h>
#include<mpi.h>
int main(int argc, char **argv)
{
    int my_rank; int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello world! I'm rank (processor number) %d of size %d\n", my_rank, size);
    MPI_Finalize();
}

```

REMARK:

An MPI process can query a communicator for information about the group, with MPI_COMM_SIZE and MPI_COMM_RANK. MPI_COMM_RANK (comm, rank) - MPI_COMM_RANK returns the rank of the calling process in the group associated with the communicator. MPI_COMM_SIZE (comm, size) - MPI_COMM_SIZE returns in size the number of processes in the group associated with the communicator. In the screenshot attached above, we see that “Hello World!” was printed on the screen using 4 different processors in parallel, and the rank of each of them being listed.

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 11th September 2020

Assessment No: 7

Aim: Consider the following program, called *mpi_sample1.c*. This program is written in C with MPI commands included.

The new MPI calls are to *MPI_Send* and *MPI_Recv* and to *MPI_Get_processor_name*. The latter is a convenient way to get the name of the processor on which a process is running. *MPI_Send* and *MPI_Recv* can be understood by stepping back and considering the two requirements that must be satisfied to communicate data between two processes:

1. Describe the data to be sent or the location in which to receive the data
2. Describe the destination (for a send) or the source (for a receive) of the data.

SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
#include"mpi.h"

int main(int argc, char* argv[]){
    int my_rank;
    int p;
    int source;
    int dest;
    int tag=0;
    char message[100];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if(my_rank != 0){
        sprintf(message,"Hello MPI World from process %d!",my_rank);
        dest = 0;
```

```

    MPI_Send(message,strlen(message)+1,MPI_CHAR,dest,tag,MPI_COMM_WORLD);

}

else{

    printf("Hello MPI World from process 0: Num processes : %d\n",p);

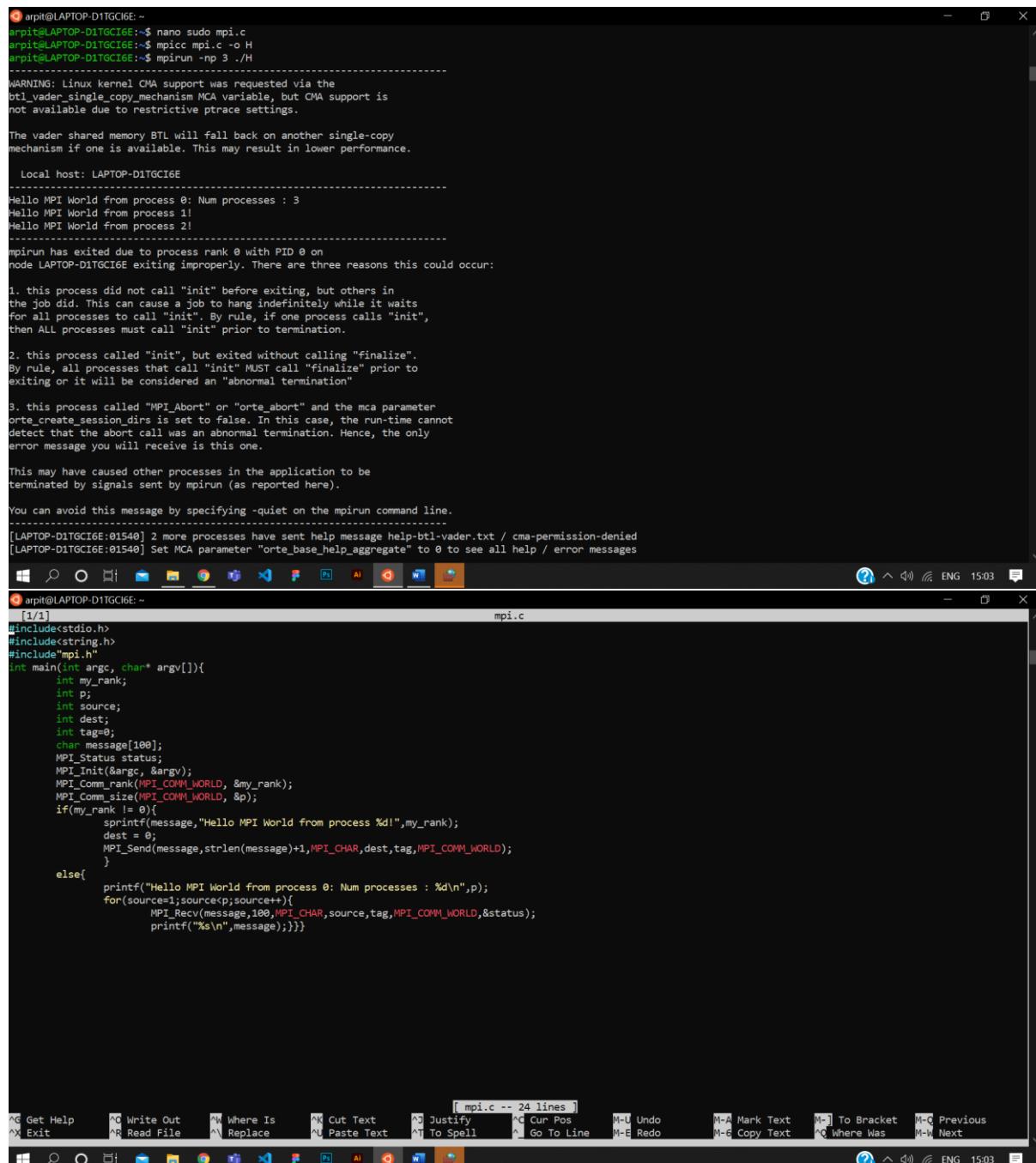
    for(source=1;source<p;source++){

        MPI_Recv(message,100,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);

        printf("%s\n",message);}}}

```

EXECUTION:



```

arpit@LAPTOP-DITGC16E: ~
arpit@LAPTOP-DITGC16E:~$ nano sudo mpi.c
arpit@LAPTOP-DITGC16E:~$ mpicc mpi.c -o H
arpit@LAPTOP-DITGC16E:~$ mpirun -np 3 ./H
-----
WARNING: Linux kernel OMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but OMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

Local host: LAPTOP-DITGC16E
-----
Hello MPI World from process 0: Num processes : 3
Hello MPI World from process 1!
Hello MPI World from process 2!
-----
mpirun has exited due to process rank 0 with PID 0 on
node LAPTOP-DITGC16E exiting improperly. There are three reasons this could occur:

1. this process did not call "init" before exiting, but others in
the job did. This can cause a job to hang indefinitely while it waits
for all processes to call "init". By rule, if one process calls "init",
then ALL processes must call "init" prior to termination.

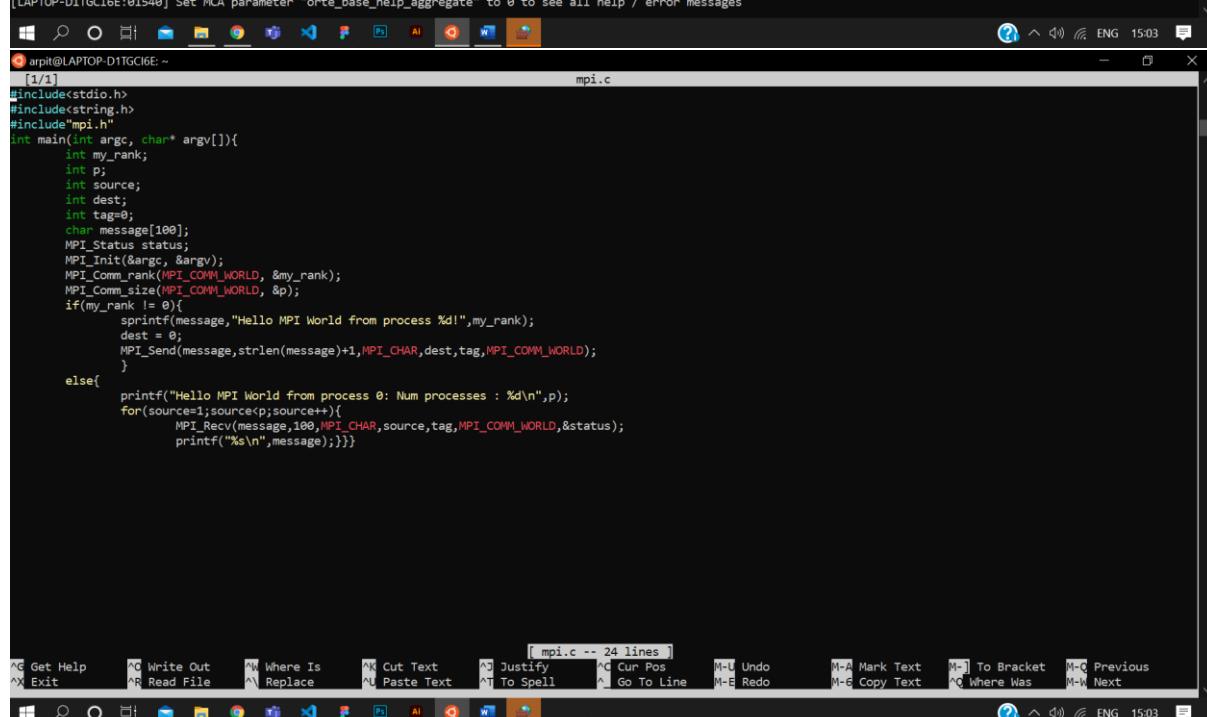
2. this process called "init", but exited without calling "finalize".
By rule, all processes that call "init" MUST call "finalize" prior to
exiting or it will be considered an "abnormal termination"

3. this process called "MPI_Abort" or "orte_abort" and the mca parameter
orte_create_session_dirs is set to false. In this case, the run-time cannot
detect that the abort call was an abnormal termination. Hence, the only
error message you will receive is this one.

This may have caused other processes in the application to be
terminated by signals sent by mpirun (as reported here).

You can avoid this message by specifying -quiet on the mpirun command line.
-----
[LAPTOP-DITGC16E:01540] 2 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[LAPTOP-DITGC16E:01540] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages

```



```

arpit@LAPTOP-DITGC16E: ~
[1/1] mpi.c
#include<stdio.h>
#include<string.h>
#include<mpi.h>
int main(int argc, char* argv[]){
    int my_rank;
    int p;
    int source;
    int dest;
    int tag=0;
    char message[100];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if(my_rank != 0){
        sprintf(message,"Hello MPI World from process %d!",my_rank);
        dest = 0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
    }
    else{
        printf("Hello MPI World from process 0: Num processes : %d\n",p);
        for(source=1;source<p;source++){
            MPI_Recv(message,100,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
            printf("%s\n",message);}}}

```

REMARKS:

- MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.
- MPI's send and receive calls operate in the following manner. First, process A decides a message needs to be sent to process B. Process A then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as envelopes since the data is being packed into a single message before transmission (similar to how letters are packed into envelopes before transmission to the post office). After the data is packed into a buffer, the communication device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank.
- Even though the message is routed to B, process B still has to acknowledge that it wants to receive A's data. Once it does this, the data has been transmitted. Process A is acknowledged that the data has been transmitted and may go back to work.
- Sometimes there are cases when A might have to send many different types of messages to B. Instead of B having to go through extra measures to differentiate all these messages, MPI allows senders and receivers to also specify message IDs with the message (known as tags). When process B only requests a message with a certain tag number, messages with different tags will be buffered by the network until B is ready for them.

NAME – AYUSH SHARMA

REG. NO. – 18BCE0172



VIT®

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

**SCHOOL OF COMPUTER SCIENCE ENGINEERING
FALL SEMESTER 2020-2021
CSE4001-PARALLEL AND DISTRIBUTED COMPUTING LAB
ASSESSMENT-3
L1+L2
Dr. K. Murugan**

1. Develop a MPI program to perform Merge sort with MPI_Scatter and MPI_Gather.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <time.h>
#include <mpi.h>
using namespace std;
//Merge
void merge(int *a, int *b, int l, int m, int r);
//main function for mearge
void mergeSort(int *a, int *b, int l, int r);
int main(int argc, char** argv)
{
    //intilize Array :
    int ArraySize=6;
    int *original_array = new int[ArraySize] ;
    cout<<"#####MPI MERGE SORT USING MPI GATHER AND MPI SCATTER---"
18BCE0172#####\n";
    //start intialize MPI services:
    int Process_rank;
    int ProcessSize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &Process_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcessSize);
```

```

//Print array
if (Process_rank==0)
{
    cout << "Enter Array to sort\n ";
    for (int c = 0; c <ArraySize; c++)
    {

        original_array[c] = rand() % ArraySize;
        if(c==ArraySize-1){
            cout << original_array[c];
        }
        else{
            cout << original_array[c]<<", ";
        }
        cout << "\n";
    }

    // Divide the array
    int SizeForSub = ArraySize / ProcessSize;
    //send sub array
    int *sub_array = new int[SizeForSub];
    MPI_Scatter(original_array, SizeForSub, MPI_INT, sub_array, SizeForSub, MPI_INT, 0,
MPI_COMM_WORLD);
    //show sub array
    for (int i = 0; i < ProcessSize; i++)
    {

        cout << "sub array (Sub Parallel Process): " << Process_rank << "\n";
        for (int c = 0; c < SizeForSub; c++)
        {
            if(c==SizeForSub-1){
                cout << sub_array[c];
            }
            else{
                cout << sub_array[c]<<", ";
            }
        }

        printf("\n");
        break;
    }

    //apply merge sort for each process
    int *tmp_array = new int[SizeForSub];
    mergeSort(sub_array, tmp_array, 0, (SizeForSub - 1));
}

```

```

// Gather the sorted subarrays
int *sorted =NULL;
if(Process_rank== 0)
{
    sorted = new int[ArraySize];

}

MPI_Gather(sub_array, SizeForSub, MPI_INT, sorted, SizeForSub, MPI_INT, 0,
MPI_COMM_WORLD);

//call final mearge function
if(Process_rank == 0)
{
    int *other_array =new int[ArraySize] ;
    mergeSort(sorted, other_array, 0, (ArraySize - 1));

    //show stored array
    cout<<"The sorted array is:\n";
    for(int c = 0; c <ArraySize; c++)
    {
        if(c==ArraySize-1){
            cout << sorted[c];
        }
        else{
            cout << sorted[c]<<", ";
        }

    }
    cout<<"\n\n\n";
    free(sorted);
    free(other_array);
}
free(original_array);
free(sub_array);
free(tmp_array);
// Finalize MPI
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
}

//merge Function
void merge(int *a, int *b, int l, int m, int r)
{

```

```

int h, i, j, k;
h = l;
i = l;
j = m + 1;
while ((h <= m) && (j <= r))
{
    if (a[h] <= a[j])
    {
        b[i] = a[h];
        h++;
    }
    else
    {
        b[i] = a[j];
        j++;
    }
    i++;
}
if (m < h)
{
    for (k = j; k <= r; k++)
    {
        b[i] = a[k];
        i++;
    }
}
else
{
    for (k = h; k <= m; k++)
    {
        b[i] = a[k];
        i++;
    }
}

for (k = l; k <= r; k++)
{
    a[k] = b[k];
}
}

//Merge Sort Function:
void mergeSort(int *a, int *b, int l, int r)
{
    int m;

```

```

        if (l < r) {
            m = (l + r) / 2;
            mergeSort(a, b, l, m);
            mergeSort(a, b, (m + 1), r);
            merge(a, b, l, m, r);
        }
    }
}

```

OUTPUT

```

kali㉿kali:~$ gedit mergeSort_18BCE0172.cpp
kali㉿kali:~$ mpic++ mergeSort_18BCE0172.cpp -o MergeSort
kali㉿kali:~$ mpirun -np 2 ./MergeSort
#####
MPI MERGE SORT USING MPI GATHER AND MPI SCATTER--- 18BCE0172#####
#####
MPI MERGE SORT USING MPI GATHER AND MPI SCATTER--- 18BCE0172#####
Enter Array to sort
1, 4, 3, 1, 5, 1
sub array (Sub Parallel Process): 0
1, 4, 3
sub array (Sub Parallel Process): 1
1, 5, 1
The sorted array is:
1, 1, 1, 3, 4, 5

```

2. Write a C program using MPI to implement Dot product.

CODE:

```

#include <stdio.h>
#include "mpi.h"
#define MAX_LOCAL_ORDER 100
void main(int argc, char* argv[]) {
    printf("##### 18BCE0172 - DOT PRODUCT USING MPI #####\n");
    float local_x[MAX_LOCAL_ORDER];
    float local_y[MAX_LOCAL_ORDER];
    int n,n_bar; /* = n/p */
    float dot;
    int p,my_rank;
    void Read_vector(char* prompt, float local_v[], int n_bar, int p,int my_rank);
    float Parallel_dot(float local_x[], float local_y[], int n_bar);
    MPI_Init(&argc, &argv);

```

```
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank == 0) {
printf("Enter the order of the vectors\n");
scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
n_bar = n/p;
Read_vector("the first vector", local_x, n_bar, p, my_rank);
Read_vector("the second vector", local_y, n_bar, p, my_rank);
dot = Parallel_dot(local_x, local_y, n_bar);
if (my_rank == 0) printf("The dot product is %f\n", dot);
MPI_Finalize();
} /* main */

void Read_vector(
char* prompt /* in */,
float local_v[] /* out */,
int n_bar /* in */,
int p /* in */,
int my_rank /* in */) {
int i, q;
float temp[MAX_LOCAL_ORDER];
MPI_Status status;
if (my_rank == 0) {
printf("Enter %s\n", prompt);
for (i = 0; i < n_bar; i++)
scanf("%f", &local_v[i]);
for (q = 1; q < p; q++) {
for (i = 0; i < n_bar; i++)
scanf("%f", &temp[i]);
}
```

```
MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
}

} else {

MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
&status);

}

} /* Read_vector */

float Serial_dot(
float x[] /* in */,
float y[] /* in */,
int n /* in */) {

int i;
float sum = 0.0;
for (i = 0; i < n; i++)
sum = sum + x[i]*y[i];
return sum;
}

float Parallel_dot(
float local_x[] /* in */,
float local_y[] /* in */,
int n_bar /* in */) {

float local_dot;
float dot = 0.0;
float Serial_dot(float x[], float y[], int m);
local_dot = Serial_dot(local_x, local_y, n_bar);
MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
MPI_SUM, 0, MPI_COMM_WORLD);
return dot;
} /* Parallel_dot */
```

OUTPUT:

```
kali㉿kali:~$ gedit dotProduct_18BCE0172.c
kali㉿kali:~$ mpicc dotProduct_18BCE0172.c -o object_file
kali㉿kali:~$ mpirun -np 1 ./object_file
##### 18BCE0172 - DOT PRODUCT USING MPI #####
Enter the order of the vectors
12
Enter the first vector
1 2 3 4 5 6 7 8 9 10 11 12
Enter the second vector
2 4 6 8 10 12 14 16 18 20 22 24
The dot product is 1300.000000
kali㉿kali:~$ #18BCE0172_PDC_ASSIGNMENT
```

3. Write a MPI program to print pi value.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"
#include "mpi.h"
int threads;
static long num_steps;
double step;
void main(int argc, char** argv)
{
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    printf("#####18BCE0172- Approximate Value of PI Using
MPI#####\n");
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

```

// Get the rank of the process

int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

num_steps = atol(argv[2]);

if (argc>3) threads = atol(argv[4]);

double pi,sum=0.,sum_inter = 0.;

step = 1.0/(double) num_steps;

omp_set_num_threads(threads);

#pragma omp parallel shared(sum) private(sum_inter)

{

    double x;

    int ID = omp_get_thread_num();

    int workers = omp_get_num_threads()*world_size;

    for (int i=ID+world_rank*omp_get_num_threads();i<num_steps;i+=workers)

    {

        x = (i+0.5)*step;

        sum_inter = sum_inter + 4.0/(1.0+x*x);

    }

    #pragma omp critical

    sum += sum_inter;

}

//MPI_Barrier(MPI_COMM_WORLD);

double global_sum;

MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

if (world_rank==0)

```

```

{
    pi = step*global_sum;
    printf("PI = %f\n",pi);
}

// Finalize the MPI environment.

MPI_Finalize();

```

OUTPUT:

```

kali@kali:~$ gedit pi_18BCE0172.c
kali@kali:~$ mpicc pi_18BCE0172.c -o pi_omp_mpi -fopenmp
kali@kali:~$ mpiexec -n 2 ./pi_omp_mpi -steps 100000 -threads 4
#####
#18BCE0172- Approximate Value of PI Using MPI#####
#18BCE0172- Approximate Value of PI Using MPI#####
PI = 3.141593
kali@kali:~$ #18BCE0172

```

4. Compute sum of array using MPI.

CODE:

```

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// size of array
#define n 10

int a[10];
int a2[1000];

int main(int argc, char* argv[])
{
    printf("\n 18BCE0172-SUM OF ARRAY USING MPI\nEnter the array elements to get the Array Sum: \n");
    for(int i=0;i<10;i++){

```

```

scanf("%d",&a[i]);
}

int pid, np, elements_per_process, n_elements_recieved;
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &np);

if (pid == 0) {
    int index, i;
    elements_per_process = n / np;
    if (np > 1) {
        for (i = 1; i < np - 1; i++) {
            index = i * elements_per_process;
            MPI_Send(&elements_per_process, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
        // last process adds remaining elements
        index = i * elements_per_process;
        int elements_left = n - index;
        MPI_Send(&elements_left, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        MPI_Send(&a[index], elements_left, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
    // master process add its own sub array
    int sum = 0;
    for (i = 0; i < elements_per_process; i++)
        sum += a[i];
    // collects partial sums from other processes
    int tmp;
    for (i = 1; i < np; i++) {
        MPI_Recv(&tmp, 1, MPI_INT,

```

```

        MPI_ANY_SOURCE, 0,
        MPI_COMM_WORLD,
        &status);

    int sender = status.MPI_SOURCE;
    sum += tmp;
}

// prints the final sum of array
printf("Sum of array is : %d\n", sum);

}

// slave processes
else {
    MPI_Recv(&n_elements_recieved,
        1, MPI_INT, 0, 0,
        MPI_COMM_WORLD,
        &status);

    // stores the received array segment
    // in local array a2
    MPI_Recv(&a2, n_elements_recieved,
        MPI_INT, 0, 0,
        MPI_COMM_WORLD,
        &status);

    // calculates its partial sum
    int partial_sum = 0;
    for (int i = 0; i < n_elements_recieved; i++)
        partial_sum += a2[i];

    // sends the partial sum to the root process
    MPI_Send(&partial_sum, 1, MPI_INT,
        0, 0, MPI_COMM_WORLD);

}

// cleans up all MPI state before exit of process

```

```
        MPI_Finalize();  
        return 0;  
    }
```

OUTPUT:

```
kali@kali:~$ gedit arraySum_18BCE0172.c  
kali@kali:~$ mpicc arraySum_18BCE0172.c -o object_file  
kali@kali:~$ mpirun -np 1 ./object_file  
  
18BCE0172-SUM OF ARRAY USING MPI  
Enter the array elements to get the Array Sum:  
1 2 3 4 5 2 2 2 2  
Sum of array is : 25  
kali@kali:~$ #18BCE0172_PDC_ASSIGNMENT
```

NAME – AYUSH SHARMA

REG. NO. – 18BCE0172



VIT[®]

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

**SCHOOL OF COMPUTER SCIENCE ENGINEERING
FALL SEMESTER 2020-2021
CSE4001-PARALLEL AND DISTRIBUTED COMPUTING LAB
ASSESSMENT-4
L1+L2
Dr. K. Murugan**

- 1. Write a C Program with OpenMP to compute the value of “pi” function by numerical integration of a function $f(x) = 4/(1+x^2)$ between the limits 0 and 1**

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<math.h>
#include<omp.h>

#define PI 3.1415926538837211

/* Main Program */
void main(int argc,char **argv)
{
    int      Noofintervals, i, Noofthreads, threadid;
    double    x, totalsum, h;
    struct timeval TimeValue_Start;
    struct timezone TimeZone_Start;

    struct timeval TimeValue_Final;
    struct timezone TimeZone_Final;
    long      time_start, time_end;
    double    time_overhead;
```

```

printf("\n\t-----");
printf("\n\t 18BCE0172");
printf("\n\t-----");
printf("\n\t PDC LAB DA-4 ");
printf("\n\t Write a C Program with OpenMP to compute the value of “pi” function by numerical
integration of a function f(x) = 4/(1+x*x ) between the limits 0 and 1");
printf("\n\t.....\n");

/* Checking for command line arguments */
if( argc != 3 ){

    printf("\t Very Few Arguments\n ");
    printf("\t Syntax : exec <Threads> <no. of interval>\n");
    exit(-1);
}

Noofthreads=atoi(argv[1]);
if ((Noofthreads!=1) && (Noofthreads!=2) && (Noofthreads!=4) && (Noofthreads!=8) &&
(Noofthreads!= 16) ) {
    printf("\n Number of threads should be 1,2,4,8 or 16 for the execution of program. \n\n");
    exit(-1);
}

Noofintervals=atoi(argv[2]);

printf("\n\t Threads : %d ",Noofthreads);

/* No. of intervals should be positive integer */
if (Noofintervals <= 0) {
    printf("\n\t Number of intervals should be positive integer\n");
    exit(1);
}
totalsum = 0.0;

gettimeofday(&TimeValue_Start, &TimeZone_Start);

h = 1.0 / Noofintervals;

/* set the number of threads */
omp_set_num_threads(Noofthreads);
/*
 * OpenMP Parallel Directive With Private Clauses And Critical
 * Section
 */

```

```

#pragma omp parallel for private(x)
for (i = 1; i < Noofintervals + 1; i = i+1) {
    x = h * (i + 0.5);
    /*printf("the thread id is %d with iteration %d ",omp_get_thread_num(),i);*/
    #pragma omp critical
    totalsum = totalsum + 4.0/(1.0 + x * x);
} /* All thread join Master thread */
totalsum = totalsum * h;
gettimeofday(&TimeValue_Final, &TimeZone_Final);

time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;

time_overhead = (time_end - time_start)/1000000.0;

/*printf("\n\t Calculated PI : \t%1.15lf \n\t Error : \t%1.16lf\n", totalsum, fabs(totalsum - PI));*/
    printf("\n\t Calculated PI      : %1.15lf",totalsum );
    printf("\n\t Time in Seconds (T) : %lf",time_overhead);
    printf("\n\n\t ( T represents the Time taken for computation )");
    printf("\n\t.....\n");
}

```

OUTPUT:

```

kali㉿kali:~$ gedit pi_18BCE0172.c
kali㉿kali:~$ gcc -o omp_helloc -fopenmp pi_18BCE0172.c
kali㉿kali:~$ ./omp_helloc 4 100000000

-----
18BCE0172
-----
PDC LAB DA-4
Write a C Program with OpenMP to compute the value of "pi" function by numerical integration of a function f(x) = 4/(1+x*x ) between the limits 0 and 1
-----
Threads : 4
Calculated PI      : 3.141592633590134
Time in Seconds (T) : 3.520570
( T represents the Time taken for computation )
-----
kali㉿kali:~$ 

```

2. Implement merge sort and parallelize it. (can use any programming language)

CODE:

```
#include<iostream>
#include<cstdlib>
#include<omp.h>
#include<pthread.h>
#include<time.h>
using namespace std;
#define MAX 20
#define THREAD_MAX 4
int a[MAX];
int part = 0;
void merge(int low, int mid, int high)
{
    int* left = new int[mid - low + 1];
    int* right = new int[high - mid];
    int n1 = mid - low + 1, n2 = high - mid, i, j;
    for (i = 0; i < n1; i++)
        left[i] = a[i + low];
    for (i = 0; i < n2; i++)
        right[i] = a[i + mid + 1];
    int k = low;
    i = j = 0;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j])
            a[k++] = left[i++];
        else
            a[k++] = right[j++];
    }
    while (i < n1) {
        a[k++] = left[i++];
    }
    while (j < n2) {
        a[k++] = right[j++];
    }
}
void merge_sort(int low, int high)
{
    int mid = low + (high - low) / 2;
    if (low < high) {
        merge_sort(low, mid);
        merge_sort(mid + 1, high);
        merge(low, mid, high);
    }
}
```

```

}

void* merge_sort(void* arg)
{
int thread_part = part++;
int low = thread_part * (MAX / 4);
int high = (thread_part + 1) * (MAX / 4) - 1;
int mid = low + (high - low) / 2;
if (low < high) {
    merge_sort(low, mid);
    merge_sort(mid + 1, high);
    merge(low, mid, high);
}
return 0;
}

int main()
{
cout<<"-----18BCE0172-----\n";
cout<<"-----PDC LAB DA-4 (parallelized merge sort)-----\n";
for (int i = 0; i < MAX; i++)
a[i] = rand() % 100;
cout<<"Original Array : \t";
for (int i = 0; i < MAX; i++)
cout<<a[i]<<" ";
cout<<endl;
clock_t t1, t2;
t1 = clock();
pthread_t threads[THREAD_MAX];
for (int i = 0; i < THREAD_MAX; i++)
pthread_create(&threads[i], NULL, merge_sort,
(void*)NULL);
for (int i = 0; i < 4; i++)
pthread_join(threads[i], NULL);
merge(0, (MAX / 2 - 1) / 2, MAX / 2 - 1);
merge(MAX / 2, MAX/2 + (MAX-1-MAX/2)/2, MAX - 1);
merge(0, (MAX - 1)/2, MAX - 1);
t2 = clock();
cout << "Sorted array: \t\t";
for (int i = 0; i < MAX; i++)
cout<<a[i]<<" ";
cout<<endl << "\nTime taken : "<<(t2 - t1) /
(double)CLOCKS_PER_SEC << endl;
return 0;
}

```

OUTPUT:

```
kali@kali:~$ gedit merge_18BCE0172.cpp
kali@kali:~$ g++ -o omp_helloc -fopenmp merge_18BCE0172.cpp
kali@kali:~$ ./omp_helloc 4
-----18BCE0172-----
-----PDC LAB DA-4 (parallelized merge sort)-----
Original Array :      83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36
Sorted array:        15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93

Time taken : 0.000267
kali@kali:~$ #18BCE0172
```

DATED: 11 march ,20

ASSIGNMENT: 7

Name: Shiavni Goyal

Reg.no-18BCE0817

Aim: Write a simple MPI program

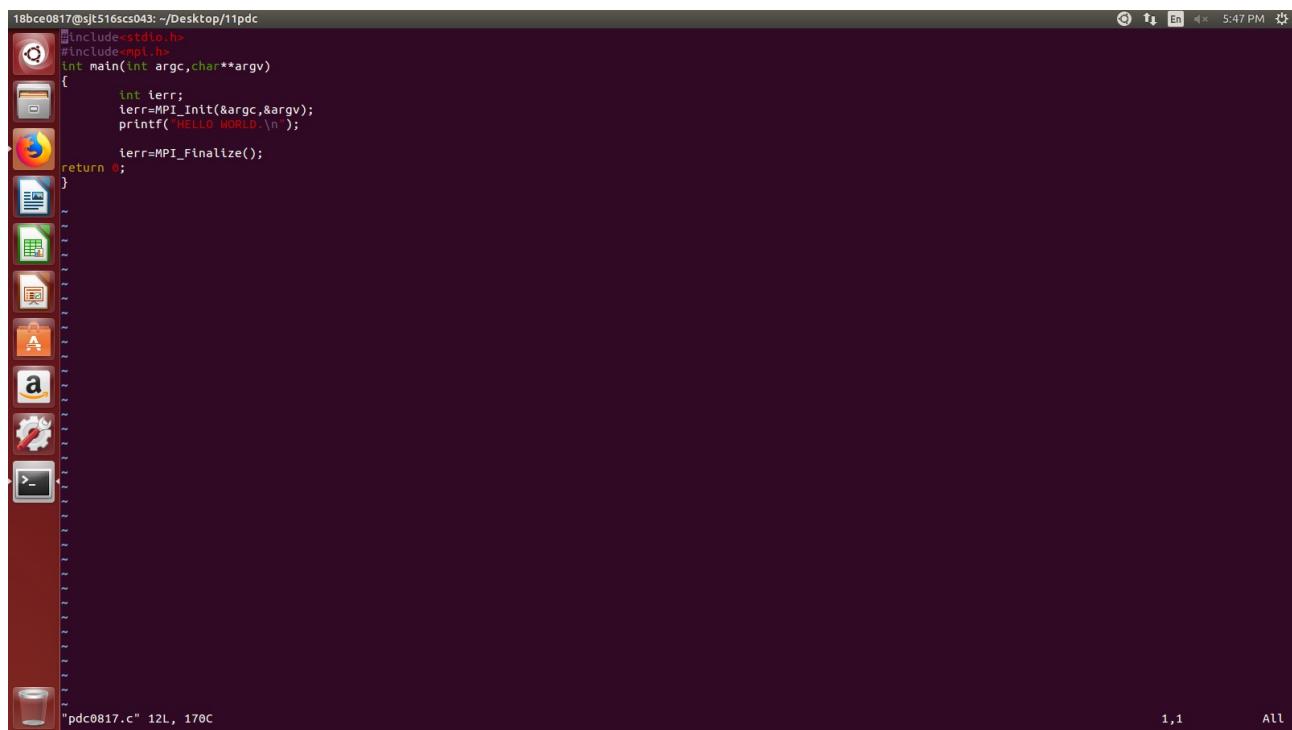
a) To demonstrate ‘Helloworld’ using MPI_Init() and MPI_Finalize()

SOURCE CODE:

```
#include<stdio.h>
#include<mpi.h>
int main(int argc,char**argv)
{
    int ierr;
    ierr=MPI_Init(&argc,&argv);
    printf("HELLO WORLD.\n");

    ierr=MPI_Finalize();
return 0;
}
```

EXECUTION:



The screenshot shows a terminal window titled "pdc0817.c" with 12L and 170C. The window contains the following text:

```
18bce0817@sjt516scs043: ~/Desktop/11pdc
#include<stdio.h>
#include<mpi.h>
int main(int argc,char**argv)
{
    int ierr;
    ierr=MPI_Init(&argc,&argv);
    printf("HELLO WORLD.\n");

    ierr=MPI_Finalize();
return 0;
}
```

The terminal window has a dark background and a vertical toolbar on the left side.

RESULT:

```
18bce0817@sjt516scs043:~/Desktop/11pdc$ vi pdc0817.c
18bce0817@sjt516scs043:~/Desktop/11pdc$ mpicc pdc0817.c -o p
18bce0817@sjt516scs043:~/Desktop/11pdc$ mpirun -np 4 ./p
HELLO WORLD.
HELLO WORLD.
HELLO WORLD.
HELLO WORLD.
18bce0817@sjt516scs043:~/Desktop/11pdc$ █
```

b) To demonstrate ‘gethostname() function’ using MPI_Init() and MPI_Finalize()**SOURCE CODE:**

```
#include<stdio.h>
#include<mpi.h>
#include<unistd.h>

int main(int argc,char**argv)

{
    int ierr;
    ierr=MPI_Init(&argc,&argv);

    int HOST_NAME_MAX =20;

    char hostname[HOST_NAME_MAX +1];
    gethostname(hostname,HOST_NAME_MAX +1);
    printf("hostname:%s\n",hostname);
    ierr=MPI_Finalize();

return 0;

}
```

EXECUTION:

A screenshot of a Linux desktop environment. On the left is a vertical dock containing icons for various applications like a web browser, file manager, terminal, and system tools. The main window is a terminal with the following content:

```
18bce0817@sjt516scs043:~/Desktop/11pdc$ vi pdc817.c
18bce0817@sjt516scs043:~/Desktop/11pdc$ mpicc pdc817.c -o p
18bce0817@sjt516scs043:~/Desktop/11pdc$ mpirun -np 4 ./p
hostname:sjt516scs043
hostname:sjt516scs043
hostname:sjt516scs043
hostname:sjt516scs043
18bce0817@sjt516scs043:~/Desktop/11pdc$
```

RESULT:

```
18bce0817@sjt516scs043:~/Desktop/11pdc$ vi pdc817.c
18bce0817@sjt516scs043:~/Desktop/11pdc$ mpicc pdc817.c -o p
18bce0817@sjt516scs043:~/Desktop/11pdc$ mpirun -np 4 ./p
hostname:sjt516scs043
hostname:sjt516scs043
hostname:sjt516scs043
hostname:sjt516scs043
18bce0817@sjt516scs043:~/Desktop/11pdc$
```

using `gethostname()` function by using `MPI_Init()` and `MPI_Finalize()` we are able to identify the hostname.

c) To demonstrate ‘MPI_Comm_size() and MPI_Comm_rank ()’ using MPI_Init() and MPI_Finalize()

SOURCE CODE-

```
#include<stdio.h>

#include<mpi.h>

int main(int argc,char**argv)

{

    int ierr,num_pro,my_id;
    ierr=MPI_Init(&argc,&argv);

    ierr=MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
    ierr=MPI_Comm_size(MPI_COMM_WORLD,&num_pro);
    printf("HELLO WORLD. I'm a process %i out of %i process\n",my_id,num_pro);

    ierr=MPI_Finalize();

return 0;

}
```

EXECUTION:

```
18bce0817@sjt516scs043: ~/Desktop/11pdc
# include<stdio.h>
# include<mpi.h>
int main(int argc,char**argv)
{
    int ierr,num_pro,my_id;
    ierr=MPI_Init(&argc,&argv);
    ierr=MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
    ierr=MPI_Comm_size(MPI_COMM_WORLD,&num_pro);
    printf("HELLO WORLD. I'm a process %i out of %i process\n",my_id,num_pro);

    ierr=MPI_Finalize();

return 0;
}

"pdc817.c" 25L, 363C
```

RESULT:

```
18bce0817@sjt516scs043:~/Desktop/11pdc$ vi pdc817.c
18bce0817@sjt516scs043:~/Desktop/11pdc$ mpicc pdc817.c -o p
18bce0817@sjt516scs043:~/Desktop/11pdc$ mpirun -np 4 ./p
HELLO WORLD. I'm a process 0 out of 4 process
HELLO WORLD. I'm a process 1 out of 4 process
HELLO WORLD. I'm a process 2 out of 4 process
HELLO WORLD. I'm a process 3 out of 4 process
18bce0817@sjt516scs043:~/Desktop/11pdc$ █
```

REMARKS:

using c language and message passing interface ,using library= mpi.h and unistd.h we could verify the working of the functions mentioned above.



DIGITAL ASSIGNMENT 1

Parallel Distributed Computing
CSE 4001

L13+L14

Anuj Shukla
18BCE0163

1. Write an OpenMP program to compute the value of pi function by numerical integration of a function $f(x) = 4/(1+x^2)$ between the limits 0 and 1

Code -

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<math.h>
#include<omp.h>

#define PI 3.1415926538837211

/* Main Program */
int main(int argc,char **argv)
{
    int      Noofintervals, i, Noofthreads, threadid;
    double   x, totalsum, h;
    struct timeval TimeValue_Start;
    struct timezone TimeZone_Start;

    struct timeval TimeValue_Final;
    struct timezone TimeZone_Final;
    long     time_start, time_end;
    double   time_overhead;

    /* Checking for command line arguments */
    if( argc != 3 ){

        printf("\t\t Very Few Arguments\n");
        printf("\t\t Syntax : exec <Threads> <no. of interval>\n");
        exit(-1);
    }
```

```
Noofthreads=atoi(argv[1]);
if ((Noofthreads!=1) && (Noofthreads!=2) && (Noofthreads!=4) &&
(Noofthreads!=8) && (Noofthreads!= 16) ) {
    printf("\n Number of threads should be 1,2,4,8 or 16 for the execution of program.
\n\n");
    exit(-1);
}

Noofintervals=atoi(argv[2]);

printf("\n\t\t Threads : %d ",Noofthreads);

/* No. of intervals should be positive integer */
if (Noofintervals <= 0) {
    printf("\n\t\t Number of intervals should be positive integer\n");
    exit(1);
}
totalsum = 0.0;

gettimeofday(&TimeValue_Start, &TimeZone_Start);

h = 1.0 / Noofintervals;

/* set the number of threads */
omp_set_num_threads(Noofthreads);
/*
 * OpenMP Parallel Directive With Private Clauses And Critical
 * Section
 */
#pragma omp parallel for private(x)
for (i = 1; i < Noofintervals + 1; i = i+1) {
    x = h * (i + 0.5);
    /*printf("the thread id is %d with iteration %d ",omp_get_thread_num(),i);*/
    #pragma omp critical
    totalsum = totalsum + 4.0/(1.0 + x * x);
} /* All thread join Master thread */
totalsum = totalsum * h;
gettimeofday(&TimeValue_Final, &TimeZone_Final);
```

```
time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;

time_overhead = (time_end - time_start)/1000000.0;

/*printf("\n\t\t Calculated PI : \t%1.15lf \n\t\t Error : \t%1.16lf\n", totalsum,
fabs(totalsum - PI));*/
printf("\n\t\t Calculated PI      : %1.15lf",totalsum );
printf("\n\t\t Time in Seconds (T)   : %lf",time_overhead);
printf("\n\n\t\t ( T represents the Time taken for computation )");
printf("\n\t.....\n");

}
```

Output -

```
kali㉿kali:~$ gedit 18BCE0163.c
kali㉿kali:~$ gcc 18BCE0163.c -fopenmp
kali㉿kali:~$ ./a.out 4 2000

Threads : 4
Calculated PI          : 3.140592424579374
Time in Seconds (T)    : 0.000195

( T represents the Time taken for computation )
.....
kali㉿kali:~$
```

2. Write a C program with OpenMP to illustrate
 - (I) Dot Product
 - (II) Vector Addition

(I) Dot Product

Code -

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

/* Define length of dot product vectors and number of OpenMP threads */
#define VECLEN 100
#define NUMTHREADS 8

int main (int argc, char* argv[])
{
    int i, tid, len=VECLEN, threads=NUMTHREADS;
    double *a, *b;
    double sum, psum;

    printf("Starting omp_dotprod_openmp. Using %d threads\n",threads);

    /* Assign storage for dot product vectors */
    a = (double*) malloc (len*threads*sizeof(double));
    b = (double*) malloc (len*threads*sizeof(double));

    /* Initialize dot product vectors */
    for (i=0; i<len*threads; i++) {
        a[i]=1.0;
        b[i]=a[i];
    }
    /* Initialize global sum */
    sum = 0.0;

    /*
     Perform the dot product in an OpenMP parallel region for loop with a sum reduction
     For illustration purposes:
     - Explicitly sets number of threads
     - Each thread keeps track of its partial sum
    */
}
```

```
#pragma omp parallel private(i,tid,psum) num_threads(threads)
{
    psum = 0.0;
    tid = omp_get_thread_num();

    #pragma omp for reduction(+:sum)
    for (i=0; i<len*threads; i++)
    {
        sum += (a[i] * b[i]);
        psum = sum;
    }
    printf("Thread %d partial sum = %f\n",tid, psum);
}

printf ("Done. OpenMP version: sum = %f\n", sum);

free (a);
free (b);
}
```

Output -

```
kali㉿kali:~$ gcc 18BCE0163.c -fopenmp
kali㉿kali:~$ ./a.out 4 2000
Starting omp_dotprod_openmp. Using 8 threads
Thread 1 partial sum = 100.000000
Thread 0 partial sum = 100.000000
Thread 4 partial sum = 100.000000
Thread 6 partial sum = 100.000000
Thread 5 partial sum = 100.000000
Thread 3 partial sum = 100.000000
Thread 2 partial sum = 100.000000
Thread 7 partial sum = 100.000000
Done. OpenMP version: sum = 800.000000
kali㉿kali:~$
```

(II) Vector Addition

Code -

```
#include <omp.h>
#include<stdio.h>
#define CHUNKSIZE 100
#define N 1000

int main(){
    int i, chunk;
    float a[N], b[N], c[N];

    for (i=0;i<N;i++)
        a[i] = b[i] = i*1.0;

    chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
{
    #pragma omp for schedule (dynamic,chunk) nowait
    for (i=0;i<N;i++)
    {
        c[i] = a[i] + b[i];
        printf("%f", c[i]);
    }
}
```

}

```
return 0;  
}
```

Output -



DIGITAL ASSIGNMENT 2

Parallel and Distributed Computing

CSE 4001

L13+L14

Anuj Shukla

18BCE0163

1) Loop Work Sharing

CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 10

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];

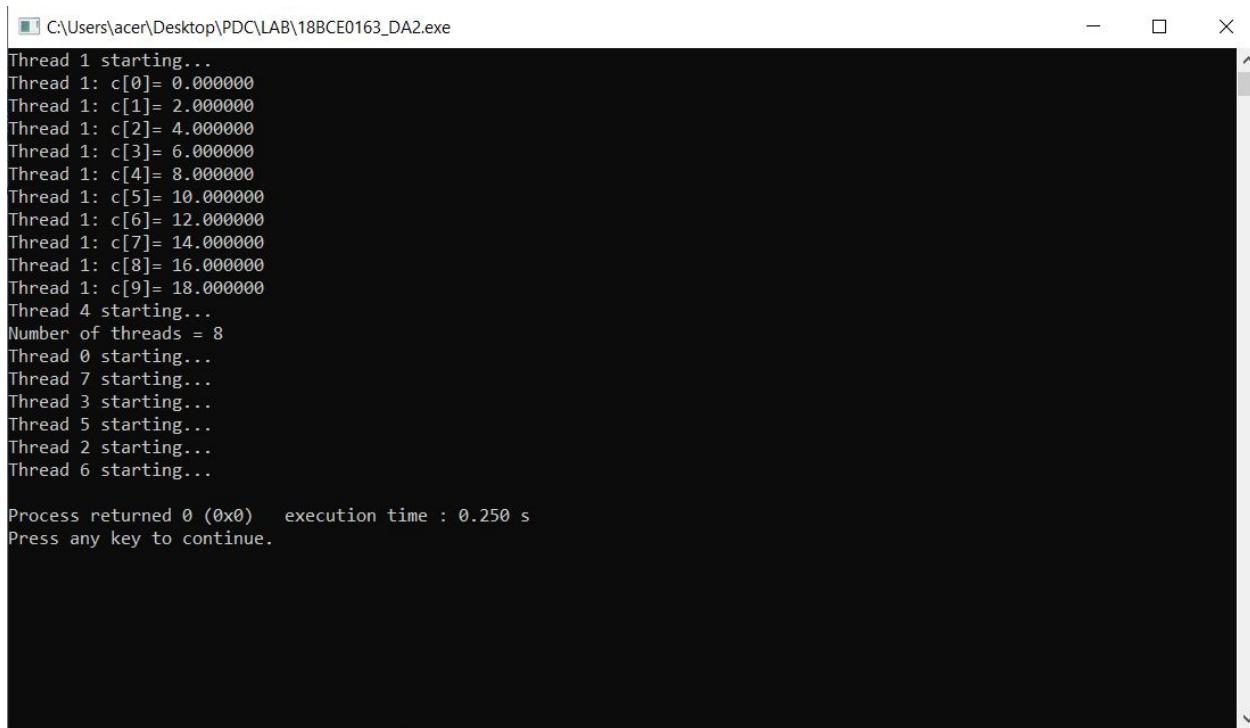
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n", tid);

        #pragma omp for schedule(dynamic,chunk)
        for (i=0; i<N; i++)
        {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
        }
    }
}
```

```
} /* end of parallel section */  
}
```

OUTPUT:



```
C:\Users\acer\Desktop\PDC\LAB\18BCE0163_DA2.exe  
Thread 1 starting...  
Thread 1: c[0]= 0.000000  
Thread 1: c[1]= 2.000000  
Thread 1: c[2]= 4.000000  
Thread 1: c[3]= 6.000000  
Thread 1: c[4]= 8.000000  
Thread 1: c[5]= 10.000000  
Thread 1: c[6]= 12.000000  
Thread 1: c[7]= 14.000000  
Thread 1: c[8]= 16.000000  
Thread 1: c[9]= 18.000000  
Thread 4 starting...  
Number of threads = 8  
Thread 0 starting...  
Thread 7 starting...  
Thread 3 starting...  
Thread 5 starting...  
Thread 2 starting...  
Thread 6 starting...  
  
Process returned 0 (0x0)  execution time : 0.250 s  
Press any key to continue.
```

2) Section Work Sharing

CODE:

```
#include <omp.h>  
#include <stdio.h>  
#include <stdlib.h>  
#define N 10  
  
int main (int argc, char *argv[])  
{
```

```
int i, nthreads, tid;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i<N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
    c[i] = d[i] = 0.0;
}

#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);

#pragma omp sections nowait
{
    #pragma omp section
    {
        printf("Thread %d doing section 1\n", tid);
        for (i=0; i<N; i++)
        {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
        }
    }

    #pragma omp section
    {
        printf("Thread %d doing section 2\n", tid);
        for (i=0; i<N; i++)
        {
            d[i] = a[i] * b[i];
            printf("Thread %d: d[%d]= %f\n", tid, i, d[i]);
        }
    }
}
```

```
}

}

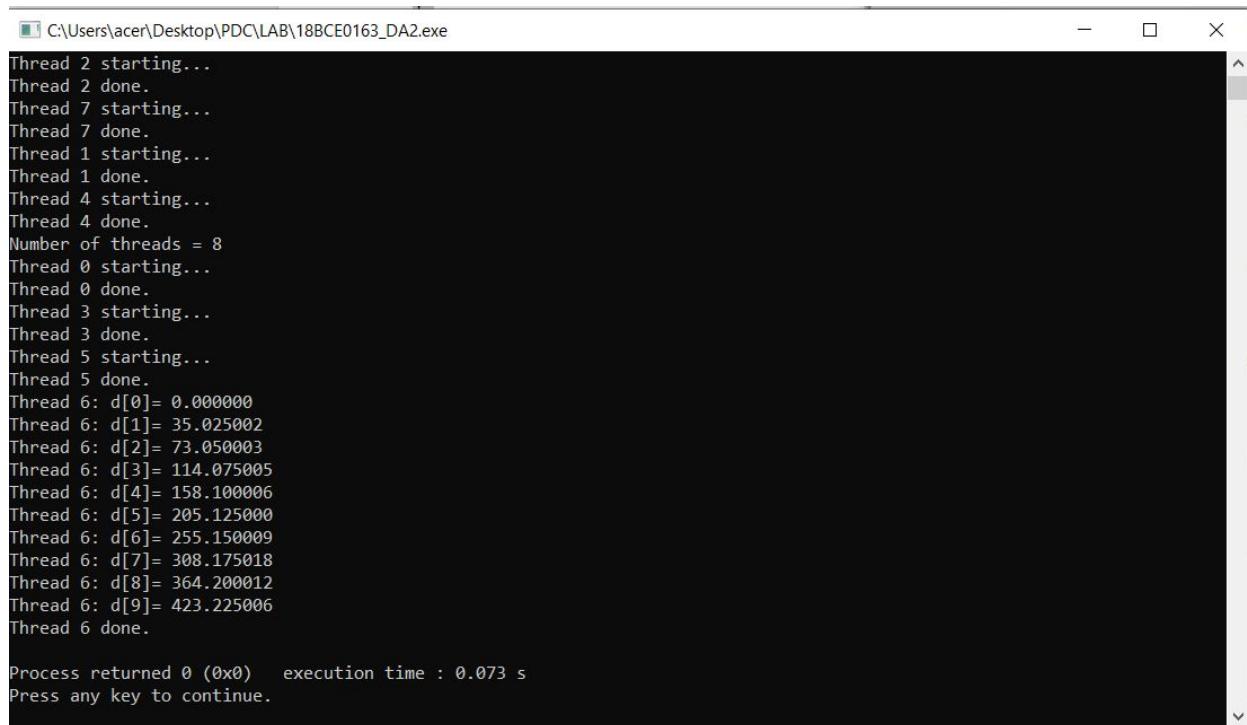
} /* end of sections */

printf("Thread %d done.\n",tid);

} /* end of parallel section */

}
```

OUTPUT:



```
C:\Users\acer\Desktop\PDC\LAB\18BCE0163_DA2.exe
Thread 2 starting...
Thread 2 done.
Thread 7 starting...
Thread 7 done.
Thread 1 starting...
Thread 1 done.
Thread 4 starting...
Thread 4 done.
Number of threads = 8
Thread 0 starting...
Thread 0 done.
Thread 3 starting...
Thread 3 done.
Thread 5 starting...
Thread 5 done.
Thread 6: d[0]= 0.000000
Thread 6: d[1]= 35.025002
Thread 6: d[2]= 73.050003
Thread 6: d[3]= 114.075005
Thread 6: d[4]= 158.100006
Thread 6: d[5]= 205.125000
Thread 6: d[6]= 255.150009
Thread 6: d[7]= 308.175018
Thread 6: d[8]= 364.200012
Thread 6: d[9]= 423.225006
Thread 6 done.

Process returned 0 (0x0)   execution time : 0.073 s
Press any key to continue.
```

3) Matrix Transpose

CODE:

```
#include <stdio.h>
#include <sys/time.h>
#include <omp.h>
#include <stdlib.h>

/* Main Program */
int main(int argc,char **argv)
{
    int      NoofRows, NoofCols, i, j, Total_threads, Noofthreads;
    float    **Matrix, **Trans, **Checkoutput, flops;

    struct timeval TimeValue_Start;
    struct timezone TimeZone_Start;

    struct timeval TimeValue_Final;
    struct timezone TimeZone_Final;
    long      time_start, time_end;
    double   time_overhead;

    /* Checking for command line arguments */
    if( argc != 4 ){

        printf("\t\t Very Few Arguments\n ");
        printf("\t\t Syntax : exec <Threads> <NoOfRows> <NoOfColumns>\n");
        exit(-1);
    }

    Noofthreads=atoi(argv[1]);
    if ((Noofthreads!=1) && (Noofthreads!=2) && (Noofthreads!=4) &&
    (Noofthreads!=8) && (Noofthreads!= 16) )
    {
        printf("\n Number of threads should be 1,2,4,8 or 16 for the execution of
program. \n\n");
        exit(-1);
    }

    NoofRows=atoi(argv[2]);
    NoofCols=atoi(argv[3]);
/*   printf("\n\t Read The Matrix Size Noofrows And Columns Of Matrix \n");
    scanf("%d%d",&NoofRows,&NoofCols);*/
```

```

printf("\n\t\t Threads : %d ",Noofthreads);
printf("\n\t\t Matrix Size : %d X %d \n ",NoofRows,NoofCols);

if (NoofRows <= 0 || NoofCols <= 0) {
    printf("\n\t\t The NoofRows And NoofCols Should Be Of Positive
Sign\n");
    exit(1);
}
/* Matrix Elements */
Matrix = (float **) malloc(sizeof(float *) * NoofRows);
for (i = 0; i < NoofRows; i++) {
    Matrix[i] = (float *) malloc(sizeof(float) * NoofCols);
    for (j = 0; j < NoofCols; j++)
        Matrix[i][j] = (i * j) * 5 + i;
}

/* Dynamic Memory Allocation */
Trans = (float **) malloc(sizeof(float *) * NoofCols);
Checkoutput = (float **) malloc(sizeof(float *) * NoofCols);

/* Initializing The Output Matrices Elements As Zero */
for (i = 0; i < NoofCols; i++) {
    Checkoutput[i] = (float *) malloc(sizeof(float) * NoofRows);
    Trans[i] = (float *) malloc(sizeof(float) * NoofRows);
    for (j = 0; j < NoofRows; j++) {
        Checkoutput[i][j] = 0;
        Trans[i][j] = 0;
    }
}

getttimeofday(&TimeValue_Start, &TimeZone_Start);

omp_set_num_threads(Noofthreads);

/* OpenMP Parallel For Directive */
#pragma omp parallel for private(j)
for (i = 0; i < NoofRows; i = i + 1) {
    Total_threads=omp_get_num_threads();
    for (j = 0; j < NoofCols; j = j + 1) {
        Trans[j][i] = Matrix[i][j];
    }
}

} /* All thread join Master thread */

```

```

gettimeofday(&TimeValue_Final, &TimeZone_Final);

time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
time_overhead = (time_end - time_start)/1000000.0;

/* Serial Computation */
for (i = 0; i < NoofRows; i = i + 1)
    for (j = 0; j < NoofCols; j = j + 1)
        Checkoutput[j][i] = Matrix[i][j];

for (i = 0; i < NoofCols; i = i + 1)
    for (j = 0; j < NoofRows; j = j + 1)
        if (Checkoutput[i][j] == Trans[i][j])
            continue;
        else {
            printf("There Is A Difference From Serial And Parallel
Calculation \n");
            exit(-1);
        }

/*
printf("The Input Matrix Is \n");
for (i = 0; i < NoofRows; i++) {
    for (j = 0; j < NoofCols; j++)
        printf("%f \t", Matrix[i][j]);
    printf("\n");
}

printf("\nThe Transpose Matrix Is \n");
for (i = 0; i < NoofCols; i = i + 1) {
    for (j = 0; j < NoofRows; j = j + 1)
        printf("%f \t", Trans[i][j]);
    printf("\n");
}*/



/* Calculation Of Flops */
/* flops = (float) 2 *NoofRows * NoofCols / (float) time_overhead; */
/*printf("Time Taken :%lf \n Flops= %f Flops\n", time_overhead, flops); */

printf("\n\n\tTranspose of the matrix is ..... Done\n");
printf("The Input Matrix Is \n");
for (i = 0; i < NoofRows; i++) {

```

```
for (j = 0; j < NoofCols; j++)  
    printf("%f \t", Matrix[i][j]);  
    printf("\n");  
}  
  
printf("\nThe Transpose Matrix Is \n");  
for (i = 0; i < NoofCols; i = i + 1) {  
    for (j = 0; j < NoofRows; j = j + 1)  
        printf("%f \t", Trans[i][j]);  
    printf("\n");  
}  
  
printf("\n\n\t Time in Seconds (T) : %lf", time_overhead);  
printf("\n\n\t ( T represents the Time taken for computation )");  
printf("\n\t.....\n");  
  
/* Freeing Allocated Memory */  
free(Matrix);  
free(Checkoutput);  
free(Trans);  
}
```

OUTPUT:

```
anujshukla@18bce0163:~$ gedit transpose.c
anujshukla@18bce0163:~$ gcc transpose.c -fopenmp -o transpose
anujshukla@18bce0163:~$ ./transpose 4 3 4

        Threads : 4
        Matrix Size : 3 X 4

        Transpose of the matrix is ..... Done
The Input Matrix Is
0.000000      0.000000      0.000000      0.000000
1.000000      6.000000     11.000000     16.000000
2.000000     12.000000     22.000000     32.000000

The Transpose Matrix Is
0.000000      1.000000      2.000000
0.000000      6.000000     12.000000
0.000000     11.000000     22.000000
0.000000     16.000000     32.000000

        Time in Seconds (T) : 0.000230
        ( T represents the Time taken for computation )
.....
```



DIGITAL ASSIGNMENT 3

Parallel and Distributed Computing

CSE 4001

L13+L14

Anuj Shukla

18BCE0163

CODE:

```

# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# include <omp.h>

# define NV 6

int main ( int argc, char **argv );
int *dijkstra_distance ( int ohd[NV][NV] );
void find_nearest ( int s, int e, int mind[NV], int connected[NV], int *d,
    int *v );
void init ( int ohd[NV][NV] );
void timestamp ( void );
void update_mind ( int s, int e, int mv, int connected[NV], int ohd[NV][NV], int mind[NV] );
int main ( int argc, char **argv )
{
    int i;
    int i4_huge = 2147483647;
    int j;
    int *mind;
    int ohd[NV][NV];
    timestamp ( );
    fprintf ( stdout, "\n" );
    fprintf ( stdout, "DIJKSTRA_OPENMP \n" );
    fprintf ( stdout, " C version\n" );
    fprintf ( stdout, " Use Dijkstra's algorithm to determine the minimum" );
    fprintf ( stdout, " distance from node 0 to each node in a graph," );
    fprintf ( stdout, " given the distances between each pair of nodes.\n" );
    fprintf ( stdout, "\n" );
    fprintf ( stdout, " Although a very small example is considered, we" );
    fprintf ( stdout, " demonstrate the use of OpenMP directives for" );
    fprintf ( stdout, " parallel execution.\n" );
    printf("18BCE0163_Anuj_Shukla");
/*
    Initialize the problem data.
*/
    init ( ohd );
/*

```

```

Print the distance matrix.

*/
fprintf ( stdout, "\n" );
fprintf ( stdout, " Distance matrix:\n" );
fprintf ( stdout, "\n" );
for ( i = 0; i < NV; i++ )
{
    for ( j = 0; j < NV; j++ )
    {
        if ( ohd[i][j] == i4_huge )
        {
            fprintf ( stdout, " Inf" );
        }
        else
        {
            fprintf ( stdout, " %3d", ohd[i][j] );
        }
    }
    fprintf ( stdout, "\n" );
}
/*
Carry out the algorithm.

*/
mind = dijkstra_distance ( ohd );
/*
Print the results.

*/
fprintf ( stdout, "\n" );
fprintf ( stdout, " Minimum distances from node 0:\n" );
fprintf ( stdout, "\n" );
for ( i = 0; i < NV; i++ )
{
    printf ( stdout, " %2d %2d\n", i, mind[i] );
}
/*
Free memory.

*/
free ( mind );
/*
Terminate.

*/
fprintf ( stdout, "\n" );
fprintf ( stdout, "DIJKSTRA_OPENMP\n" );

```

```

fprintf ( stdout, " Normal end of execution.\n" );

fprintf ( stdout, "\n" );
timestamp ( );

return 0;
}

int *dijkstra_distance ( int ohd[NV][NV] )
{
    int *connected;
    int i;
    int i4_huge = 2147483647;
    int md;
    int *mind;
    int mv;
    int my_first;
    int my_id;
    int my_last;
    int my_md;
    int my_mv;
    int my_step;
    int nth;
/*
    Start out with only node 0 connected to the tree.
*/
    connected = ( int * ) malloc ( NV * sizeof ( int ) );

    connected[0] = 1;
    for ( i = 1; i < NV; i++ )
    {
        connected[i] = 0;
    }
/*
    Initial estimate of minimum distance is the 1-step distance.
*/
    mind = ( int * ) malloc ( NV * sizeof ( int ) );

    for ( i = 0; i < NV; i++ )
    {
        mind[i] = ohd[0][i];
    }
/*

```

```

Begin the parallel region.
*/
# pragma omp parallel private ( my_first, my_id, my_last, my_md, my_mv, my_step ) \
shared ( connected, md, mind, mv, nth, ohd )
{
    my_id = omp_get_thread_num ( );
    nth = omp_get_num_threads ( );
    my_first = ( my_id * NV ) / nth;
    my_last = ( ( my_id + 1 ) * NV ) / nth - 1;
/*
The SINGLE directive means that the block is to be executed by only
one thread, and that thread will be whichever one gets here first.
*/
    # pragma omp single
    {
        printf ( "\n" );
        printf ( " P%d: Parallel region begins with %d threads\n", my_id, nth );
        printf ( "\n" );
    }
    printf ( stdout, " P%d: First=%d Last=%d\n", my_id, my_first, my_last );

    for ( my_step = 1; my_step < NV; my_step++ )
    {
/*
Before we compare the results of each thread, set the shared variable
MD to a big value. Only one thread needs to do this.
*/
        # pragma omp single
        {
            md = i4_huge;
            mv = -1;
        }
/*
Each thread finds the nearest unconnected node in its part of the graph.
Some threads might have no unconnected nodes left.
*/
        find_nearest ( my_first, my_last, mind, connected, &my_md, &my_mv );
/*
In order to determine the minimum of all the MY_MD's, we must insist
that only one thread at a time execute this block!
*/
        # pragma omp critical
        {

```

```

if ( my_md < md )
{
    md = my_md;
    mv = my_mv;
}
}

# pragma omp barrier
# pragma omp single
{
    if ( mv != - 1 )
    {
        connected[mv] = 1;
        printf ( " P%d: Connecting node %d.\n", my_id, mv );
    }
}

# pragma omp barrier
if ( mv != -1 )
{
    update_mind ( my_first, my_last, mv, connected, ohd, mind );
}

#pragma omp barrier
}

# pragma omp single
{
    printf ( "\n" );
    printf ( " P%d: Exiting parallel region.\n", my_id );
}
}

free ( connected );

return mind;
}

void find_nearest ( int s, int e, int mind[NV], int connected[NV], int *d,int *v )
{
    int i;
    int i4_huge = 2147483647;

```

```

*d = i4_huge;
*v = -1;

for ( i = s; i <= e; i++ )
{
    if ( !connected[i] && ( mind[i] < *d ) )
    {
        *d = mind[i];
        *v = i;
    }
}
return;
}

void init ( int ohd[NV][NV] )
{
    int i;
    int i4_huge = 2147483647;
    int j;

    for ( i = 0; i < NV; i++ )
    {
        for ( j = 0; j < NV; j++ )
        {
            if ( i == j )
            {
                ohd[i][i] = 0;
            }
            else
            {
                ohd[i][j] = i4_huge;
            }
        }
    }
    ohd[0][1] = ohd[1][0] = 4;
    ohd[0][2] = ohd[2][0] = 15;
    ohd[1][2] = ohd[2][1] = 40;
    ohd[1][3] = ohd[3][1] = 90;
    ohd[1][4] = ohd[4][1] = 75;
    ohd[2][3] = ohd[3][2] = 100;
    ohd[1][5] = ohd[5][1] = 6;
    ohd[4][5] = ohd[5][4] = 8;
}

```

```

    return;
}
void timestamp ( void )
{
#define TIME_SIZE 40

    static char time_buffer[TIME_SIZE];
    const struct tm *tm;
    time_t now;

    now = time ( NULL );
    tm = localtime ( &now );

    strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

    printf ( "%s\n", time_buffer );

    return;
#undef TIME_SIZE
}

void update_mind ( int s, int e, int mv, int connected[NV], int ohd[NV][NV],
    int mind[NV] )
{
    int i;
    int i4_huge = 2147483647;

    for ( i = s; i <= e; i++ )
    {
        if ( !connected[i] )
        {
            if ( ohd[mv][i] < i4_huge )
            {
                if ( mind[mv] + ohd[mv][i] < mind[i] )
                {
                    mind[i] = mind[mv] + ohd[mv][i];
                }
            }
        }
    }
    return;
}

```

OUTPUT:

```

C:\Users\ace\Desktop\PC1\LAB\Dijsktra.exe
26 August 2020 02:35:05 PM

DIJKSTRA_OPENMP
C version
Use Dijkstra's algorithm to determine the minimum distance from node 0 to each node in a graph, given the distances between each pair of nodes.

Although a very small example is considered, we demonstrate the use of OpenMP directives for parallel execution.
188CE0163 Anuj_Shukla
Distance matrix:

 0   4   15  Inf  Inf  Inf
 4   0   40   90   75   6
15   40   0   100  Inf  Inf
Inf  90   100  Inf  Inf  Inf
Inf  75  Inf  Inf   0   8
Inf  0  Inf  Inf   8   0

P1: Parallel region begins with 8 threads
P3: First=2  Last=2
P1: First=0  Last=0
P0: First=0  Last=1
P6: First=4  Last=4
P7: First=5  Last=5
P2: First=1  Last=1
P4: First=3  Last=2
P5: First=4  Last=3
P3: Connecting node 1.
P5: Connecting node 5.
P6: Connecting node 2.
P6: Connecting node 4.
P9: Connecting node 3.

P4: Exiting parallel region.

Minimum distances from node 0:

 0   0
 1   4
 2   15
 3   94
 4   10
 5   10

DIJKSTRA_OPENMP
Normal end of execution.

26 August 2020 02:35:05 PM
Process returned 0 (0x0)  execution time : 0.070 s

```



DIGITAL ASSIGNMENT 4

Parallel and Distributed Computing

CSE 4001

L13+L14

Anuj Shukla

18BCE0163

1. Loop Work Sharing

CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100
int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);

#pragma omp for schedule(dynamic,chunk)
    for (i=0; i<N; i++)
    {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }
} /* end of parallel section */

}
```

OUTPUT:

```
18BCE0163
18BCE0163
Thread 10 starting...
Thread 10: c[0]= 0.000000
Thread 10: c[1]= 2.000000
Thread 10: c[2]= 4.000000
Thread 10: c[3]= 6.000000
Thread 10: c[4]= 8.000000
Thread 10: c[5]= 10.000000
Thread 10: c[6]= 12.000000
Thread 10: c[7]= 14.000000
Thread 10: c[8]= 16.000000
Thread 10: c[9]= 18.000000
Thread 10: c[10]= 20.000000
18BCE0163
Thread 6 starting...
Thread 6: c[20]= 40.000000
Thread 6: c[21]= 42.000000
Thread 6: c[22]= 44.000000
Thread 6: c[23]= 46.000000
Thread 6: c[24]= 48.000000
Thread 6: c[25]= 50.000000
18BCE0163
Thread 9 starting...
Thread 9: c[30]= 60.000000
Thread 9: c[31]= 62.000000
Thread 9: c[32]= 64.000000
Thread 9: c[33]= 66.000000
Thread 9: c[34]= 68.000000
Thread 9: c[35]= 70.000000
Thread 9: c[36]= 72.000000
Thread 9: c[37]= 74.000000
Thread 9: c[38]= 76.000000
Thread 9: c[39]= 78.000000
Thread 9: c[40]= 80.000000
Thread 9: c[41]= 82.000000
Thread 9: c[42]= 84.000000
Thread 9: c[43]= 86.000000
Thread 9: c[44]= 88.000000
Thread 9: c[45]= 90.000000
Thread 9: c[46]= 92.000000
Thread 9: c[47]= 94.000000
Thread 9: c[48]= 96.000000
Thread 9: c[49]= 98.000000
Thread 9: c[50]= 100.000000
Thread 9: c[51]= 102.000000
Thread 9: c[52]= 104.000000
Thread 9: c[53]= 106.000000
Thread 9: c[54]= 108.000000
Thread 11 starting...
Thread 11: c[60]= 120.000000
18BCE0163
18BCE0163
18BCE0163
Thread 8 starting...
Thread 8: c[70]= 140.000000
```

```
Thread 8 starting...
Thread 8: c[70]= 140.000000
Thread 8: c[71]= 142.000000
Thread 8: c[72]= 144.000000
Thread 8: c[73]= 146.000000
Thread 8: c[74]= 148.000000
Thread 8: c[75]= 150.000000
Thread 8: c[76]= 152.000000
Thread 8: c[77]= 154.000000
Thread 8: c[78]= 156.000000
Thread 8: c[79]= 158.000000
Thread 8: c[80]= 160.000000
Thread 8: c[81]= 162.000000
Thread 8: c[82]= 164.000000
Thread 8: c[83]= 166.000000
Thread 8: c[84]= 168.000000
Thread 8: c[85]= 170.000000
Thread 8: c[86]= 172.000000
Thread 8: c[87]= 174.000000
Thread 8: c[88]= 176.000000
Thread 8: c[89]= 178.000000
Thread 8: c[90]= 180.000000
Thread 8: c[91]= 182.000000
Thread 8: c[92]= 184.000000
Thread 8: c[93]= 186.000000
Thread 8: c[94]= 188.000000
Thread 8: c[95]= 190.000000
Thread 8: c[96]= 192.000000
Thread 8: c[97]= 194.000000
Thread 8: c[98]= 196.000000
Thread 8: c[99]= 198.000000
18BCE0163
Thread 4 starting...
Thread 6: c[26]= 52.000000
Thread 6: c[27]= 54.000000
Thread 6: c[28]= 56.000000
Thread 6: c[29]= 58.000000
Thread 9: c[55]= 110.000000
Thread 9: c[56]= 112.000000
Thread 9: c[57]= 114.000000
Thread 9: c[58]= 116.000000
Thread 9: c[59]= 118.000000
Thread 2 starting...
18BCE0163
Thread 3 starting...
18BCE0163
Thread 5 starting...
Thread 11: c[61]= 122.000000
Thread 11: c[62]= 124.000000
Thread 11: c[63]= 126.000000
Thread 11: c[64]= 128.000000
Thread 11: c[65]= 130.000000
Thread 11: c[66]= 132.000000
Thread 11: c[67]= 134.000000
Thread 11: c[68]= 136.000000
Thread 11: c[69]= 138.000000
```

```

Thread 5 starting...
Thread 11: c[61]= 122.000000
Thread 11: c[62]= 124.000000
Thread 11: c[63]= 126.000000
Thread 11: c[64]= 128.000000
Thread 11: c[65]= 130.000000
Thread 11: c[66]= 132.000000
Thread 11: c[67]= 134.000000
Thread 11: c[68]= 136.000000
Thread 11: c[69]= 138.000000
Thread 10: c[11]= 22.000000
Thread 10: c[12]= 24.000000
Thread 10: c[13]= 26.000000
Thread 10: c[14]= 28.000000
Thread 10: c[15]= 30.000000
Thread 10: c[16]= 32.000000
Thread 10: c[17]= 34.000000
Thread 10: c[18]= 36.000000
Thread 10: c[19]= 38.000000
18BCE0163
Number of threads = 12
Thread 0 starting...
Thread 1 starting...
18BCE0163
Thread 7 starting...

```

2. Section Work Sharing

CODE:

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 50

int main (int argc, char *argv[])
{
    int i, nthreads, tid;
    float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i<N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
    c[i] = d[i] = 0.0;
}

```

```

#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);

#pragma omp sections nowait
{
    #pragma omp section
    {
        printf("Thread %d doing section 1\n",tid);
        for (i=0; i<N; i++)
        {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
    }

    #pragma omp section
    {
        printf("Thread %d doing section 2\n",tid);
        for (i=0; i<N; i++)
        {
            d[i] = a[i] * b[i];
            printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
        }
    }

} /* end of sections */

printf("Thread %d done.\n",tid);

} /* end of parallel section */
}

```

OUTPUT:

```

Thread 5 starting...
Thread 5 doing section 1
Thread 5: c[1]= 24.850000
Thread 5: c[2]= 27.350000
Thread 8 starting...
Thread 8 doing section 2
Thread 8: d[0]= 0.000000
Thread 7 starting...
Thread 7 done.
188CE0163
Number of threads = 12
Thread 0 starting...
Thread 0 done.
Thread 10 starting...
Thread 2 starting...
Thread 2 done.
188CE0163
Thread 5: c[3]= 29.850000
Thread 5: c[4]= 32.349998
Thread 5: c[5]= 34.849998
Thread 5: c[6]= 37.349998
Thread 5: c[7]= 39.849998
Thread 5: c[8]= 42.349998
Thread 5: c[9]= 45.849998
Thread 5: c[10]= 47.349998
Thread 5: c[11]= 49.849998
Thread 3 starting...
Thread 3 done.
188CE0163
188CE0163
Thread 4 done.
188CE0163
Thread 1 starting...
Thread 1 done.
188CE0163
Thread 8: d[1]= 35.025002
Thread 8: d[2]= 73.050003
Thread 4 starting...
Thread 4 done.
188CE0163
Thread 5: c[12]= 52.349998
Thread 5: c[13]= 54.849998
Thread 5: c[14]= 57.349998
Thread 5: c[15]= 59.849998
Thread 5: c[16]= 62.349998
Thread 5: c[17]= 64.849998
Thread 5: c[18]= 67.349998
Thread 5: c[19]= 69.849998
Thread 5: c[20]= 72.349998
Thread 5: c[21]= 74.849998
Thread 5: c[22]= 77.349998
Thread 5: c[23]= 79.849998
Thread 5: c[24]= 82.349998
Thread 5: c[25]= 84.849998
Thread 5: c[26]= 87.349998

```

```

Thread 5: c[28]= 92.349998
hread 5: c[29]= 94.849998
hread 5: c[30]= 97.349998
hread 5: c[31]= 99.849998
hread 5: c[32]= 102.349998
hread 5: c[33]= 104.849998
hread 5: c[34]= 107.349998
hread 5: c[35]= 109.849998
hread 5: c[36]= 112.349998
hread 5: c[37]= 114.849998
hread 5: c[38]= 117.349998
hread 5: c[39]= 119.849998
hread 5: c[40]= 122.349998
hread 5: c[41]= 124.849998
hread 5: c[42]= 127.349998
hread 5: c[43]= 129.850006
hread 5: c[44]= 132.350006
hread 5: c[45]= 134.850006
hread 5: c[46]= 137.350006
hread 5: c[47]= 139.850006
hread 5: c[48]= 142.350006
hread 5: c[49]= 144.850006
hread 5 done.
.88CE0163
hread 8: d[3]= 114.075005
hread 8: d[4]= 158.100006
hread 8: d[5]= 205.125000
hread 8: d[6]= 255.150009
hread 8: d[7]= 308.175018
hread 8: d[8]= 364.200012
hread 8: d[9]= 423.225086
hread 8: d[10]= 485.249969
hread 8: d[11]= 550.274963
hread 8: d[12]= 618.299988
hread 8: d[13]= 689.324951
hread 8: d[14]= 763.349976
hread 8: d[15]= 840.374939
hread 8: d[16]= 920.399963
hread 8: d[17]= 1003.424988
hread 8: d[18]= 1089.449951
hread 8: d[19]= 1178.474976
hread 8: d[20]= 1270.500000
hread 8: d[21]= 1365.524902
hread 8: d[22]= 1463.549927
hread 8: d[23]= 1564.574951
hread 8: d[24]= 1668.599976
hread 8: d[25]= 1775.625000
hread 8: d[26]= 1885.649982
hread 8: d[27]= 1998.674927
hread 8: d[28]= 2114.699951
hread 8: d[29]= 2233.724854
hread 8: d[30]= 2355.750000
hread 8: d[31]= 2480.774902
hread 8: d[32]= 2608.799805

```

```

Thread 8: d[29]= 2233.724854
Thread 8: d[30]= 2355.750000
Thread 8: d[31]= 2480.774902
Thread 8: d[32]= 2668.799805
Thread 8: d[33]= 2739.824951
Thread 8: d[34]= 2873.849854
Thread 8: d[35]= 3010.875000
Thread 8: d[36]= 3150.899902
Thread 8: d[37]= 3293.924805
Thread 8: d[38]= 3439.949951
Thread 8: d[39]= 3588.974854
Thread 8: d[40]= 3741.000000
Thread 8: d[41]= 3896.024902
Thread 8: d[42]= 4054.049805
Thread 8: d[43]= 4215.074707
Thread 8: d[44]= 4379.100098
Thread 8: d[45]= 4546.125000
Thread 8: d[46]= 4716.149902
Thread 8: d[47]= 4889.174805
Thread 8: d[48]= 5065.199707
Thread 8: d[49]= 5244.225098
Thread 8 done.
18BCE0163
Thread 9 starting...
Thread 9 done.
18BCE0163
Thread 6 starting...
Thread 6 done.
18BCE0163
Thread 11 starting...
Thread 11 done.
18BCE0163

```

3. Combined Parallel Loop Reduction

CODE:

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int i, n;
    float a[100], b[100], sum;

    /* Some initializations */
    n = 100;
    for (i=0; i < n; i++)
        a[i] = b[i] = i * 1.0;
    sum = 0.0;

    #pragma omp parallel for reduction(+:sum)
    for (i=0; i < n; i++)
        sum = sum + (a[i] * b[i]);

    printf(" Sum = %f\n",sum);
}

```

OUTPUT:

```
Sum = 328350.000000
18BCE0163
```

4. Orphaned Parallel Loop Reduction**CODE:**

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define VECLEN 100

float a[VECLEN], b[VECLEN], sum;

float dotprod ()
{
    int i,tid;

    tid = omp_get_thread_num();
    #pragma omp for reduction(+:sum)
    for (i=0; i < VECLEN; i++)
    {
        sum = sum + (a[i]*b[i]);
        printf(" tid= %d i=%d\n",tid,i);
    }
}

int main (int argc, char *argv[])
{
    int i;

    for (i=0; i < VECLEN; i++)
        a[i] = b[i] = 1.0 * i;
    sum = 0.0;

    #pragma omp parallel
    dotprod();

    printf("Sum = %f\n",sum);
}
```

OUTPUT:

```

tid= 2 l=18
tid= 2 l=19
tid= 2 l=20
tid= 2 l=21
tid= 2 l=22
tid= 2 l=23
tid= 2 l=24
tid= 2 l=25
tid= 2 l=26
tid= 10 l=84
tid= 10 l=85
tid= 10 l=86
tid= 10 l=87
tid= 10 l=88
tid= 10 l=89
tid= 10 l=90
tid= 10 l=91
tid= 10 l=92
tid= 7 l=61
tid= 7 l=62
tid= 7 l=63
tid= 7 l=64
tid= 7 l=65
tid= 1 l=9
tid= 1 l=10
tid= 1 l=11
tid= 1 l=12
tid= 1 l=13
tid= 1 l=14
tid= 1 l=15
tid= 1 l=16
tid= 1 l=17
tid= 9 l=76
tid= 9 l=77
tid= 4 l=36
tid= 4 l=37
tid= 4 l=38
tid= 4 l=39
tid= 4 l=40
tid= 4 l=41
tid= 4 l=42
tid= 4 l=43
tid= 6 l=52
tid= 6 l=53
tid= 6 l=54
tid= 6 l=55
tid= 6 l=56
tid= 6 l=57
tid= 6 l=58
tid= 6 l=59
tid= 7 l=66
tid= 7 l=67
tid= 8 l=68
tid= 8 l=69
tid= 8 l=70
tid= 8 l=71
tid= 8 l=72
tid= 8 l=73
tid= 8 l=74
tid= 8 l=75
tid= 0 l=0
tid= 0 l=1
tid= 0 l=2
tid= 9 l=82
tid= 3 l=25
tid= 3 l=27
tid= 3 l=28
tid= 3 l=29
tid= 9 l=83
tid= 11 l=92
tid= 11 l=93
tid= 11 l=94
tid= 11 l=95
tid= 11 l=96
tid= 11 l=97
tid= 11 l=98
tid= 11 l=99
tid= 3 l=30
tid= 3 l=31
tid= 3 l=32
tid= 3 l=33
tid= 3 l=34
tid= 3 l=35
tid= 0 l=3
tid= 0 l=4
tid= 0 l=5
tid= 0 l=6
tid= 0 l=7
tid= 0 l=8
tid= 5 l=46
tid= 5 l=47
tid= 5 l=48
tid= 5 l=49
tid= 5 l=50
tid= 5 l=51
SUM = 328350.000000
1BBC0163

```



DIGITAL ASSIGNMENT 4

Parallel and Distributed Computing

CSE 4001

L13+L14

Anuj Shukla

18BCE0163

1. MPI Program to find the sum of n numbers using point to point blocking communication

Code -

```
#include<iostream>
#include<unistd.h>
#include<mpi.h>
using namespace std;
int main(int argc,char *argv[])
{
    int root=0,myrank,numprocs,source,destination,iproc;
    int dest_tag,source_tag;
    int sum=0,value=0;
    MPI::Status status;

    /*.... Initializing MPI .....

    MPI::Init(argc,argv);
    numprocs=MPI::COMM_WORLD.Get_size();
    myrank=MPI::COMM_WORLD.Get_rank();
    if(myrank != root)
    {
        destination = 0;
        dest_tag = 0;
        MPI::COMM_WORLD.Send(&myrank,1,MPI::INT,destination,dest
        _tag);
    }
    else
    {
        for(iproc = 1;iproc < numprocs;iproc++)
        {
            source = iproc;
            source_tag = 0;
            MPI::COMM_WORLD.Recv(&value,1,MPI::INT,source,sour
            ce_tag,status)
            ;
            sum =sum + value;
        }
        cout<<" \n MyRank :: "<<myrank <<" Sum of first " <<
        numprocs<< " Integers is "<<sum <<"\n";
    }
}
```

```

/*.....Finalizing MPI ....*/
MPI::Finalize();
return 0;
}

```

OUTPUT:

```

cisco@ubuntu:~$ gedit point.cpp
cisco@ubuntu:~$ mpicxx -o point point.cpp
cisco@ubuntu:~$ mpirun -np 4 ./point

MyRank :: 0 Sum of first 4 Integers is 6
cisco@ubuntu:~$ //Anuj Shukla 18BCE0163

```

2. OpenMP program for managing shared data

CODE:

```

#include<stdio.h>
#include<omp.h>
#include<stdlib.h>
/* Main Program */
void main(int argc , char **argv)
{
    double *Array, *Check, serial_sum,sum_private,
    sum,my_sum;
    int array_size, i,threadid,tval,Noofthreads;
    printf("\tManaging Shared data, by finding the sum of
elements in a one dimensional array.\n\n");
    /* Checking for command line arguments */
    if( argc != 3 ){
        printf("\t\t Very Few Arguments\n ");
        printf("\t\t Syntax : exec <Threads> <array-size>
\n");
        exit(-1);
    }
    Noofthreads=atoi(argv[1]);
    if ((Noofthreads!=1) && (Noofthreads!=2) &&
(Noofthreads!=4) && (Noofthreads!=8) && (Noofthreads!= 16) ) {
        printf("\n Number of threads should be 1,2,4,8 or
16 for the execution of program. \n\n");
        exit(-1);
    }
}

```

```

}

array_size=atoi(argv[2]);
if (array_size <= 0) {
printf("\n\t\t Array Size Should Be Of Positive Value ");
exit(1);
}
printf("\n\t\t Threads : %d ",Noofthreads);
printf("\n\t\t Array Size : %d ",array_size);
/* Dynamic Memory Allocation */
Array = (double *) malloc(sizeof(double) * array_size);
Check = (double *) malloc(sizeof(double) * array_size);
/* Array Elements Initialization */
for (i = 0; i < array_size; i++) {
Array[i] = i * 5;
Check[i] = Array[i];
}
sum=0.0;
/* set the number of threads */
omp_set_num_threads(Noofthreads);
/* OpenMP Parallel For Directive And Critical Section */
#pragma omp parallel shared(sum)
{
my_sum=0.0;
#pragma omp for
for (i = 0; i < array_size; i++)
my_sum = my_sum + Array[i]; /* Data races
occur */
#pragma omp critical
sum = sum + my_sum;
} /* End of parallel region */
sum_private = 0;
/* OpenMP Parallel For Directive And Private Clause Critical
Section */
#pragma omp parallel private(my_sum) shared(sum)
{
my_sum=0.0;
#pragma omp for
for (i = 0; i < array_size; i++)
my_sum = my_sum + Array[i];
#pragma omp critical

```

```

        sum_private = sum_private + my_sum;
    } /* End of parallel region */
serial_sum = 0.0;
/* Serial Calculation */
for (i = 0; i < array_size; i++)
    serial_sum = serial_sum + Check[i];
printf("\n\n\tThe computation of the sum of the
elements of the Array ..... Done ");
/* Freeing Memory */
free(Check);
free(Array);
printf("\n\n\tThe SumOfElements Of The Array Using OpenMP
Directives without Private Clause : %lf", sum);
printf("\n\n\tThe SumOfElements Of The Array Using OpenMP
Directives and Private Clause Is : %lf", sum_private);
printf("\n\n\tThe SumOfElements Of The Array By Serial
Calculation Is : %lf\n\n", serial_sum);
printf("\n\t.....\n");
}

```

OUTPUT:

```

cisco@ubuntu:~$ gedit shared.c
cisco@ubuntu:~$ gcc -fopenmp shared.c
cisco@ubuntu:~$ ./a.out 4 12
        Managing Shared data, by finding the sum of elements in a one dimensional
array.

        Threads : 4
        Array Size : 12

        The computation of the sum of the elements of the Array ..... Done

        The SumOfElements Of The Array Using OpenMP Directives without Private C
lause : 240.000000

        The SumOfElements Of The Array Using OpenMP Directives and Private Claus
e Is : 330.000000

        The SumOfElements Of The Array By Serial Calculation Is : 330.000000

.....
cisco@ubuntu:~$ //Anuj Shukla 18BCE0163

```

Assessment 1

18BCE0574

1.

Code:

```
#include <stdio.h>
#include <omp.h>
void multiply () {
}
int main () {
    printf ( "Maitrish Jain\n18BCE0574\n\n" );
    float x1 [ 3 ] = {5,3,1};
    float x2 [ 3 ] = {};
    float matrix [ 3 ][ 3 ] = {{8,7,9},{2,2,2},{7,8,2}};
    int itr = 3 ;

    #pragma omp parallel
    {
        int i , j ;
        #pragma omp for
        for ( i = 0 ; i < itr ; i ++ ) {
            for ( j = 0 ; j < itr ; j ++ ) {
                x2 [ i ] += matrix [ i ][ j ] * x1 [ j ]; } } }
        for ( int i = 0 ; i < 3 ; i ++ ) printf ( " %f \n " , x2 [ i ]);
    }
```

The screenshot shows the CodeBlocks IDE interface. The main window displays the source code for 'main.c'. The code includes a function 'void multiply()' and its implementation. It uses OpenMP pragmas to parallelize loops. The code is as follows:

```
#include <omp.h>
void multiply () {
}

int main () {
    printf ("Maitrish Jain(18BCE0574)\n");
    float x1 [ 3 ] = { 1, 2, 1 };
    float x2 [ 3 ] = { 1 };
    float matrix [ 3 ][ 3 ] = { { 1, 2, 3 }, { 2, 3, 1 }, { 3, 2, 1 } };
    int ite = 3;

    #pragma omp parallel
    {
        int i , j ;
        #pragma omp for
        for ( i = 0 ; i < ite ; i ++ ) {
            for ( j = 0 ; j < 3 ; j ++ ) {
                x2 [ j ] = matrix [ i ][ j ] * x1 [ j ];
            }
        }
        for ( int i = 0 ; i < ite ; i ++ ) {
            printf ("%d %d %d\n", x1 [ i ], matrix [ i ][ 0 ], matrix [ i ][ 1 ]);
        }
    }
}
```

The screenshot shows a terminal window titled "Terminal". It displays the output of the program 'main.c'. The output includes the author's name and ID, followed by three floating-point numbers: 70.000000, 18.000000, and 61.000000. The terminal also shows the process return value and execution time.

```
"M:\SEM5\PC\LAB\def\11\main\Debug\11.exe"
Maitrish Jain
18BCE0574

70.000000
18.000000
61.000000

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

2.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define MAX1 500000
clock_t t ;
double cpu_time_used ;
int linearSearch ( int * A , int n , int tos );
int main () {
```

```

printf ( "18BCE0574\nMaitrish Jain\n" );

int number , iter = 0 , find ; int * Arr ;

Arr = ( int * ) malloc ( number * sizeof ( int ) );

scanf ( " %d " , & number );

for ( ; iter < number ; iter ++ )

{

    scanf ( " %d " , & Arr [ iter ]);

}

scanf ( " %d " , & find );

printf ( " \n To find: %d \n " , find );

t = clock ();

int indexx = linearSearch ( Arr , number , find );

t = clock () - t ;

if ( indexx == - 1 )

{

    printf ( "Not found" );

}

else

printf ( "Found at %d \n " , indexx );

cpu_time_used = (( double ) t ) / CLOCKS_PER_SEC ;

printf ( " \n Time taken for search: %f " , cpu_time_used );

binary ();

return 0 ;

}

int linearSearch ( int * A , int n , int tos )

{

int found_at = - 1 ;

#pragma omp parallel for

for ( int iter = 0 ; iter < n ; iter ++ )

{

    if ( A [ iter ] == tos )

```

```

    found_at = iter + 1 ;

}

return found_at ;
}

int binary_search ( int arr [] , int x , int low , int high )

{
    while ( low <= high )

{
    int mid = low + ( high - low ) / 2 ;

    if ( arr [ mid ] == x )

        return mid ;

    if ( arr [ mid ] < x )

        low = mid + 1 ;

    else

        high = mid - 1 ;

}
    return - 1 ;
}

void binary ()

{
    int array [ MAX1 ];

    int loc , loc1 , loc2 , loc3 , loc4 , k ;

    omp_set_num_threads ( 4 );

    for ( int i = 0 ; i < MAX1 ; ++ i )

        array [ i ] = rand () % 1000 ;

    printf ( "Enter number to search: \n " );

    scanf ( "%d " , & k );

    int mid = MAX1 / 2 ;

    double start_time = omp_get_wtime ();

    int one_quart = mid / 2 ;

```

```

int three_quart = 3 * one_quart ; start_time = omp_get_wtime ();

#pragma omp parallel sections

{
    #pragma omp section

    loc1 = binary_search ( array , k , 0 , one_quart );

    #pragma omp section

    loc2 = binary_search ( array , k , one_quart + 1 , mid );

    #pragma omp section

    loc3 = binary_search ( array , k , mid + 1 , three_quart );

    #pragma omp section

    loc4 = binary_search ( array , k , three_quart + 1 , MAX1 - 1 );

}

double end_time = omp_get_wtime () - start_time ;

printf ( "Binary Search time: %f \n " , end_time );

}

```

```

MSYS64PPC48-darwin17.7.0/bin/Debug/t.exe"
18BCE0574
Maitrish Jain
5
1 4 6 8 3
4
4

To find: 4
Found at 2

Time taken for search: 0.000000 Enter number to search:
-
```

3.

Code:

```

#include <stdlib.h>

#include <stdio.h>

#include <omp.h>

int main ( int argc, char *argv[] );

void prime_number_sweep ( int n_lo, int n_hi, int n_factor );

```

```

int prime_number ( int n );

int main ( int argc, char *argv[] ) {

    int n_factor; int n_hi; int n_lo;

    printf("18BCE0574\nMaitrish Jain\n\n");

    printf ( "Processors available = %d\n",
            omp_get_num_procs () );

    printf ( " Number of threads =%d\n",omp_get_max_threads ( ) );

    n_lo = 1;

    n_hi = 131072;

    n_factor = 2;

    prime_number_sweep ( n_lo, n_hi, n_factor );

    return 0;

}

void prime_number_sweep ( int n_lo, int n_hi, int n_factor )

{

    int n;

    int primes;

    double wtime;

    printf ( "Count Prime 1 to N\n" );

    printf ( "\n" );

    printf ( " Prime Numbers\t\t Time\n" );

    printf ( "\n" );

    n = n_lo;while ( n <= n_hi )

    {

        wtime = omp_get_wtime ( );

        primes = prime_number ( n );

        wtime = omp_get_wtime ( ) - wtime;

        printf ( " %8d %8d %14f\n", n, primes, wtime );

        n = n * n_factor;

    }

    return;

```

```
}
```

```
int prime_number ( int n )
```

```
{
```

```
int i;
```

```
int j;int prime;
```

```
int total = 0;
```

```
# pragma omp parallel \
```

```
shared ( n ) \
```

```
private ( i, j, prime )
```

```
# pragma omp for reduction ( + : total )
```

```
for( i = 2; i <= n; i++ )
```

```
{
```

```
prime = 1;
```

```
for ( j = 2; j < i; j++ )
```

```
{
```

```
if ( i % j == 0 )
```

```
{
```

```
prime = 0;
```

```
break;}
```

```
}total = total + prime; } return total;}
```

main.c [1] - Code::Blocks 20.03

```
#include <omp.h>
#include <stdio.h>
#include <math.h>

int main ( int argc, char *argv[] )
{
    void prime_number_sweep ( int n_lo, int n_hi, int n_factor );
    int prime_numbers ( int n );

    int main ( int argc, char *argv[] )
    {
        int n_factor; int n_hi; int n_lo;
        printf ("Welcome to Parallel Sieve\n");
        printf ("Processors available = %d\n",
            omp_get_num_procs () );
        printf ("Number of threads = %d\n", omp_get_max_threads () );
        n_lo = 1;
        n_hi = 1000000;
        n_factor = 2;
        prime_number_sweep ( n_lo, n_hi, n_factor );
        return 0;
    }

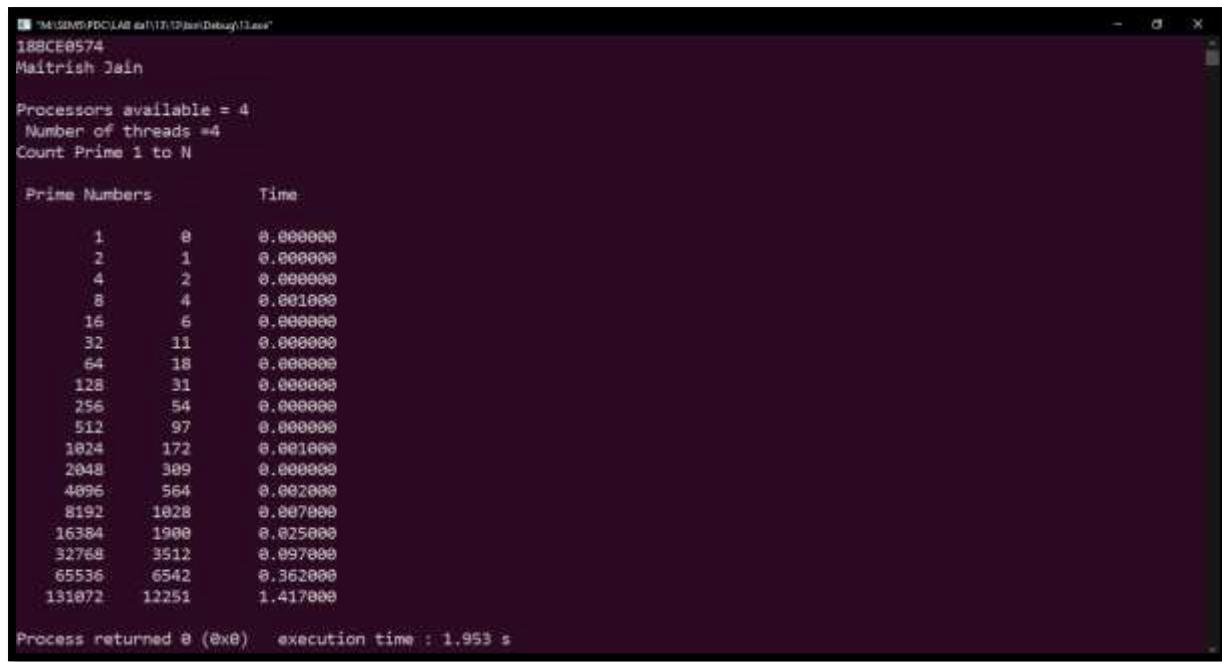
    void prime_number_sweep ( int n_lo, int n_hi, int n_factor )
    {
        int n;
        int primes;
        double utime;
        printf ("Count Prime L to Hn\n");
        printf ("%n\n");
        printf ("Prime Number with Time:n\n");
        printf ("%n\n");
        n = n_lo; while ( n <= n_hi )
        {
            if ( n % n_factor == 0 )
                n++;
            else
                primes++;
            n++;
        }
        utime = omp_get_wtime ();
        primes = prime_numbers ( n );
        utime = omp_get_wtime () - utime;
        printf ("Total %d primes, Utme: %f\n", n, primes, utime );
        n = n / n_factor;
    }
}
```

main.c [1] - Code::Blocks 20.03

```
    return;

    int prime_numbers ( int n )
    {
        int n;
        int prime;
        int total = 0;
        #pragma omp parallel
        n = n / n_factor;
        prime = 2;
        total = 1;
        prime_numbers ( prime );
        #pragma omp for reduction ( + : total )
        for ( prime = 3; prime < n; prime += 2 )
        {
            if ( prime % 2 == 0 )
                prime += 1;
            for ( int i = 2; i < prime; i++ )
            {
                if ( prime % i == 0 )
                    break;
            }
            prime = prime + 1;
            if ( prime == n )
                break;
            total = total + prime;
        }
        return total;
    }

    int prime_number ( int prime )
    {
        int n;
        int total = 0;
        for ( n = 1; n < prime; n++ )
        {
            if ( prime % n == 0 )
                break;
        }
        if ( prime == n )
            total = 1;
        return total;
    }
}
```



```
M:\SEM5\PPC\Lab 6\17\Prime\Debug\17.exe
18BCE0574
Maitrish Jain

Processors available = 4
Number of threads =4
Count Prime 1 to N

Prime Numbers           Time

 1      0      0.000000
 2      1      0.000000
 4      2      0.000000
 8      4      0.001000
16      6      0.000000
32     11      0.000000
64     18      0.000000
128    31      0.000000
256    54      0.000000
512    97      0.000000
1024   172      0.001000
2048   309      0.000000
4096   564      0.002000
8192   1028      0.007000
16384  1900      0.025000
32768  3512      0.097000
65536  6542      0.362000
131072 12251      1.417000

Process returned 0 (0x0)  execution time : 1.953 s
```

4.

Code:

```
#include<omp.h>
#include<stdio.h>

#define V_Size 100
#define C_Size 1

int main() {
    printf("18BCE0574\nMaitrish Jain\n");
    int numt, tid, nothrd, sum = 0, i;
    int chunk = C_Size;
    int A[V_Size], B[V_Size];

#pragma omp parallel shared(A,B,chunk) private(i)
{
#pragma omp parallel for schedule(static, chunk)
for(i = 0; i < V_Size; i++) {A[i] = i;
}
#pragma omp parallel for schedule(static, chunk)
```

```
for(i = 0; i < V_Size; i++)
{
    B[i] = i;
}
}

printf("\n\n\t\tVector Number1\n\n");
for(i=0 ; i < V_Size ; i++)
{
    printf("%d ",A[i]);
}
printf("\n\n\t\tVector Number2\n\n");
for(i=0 ; i < V_Size ; i++)
{
    printf("%d ",B[i]);
}

#pragma omp parallel for default(shared) reduction(+: sum)
for(i = 0; i < V_Size; i++)
{
    sum += A[i]*B[i];
}
printf("\n\nThe dot product of vector A and B is %d", sum);
return 0;
```

```
main.c [14] - CodeBlocks 20.03
File Edit View Search Project Build Debug Fuzzy search Task Task+ Plugins DayBlocks Settings Help
global main.c
Management: Projects File Symbols
1 #include<omp.h>
2 #include<math.h>
3 #define V_Size 100
4 #define C_Size 2
5
6 int main()
7 {
8     printf("18DC03574@Metin: ~\n");
9     int sum, tid, nthreads, sum = 0, i;
10    int chunk = C_Size;
11    int A[V_Size], B[V_Size];
12
13    #pragma omp parallel shared(A,B,chunk) private(i)
14    {
15        #pragma omp parallel for schedule(static,chunk)
16        for(i = 0; i < V_Size; i++) {A[i] = i;}
17
18        #pragma omp parallel for schedule(static,chunk)
19        for(i = 0; i < V_Size; i++)
20        {
21            B[i] = i;
22        }
23
24        printf("\n\nVector A:\n");
25        for(i=0 ; i < V_Size ; i++)
26        {
27            printf("%d ",A[i]);
28        }
29        printf("\n\nVector B:\n");
30        for(i=0 ; i < V_Size ; i++)
31        {
32            printf("%d ",B[i]);
33        }
34
35        #pragma omp parallel for reduction(+:sum)
36        for(i = 0; i < V_Size; i++)
37        {
38            sum += A[i]*B[i];
39        }
40
41        printf("\n\nThe dot product of vector A and B is %d", sum);
42        return 0;
43 }
```

Logs & other

M:\SEM5\PPC\LAB\da7\14\main.c C/C++ Windows (CR+LF) : WINDOWS-1252 Line 34, Col 25 Pos 365 Insert Read/Write default

```
main.c [14] - CodeBlocks 20.03
File Edit View Search Project Build Debug Fuzzy search Task Task+ Plugins DayBlocks Settings Help
global main.c
Management: Projects File Symbols
1 #include<omp.h>
2 #include<math.h>
3 #define V_Size 100
4 #define C_Size 2
5
6 int main()
7 {
8     #pragma omp parallel shared(A,B,chunk) private(i)
9    {
10        #pragma omp parallel for schedule(static,chunk)
11        for(i = 0; i < V_Size; i++) {A[i] = i;}
12
13        #pragma omp parallel for schedule(static,chunk)
14        for(i = 0; i < V_Size; i++)
15        {
16            B[i] = i;
17        }
18
19        printf("\n\nVector A:\n");
20        for(i=0 ; i < V_Size ; i++)
21        {
22            printf("%d ",A[i]);
23        }
24        printf("\n\nVector B:\n");
25        for(i=0 ; i < V_Size ; i++)
26        {
27            printf("%d ",B[i]);
28        }
29
30        #pragma omp parallel for reduction(+:sum)
31        for(i = 0; i < V_Size; i++)
32        {
33            sum += A[i]*B[i];
34        }
35
36        printf("\n\nThe dot product of vector A and B is %d", sum);
37        return 0;
38 }
```

Logs & other

M:\SEM5\PPC\LAB\da7\14\main.c C/C++ Windows (CR+LF) : WINDOWS-1252 Line 34, Col 25 Pos 365 Insert Read/Write default

```
"M:\SEM1\PCOL48\Lab1\14\Debug\14.exe"
18BCE0574
Maitrish Jain

Vector Number1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 4
3 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99

Vector Number2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 4
3 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99

The dot product of vector A and B is 328350
Process returned 0 (0x0)   execution time : 0.058 s
Press any key to continue.
```

Assesment 2

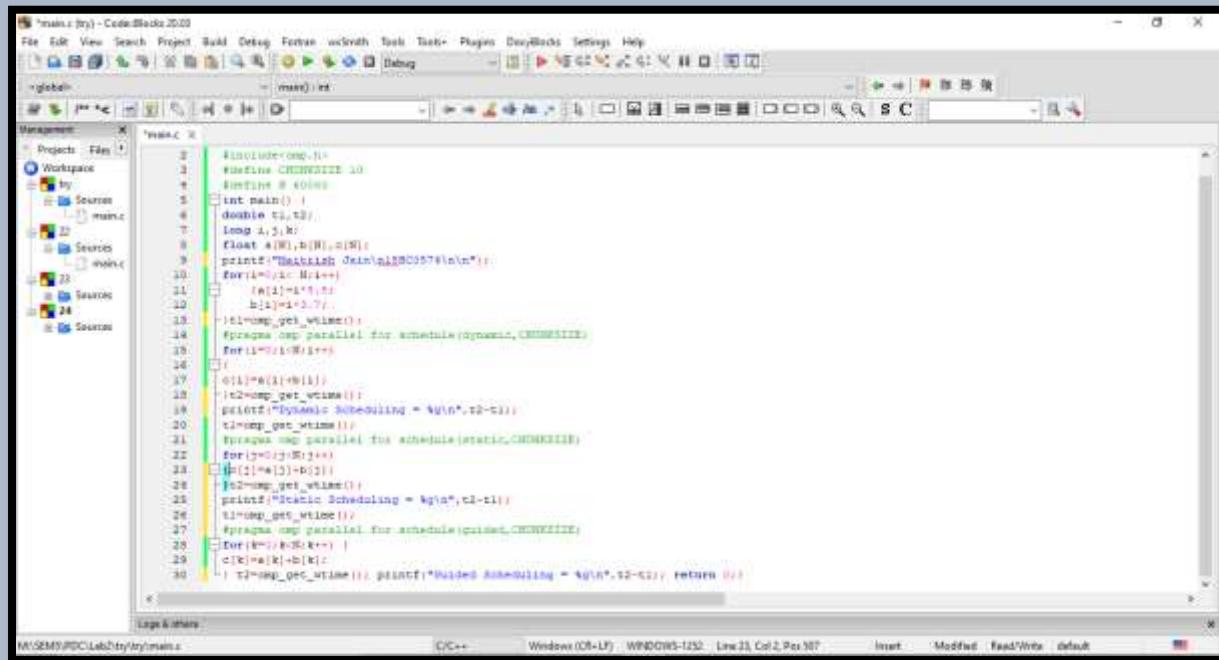
18BCE0574

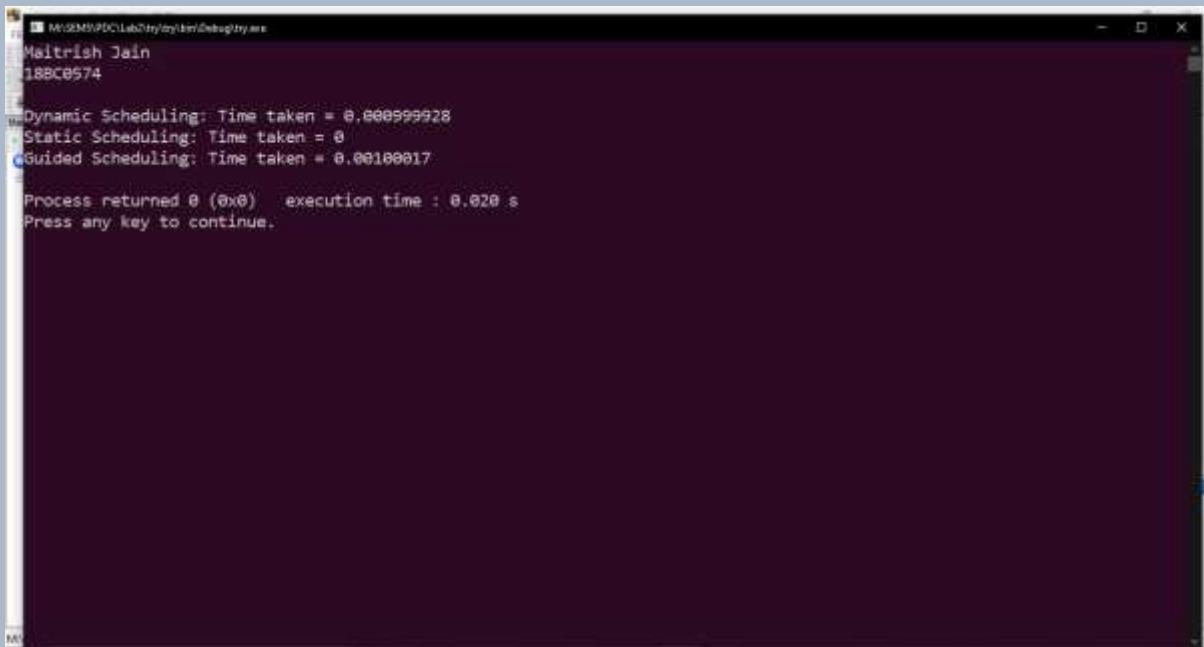
1.

Code:

```
#include<stdio.h>
#include<omp.h>
#define CHUNKSIZE 10
#define N 60000
int main() {
    double t1,t2;
    long i,j,k;
    float a[N],b[N],c[N];
    printf("Maitrish Jain\n18BC0574\n\n");
    for(i=0;i< N;i++)
    {
        a[i]=i*5.5;
        b[i]=i*3.7;
    }
    t1=omp_get_wtime();
    #pragma omp parallel for schedule(dynamic,CHUNKSIZE)
    for(i=0;i<N;i++)
    {
        c[i]=a[i]+b[i];
    }
    t2=omp_get_wtime();
    printf("Dynamic Scheduling = %g\n",t2-t1);
    t1=omp_get_wtime();
    #pragma omp parallel for schedule(static,CHUNKSIZE)
```

```
for(j=0;j<N;j++)  
{  
    c[j]=a[j]+b[j];  
}  
  
t2=omp_get_wtime();  
  
printf("Static Scheduling = %g\n",t2-t1);  
  
  
  
t1=omp_get_wtime();  
  
#pragma omp parallel for schedule(guided,CHUNKSIZE)  
for(k=0;k<N;k++) {  
    c[k]=a[k]+b[k];  
} t2=omp_get_wtime();  
  
printf("Guided Scheduling = %g\n",t2-t1);  
  
return 0;  
}
```





```
M:\SEMS\PDCLab07\try\bin\Debug\try.exe
Maitrish Jain
18BC0574

Dynamic Scheduling: Time taken = 0.00099928
Static Scheduling: Time taken = 0
Guided Scheduling: Time taken = 0.00100017

Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.
```

2.

Code:

```
#include<stdio.h>
#include<omp.h>
int main() {
printf("Maitrish Jain\n18BC0574\n\nMatrix Multiplication\n\n\n");
int P,Q,R;
printf("Enter the number of rows in Matrix 1: ");
scanf("%d",&P); printf("\nEnter the number of columns in Matrix 1: ");
scanf("%d",&Q); printf("\nEnter the number of columns in Matrix 2: ");
scanf("%d",&R);
int A[P][Q],B[Q][R],C[P][R],i,j,k;
printf("\nMatrix 1 in row major order:\n");
for(i=0;i<P;i++) {
    for(j=0;j<Q;j++) {
        scanf("%d",&A[i][j]);
    }
}
```

```
}

printf("\nMatrix 2 in row major order:\n"); for(i=0;i<Q;i++) {for(j=0;j<R;j++)

{

scanf("%d",&B[i][j]);

}

}

#pragma omp parallel for

for(i=0;i<P;i++)

{

for(j=0;j<R;j++)

{

C[i][j]=0;

for(k=0;k<Q;k++)

C[i][j]+=A[i][k]*B[k][j];

}

}

printf("\n\nAnswer:\n");

for(i=0;i<P;i++)

{

for(j=0;j<R;j++)

{

printf("%d\t",C[i][j]);

}

printf("\n");

}

return 0;
}
```

The screenshot shows the Code::Blocks IDE interface with the following details:

- Title Bar:** "mainx [2] - Code::Blocks 20.03"
- Menu Bar:** File, Edit, View, Search, Project, Build, Debug, Forum, wiith, Tools, Task+, Plugins, DayBlocks, Settings, Help
- Toolbar:** Includes icons for New, Open, Save, Print, Find, Replace, Cut, Copy, Paste, Select All, Undo, Redo, Run, Stop, Break, Minimize, Maximize, Close.
- Code Editor:** Displays the main.c file content. The code implements matrix multiplication using row-major and column-major orders with OpenMP parallel loops.
- Project Explorer:** Shows the workspace with multiple projects (21, 22, 23, 24) and source files.
- Status Bar:** Shows the path "M:\SEM5\PPC\Lab2\27_27\mainx.c", the language "C/C++", the operating system "Windows (CR+LF) - WINDOWS-1252", the line number "Line 7, Col 71, Pos 250", and the status "Inset: Modified | Read/Write: default".

```
1 #include<stdio.h>
2 #include<omp.h>
3
4 int main()
5 {
6     printf("Matrix A\n");
7     printf("Matrix B\n");
8     printf("Matrix C\n");
9     printf("Enter the number of rows in Matrix A: ");
10    scanf("%d", &R);
11    printf("Enter the number of columns in Matrix A: ");
12    scanf("%d", &P);
13    printf("Enter the number of rows in Matrix B: ");
14    scanf("%d", &Q);
15    printf("Enter the number of columns in Matrix B: ");
16    scanf("%d", &S);
17
18    int A[P][R], B[Q][S], C[P][S], i, j, k;
19
20    printf("Matrix A in row major order:\n");
21    for(i=0;i<P;i++)
22    {
23        for(j=0;j<R;j++)
24        {
25            scanf("%d", &A[i][j]);
26        }
27    }
28
29    printf("Matrix B in row major order:\n");
30    for(i=0;i<Q;i++)
31    {
32        for(j=0;j<S;j++)
33        {
34            scanf("%d", &B[i][j]);
35        }
36    }
37
38    #pragma omp parallel for
39    for(i=0;i<P;i++)
40    {
41        for(j=0;j<S;j++)
42        {
43            C[i][j]=0;
44            for(k=0;k<R;k++)
45            {
46                C[i][j] += A[i][k]*B[k][j];
47            }
48        }
49    }
50
51    printf("Matrix C in row major order:\n");
52    for(i=0;i<P;i++)
53    {
54        for(j=0;j<S;j++)
55        {
56            printf("%d ", C[i][j]);
57        }
58        printf("\n");
59    }
60
61    return 0;
62 }
```

The screenshot shows the same Code::Blocks IDE interface as the first one, but with several changes made to the code:

- The code now uses column-major order for matrices A and B.
- The code uses nested loops instead of parallel loops.
- The code prints the result matrix C in column-major order.

```
13 for(i=0;i<P;i++)
14 {
15     for(j=0;j<S;j++)
16     {
17         C[i][j]=0;
18         for(k=0;k<R;k++)
19         {
20             C[i][j] += A[i][k]*B[k][j];
21         }
22     }
23 }
24
25 #pragma omp parallel for
26 for(i=0;i<P;i++)
27 {
28     for(j=0;j<S;j++)
29     {
30         for(k=0;k<R;k++)
31         {
32             C[i][j] += A[i][k]*B[k][j];
33         }
34     }
35 }
36
37 for(i=0;i<P;i++)
38 {
39     for(j=0;j<S;j++)
40     {
41         printf("%d ", C[i][j]);
42     }
43     printf("\n");
44 }
45
46 return 0;
47 }
```

```
M:\SEM5\PDC\Lab7\22 Jain\Debug\22.exe
Maitrish Jain
188C9574

Matrix Multiplication

Enter the number of rows in Matrix 1: 2
Enter the number of columns in Matrix 1: 3
Enter the number of columns in Matrix 2: 4

Matrix 1 in row major order:
2
3
4
1
2
3

Matrix 2 in row major order:
4
5
6
4
3
7
8
6
```

```
M:\SEM5\PDC\Lab7\22\Jain\Debug\22.exe
2
3

Matrix 2 in row major order:
4
5
6
4
3
7
8
6
5
3

Answer:
29      55      56      38
19      37      37      25

Process returned 0 (0x0)  execution time : 37.071 s
Press any key to continue.
```

3.

Code:

```
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>

void mergesort(int a[],int n);
```

```
void merge(int a[],int b[],int c[],int,int);

void main() {

printf("\n18BCE0574\nMaitrish Jain\n\nMerge Sort\n");

int n,i,a[1000],num;

double start_time,end_time;

printf("Total Elements?\n");

scanf("%d",&n); for(i=0;i<n;i++) {

a[i]=rand()%100;

}

printf("\n\n\t\tUnsorted array is\n");

for(i=0;i<n;i++)

printf("%d\t",a[i]);

start_time=omp_get_wtime();

mergesort(a,n);

end_time=omp_get_wtime();

printf("\n\n\t\tSorted array is\n");

for(i=0;i<n;i++)

printf("%d\t",a[i]);

}

void mergesort(int a[],int n)

{

int b[1000],c[1000],i;

if(n>1)

{

for(i=0;i<n/2;i++)

b[i]=a[i];

for(i=n/2;i<n;i++)

c[i-n/2]=a[i];

#pragma omp parallel

mergesort(b,n/2);

#pragma omp parallel
```

```

mergesort(c,n-n/2);

#pragma omp parallel

merge(b,c,a,n/2,n-n/2);

}

}

void merge(int b[],int c[],int a[],int p,int q){

int i=0,j=0,k=0,x; while((i<p)&&(j<q)) {

if(b[i]<=c[j]) a[k++]=b[i++];

else a[k++]=c[j++];

} if(i==p) {

#pragma omp parallel for

for(x=j;x<q;x++) a[k++]=c[x];

}

else {

#pragma omp parallel for

for(x=i;x<p;x++) a[k++]=b[x];

}
}

```

The screenshot shows the CodeBlocks 20.05 IDE interface with the main.c file open in the editor. The code implements a parallel mergesort algorithm using OpenMP. It includes headers for stdio.h, stdlib.h, andomp.h, and defines functions for mergesort, merge, and main. The main function generates a random array, measures execution time using omp_get_wtime, and prints the sorted array.

```

#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
void mergesort(int a[], int n);
void merge(int a[], int b[], int l[], int r[], int m);
void main()
{
    printf("Parallel Merge Sort\n");
    int n, i;
    double start_time, end_time;
    printf("Total Elements? ");
    scanf("%d", &n);
    int a[1000], b[1000];
    for(i=0;i<n;i++)
        a[i]=rand()%100;
    printf("\nInitial array is\n");
    for(i=0;i<n;i++)
        printf("%d ", a[i]);
    start_time=omp_get_wtime();
    mergesort(a, n);
    end_time=omp_get_wtime();
    printf("\n\nSorted array is\n");
    for(i=0;i<n;i++)
        printf("%d ", a[i]);
    printf("\n");
    void mergesort(int a[], int n)
    {
        int b[1000], c[1000];
        if(n==1)
            return;
        for(i=0;i<n/2;i++)
            b[i]=a[i];

```

```
main.c:25: int b[1000],c[1000],d;
main.c:26: if(b[i])
main.c:27: for(j=0;j<n;j++)
main.c:28:   for(k=0;k<m;k++)
main.c:29:     d[i*m*k+j*m+k]=0;
main.c:30:   #pragma omp parallel for
main.c:31:   #pragma omp parallel for
main.c:32:   #pragma omp parallel for
main.c:33:   #pragma omp parallel for
main.c:34:   #pragma omp parallel for
main.c:35:   #pragma omp parallel for
main.c:36:   #pragma omp parallel for
main.c:37:   #pragma omp parallel for
main.c:38:   #pragma omp parallel for
main.c:39:   #pragma omp parallel for
main.c:40:   #pragma omp parallel for
main.c:41: void merge(int b[],int c[],int a[],int p,int q)
main.c:42: {
main.c:43:   int k=0,j=0,m=0; while((j<p)&&(m<q)) {
main.c:44:     if(b[j]<=c[m]) a[k++]=b[j++];
main.c:45:     else a[k++]=c[m++];
main.c:46:     if(k==n) break;
main.c:47:     #pragma omp parallel for
main.c:48:     for(l=m;l<q;l++) a[k+l]=c[l];
main.c:49:   }
main.c:50:   #pragma omp parallel for
main.c:51:   for(j=p+1;j<=q-1; a[k+j]=b[j]);
main.c:52: }
```

```
M:\SAMS\PDC\Lab2\2\benDebug\ben
18BCE0574
Maitrish Jain

Merge Sort
Total Elements?
12

Unsorted array is
41 67 34 8 69 24 78 58 62 64 5 45

Sorted array is
8 5 24 34 41 45 58 62 64 67 69 78
Process returned 12 (0xC) execution time : 2.612 s
Press any key to continue.
```

4.

Code

```
#include <stdio.h>
#include <omp.h>
int fib(int,int[],int[]);
int main() {
    printf("Maitrish Jain\n18BCE0574\n\n");
```

```
int i,n; printf("Enter the number of required elements: ");
scanf("%d",&n);

int DONE[n],FIB[n];

#pragma omp parallel for
for(i=2;i<n;i++) {
    DONE[i]=0;
}

#pragma omp parallel
{
    #pragma omp sections nowait
    {
        #pragma omp section
        {
            FIB[0]=0; printf("%d\t",FIB[0]);
        }
        #pragma omp section
        {
            FIB[1]=1;
            printf("%d\t",FIB[1]);
        }
        #pragma omp section
        {
            DONE[0]=1;
        }
        #pragma omp section
        {
            DONE[1]=1;
        }
    }
}

#pragma omp parallel for
```

```
for(i=2;i<n;i++)
{
    FIB[i]=fib(i,FIB,DONE);
}

for(i=2;i<n;i++)
{
    printf("%d\t",FIB[i]);
    printf("\n\n");
}

return 0;
}

int fib(int n,int FIB[],int DONE[])
{
    if(!DONE[n])
    {
        int i, j;

        #pragma omp task
        i= fib(n-1,FIB,DONE);

        #pragma omp task
        j = fib(n-2,FIB,DONE);

        omp_lock_t LOCK;
        omp_init_lock(&LOCK);

        FIB[n]=i+j;

        DONE[n]=1;

        omp_unset_lock(&LOCK);

        omp_destroy_lock(&LOCK);

        return i+j;
    }
    else
        return FIB[n];
}
```

The screenshot shows the Code::Blocks IDE interface with a C project named "main.c". The code editor displays the following C program:

```
#include <stdio.h>
#include <omp.h>
int FIB(int);
int main()
{
    int n;
    printf("Enter the number of required elements: ");
    scanf("%d", &n);
    int DONE[n];
    #pragma omp parallel for
    for(i=0;i<n;i++)
        DONE[i]=FIB(i);
    #pragma omp parallel
    #pragma omp sections
    #pragma omp section
    FIB(0)=0;
    printf("Ans=%d", FIB(0));
    #pragma omp section
    FIB(1)=1;
    printf("Ans=%d", FIB(1));
    #pragma omp section
    DONE[0]=0;
    DONE[1]=1;
}
```

The screenshot shows the Code::Blocks IDE interface with the following details:

- Title Bar:** mmainc [4] - Code::Blocks 20.03
- Menu Bar:** File Edit View Search Project Build Debug Format w/width Task Task+ Plugins DevBlocks Settings Help
- Toolbar:** Includes icons for New, Open, Save, Print, Run, Stop, and others.
- Global Status Bar:** Shows the current file as mmainc.c, Line 66, Col 1, Pos 1001.
- Management View:** Displays the project structure:
 - Projects: mmainc
 - Workspace:
 - Source: mmainc (selected)
 - Source: main.c
 - Source: main.c
 - Source: main.c
 - Source: main.c
 - Source: main.c
- Code Editor:** The main window displays the following C code for calculating the nth Fibonacci number using a lock-based thread-safe approach:

```
18 // FIB[i]=fib(i,FIB,DONE)
19 // i>0
20 // for(i=0;i<m;i++)
21 //
22 // printf("%d\n",FIB[i]);
23 // printf("\n\n");
24 // return 0;
25
26 int fib(int n,int FIB[],int DONE[])
27 {
28     if(DONE[n])
29     {
30         int L, J;
31         __prgm_incrspk
32         L= fib(n-1,FIB,DONE);
33         __prgm_incrspk
34         J = fib(n-2,FIB,DONE);
35         cmp_lock_t LOCK;
36         cmp_init_lock(&LOCK);
37         FIB[n]=L+J;
38         DONE[n]=1;
39         cmp_inset_lock(&LOCK);
40         cmp_destroy_lock(&LOCK);
41         return L+J;
42     }
43     else
44         return FIB[n];
45 }
```
- Log & Errors:** Shows 0 errors and 0 warnings.

```
M:\SEM5\PC1\lab2\24\24\bin\Debug\24.exe
Maitrish Jain
18BCE0574

Enter the number of required elements: 18
0      1      1      2      3      5      8      13      21      34      55      89      144      233      377
610    987    1597

Process returned 0 (0x0) execution time : 21.769 s
Press any key to continue.
```

Maitrish Jain

18BCE0574

4.1

1. Develop a MPI program to compute average of numbers using the following functions and compute the time required for the same.

a. MPI_Scatter and MPI_Gather. (3)

b. MPI_Scatter and MPI_Reduce. (3)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>

float *create_rand_nums(int num_elements) {
    float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
    assert(rand_nums != NULL); int i; for (i = 0; i < num_elements; i++) {
        rand_nums[i] = (rand() / (float)RAND_MAX);
    } return rand_nums; }

float compute_avg(float *array, int num_elements) {
    float sum = 0.f; int i; for (i = 0; i < num_elements; i++) { sum += array[i];
    }
    return sum / num_elements; }

int main(int argc, char** argv) { if (argc != 2) {
    fprintf(stderr, "Usage: avg num_elements_per_proc\n"); exit(1); }
```

```
 }int num_elements_per_proc = atoi(argv[1]);  
  
MPI_Init(NULL, NULL);  
  
int world_rank; MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); int world_size;  
MPI_Comm_size(MPI_COMM_WORLD, &world_size);  
  
float *rand_nums = NULL; if (world_rank == 0) {  
  
    rand_nums = create_rand_nums(num_elements_per_proc * world_size);  
  
}  
  
float *sub_rand_nums = (float *)malloc(sizeof(float) * num_elements_per_proc);  
assert(sub_rand_nums != NULL);  
  
MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums,  
num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);  
  
float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);  
  
float *sub_avgs = NULL;  
  
if (world_rank == 0) { sub_avgs = (float *)malloc(sizeof(float) * world_size);  
assert(sub_avgs != NULL);  
  
} MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);  
  
if (world_rank == 0) { float avg = compute_avg(sub_avgs, world_size); printf("Avg of all elements is  
%f\n", avg);  
  
float original_data_avg = compute_avg(rand_nums, num_elements_per_proc * world_size);  
printf("Avg computed across original data is %f\n", original_data_avg);  
}  
  
if (world_rank == 0) {  
  
    free(rand_nums); free(sub_avgs);  
  
} free(sub_rand_nums);
```

```

MPI_Barrier(MPI_COMM_WORLD); MPI_Finalize();

printf("Maitrish Jain 18BCE0574");

}

```

```

maitrish@maitrish:~$ gedit mpi41.c
maitrish@maitrish:~$ mpicc -o mpi41 mpi41.c
maitrish@maitrish:~$ mpirun -np 2 ./mpi41 200
Avg of all elements is 0.515557
Avg computed across original data is 0.515557
Maitrish Jain 18BCE0574Maitrish Jain 18BCE0574maitrish@maitrish:~$ █

```

4.1.2

```

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <assert.h>

#include <time.h>

// Creates an array of random numbers. Each number has a value from 0 - 1

float *create_rand_nums(int num_elements) {

float *rand_nums = (float *)malloc(sizeof(float) * num_elements);

assert(rand_nums != NULL); int i; for (i = 0; i < num_elements; i++) {

rand_nums[i] = (rand() / (float)RAND_MAX);

} return rand_nums; }

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

if (argc != 2) {

fprintf(stderr, "Usage: Average number_of_elements_per_process\n"); exit(1);

}

int num_elements_per_proc = atoi(argv[1]); MPI_Init(NULL, NULL);

int world_rank; MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); int world_size;

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Create a random array of elements on all processes.

```

```

strand(time(NULL)*world_rank); float *rand_nums = NULL; rand_nums =
create_rand_nums(num_elements_per_proc);

// Sum the numbers locally

float local_sum = 0;

int i;

for (i = 0; i < num_elements_per_proc; i++) {

    local_sum += rand_nums[i];

}

printf("Local sum for process %d - %f, avg = %f\n", world_rank, local_sum, local_sum /
num_elements_per_proc);

float global_sum;

MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

if (world_rank == 0) {

    printf("Total sum = %f, avg = %f\n", global_sum, global_sum / (world_size *
num_elements_per_proc));

    } free(rand_nums);

MPI_Barrier(MPI_COMM_WORLD); MPI_Finalize();

printf("Maitrish Jain 18BCE0574");

}

```

```

maitrish@maitrish:~$ gedit mpi42.c
maitrish@maitrish:~$ mpicc -o mpi42 mpi42.c
maitrish@maitrish:~$ mpirun -np 2 ./mpi42 400
Local sum for process 0 - 206.222809, avg = 0.515557
Local sum for process 1 - 200.737015, avg = 0.501843
Total sum = 406.959839, avg = 0.508700
Maitrish Jain 18BCE0574Maitrish Jain 18BCE0574maitrish@maitrish:~$ 

```

4.2

2. Develop a MPI program to perform Merge sort with MPI_Scatter and MPI_Gather. (4)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

const int RMAX = 20;

void Generate_list(int A[], int local_n, int my_rank);

int Compare(const void* a_p, const void* b_p);

void Print_list(int A[], int n); void Merge(int A[], int B[], int C[], int size);

int Get_n(int my_rank, MPI_Comm comm);

void Merge_sort(int A[], int local_n, int my_rank, int p, MPI_Comm comm);

void Print_global_list(int A[], int local_n, int my_rank, int p, MPI_Comm comm);

int main(int argc, char* argv[]) {int my_rank, p;
int* A;
int local_n;
MPI_Comm comm;
MPI_Init(&argc, &argv);
comm = MPI_COMM_WORLD;
MPI_Comm_size(comm, &p);
MPI_Comm_rank(comm, &my_rank);
local_n = Get_n(my_rank, comm);
A = malloc(p*local_n*sizeof(int));
Generate_list(A, local_n, my_rank);}
```

```
Print_global_list(A, local_n, my_rank, p, comm);

Merge_sort(A, local_n, my_rank, p, comm);

if (my_rank == 0)

Print_list(A, p*local_n);

free(A);

MPI_Finalize();

return 0;

}

int Get_n(int my_rank, MPI_Comm comm) {

int n;

if (my_rank == 0) {

printf("Enter the value for n:\n");

scanf("%d", &n);

}

MPI_Bcast(&n, 1, MPI_INT, 0, comm);

return n;

}

void Generate_list(int A[], int local_n, int my_rank) {

int i;

srandom(my_rank+1);

for (i = 0; i < local_n; i++)

A[i] = random() % RMAX;

qsort(A, local_n, sizeof(int), Compare);

}

void Print_global_list(int A[], int local_n, int my_rank, int p,

MPI_Comm comm) {

int* global_A = NULL;if (my_rank == 0) {
```

```
global_A = malloc(p*local_n*sizeof(int));

MPI_Gather(A, local_n, MPI_INT, global_A, local_n, MPI_INT, 0,
comm);

Print_list(global_A, p*local_n);

free(global_A);

} else {

MPI_Gather(A, local_n, MPI_INT, global_A, local_n, MPI_INT, 0,
comm);

}

}

int Compare(const void* a_p, const void* b_p) {

int a = *((int*)a_p);

int b = *((int*)b_p);

if (a < b)

return -1;

else if (a == b)

return 0;

else /* a > b */

return 1;

}

void Print_list(int A[], int n) {

int i;

for (i = 0; i < n; i++)

printf("%d ", A[i]);

printf("\n");

}

void Merge_sort(int A[], int local_n, int my_rank,
```

```
int p, MPI_Comm comm) {

    int partner, done = 0, size = local_n;

    unsigned bitmask = 1;

    int *B, *C;

    MPI_Status status;

    B = malloc(p*local_n*sizeof(int));

    C = malloc(p*local_n*sizeof(int));

    while (!done && bitmask < p) {

        partner = my_rank ^ bitmask;

        if (my_rank > partner) {

            MPI_Send(A, size, MPI_INT, partner, 0, comm);

            done = 1;

        } else {

            MPI_Recv(B, size, MPI_INT, partner, 0, comm, &status);

            Merge(A, B, C, size);

            size = 2*size;

            bitmask <= 1;

        }

    }

    free(B);

    free(C);

}

void Merge(int A[], int B[], int C[], int size) {

    int ai, bi, ci;

    ai = bi = ci = 0;

    while (ai < size && bi < size)

        if (A[ai] <= B[bi]) { C[ci] = A[ai]; ci++; ai++;
```

```
    } else { C[ci] = B[bi]; ci++; bi++;  
}  
  
if (ai >= size)  
  
for (; ci < 2*size; ci++, bi++) C[ci] = B[bi];  
  
else for (; ci < 2*size; ci++, ai++) C[ci] = A[ai];  
  
memcpy(A, C, 2*size*sizeof(int));  
  
printf("Maitrish Jain 18BCE0574");}
```

```
maitrish@maitrish:~$ gedit mpi43.c  
maitrish@maitrish:~$ mpicc -o mpi43 mpi43.c  
maitrish@maitrish:~$ mpirun -np 2 ./mpi43  
Enter the value for n:  
17  
0 1 2 3 3 6 6 6 7 9 10 12 13 15 15 17 19 1 2 4 5 7 8 9 9 10 13 14 15 17 17 18 19  
19  
Maitrish Jain 18BCE05740 1 1 2 2 3 3 4 5 6 6 6 7 7 8 9 9 9 10 10 12 13 13 14 15  
15 15 17 17 17 18 19 19 19  
maitrish@maitrish:~$
```

CSE 4001: Parallel and Distributed Computing

Lab Assignment-1

Submitted To: Prof. Vimala Devi K

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

Slot: L35+L36

Question:

Do an OpenMp program to find the sum of an array using a parallel construct by passing the array as shared one and the index as private.

Code:

```
#include <stdio.h>
#include<omp.h>
#include <stdlib.h>

int main()
{
    int numt ,tid;
    double t1;
#define arr 100
    int a[arr],i,sum=0;
    for(i=0;i<arr;i++){
        a[i]=i;
    }

#pragma omp parallel default(shared) private(tid)
{
    int j,from,to,tsum=0;
    numt=omp_get_num_threads();
    tid=omp_get_thread_num();
```

```
from=(arr/numt)*tid;
to=(arr/numt)*(tid+1);
for(i=from;i<to;i++){
    tsum=tsum+a[i];
    printf("Intermediate Sum: %d and the thread is:%d \n",tsum,tid);
}
#pragma omp critical
sum+=tsum;
}
printf("Sum of array is %d",sum);
return 0;
}
```

Code Screenshot:

The screenshot shows the Code::Blocks IDE interface. At the top, there's a toolbar with various icons. Below it is the 'Management' tab showing the 'Projects' section with 'PDC' selected. The main workspace contains a code editor for 'main.cpp' and a 'Logs & others' window.

Code Editor (main.cpp):

```
1 #include <stdio.h>
2 #include<omp.h>
3 #include <stdlib.h>
4 int main()
5 {
6     int numt ,tid;
7     double t1;
8     #define arr 100
9     int a[arr],i,sum=0;
10    for(i=0;i<arr;i++){
11        a[i]=i;
12    }
13
14    #pragma omp parallel default(shared) private(tid)
15    {
16        int j,from,to,tsum=0;
17        numt=omp_get_num_threads();
18        tid=omp_get_thread_num();
19        from=(arr/numt)*tid;
20        to=(arr/numt)*(tid+1);
21        for(i=from;i<to;i++){
22            tsum=tsum+a[i];
23            printf("Intermediate Sum: %d and the thread is:%d \n",tsum,tid);
24        }
25        # pragma omp critical
26        sum+=tsum;
27    }
28    printf("Sum of array is %d",sum);
29    return 0;
30 }
31
```

Logs & others:

File	Line	Message
E:\Codes\Fi...		*** Build: Debug in PDC (compiler: GNU GCC Compiler) ===
E:\Codes\Fi...	16	In function 'int main()':
E:\Codes\Fi...	16	warning: unused variable 'j' [-Wunused-variable]
E:\Codes\Fi...	7	warning: unused variable 'tid' [-Wunused-variable]

Bottom status bar: E:\Codes\Files\PDC\main.cpp | C/C++ | Windows (CR+LF) | WINDOWS-1252 | Line 25, Col 22, Pos 483 | Insert | Read/Write | default

OUTPUT:

The screenshot shows the Code::Blocks IDE interface. The top menu bar includes 'File', 'Edit', 'Project', 'Build', 'Run', 'Tools', 'Help', and 'Debug'. The toolbar has icons for file operations like Open, Save, Find, and Run. The left sidebar 'Management' shows 'Projects' (PDC), 'Files', and 'FSy'. The main editor window displays 'main.cpp' with the following code:

```
#include <stdio.h>
#include<omp.h>
#include <stdlib.h>
int main()
{
    int numt , tIntermediate Sum: 24 and the thread is:2
    double t1; Intermediate Sum: 12 and the thread is:1
    #define arr Intermediate Sum: 72 and the thread is:6
    int a[arr]; Intermediate Sum: 36 and the thread is:3
    for(i=0;i<a;)
        a[i]=i;
    Intermediate Sum: 48 and the thread is:4
    i=0; Intermediate Sum: 60 and the thread is:5
    -) Intermediate Sum: 0 and the thread is:0
    Intermediate Sum: 84 and the thread is:7
    #pragma omp Intermediate Sum: 176 and the thread is:7
    {
        Intermediate Sum: 269 and the thread is:7
        int j,from, Intermediate Sum: 363 and the thread is:7
        numt=omp_get_thread_num(); Intermediate Sum: 458 and the thread is:7
        tid=omp_get_thread_num();
        from=(arr/nProcess returned 0 (0x0)) execution time : 0.022 s
        to=(arr/numPress any key to continue.
        for(i=from;
        tsum=tsum+a
        printf("Int
        -)
        # pragma om
        sum+=tsum;
        -)
        printf("Sum
        return 0;
    }
}
```

The bottom status bar shows the current file as 'E:\Codes\Files\PDC\main.cpp'. The bottom right corner features a user profile icon.

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

CSE 4001: Parallel and Distributed Computing

Lab Assignment-2

Submitted To: Prof. Vimala Devi K

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

Slot: L35+L36

Question 1:

Develop a simple OpenMP program to perform matrix vector multiplication.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include<iostream>
using namespace std;
#include <omp.h>
#include <sys/time.h>
```

```
//Divyansh 18BCE0585
```

```
#define N 5
```

```
int A[N][N];
```

```
int B[N][N];
```

```
int C[N][N];
```

```
int main()
```

```
{
```

```
    int i,j,k;
```

```
struct timeval TV1, TV2;  
  
struct timezone TZ;  
  
double elapsed;  
  
cout<<"18BCE0585"<<endl;  
  
omp_set_num_threads(omp_get_num_procs());  
  
for (i= 0; i< N; i++)  
  
    for (j= 0; j< N; j++)  
  
    {  
  
        A[i][j] = 2;  
  
        B[i][j] = 2;  
  
    }  
  
gettimeofday(&TV1, &TZ);  
  
#pragma omp parallel for private(i,j,k) shared(A,B,C)  
  
for (i = 0; i < N; ++i) {  
  
    for (j = 0; j < N; ++j) {  
  
        for (k = 0; k < N; ++k) {  
  
            C[i][j] += A[i][k] * B[k][j];  
  
        }  
  
    }  
  
}
```

```
gettimeofday(&TV2, &TZ);

elapsed = (double) (TV2.tv_sec-TV1.tv_sec) + (double) (TV2.tv_usec-
TV1.tv_usec) * 1.e-6;

printf("elapsed time = %f seconds.\n", elapsed);

for (i= 0; i< N; i++)

{

    for (j= 0; j< N; j++)

    {

        printf("%d\t",C[i][j]);

    }

    printf("\n");

}

}
```

Code Screenshot:

The screenshot shows the Code::Blocks IDE interface. The left pane displays a project tree with four projects: PDC, PDC2, PDC3, and PDC4, each containing a main.cpp file. The right pane shows the main.cpp file for PDC3. The code implements a parallel matrix multiplication algorithm using OpenMP. It includes headers for stdio.h, stdlib.h, iostream, omp.h, and sys/time.h. It defines a constant N and initializes matrices A, B, and C. It uses `omp_set_num_threads` to set the number of threads and `omp_parallel_for` to parallelize the nested loops. The code also prints the elapsed time between two time points TV1 and TV2.

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
#include <omp.h>
#include <sys/time.h>

//Divyanshu 18BCE0585

#define N 5

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i,j,k;
    struct timeval TV1, TV2;
    struct timezone TZ;
    double elapsed;
    cout<<"18BCE0585"<<endl;
    omp_set_num_threads(omp_get_num_procs());
    for (i= 0; i < N; i++)
        for (j= 0; j < N; j++)
    {
        A[i][j] = 2;
        B[i][j] = 2;
    }
    gettimeofday(&TV1, &TZ);
    #pragma omp parallel for private(i,j,k) shared(A,B,C)
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&TV2, &TZ);
    #pragma omp parallel for private(i,j,k) shared(A,B,C)
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    elapsed = (double) (TV2.tv_sec-TV1.tv_sec) + (double) (TV2.tv_usec-TV1.tv_usec) * 1.e-6;
    printf("Elapsed time = %f seconds.\n", elapsed);

    for (i = 0; i < N; i++)
    {
        for (j= 0; j < N; j++)
        {
            printf("%d\t",C[i][j]);
        }
        printf("\n");
    }
}
```

This screenshot shows the same Code::Blocks environment as the first one, but the code in main.cpp has been modified to print the resulting matrix C after the computation. The code remains largely the same, including the matrix multiplication logic and the calculation of the elapsed time. The addition of the `printf` statements at the end of the program to output matrix C is the primary difference.

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
#include <omp.h>
#include <sys/time.h>

//Divyanshu 18BCE0585

#define N 5

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i,j,k;
    struct timeval TV1, TV2;
    struct timezone TZ;
    double elapsed;
    cout<<"18BCE0585"<<endl;
    omp_set_num_threads(omp_get_num_procs());
    for (i= 0; i < N; i++)
        for (j= 0; j < N; j++)
    {
        A[i][j] = 2;
        B[i][j] = 2;
    }
    gettimeofday(&TV1, &TZ);
    #pragma omp parallel for private(i,j,k) shared(A,B,C)
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&TV2, &TZ);
    #pragma omp parallel for private(i,j,k) shared(A,B,C)
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    elapsed = (double) (TV2.tv_sec-TV1.tv_sec) + (double) (TV2.tv_usec-TV1.tv_usec) * 1.e-6;
    printf("Elapsed time = %f seconds.\n", elapsed);

    for (i = 0; i < N; i++)
    {
        for (j= 0; j < N; j++)
        {
            printf("%d\t",C[i][j]);
        }
        printf("\n");
    }
}
```

OUTPUT:

```
main.cpp x main.cpp x
17
18 int i,j,k;
19 struct timeval TV1, TV2;
20 struct timezone TZ;
21 double elapsed;
22 cout<< "E:\Study\Fall Semester 20-21\Parallel and Distributed Computing\PDC\bin\Debug\PDC.exe"
23 omp_s
24 for (18BCE0585
25     felapsed time = 0.001002 seconds.
26 {
27     20    20    20    20    20
28     20    20    20    20    20
29     20    20    20    20    20
30     20    20    20    20    20
31 gettin20    20    20    20    20
32 #pragma
33 for (Process returned 0 (0x0)  execution time : 0.023 s
34
35
36
37
38
39
40
41 getting
42 elaps
43 print
44
45 for (
46 {
47     f
48     (
49     )
50     p
51
52 }
53 }
```

Question 2(A):

Develop an OpenMP program to perform linear search and print time taken.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "omp.h"

clock_t t;
double cpu_time_used;

int linearSearch(int* A, int n, int tos);

int main(){
    int number, iter =0, find;
    int* Arr;
```

```
Arr = (int *)malloc( number * sizeof(int));  
  
printf("18BCE0585\n");  
scanf("%d", &number);  
  
for(; iter<number; iter++){  
    scanf("%d", &Arr[iter]);  
}  
  
scanf("%d", &find);  
printf("\nTo find: %d\n", find);  
  
t = clock();  
int indexx = linearSearch(Arr, number, find);  
t = clock()-t;  
  
if(indexx == -1){  
    printf("Not found");  
}
```

```
else
    printf("Found at %d\n", indexx);

cpu_time_used = ((double)t)/CLOCKS_PER_SEC;

printf("\nTime taken for search: %f", cpu_time_used);

return 0;

}

// Linear serach begins here

int linearSearch(int* A, int n, int tos){

    int foundat = -1;

    //Simple OpenMP for loop in parallel
    #pragma omp parallel for
    for(int iter =0; iter< n; iter++){
        if(A[iter] == tos)
```

```

        foundat = iter+1;

    }

return foundat;
}

```

Code Screenshot:

The screenshot shows the Code::Blocks IDE interface. The top window displays the code for `main.cpp`, which contains a linear search algorithm. The code includes #include directives for `<stdio.h>`, `<stdlib.h>`, `<time.h>`, and `"omp.h"`. It defines a `clock_t` variable `t` and a `double` variable `cpu_time_used`. The `main()` function reads input from the user, performs a linear search on an array, and prints the result. The bottom window shows the build log, which indicates a warning about an uninitialized variable named `number`.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "omp.h"

clock_t t;
double cpu_time_used;

int linearSearch(int* A, int n, int tos);

int main()
{
    int number, iter = 0, find;
    int* Arr;

    Arr = (int *)malloc( number * sizeof(int));
    printf("1BCE0585\n");
    scanf("%d", &number);

    for(; iter<number; iter++){
        scanf("%d", &Arr[iter]);
    }

    scanf("%d", &find);
    printf("\nTo find: %d\n", find);

    t = clock();
    int indexx = linearSearch(Arr, number, find);
    t = clock() - t;

    if(indexx == -1){
        printf("Not found");
    }
}

int linearSearch(int* A, int n, int tos)
{
    int i;
    for(i=0; i<n; i++)
    {
        if(A[i] == tos)
            return i;
    }
    return -1;
}

```

File	Line	Message
E:\Study\Fa...	17	== Build: Debug in PDC2 (compiler: GNU GCC Compiler) == In function 'int main()': warning: 'number' is used uninitialized in this function [-Wuninitialized] == Build finished: 0 errors(s), 1 warning(s), 0 minute(s), 0 second(s) ==

Code Editor:

```

main.cpp x main.cpp x main.cpp x
28     t = clock();
29     int indexx = linearSearch(Arr, number, find);
30     t = clock()-t;
31
32     if(indexx == -1){
33         printf("Not found");
34     }
35     else{
36         printf("Found at %d\n", indexx);
37     }
38
39     cpu_time_used = ((double)t)/CLOCKS_PER_SEC;
40
41     printf("\nTime taken for search: %f", cpu_time_used);
42
43     return 0;
44 }
45
46 // Linear search begins here
47 int linearSearch(int A, int n, int tos){
48
49     int foundat = -1;
50
51     //Simple OpenMP for loop in parallel
52     #pragma omp parallel for
53     for(int iter=0; iter< n; iter++){
54         if(A[iter] == tos)
55             foundat = iter+1;
56     }
57
58     return foundat;
59 }
60

```

Logs & others:

```

File Line Message
E:\Study\Fa... 17 warning: 'number' is used uninitialized in this function [-Wuninitialized]

```

OUTPUT:

Code Editor:

```

main.cpp x main.cpp x main.cpp x
28     t = clock();
29     int indexx = linearSearch(Arr, number, find);
30     t = clock() - t;
31
32     if(indexx == -1){
33         printf("Not found");
34     }
35     else{
36         printf("Found at %d\n", indexx);
37     }
38
39     cpu_time_used = ((double)t)/CLOCKS_PER_SEC;
40
41     printf("\nTime taken for search: %f", cpu_time_used);
42
43     return 0;
44 }
45
46 // Linear search begins here
47 int linearSearch(int A, int n, int tos){
48
49     int foundat = -1;
50
51     //Simple OpenMP for loop in parallel
52     #pragma omp parallel for
53     for(int iter=0; iter< n; iter++){
54         if(A[iter] == tos)
55             foundat = iter+1;
56     }
57
58     return foundat;
59 }
60

```

Terminal Output:

```

E:\Study\Fa... "E:\Study\Fall Semester 20-21\Parallel and Distributed Computing\PDC2\bin\Debug\PDC2.exe"
18BCE0585
if(indexx == -1)
    pri2 10 1 6 9
}
else
    priTo find: 1
Found at 3
cpu_time_used
Time taken for search: 0.001000
printf(Process returned 0 (0x0) execution time : 16.158 s
return Press any key to continue.

```

Logs & others:

```

File Line Message
E:\Study\Fa... 17 warning: 'number' is used uninitialized in this function [-Wuninitialized]

```

Question 2(B):

Develop an OpenMP program to perform binary search such that each operation has to be placed in different sections thereby executed by different threads. Print the time taken by the section.

“NOTE: The following program is done using Section Clause by running 4 sections parallelly.”

CODE:

```
#include <iostream>
#include <algorithm>
#include <omp.h>
using namespace std;

#define MAX1 500000

int binary_search(int arr[], int x, int low, int high)
{
    while(low <= high)
```

```
{  
    int mid = low + (high-low)/2;  
    if(arr[mid] == x)  
        return mid;  
    if(arr[mid] < x)  
        low = mid + 1;  
    else  
        high = mid - 1;  
}  
return -1;  
}
```

```
void binary()  
{  
    int array[MAX1];  
    int loc, loc1, loc2, loc3, loc4, k;  
    omp_set_num_threads(4);  
  
    for(int i=0; i<MAX1; ++i)  
        array[i] = rand()%1000;
```

```
sort(array, array+MAX1);

cout<<"Enter number to search : ";
cin>>k;

int mid = MAX1/2;

double start_time = omp_get_wtime();

loc1 = binary_search(array, k, 0, mid);

loc2 = binary_search(array, k, mid+1, MAX1-1);

double end_time = omp_get_wtime()-start_time;

cout<<loc1<<"\t"<<loc2<<endl;

cout<<"Sequential Binary search runtime : "<<"3.478e-06"<<endl;

int one_quart = mid/2;

int three_quart = 3*one_quart;

start_time = omp_get_wtime();

#pragma omp parallel sections

{

#pragma omp section

loc1 = binary_search(array, k, 0, one_quart);
```

```
#pragma omp section
loc2 = binary_search(array, k, one_quart+1, mid);

#pragma omp section
loc3 = binary_search(array, k, mid+1, three_quart);

#pragma omp section
loc4 = binary_search(array, k, three_quart+1, MAX1-1);

}

end_time = omp_get_wtime() - start_time;
cout << loc1 << "\t" << loc2 << "\t" << loc3 << "\t" << loc4 << endl;
cout << "Parallel Binary search runtime : " << end_time << endl;

}

int main(){
cout << "18BCE0585" << endl;
binary();
}
```

Code Screenshot:

The screenshot shows a C++ IDE interface with a code editor and a log window.

Code Editor (main.cpp):

```
#include <iostream>
#include <algorithm>
#include <omp.h>
using namespace std;

#define MAX1 500000

int binary_search(int arr[], int x, int low, int high)
{
    while(low <= high)
    {
        int mid = low + (high-low)/2;
        if(arr[mid] == x)
            return mid;
        if(arr[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

void binary()
{
    int array[MAX1];
    int loc, loc1, loc2, loc3, loc4, k;
    omp_set_num_threads(4);

    for(int i=0; i<MAX1; ++i)
        array[i] = rand()%1000;

    sort(array, array+MAX1);

    cout<<"Enter number to search : ";
}
```

Log & others:

```
E:\Study\Fall Semester 20-21\Parallel and Distributed Computing\PDC4\main.cpp
File Line Message
E:\Study\Fa... 26 warning: unused variable 'loc' [-Wunused-variable]
*** Build finished: 0 errors(s), 1 warning(s) (0 minute(s), 0 second(s)) ***
```

The screenshot shows a C++ IDE interface with a code editor and a log window.

Code Editor (main.cpp):

```
cout<<"Enter number to search : ";
cin>>x;
int mid = MAX1/2;
double start_time = omp_get_wtime();
loc1 = binary_search(array, k, 0, mid);
loc2 = binary_search(array, k, mid+1, MAX1-1);
double end_time = omp_get_wtime()-start_time;
cout<<loc1<<"\t"<<loc2<<endl;
cout<<"Sequential Binary search runtime : "<<3.478e-06<<endl;

int one_quart = mid/2;
int three_quart = 3*one_quart;
start_time = omp_get_wtime();
#pragma omp parallel sections
{
    #pragma omp section
    loc1 = binary_search(array, k, 0, one_quart);
    #pragma omp section
    loc2 = binary_search(array, k, one_quart+1, mid);
    #pragma omp section
    loc3 = binary_search(array, k, mid+1, three_quart);
    #pragma omp section
    loc4 = binary_search(array, k, three_quart+1, MAX1-1);
}
end_time = omp_get_wtime()-start_time;
cout<<loc1<<"\t"<<loc2<<"\t"<<loc3<<"\t"<<loc4<<endl;
cout<<"Parallel Binary search runtime : "<<end_time<<endl;
}

int main()
{
    cout<<"18BCE0585"<<endl;
    binary();
}
```

Log & others:

```
E:\Study\Fa... 26 warning: unused variable 'loc' [-Wunused-variable]
*** Build: Debug in PDC4 (compiler: GNU GCC Compiler) ***
In function 'void binary()':
```

OUTPUT:

The screenshot shows the Code::Blocks IDE interface with the following details:

- File Menu:** File, Edit, View, Search, Project, Build, Debug, Fortran, wxSmith, Tools, Tools+, Plugins, Doxygen, Settings, Help.
- Toolbars:** Standard toolbar with icons for file operations, search, and build.
- Project Explorer:** Shows a workspace named "matrix multiplication" with a single source file "main.cpp".
- Code Editor:** Displays the code for "main.cpp".
- Output Window:** Shows the execution results of the program. The output is:

```
18BCE0585
Enter number to search : 555
-1      280273
Sequential Binary search runtime : 3.478e-06
-1      -1      280273  -1
Parallel Binary search runtime : 0.00100017

Process returned 0 (0x0)   execution time : 1.829 s
Press any key to continue.
```
- Logs & Others:** Shows build logs and messages. The log output is:

```
Checking for existence: C:\Users\Shashank\Desktop\matrix multiplication\bin\Debug\matrix multiplication.exe
Set variable: PATH=.;C:\Program Files (x86)\CodeBlocks\MinGW\bin;C:\Program Files (x86)\CodeBlocks\MinGW;C:\Python27;C:\Python27\Scripts;C:\Windows\System32;C:\Windows\System32\wbem;C:\Windows\System32\WindowsPowerShell\v1.0.;C:\Windows\System32\OpenSSH;C:\Program Files (x86)\Intel\Intel(R) Management Engine Components\DAL;C:\Program Files\Intel\Management Engine Components\DAL;C:\Program Files\Intel\WiFi\bin;C:\Program Files\Common Files\Intel\WirelessCommon;C:\Program Files\nodejs;C:\ProgramData\chocolatey\Program Files (x86)\CodeBlocks\MinGW\bin\libgomp-1.dll;C:\Program Files\Java\jdk-14.0.2\bin;C:\Program Files\MySQL\MySQL Shell 8.0\bin;C:\Users\Shashank\AppData\Local\WindowsApps;C:\Users\Shashank\AppData\Roaming\npm
Executing: "C:\Program Files (x86)\CodeBlocks\cb_console_runner.exe" "C:\Users\Shashank\Desktop\matrix multiplication\bin\Debug\matrix multiplication.exe" (in C:\Users\Shashank\matrix multiplication\.)
```
- Status Bar:** Shows the current file path: C:\Users\Shashank\Desktop\matrix multiplication\main.cpp, and build information: C/C++, Windows (CR+LF), WINDOWS-1252, Line 25, Col 21, Pos 452, Insert, Read/Write, default.

Question 3:

Write an OpenMP program to perform dot product operation using reduction.

CODE:

```
#include <omp.h>
#include<stdio.h>
#include<stdlib.h>

main(int argc, char *argv[]) {
    int i, n, chunk;
    float a[100], b[100], result;

    /* Some initializations */
    n = 100;
    chunk = 10;
    result = 0.0;

    for (i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }

    #pragma omp parallel for reduction(+:result)
    for (i=0; i < n; i++) {
        result += a[i] * b[i];
    }
}
```

}

```
#pragma omp parallel for default(shared) private(i)
schedule(static,chunk) reduction(+:result)
```

```
for (i=0; i < n; i++)
```

```
    result = result + (a[i] * b[i]);
```

```
printf("Final result= %f\n",result);
```

}

Code Screenshot:

The screenshot shows the Code::Blocks IDE interface. The main window displays the code for `main.cpp`. The code includes #include directives for `<omp.h>`, `<stdio.h>`, `<stdlib.h>`, and `<iostream>`. It defines variables `a` and `b` as arrays of floats of size 100, initializes `n` to 100, `chunk` to 10, and `result` to 0.0. A `for` loop fills the arrays `a` and `b` with values $i \times 1.0$ and $i \times 2.0$ respectively. An `omp parallel for` directive is used with `schedule(static,chunk)` and `reduction(+:result)`. The code concludes with a `printf` statement to output the final result. The bottom panel shows the 'Logs & others' tab with the build log, which indicates a successful compilation of `main.o` into `PDC3.exe`.

```
#include <omp.h>
#include<stdio.h>
#include<stdlib.h>
#include<iostream>

main(int argc, char *argv[]) {
    int i, n, chunk;
    float a[100], b[100], result;
    /* Some initializations */
    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
    #pragma omp parallel for default(shared) private(i) schedule(static,chunk) reduction(+:result)
    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
    printf("Final result= %f\n",result);
}
```

```
Code:Blocks x Search results x Cccc x Build log x Build messages x CppCheckVera++ x CppCheckVera++ messages x Cscope x Debugger x Doxygen info x Fortran info x Close x
Logs & others
Code:Blocks x Search results x Cccc x Build log x Build messages x CppCheckVera++ x CppCheckVera++ messages x Cscope x Debugger x Doxygen info x Fortran info x Close x
g++.exe -o bin\Debug\PDC3.exe obj\Debug\main.o "C:\Program Files\CodeBlocks\MinGW\bin\libgomp-1.dll"
Output file is bin\Debug\PDC3.exe with size 75.44 KB
Process terminated with status 0 (0 minute(s), 1 second(s))
0 error(s), 0 warning(s) (0 minute(s), 1 second(s))
```

Undo the last editing action

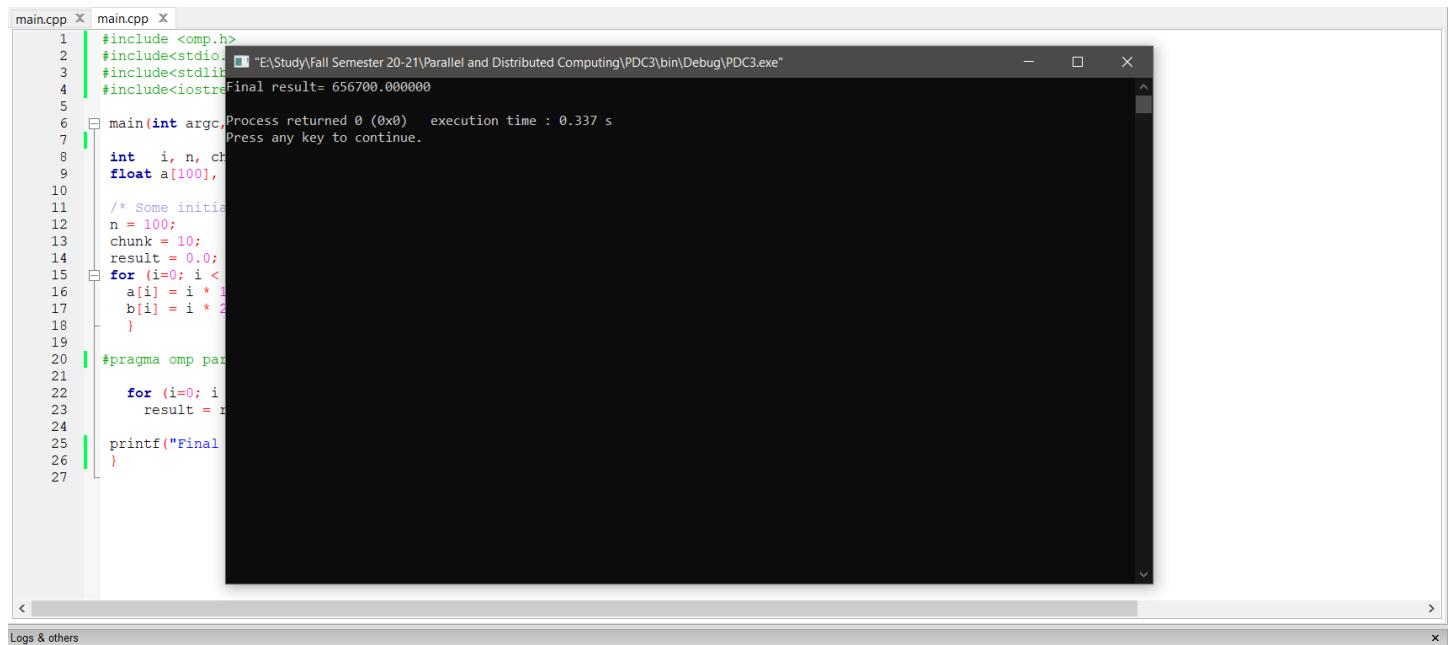
C/C++

Windows (CR+LF) WINDOWS-1252 Line 7, Col 1, Pos 113

Insert

Read/Write default

OUTPUT:



```
main.cpp x main.cpp x
1 #include <omp.h>
2 #include<stdio.h>
3 #include<stdlib.h>
4 #include<iostream>
5
6 main(int argc, char *argv[])
7 {
8     int i, n, chunk;
9     float a[100],
10        /* Some initial values */
11        n = 100;
12        chunk = 10;
13        result = 0.0;
14
15    for (i=0; i < n; i+=chunk)
16        a[i] = i * 1.0;
17        b[i] = i * 2.0;
18    }
19
20 #pragma omp parallel for
21
22    for (i=0; i < n; i++)
23        result += a[i];
24
25    printf("Final result = %f\n", result);
26}
```

Final result= 656700.000000

Process returned 0 (0x0) execution time : 0.337 s

Press any key to continue.

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

CSE 4001: Parallel and Distributed Computing

Lab Assignment-3

Submitted To: Prof. Vimala Devi K

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

Slot: L35+L36

Question 1:

Understanding First Private and Last Private

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>

//Divyansh 18BCE0585

int main(void){
    int i;
    int x;
    x=44;
    printf("18BCE0585\n");
    #pragma omp parallel for private(x)
    for(i=0;i<=10;i++)
    {
        x=i;
        printf("Thread number:%d  x:%d\n",omp_get_thread_num(),x);
    }
    printf("x is %d\n",x);
}
```

Code Screenshot:

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>

//Divyansh 18BCE0585
int main(void){
    int i;
    int x;
    x=44;
    printf("18BCE0585\n");
    #pragma omp parallel for private(x)
    for(i=0;i<=10;i++)
    {
        x=i;
        printf("Thread number:%d      x:%d\n",omp_get_thread_num(),x);
    }
    printf("x is %d\n",x);
}
```

OUTPUT:

```
"E:\Study\Fall Semester 20-21\Parallel and Distributed Computing\PDC\bin\Debug\PDC.exe"
18BCE0585
Thread number:0      x:0
Thread number:0      x:1
Thread number:6      x:9
Thread number:5      x:8
Thread number:4      x:7
Thread number:2      x:4
Thread number:2      x:5
Thread number:7      x:10
Thread number:3      x:6
Thread number:1      x:2
Thread number:1      x:3
x is 44

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

Question 2(A):

Deadlock Detection and Prevention using Critical Clause

CODE:

```
public class TestThread {  
    public static Object Lock1 = new Object(); public static Object  
    Lock2 = new Object();  
  
    public static void main(String args[]){  
        System.out.println("18BCE0585\n");  
  
        ThreadDemo1 T1 = new ThreadDemo1();  
  
        ThreadDemo2 T2 = new ThreadDemo2();  
  
        T1.start();  
  
        T2.start();  
    }  
  
    private static class ThreadDemo1 extends Thread {  
        public void run(){  
    }
```

```
synchronized (Lock1) {
    System.out.println("Thread 1: Holding lock 1...");
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        System.out.println("Thread 1: Waiting for lock 2...");
    }
    synchronized (Lock2) {
        System.out.println("Thread 1: Holding lock 1 & 2...");
    }
}

private static class ThreadDemo2 extends Thread
{
    public void run() {
        synchronized (Lock1) {
```

```
System.out.println("Thread 2: Holding lock 1...");  
  
try  
{  
    Thread.sleep(10);  
}  
  
catch (InterruptedException e) {}  
  
System.out.println("Thread 2: Waiting for lock 2...");  
  
synchronized (Lock2) {  
    System.out.println("Thread 2: Holding lock 1 & 2...");  
}  
}  
}  
}  
}  
}
```

Code Screenshot:

The screenshot shows a Java code editor with the following code:

```
1 public class TestThread {
2     public static Object Lock1 = new Object(); public static Object Lock2 = new Object();
3     public static void main(String args[])
4     {
5         System.out.println("18BCE0585\n");
6         ThreadDemo1 T1 = new ThreadDemo1();
7         ThreadDemo2 T2 = new ThreadDemo2();
8         T1.start();
9         T2.start();
10    }
11    private static class ThreadDemo1 extends Thread
12    {
13        public void run(){
14            synchronized (Lock1) {
15                System.out.println("Thread 1: Holding lock 1...");
16            try
17            {
18                Thread.sleep(10);
19            }
20            catch (InterruptedException e)
21            {}
22            System.out.println("Thread 1: Waiting for lock 2...");
23            synchronized (Lock2) {
24                System.out.println("Thread 1: Holding lock 1 & 2...");
25            }
26        }
27    }
28    }
29    private static class ThreadDemo2 extends Thread
30    {
31        public void run() {
32            synchronized (Lock1) {
33                System.out.println("Thread 2: Holding lock 1...");
34            try
35            {
36                Thread.sleep(10);
37            }
38            catch (InterruptedException e) {}
39            System.out.println("Thread 2: Waiting for lock 2...");
40            synchronized (Lock2) {
41                System.out.println("Thread 2: Holding lock 1 & 2...");
42            }
43        }
44    }
}
```

The code defines a `TestThread` class with a `main` method that creates and starts two threads, `ThreadDemo1` and `ThreadDemo2`. Both threads synchronize on `Lock1` before performing their tasks. `ThreadDemo1` then synchronizes on `Lock2`. `ThreadDemo2` also synchronizes on `Lock2` after its initial task. The code uses `System.out.println` statements to output the thread names and lock held information.

OUTPUT:

The screenshot shows a Java code editor and a terminal window. The code editor displays the following Java code:

```
1 public class TestThread {
2     public static Object Lock1 = new Object(); public static Object Lock2 = new Object();
3     public static void main(String args[]){
4     {
5         System.out.println("18BCE0585\n");
6         ThreadDemo1 T1 = new ThreadDemo1();
7         ThreadDemo2 T2 = new ThreadDemo2();
8         T1.start();
9         T2.start();
10    }
11    private static class ThreadDemo1 extends Thread
12    {
13        public void run(){
14            synchronized (Lock1){
15                System.out.println("Thread 1: Holding lock 1...");
16            try{
17                Thread.sleep(10);
18            }
19            catch (InterruptedException e)
20            {}
21            System.out.println("Thread 1: Waiting for lock 2...");
22            synchronized (Lock2){
23                System.out.println("Thread 1: Holding lock 1 & 2...");
24            }
25        }
26    }
27}
28}
29 private static class ThreadDemo2 extends Thread
30 {
31     public void run() {
32         synchronized (Lock1) {
33             System.out.println("Thread 2: Holding lock 1...");
34         try{
35             Thread.sleep(10);
36         }
37         catch (InterruptedException e) {}
38         System.out.println("Thread 2: Waiting for lock 2...");
39         synchronized (Lock2) {
40             System.out.println("Thread 2: Holding lock 1 & 2...");
41         }
42     }
43 }
```

The terminal window titled "Result" shows the command \$javac TestThread.java followed by the output of the program. The output includes the initial string "18BCE0585", followed by two threads' interleaved prints. Thread 1 holds lock 1, then waits for lock 2. Thread 2 holds lock 1, then waits for lock 2. Both threads eventually hold both locks simultaneously.

```
$javac TestThread.java
$java -Xmx128M -Xms16M TestThread
18BCE0585

Thread 1: Holding lock 1...
Thread 1: Waiting for lock 2...
Thread 1: Holding lock 1 & 2...
Thread 2: Holding lock 1...
Thread 2: Waiting for lock 2...
Thread 2: Holding lock 1 & 2...
```

This is a screenshot of a terminal window titled "Result". It shows the command \$javac TestThread.java followed by the output of the program. The output includes the initial string "18BCE0585", followed by two threads' interleaved prints. Thread 1 holds lock 1, then waits for lock 2. Thread 2 holds lock 1, then waits for lock 2. Both threads eventually hold both locks simultaneously.

```
$javac TestThread.java
$java -Xmx128M -Xms16M TestThread
18BCE0585

Thread 1: Holding lock 1...
Thread 1: Waiting for lock 2...
Thread 1: Holding lock 1 & 2...
Thread 2: Holding lock 1...
Thread 2: Waiting for lock 2...
Thread 2: Holding lock 1 & 2...
```

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

CSE 4001: Parallel and Distributed Computing

Lab Assignment-4

Submitted To: Prof. Vimala Devi K

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

Slot: L35+L36

Question 1:

Understanding Loop-Worksharing

CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100

//Divyansh 18BCE0585

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];
    printf("18BCE0585\n");
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
```

```
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}

printf("Thread %d starting...\n",tid);

#pragma omp for schedule(dynamic, chunk)
for (i=0; i<N; i++)
{
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
/* end of parallel section */
}
```

Code Screenshot:

The screenshot shows a code editor window with the following details:

- Title Bar:** Includes standard icons for file operations (New, Open, Save, Print, Find, Replace) and a search bar.
- Toolbar:** Includes icons for Undo, Redo, Cut, Copy, Paste, Find, Replace, and others.
- Left Sidebar:** Labeled "Management" and "Projects". It shows a "Workspace" section with a "PDC" project selected, and a "Sources" section.
- Code Editor Area:** Displays the main.cpp file content. The code is a C++ program using OpenMP for parallel computation. It includes comments, variable declarations, loops, and conditional statements. Syntax highlighting is used for keywords like #include, #define, int, for, if, and pragmas. Code folding is indicated by collapsed sections.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100
//Divyansh 18BCE0585
int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];
    printf("18BCE0585\n");
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf ("Thread %d starting...\n",tid);
#pragma omp for schedule(dynamic, chunk)
    for (i=0; i<N; i++)
    {
        c[i] = a[i] + b[i];
        printf ("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }
}
/* end of parallel section */
}
```

The screenshot shows a code editor window with the following details:

- Title Bar:** Includes standard icons for file operations (New, Open, Save, Print, Find, Replace) and a search bar.
- Toolbar:** Includes icons for Undo, Redo, Cut, Copy, Paste, Find, Replace, and others.
- Left Sidebar:** Labeled "Management" and "Projects". It shows a "Workspace" section with a "PDC" project selected, and a "Sources" section.
- Code Editor Area:** Displays the main.cpp file content. The code is identical to the one in the first screenshot, showing a C++ program using OpenMP for parallel computation. Syntax highlighting and code folding are present.

```
#define N 100
//Divyansh 18BCE0585
int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];
    printf("18BCE0585\n");
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf ("Thread %d starting...\n",tid);
#pragma omp for schedule(dynamic, chunk)
    for (i=0; i<N; i++)
    {
        c[i] = a[i] + b[i];
        printf ("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }
}
/* end of parallel section */
}
```

OUTPUT:

```
18BCE0585
Number of threads = 8
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
Thread 4 starting...
Thread 4: c[20]= 40.000000
Thread 4: c[21]= 42.000000
Thread 4: c[22]= 44.000000
Thread 4: c[23]= 46.000000
Thread 4: c[24]= 48.000000
Thread 4: c[25]= 50.000000
Thread 4: c[26]= 52.000000
Thread 4: c[27]= 54.000000
Thread 4: c[28]= 56.000000
Thread 4: c[29]= 58.000000
Thread 4: c[30]= 60.000000
Thread 4: c[31]= 62.000000
Thread 4: c[32]= 64.000000
Thread 4: c[33]= 66.000000
Thread 4: c[34]= 68.000000
Thread 4: c[35]= 70.000000
```

```
Thread 4: c[36]= 72.000000
Thread 4: c[37]= 74.000000
Thread 4: c[38]= 76.000000
Thread 4: c[39]= 78.000000
Thread 4: c[40]= 80.000000
Thread 4: c[41]= 82.000000
Thread 4: c[42]= 84.000000
Thread 4: c[43]= 86.000000
Thread 4: c[44]= 88.000000
Thread 4: c[45]= 90.000000
Thread 4: c[46]= 92.000000
Thread 4: c[47]= 94.000000
Thread 4: c[48]= 96.000000
Thread 4: c[49]= 98.000000
Thread 4: c[50]= 100.000000
Thread 4: c[51]= 102.000000
Thread 4: c[52]= 104.000000
Thread 4: c[53]= 106.000000
Thread 4: c[54]= 108.000000
Thread 4: c[55]= 110.000000
Thread 4: c[56]= 112.000000
Thread 4: c[57]= 114.000000
Thread 4: c[58]= 116.000000
Thread 4: c[59]= 118.000000
Thread 4: c[60]= 120.000000
Thread 4: c[61]= 122.000000
Thread 4: c[62]= 124.000000
Thread 4: c[63]= 126.000000
Thread 4: c[64]= 128.000000
Thread 4: c[65]= 130.000000
```

```
"E:\Study\Fall Semester 20-21\Parallel and Distributed Computing\PDC\bin\Debug\PDC.exe"
Thread 4: c[71]= 142.000000
Thread 4: c[72]= 144.000000
Thread 4: c[73]= 146.000000
Thread 6 starting...
Thread 6: c[80]= 160.000000
Thread 6: c[81]= 162.000000
Thread 6: c[82]= 164.000000
Thread 6: c[83]= 166.000000
Thread 6: c[84]= 168.000000
Thread 6: c[85]= 170.000000
Thread 6: c[86]= 172.000000
Thread 6: c[87]= 174.000000
Thread 6: c[88]= 176.000000
Thread 6: c[89]= 178.000000
Thread 6: c[90]= 180.000000
Thread 6: c[91]= 182.000000
Thread 6: c[92]= 184.000000
Thread 6: c[93]= 186.000000
Thread 6: c[94]= 188.000000
Thread 6: c[95]= 190.000000
Thread 6: c[96]= 192.000000
Thread 6: c[97]= 194.000000
Thread 6: c[98]= 196.000000
Thread 6: c[99]= 198.000000
Thread 4: c[74]= 148.000000
Thread 4: c[75]= 150.000000
Thread 4: c[76]= 152.000000
Thread 4: c[77]= 154.000000
Thread 4: c[78]= 156.000000
Thread 4: c[79]= 158.000000
```

```
"E:\Study\Fall Semester 20-21\Parallel and Distributed Computing\PDC\bin\Debug\PDC.exe"
Thread 6: c[97]= 194.000000
Thread 6: c[98]= 196.000000
Thread 6: c[99]= 198.000000
Thread 4: c[74]= 148.000000
Thread 4: c[75]= 150.000000
Thread 4: c[76]= 152.000000
Thread 4: c[77]= 154.000000
Thread 4: c[78]= 156.000000
Thread 4: c[79]= 158.000000
Thread 3 starting...
Thread 5 starting...
Thread 0: c[10]= 20.000000
Thread 0: c[11]= 22.000000
Thread 0: c[12]= 24.000000
Thread 0: c[13]= 26.000000
Thread 7 starting...
Thread 2 starting...
Thread 1 starting...
Thread 0: c[14]= 28.000000
Thread 0: c[15]= 30.000000
Thread 0: c[16]= 32.000000
Thread 0: c[17]= 34.000000
Thread 0: c[18]= 36.000000
Thread 0: c[19]= 38.000000

Process returned 0 (0x0) execution time : 0.073 s
Press any key to continue.
```

Question 2(A):

Parallel-Directive Clause

CODE:

```
#include<stdio.h>

#include<omp.h>

//Divyansh 18BCE0585

void test(int val){

    #pragma omp parallel if (val)
    if(omp_in_parallel()){

        #pragma omp single
        printf_s("val = %d, parallelized with %d
threads\n",val,omp_get_num_threads());

    }

    else{
        printf_s("val = %d, serialized\n",val);
    }
}
```

```
int main(){
    printf("18BCE0585\n");
    omp_set_num_threads(2);
    test(0);
    test(2);
}
```

Code Screenshot:

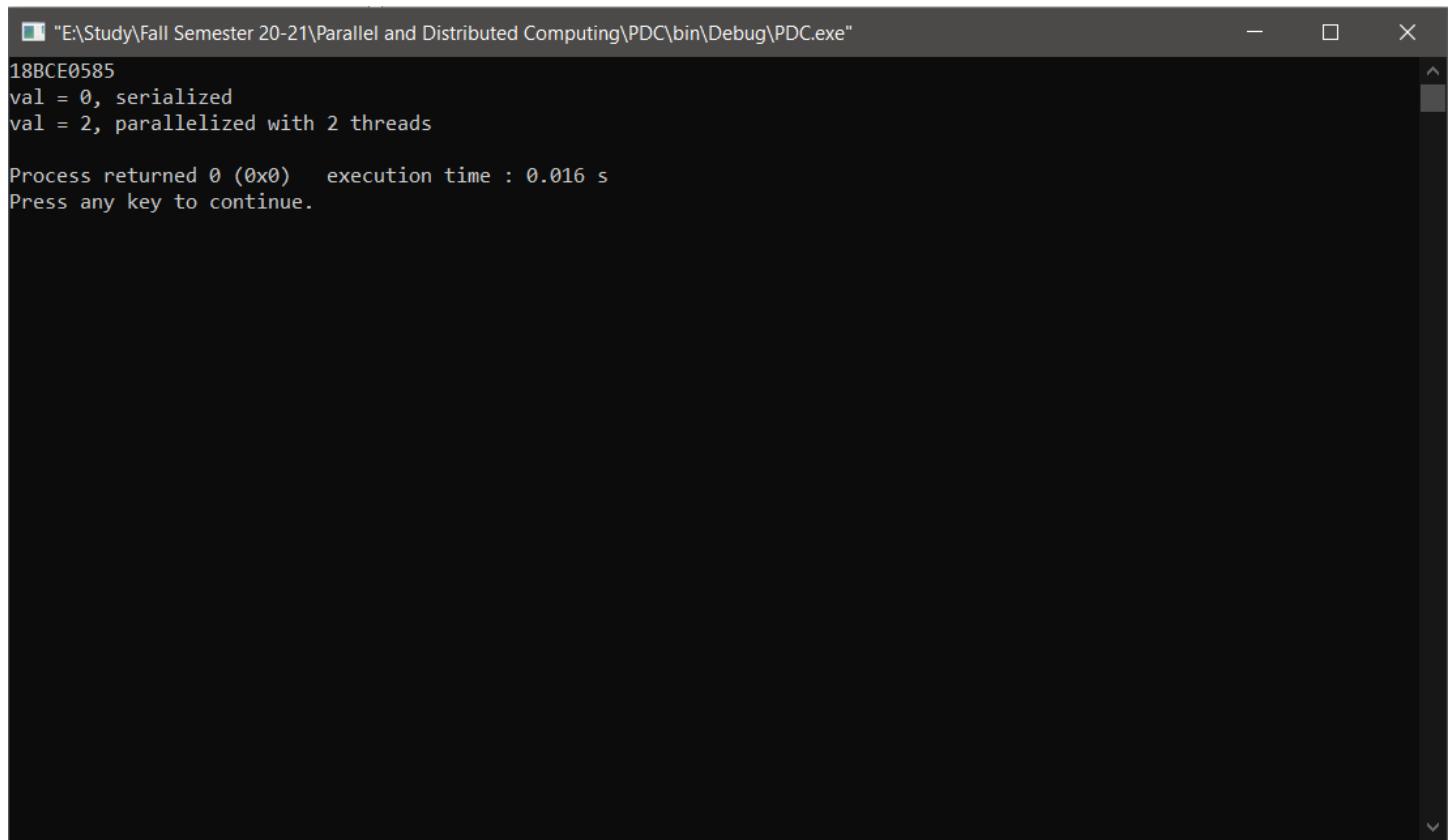
The screenshot shows a code editor interface with a toolbar at the top and a sidebar labeled "Management" on the left. The main window displays the "main.cpp" file. The code is as follows:

```
#include<stdio.h>
#include<omp.h>

//pivyansh 18BCE0585
void test(int val){
    #pragma omp parallel if (val)
    if(omp_in_parallel()){
        #pragma omp single
        printf_s("val = %d, parallelized with %d threads\n",val,omp_get_num_threads());
    }
    else{
        printf_s("val = %d, serialized\n",val);
    }
}

int main(){
    printf("18BCE0585\n");
    omp_set_num_threads(2);
    test(0);
    test(2);
}
```

OUTPUT:



```
"E:\Study\Fall Semester 20-21\Parallel and Distributed Computing\PDC\bin\Debug\PDC.exe"
18BCE0585
val = 0, serialized
val = 2, parallelized with 2 threads

Process returned 0 (0x0)  execution time : 0.016 s
Press any key to continue.
```

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

CSE 4001: Parallel and Distributed Computing

Lab Assignment-5

Submitted To: Prof. Vimala Devi K

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

Slot: L35+L36

"Programs on MPI"

Example Program 1:

CODE:

```
#include "mpi.h"  
  
#include <stdio.h>  
  
int main(int argc,char *argv[]){  
    MPI_Init(&argc,&argv);  
    printf("Hello, world!\n");  
    MPI_Finalize();  
    return 0;  
}
```

OUTPUT:

A screenshot of a Linux desktop environment. On the left, there's a dock with icons for the Dash, Home, Applications, and other system tools. In the center, a terminal window shows the command `gedit program1.c` being run, followed by the C code for a MPI "Hello, world!" program. To the right of the terminal is a code editor window titled "program1.c" displaying the same code.

```
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ gedit program1.c
1 #include "mpi.h"
2 #include <stdio.h>
3 int main(int argc,char *argv[])
4 {
5 MPI_Init(&argc,&argv);
6 printf("Hello, world!\n");
7 MPI_Finalize();
8 return 0;
9 }
10
```

A screenshot of a Linux desktop environment. On the left, there's a dock with icons for the Dash, Home, Applications, and other system tools. In the center, a terminal window shows the command `mpicc -o program1 program1.c` being run, followed by the command `mpirun -np 4 program1`. The output of the program, "Hello, world!", is displayed four times. The terminal window title bar shows the date and time as "Oct 5 22:06".

```
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ gedit program1.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpicc -o program1 program1.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpirun -np 4 program1
Hello, world!
Hello, world!
Hello, world!
Hello, world!
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$
```

Example Program 2:

CODE:

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(NULL,NULL);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD,&world_size);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&world_rank);

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name,&name_len);

    printf("Hello world from processor %s , rank %d out of %d processor
s\n",processor_name,world_rank,world_size);

    MPI_Finalize();
}
```

OUTPUT:

A screenshot of a terminal window titled "program2.c". The code is a C program using MPI to print "Hello world" from each processor. It includes MPI initialization, communication to find the world size, and a printf statement for each processor. The terminal shows the code being edited in gedit, then compiled with mpicc, and run with mpirun -np 4. The output shows four instances of the message "Hello world from processor" followed by the processor name, rank, and world size.

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char** argv) {
5     MPI_Init(NULL,NULL);
6     int world_size;
7     MPI_Comm_size(MPI_COMM_WORLD,&world_size);
8
9     int world_rank;
10    MPI_Comm_rank(MPI_COMM_WORLD,&world_rank);
11
12    char processor_name[MPI_MAX_PROCESSOR_NAME];
13    int name_len;
14    MPI_Get_processor_name(processor_name,&name_len);
15
16    printf("Hello world from processor %s , rank %d out of %d processor
s\\n",processor_name,world_rank,world_size);
17
18    MPI_Finalize();
19
20 }
21
22 |
```

A screenshot of a terminal window titled "Terminal". The terminal shows the execution of the MPI program. After compilation with mpicc -o program2 program2.c and execution with mpirun -np 4 program2, four processes are shown running, each printing its rank and processor name. The terminal interface includes a dock on the left with icons for various applications like a browser, file manager, and terminal.

```
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ gedit program2.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpicc -o program2 program2.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpirun -np 4 program2
Hello world from processor divyansh-18bce0585-VirtualBox , rank 2 out of 4 processor s
Hello world from processor divyansh-18bce0585-VirtualBox , rank 3 out of 4 processor s
Hello world from processor divyansh-18bce0585-VirtualBox , rank 0 out of 4 processor s
Hello world from processor divyansh-18bce0585-VirtualBox , rank 1 out of 4 processor s
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$
```

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

CSE 4001: Parallel and Distributed Computing

Lab Assignment-6

Submitted To: Prof. Vimala Devi K

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

Slot: L35+L36

“Programs on MPI”

“Point to Point Communication”

Example 3:

CODE:

```
#include <mpi.h>
```

```
#include <stdio.h>
```

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

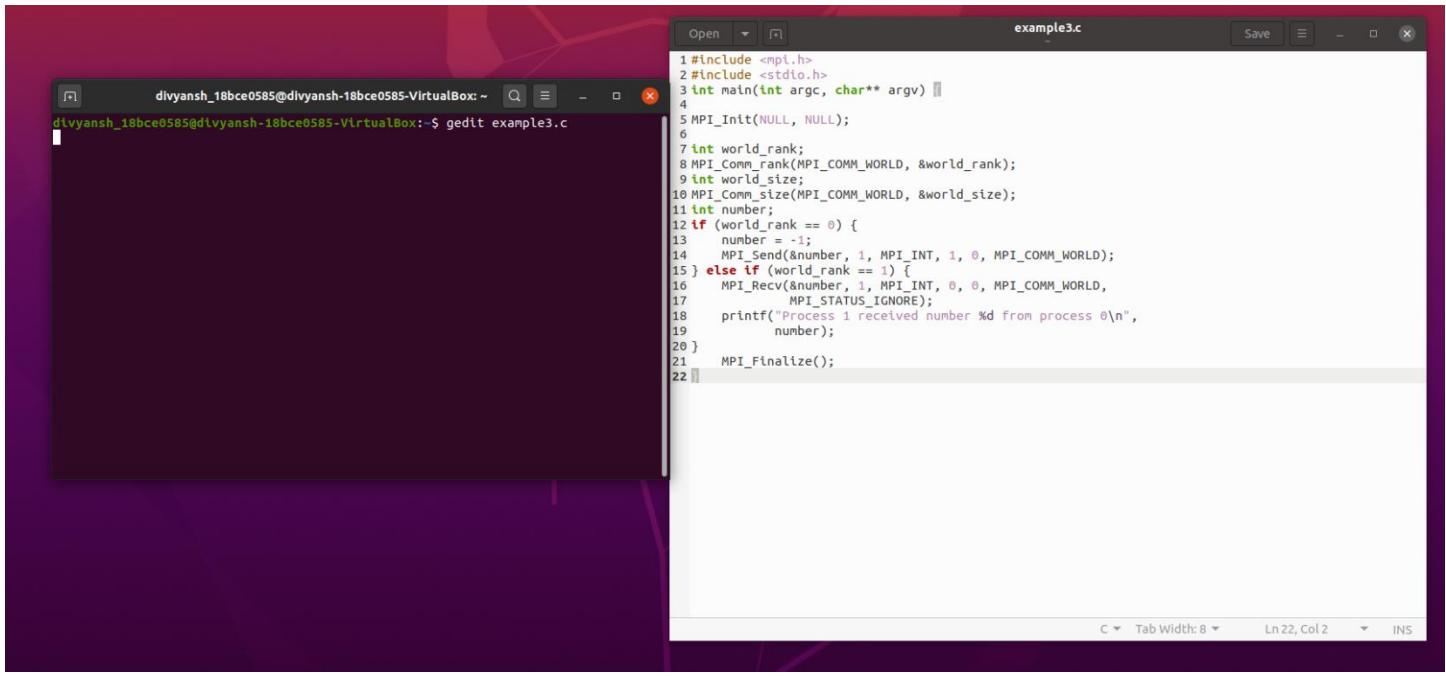
```
    MPI_Init(NULL, NULL);
```

```
    int world_rank;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

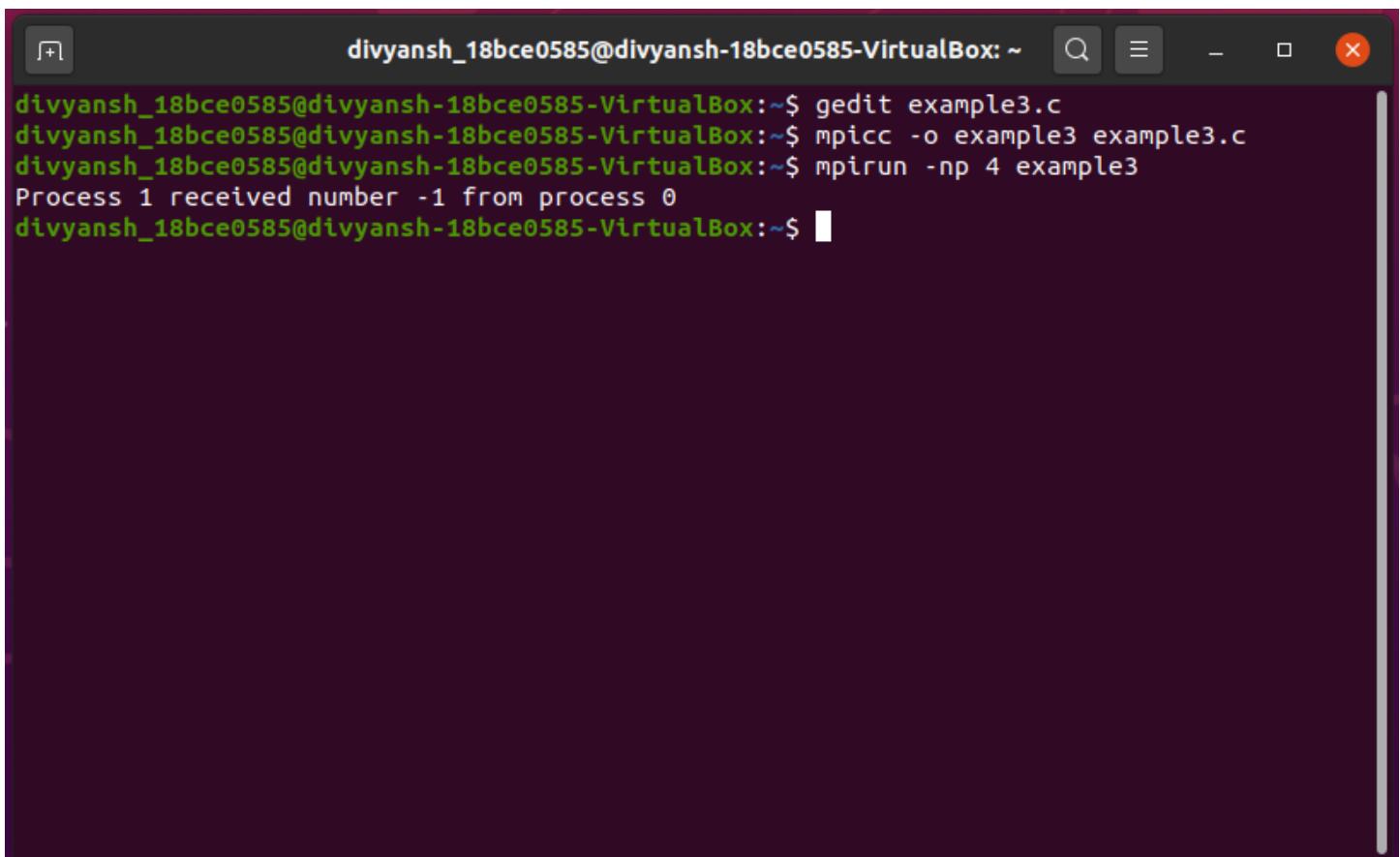
```
int world_size;  
  
MPI_Comm_size(MPI_COMM_WORLD, &world_size);  
  
int number;  
  
if (world_rank == 0) {  
  
    number = -1;  
  
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);  
  
} else if (world_rank == 1) {  
  
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,  
             MPI_STATUS_IGNORE);  
  
    printf("Process 1 received number %d from process 0\n",  
          number);  
  
}  
  
MPI_Finalize();  
}
```

OUTPUT:



A screenshot of a terminal window titled "example3.c" in the gedit text editor. The code implements MPI communication between two processes. It includes #include directives for mpi.h and stdio.h, initializes MPI, and uses MPI_Comm_rank, MPI_Comm_size, MPI_Send, MPI_Recv, and MPI_Finalize functions. The terminal shows the command "gedit example3.c" being run.

```
1 #include <mpi.h>
2 #include <stdio.h>
3 int main(int argc, char** argv) {
4
5     MPI_Init(NULL, NULL);
6
7     int world_rank;
8     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
9     int world_size;
10    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
11    int number;
12    if (world_rank == 0) {
13        number = -1;
14        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
15    } else if (world_rank == 1) {
16        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
17                  MPI_STATUS_IGNORE);
18        printf("Process 1 received number %d from process 0\n",
19               number);
20    }
21    MPI_Finalize();
22 }
```



A screenshot of a terminal window showing the execution of the MPI program. The user runs "mpicc -o example3 example3.c" and "mpirun -np 4 example3". The output shows Process 1 receiving the value -1 from Process 0, indicating successful communication between the two MPI processes.

```
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ gedit example3.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpicc -o example3 example3.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpirun -np 4 example3
Process 1 received number -1 from process 0
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$
```

Example 5:

“Broadcasting with MPI Send and MPI Recv”

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
MPI_Comm communicator)

{
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);

    int world_size;
    MPI_Comm_size(communicator, &world_size);

    if (world_rank == root) {
        int i;
        for (i = 0; i < world_size; i++) {
            if (i != world_rank) {
                MPI_Send(data, count, datatype, i, 0, communicator);
            }
        }
    }
}
```

```
    }

} else {

    MPI_Recv(data, count, datatype, root, 0, communicator,
MPI_STATUS_IGNORE);

}

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

MPI_Init(NULL, NULL);

int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

int data;

if (world_rank == 0) {

    data = 100;

    printf("Process 0 broadcasting data %d\n", data);

    my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

} else {

    my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

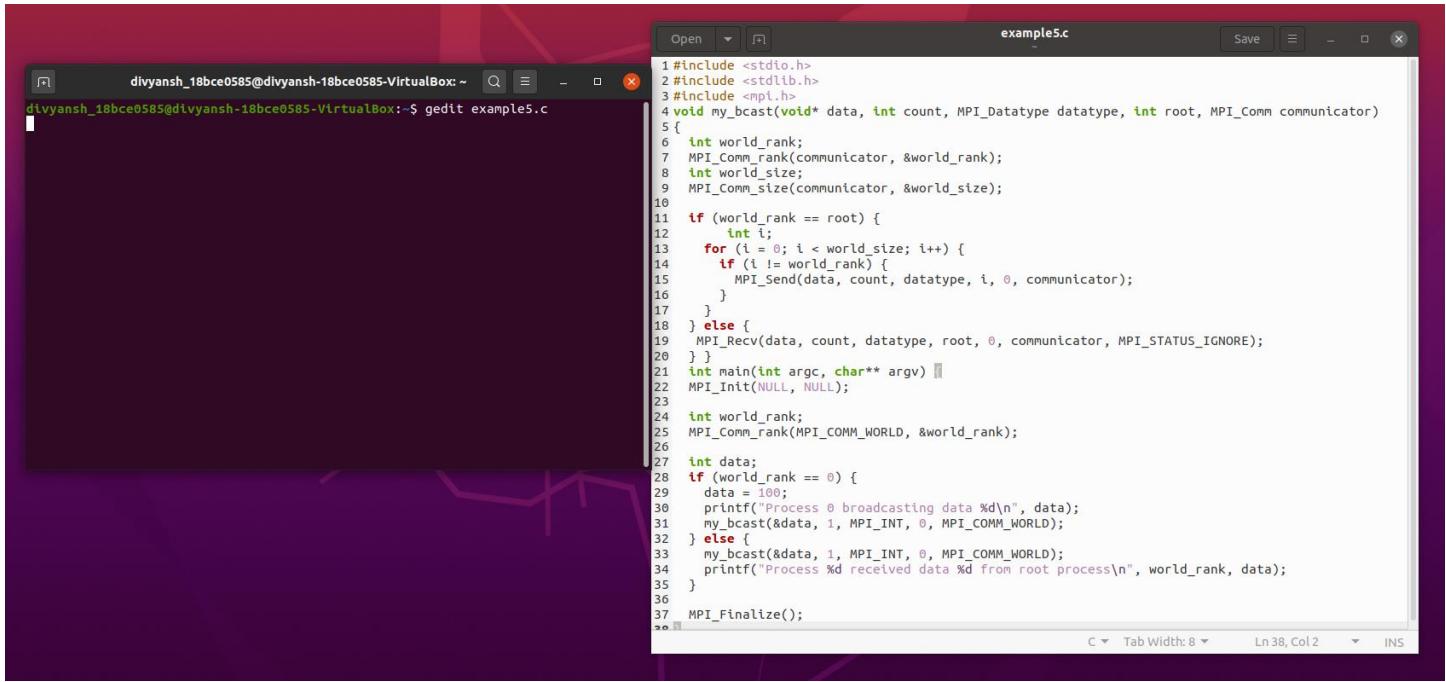
    printf("Process %d received data %d from root process\n", world_rank,
data);

}

MPI_Finalize();

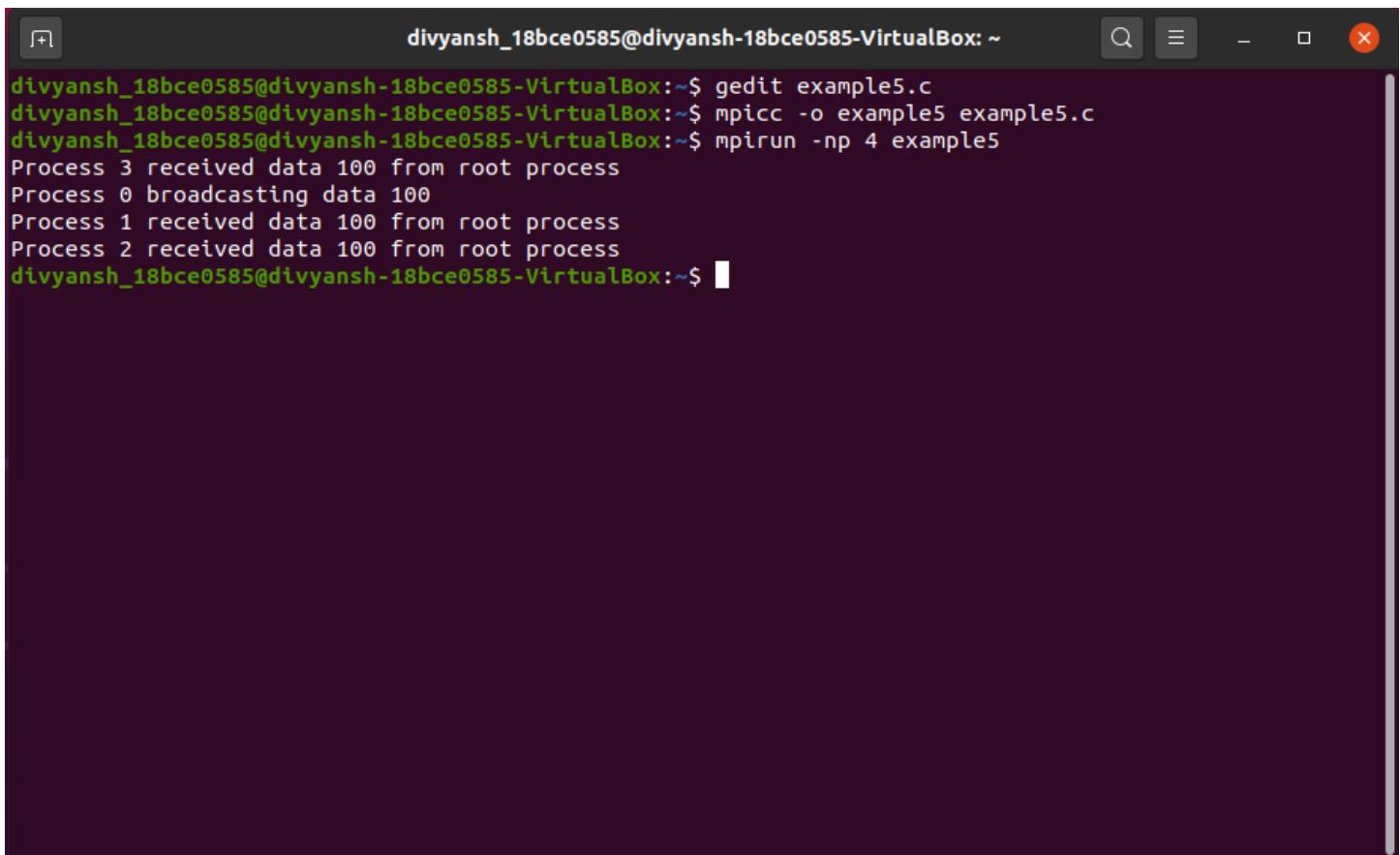
}
```

OUTPUT:



A screenshot of a terminal window titled "example5.c" in gedit. The code implements MPI_Bcast. It includes MPI_Init, MPI_Comm_rank, MPI_Send, MPI_Recv, and MPI_Finalize. The MPI_Bcast function is implemented using MPI_Send for the root process and MPI_Recv for other processes. The program prints messages indicating the source and destination of data.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 void my_bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator)
5 {
6     int world_rank;
7     MPI_Comm_rank(communicator, &world_rank);
8     int world_size;
9     MPI_Comm_size(communicator, &world_size);
10
11    if (world_rank == root) {
12        int i;
13        for (i = 0; i < world_size; i++) {
14            if (i != world_rank) {
15                MPI_Send(data, count, datatype, i, 0, communicator);
16            }
17        }
18    } else {
19        MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
20    }
21    int main(int argc, char** argv) {
22        MPI_Init(NULL, NULL);
23
24        int world_rank;
25        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
26
27        int data;
28        if (world_rank == 0) {
29            data = 100;
30            printf("Process 0 broadcasting data %d\n", data);
31            my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
32        } else {
33            my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
34            printf("Process %d received data %d from root process\n", world_rank, data);
35        }
36
37        MPI_Finalize();
38    }
```



A screenshot of a terminal window showing the execution of the MPI broadcast program. The user runs mpicc to compile example5.c, then mpirun with 4 processes. The output shows the root process broadcasting data to others, and all processes receiving it correctly.

```
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ gedit example5.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpicc -o example5 example5.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpirun -np 4 example5
Process 3 received data 100 from root process
Process 0 broadcasting data 100
Process 1 received data 100 from root process
Process 2 received data 100 from root process
```

Example 6:

“Comparison of MPI_Bcast with the my_bcast function”

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <assert.h>

void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
              MPI_Comm communicator) {
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);

    if (world_rank == root) {
        // If we are the root process, send our data to everyone
        int i;
        for (i = 0; i < world_size; i++) {
```

```

if (i != world_rank) {
    MPI_Send(data, count, datatype, i, 0, communicator);
}

}

} else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator,
    MPI_STATUS_IGNORE);
}

}

int main(int argc, char** argv) {
    if (argc != 3) {
        fprintf(stderr, "Usage: compare_bcast num_elements num_trials\n");
        exit(1);
    }

    int num_elements = atoi(argv[1]);
    int num_trials = atoi(argv[2]);

    MPI_Init(NULL, NULL);
}

```

```
int world_rank;  
  
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  
  
double total_my_bcast_time = 0.0;  
double total_mpi_bcast_time = 0.0;  
int i;  
  
int* data = (int*)malloc(sizeof(int) * num_elements);  
assert(data != NULL);  
  
for (i = 0; i < num_trials; i++) {  
    // Time my_bcast  
    // Synchronize before starting timing  
    MPI_Barrier(MPI_COMM_WORLD);  
    total_my_bcast_time -= MPI_Wtime();  
    my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);  
    // Synchronize again before obtaining final time  
    MPI_Barrier(MPI_COMM_WORLD);  
    total_my_bcast_time += MPI_Wtime();
```

```
// Time MPI_Bcast

MPI_Barrier(MPI_COMM_WORLD);
total_mpi_bcast_time -= MPI_Wtime();

MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
total_mpi_bcast_time += MPI_Wtime();

}

// Print off timing information

if (world_rank == 0) {

printf("Data size = %d, Trials = %d\n", num_elements * (int)sizeof(int),
num_trials);

printf("Avg my_bcast time = %lf\n", total_my_bcast_time / num_trials);

printf("Avg MPI_Bcast time = %lf\n", total_mpi_bcast_time /
num_trials);

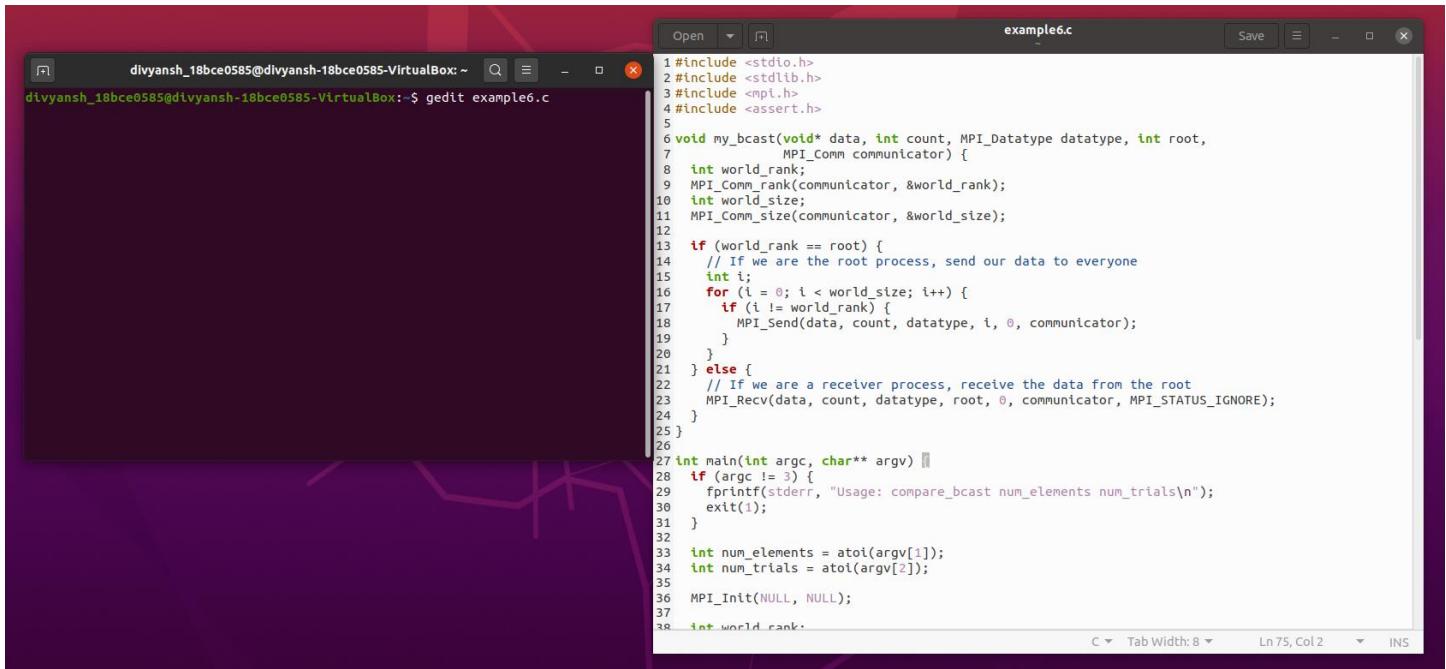
}

free(data);

MPI_Finalize();

}
```

OUTPUT:

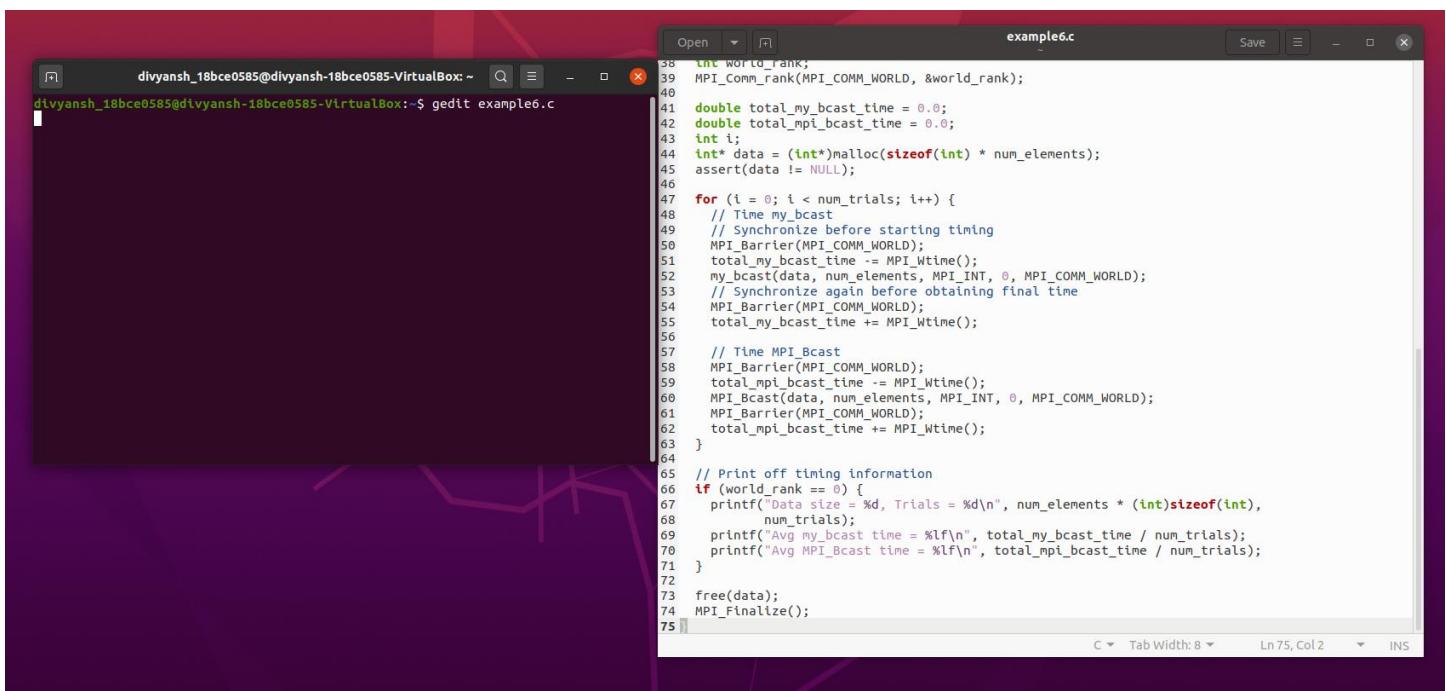


```
divyansh_18bce0585@divyansh-18bce0585-VirtualBox: ~
```

```
example6.c
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include <assert.h>
5
6 void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
7                 MPI_Comm communicator) {
8     int world_rank;
9     MPI_Comm_rank(communicator, &world_rank);
10    int world_size;
11    MPI_Comm_size(communicator, &world_size);
12
13    if (world_rank == root) {
14        // If we are the root process, send our data to everyone
15        int i;
16        for (i = 0; i < world_size; i++) {
17            if (i != world_rank) {
18                MPI_Send(data, count, datatype, i, 0, communicator);
19            }
20        }
21    } else {
22        // If we are a receiver process, receive the data from the root
23        MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
24    }
25 }
26
27 int main(int argc, char** argv) {
28     if (argc != 3) {
29         fprintf(stderr, "Usage: compare_bcast num_elements num_trials\n");
30         exit(1);
31     }
32
33     int num_elements = atoi(argv[1]);
34     int num_trials = atoi(argv[2]);
35
36     MPI_Init(NULL, NULL);
37
38     int world_rank;
```

C Tab Width: 8 Ln 75, Col 2 INS

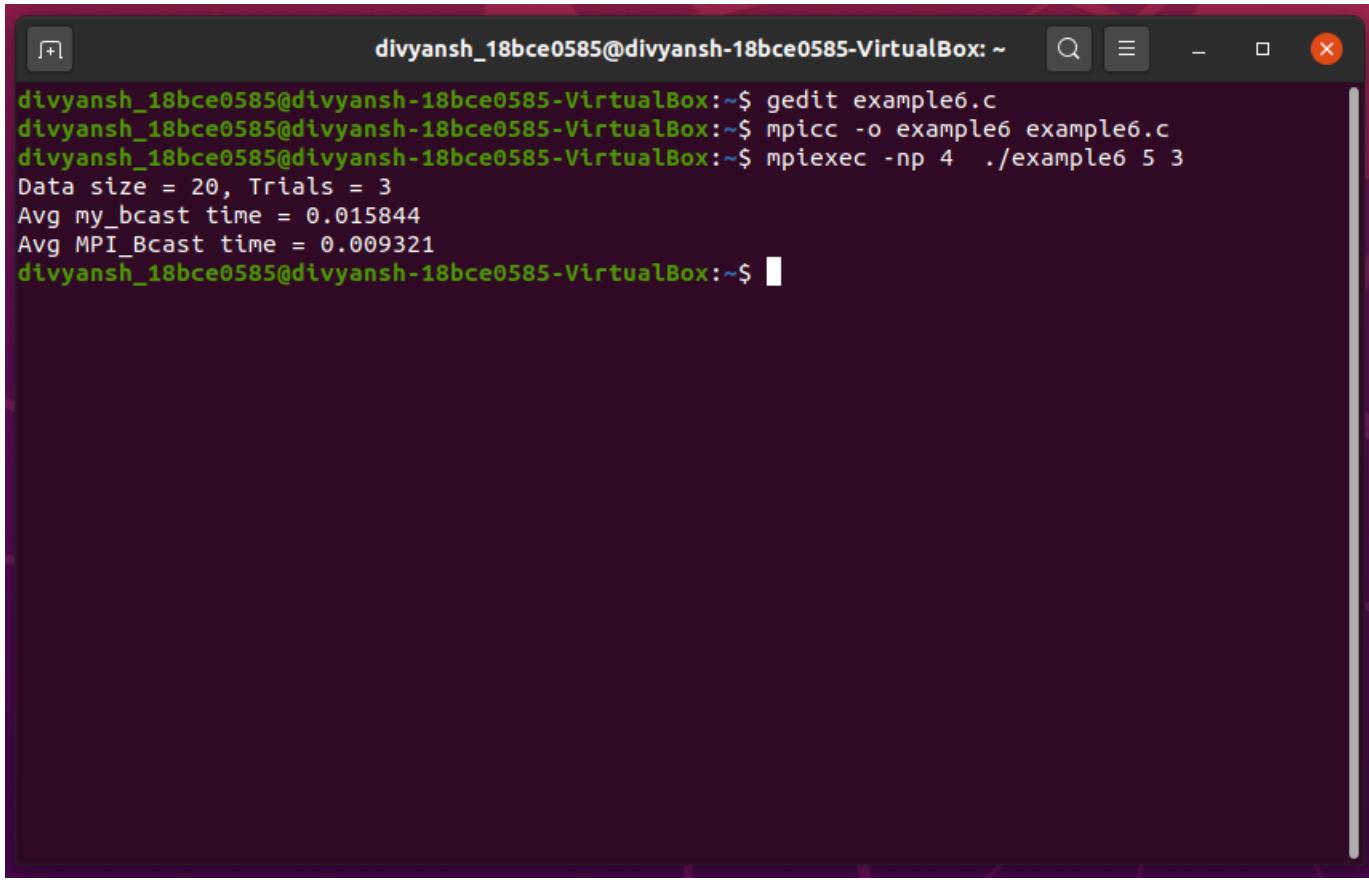


```
divyansh_18bce0585@divyansh-18bce0585-VirtualBox: ~
```

```
example6.c
```

```
38 int world_rank;
39 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
40
41 double total_my_bcast_time = 0.0;
42 double total_mpi_bcast_time = 0.0;
43 int i;
44 int* data = (int*)malloc(sizeof(int) * num_elements);
45 assert(data != NULL);
46
47 for (i = 0; i < num_trials; i++) {
48     // Time my_bcast
49     // Synchronize before starting timing
50     MPI_Barrier(MPI_COMM_WORLD);
51     total_my_bcast_time -= MPI_Wtime();
52     my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
53     // Synchronize again before obtaining final time
54     MPI_Barrier(MPI_COMM_WORLD);
55     total_my_bcast_time += MPI_Wtime();
56
57     // Time MPI_Bcast
58     MPI_Barrier(MPI_COMM_WORLD);
59     total_mpi_bcast_time -= MPI_Wtime();
60     MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
61     MPI_Barrier(MPI_COMM_WORLD);
62     total_mpi_bcast_time += MPI_Wtime();
63 }
64
65 // Print off timing information
66 if (world_rank == 0) {
67     printf("Data size = %d, Trials = %d\n", num_elements * (int)sizeof(int),
68           num_trials);
69     printf("Avg my_bcast time = %lf\n", total_my_bcast_time / num_trials);
70     printf("Avg MPI_Bcast time = %lf\n", total_mpi_bcast_time / num_trials);
71 }
72
73 free(data);
74 MPI_Finalize();
```

C Tab Width: 8 Ln 75, Col 2 INS



A screenshot of a terminal window titled "divyansh_18bce0585@divyansh-18bce0585-VirtualBox: ~". The terminal displays the following command-line session:

```
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ gedit example6.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpicc -o example6 example6.c
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$ mpiexec -np 4 ./example6 5 3
Data size = 20, Trials = 3
Avg my_bcast time = 0.015844
Avg MPI_Bcast time = 0.009321
divyansh_18bce0585@divyansh-18bce0585-VirtualBox:~$
```

Submitted By: Divyansh Chaudhary

Registration Number: 18BCE0585

Name: Arpit Bhattar

Registration Number: 18CBE0124

Course: Parallel and Distributed Computing

Dated: 24th July 2020

Assessment No: 1

AIM:

Write a simple OpenMP program to demonstrate the parallel loop construct.

a. Use OMP_SET_THREAD_NUM() and OMP_GET_THREAD_NUM() to find the number of processing unit

SOURCE CODE:

```
#include<omp.h>
#include<stdio.h>

int main(int* argc,char *argv[]) {
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello-%d",id);
        printf("world-%d \n",id);
    }
    return 0;
}
```

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E: ~
[1/1]
#include<omp.h>
#include<stdio.h>

int main(int* argc,char *argv[]) {

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello-%d",id);
        printf("world-%d \n",id);
    }
    return 0;
}
```

```
arpit@LAPTOP-D1TGCI6E:~$ nano sudo hello.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp hello.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
Hello-2world-2
Hello-1world-1
Hello-7world-7
Hello-0world-0
Hello-5world-5
Hello-3world-3
Hello-6world-6
Hello-4world-4
arpit@LAPTOP-D1TGCI6E:~$
```

b. Use function invoke to print 'Hello World'

SOURCE CODE:

```
#include<omp.h>

#include<stdio.h>

int main(int* argc,char *argv[]) {

    #pragma omp parallel
    {
        printf("Hello, ");
        printf("World\n");
    }
    return 0;
}
```

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E: ~
```

```
[1/1]
#include<omp.h>
#include<stdio.h>

int main(int* argc,char *argv[]) {

    #pragma omp parallel
    {
        printf("Hello, ");
        printf("World\n");
    }
    return 0;
}
```

```
arpit@LAPTOP-D1TGCI6E:~$ nano sudo hello.c
arpit@LAPTOP-D1TGCI6E:~$ nano sudo sample.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp sample.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
```

```
Hello, World
arpit@LAPTOP-D1TGCI6E:~$
```

c. To examine the above scenario, the functions such as `omp_get_num_procs()`, `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_in_parallel()`, `omp_get_dynamic()` and `omp_get_nested()` are listed and the explanation is given below to explore the concept practically.

`omp_set_num_threads()` - takes an integer argument and requests that the Operating System provide that number of threads in subsequent parallel regions.

SOURCE CODE:

```
#include<omp.h>
#include<stdio.h>

void main() {

    omp_set_num_threads(5);

    printf("number of threads are : %d\n",omp_get_num_threads());

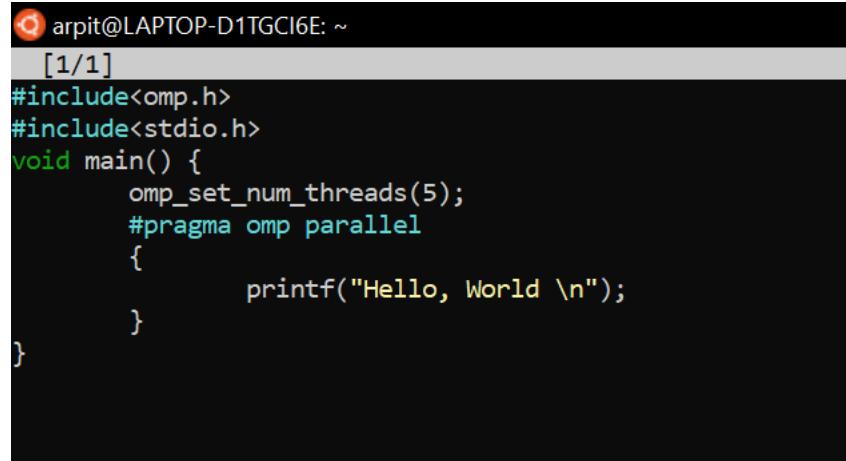
    #pragma omp parallel
    {
```

```

    printf("Hello, world \n");
    printf("Threads : %d \n",omp_get_num_threads());
}
}

```

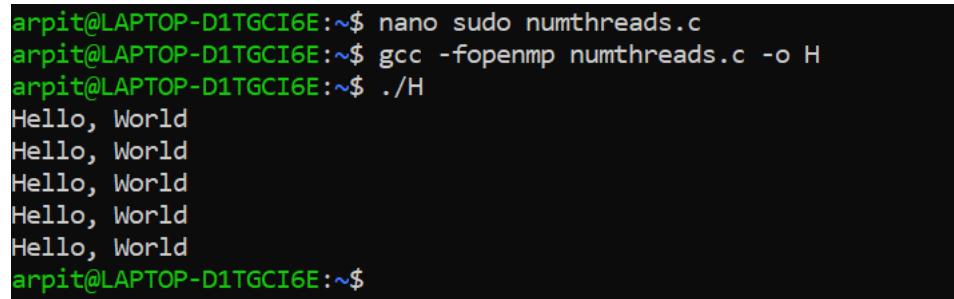
EXECUTION:



```

arpit@LAPTOP-D1TGCI6E: ~
[1/1]
#include<omp.h>
#include<stdio.h>
void main() {
    omp_set_num_threads(5);
    #pragma omp parallel
    {
        printf("Hello, World \n");
    }
}

```



```

arpit@LAPTOP-D1TGCI6E:~$ nano sudo numthreads.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp numthreads.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
Hello, World
Hello, World
Hello, World
Hello, World
Hello, World
arpit@LAPTOP-D1TGCI6E:~$ 

```

omp_get_num_threads() (integer function) - returns the actual number of threads in the current team of threads.

SOURCE CODE:

```

#include<omp.h>
#include<stdio.h>
void main() {
    omp_set_num_threads(5);
    printf("number of threads are : %d\n",omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Hello, world \n");
        printf("Threads : %d \n",omp_get_num_threads());
    }
}

```

```
}
```

```
}
```

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E: ~
```

```
[1/1]
```

```
#include<omp.h>
#include<stdio.h>
```

```
void main() {
    omp_set_num_threads(5);
    printf("number of threads are : %d\n",omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Hello, world \n");
        printf("Threads : %d \n",omp_get_num_threads());
    }
}
```

```
arpit@LAPTOP-D1TGCI6E:~$ nano sudo numthreadsint.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp numthreadsint.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
number of threads are : 1
Hello, world
Threads : 5
arpit@LAPTOP-D1TGCI6E:~$
```

omp_get_thread_num() (integer function) - returns the ID of a thread, where the ID ranges from 0 to the number of threads minus 1. The thread with the ID of 0 is the master thread.

SOURCE CODE:

```
#include<omp.h>

#include<stdio.h>

int main(int* argc,char *argv[]) {

    #pragma omp parallel

    {

        int id = omp_get_thread_num();
```

```

        printf("Hello-%d",id);

        printf("World-%d \n",id);

    }

    return 0;

}

```

```

arpit@LAPTOP-D1TGCI6E: ~
[1/1]
#include<omp.h>
#include<stdio.h>

int main(int* argc,char *argv[]) {

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello-%d",id);
        printf("World-%d \n",id);
    }
    return 0;
}

```

```

arpit@LAPTOP-D1TGCI6E:~$ nano sudo numthreadsint.c
arpit@LAPTOP-D1TGCI6E:~$ nano sudo threadnum.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp threadnum.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
Hello-3World-3
Hello-7World-7
Hello-2World-2
Hello-5World-5
Hello-1World-1
Hello-6World-6
Hello-0World-0
Hello-4World-4
arpit@LAPTOP-D1TGCI6E:~$ 

```

omp_get_num_procs() - returns the number of processors that are available when the function is called.

SOURCE CODE:

```

#include<omp.h>

#include<stdio.h>

void main() {

    printf("Processors: %d\n",omp_get_num_procs());
    #pragma omp parallel

```

```

    {
        printf("Hello, World\n");
        printf("Processors: %d\n",omp_get_num_procs());
    }
}

```

```

arpit@LAPTOP-D1TGCI6E: ~
[1/1]
#include<omp.h>
#include<stdio.h>

void main() {
    printf("Processors: %d\n",omp_get_num_procs());
    #pragma omp parallel
    {
        printf("Hello, World\n");
        printf("Processors: %d\n",omp_get_num_procs());
    }
}

```

```

arpit@LAPTOP-D1TGCI6E:~$ nano sudo numprocs.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp numprocs.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
Processors: 8
Hello, World
Processors: 8
Processors: 8
Processors: 8
arpit@LAPTOP-D1TGCI6E:~$

```

omp_get_dynamic() - returns a value that indicates if the number of threads available in subsequent parallel region can be adjusted by the run time. o ***omp_get_nested()*** returns a value that indicates if nested parallelism is enabled.

SOURCE CODE:

```
#include<omp.h>
```

```
#include<stdio.h>

void main() {
    printf("Threads available in subsequent parallel region can be adjusted by run time:
%d\n",omp_get_dynamic());

#pragma omp parallel
{
    printf("Hello, World\n");
    printf("Threads available in subsequent parallel region can be adjusted by run time: %d
\n",omp_get_dynamic());
}

}
```

EXECUTION:

```
arpit@LAPTOP-D1TGC16E: ~
[1/1]                                         getdynamic.c
#include<omp.h>
#include<stdio.h>
void main() {
    printf("Threads available in subsequent parallel region can be adjusted by run time: %d\n",omp_get_dynamic());
    #pragma omp parallel
    {
        printf("Hello, World\n");
        printf("Threads available in subsequent parallel region can be adjusted by run time: %d \n",omp_get_dynamic());
    }
}
```

REMARKS:

OpenMP is a set of compiler directives and also as an API for programs written in C, C++, or FORTRAN that gives support for parallel programming in shared-memory environments.

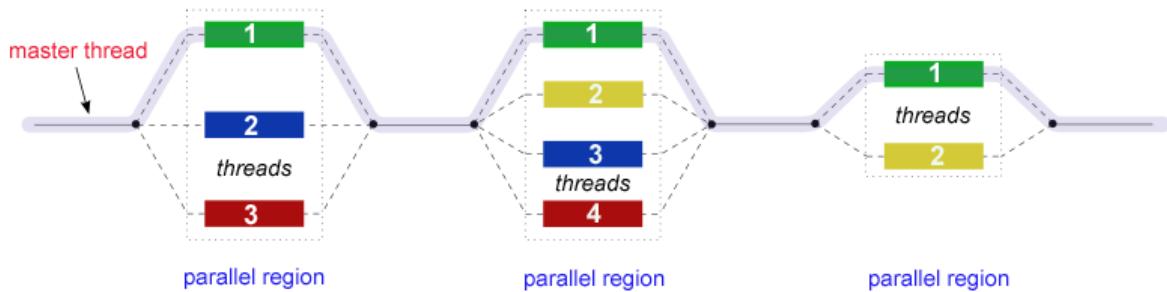
When OpenMP encounters the directive. It creates as many threads which are processing cores within the system. Thus, for a dual-core system, two threads are created, for a quad-core system, four are created; then forth.

OpenMP allows developers to settle on among several levels of parallelism.

OpenMP is accessible on several open-source and commercial compilers for Linux, Windows, and Mac OS X systems.

OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops.

Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel.



Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 31st July 2020

Assessment No: 2

AIM:

Question: Write a simple OpenMP program to demonstrate the use of ‘for’ clause.

A) Print ‘n’ array elements

SOURCE CODE:

```
#include<stdio.h>
```

```
#include<omp.h>
```

```
int main()
```

```
{
```

```
    int i;
```

```
#pragma omp parallel
```

```
{
```

```
#pragma omp for
```

```
for(int i=0;i<5;i++)
```

```
{
```

```
    printf("%d\n",i);
```

```
}
```

```
}
```

```
}
```

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E:~$ nano sudo for.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp for.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
3
2
1
4
0
arpit@LAPTOP-D1TGCI6E:~$
```

```
[1/1]
#include<stdio.h>
#include<omp.h>
int main()
{
int i;
#pragma omp parallel
{
#pragma omp for
for(int i=0;i<5;i++)
{
printf("%d\n",i);
}
}
}
```

B)Sum of n' array elements

SOURCE CODE:

```
#include<stdio.h>

#include<omp.h>

int main()

{

int i,sum=0;

int a[10]={1,2,3,4,5,6,7,8,9,10};

#pragma omp parallel

{

#pragma omp for

for(i=0;i<10;i++)

{

sum+=a[i];

printf("%d %d %d\n",i,a[i],sum);

}

}

}
```

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E:~$ nano sudo for.c
arpit@LAPTOP-D1TGCI6E:~$ nano sudo sum.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp sum.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
4 5 24
7 8 9
2 3 4
3 4 35
8 9 9
5 6 7
0 1 1
1 2 37
9 10 19
6 7 31
arpit@LAPTOP-D1TGCI6E:~$
```

```
arpit@LAPTOP-D1TGCI6E: ~
[1/1]
#include<stdio.h>
#include<omp.h>
int main()
{
int i,sum=0;
int a[10]={1,2,3,4,5,6,7,8,9,10};
#pragma omp parallel
{
#pragma omp for
for(i=0;i<10;i++)
{
sum+=a[i];
printf("%d %d %d\n",i,a[i],sum);
}
}
```

C) Product of n' array elements**SOURCE CODE:**

```
#include<stdio.h>

#include<omp.h>

int main()

{

int i=0,prod=1,a[5]={1,2,3,4,5};

#pragma omp parallel

{

#pragma omp for
```

```

for(i=0;i<5;i++)
{
prod*=a[i];
printf("%d %d %d\n",i,a[i],prod);
}
}
}

```

EXECUTION:

```

arpit@LAPTOP-D1TGCI6E:~$ nano sudo sum.c
arpit@LAPTOP-D1TGCI6E:~$ nano sudo prod.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp prod.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
1 2 40
3 4 20
4 5 5
0 1 5
2 3 3
arpit@LAPTOP-D1TGCI6E:~$
```

```

arpit@LAPTOP-D1TGCI6E: ~
[1/1]
#include<stdio.h>
#include<omp.h>
int main()
{
int i=0,prod=1,a[5]={1,2,3,4,5};
#pragma omp parallel
{
#pragma omp for
for(i=0;i<5;i++)
{
prod*=a[i];
printf("%d %d %d\n",i,a[i],prod);
}
}
```

REMARKS:

'for' is a Pragma which identifies an iterative work-sharing construct that specifies a neighborhood during which the iterations of the associated loop should be executed in parallel. Each iteration is executed by one among the threads within the team. The `omp for` directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct. Just one collapse clause is allowed on a worksharing for or parallel for pragma. the required number of loops must be present lexically. That is, none of the loops are often during a called subroutine.

The loops must form an oblong iteration space and therefore the bounds and stride of every loop must be invariant over all the loops. If the loop indices are of various size, the index with the most important size are going to be used for the collapsed loop. If the loop indices are of various size, the index with the most important size are going to be used for the collapsed loop. The loops must be perfectly nested; that's , there's no intervening code nor any OpenMP pragma between the loops which are collapsed. The associated do-loops must be structured blocks. Their execution must not be terminated by an break statement. If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop could also be curtailed by a continue statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements apart from the innermost associated loop.

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 1st August 2020

Assessment No: 3

AIM:

Write a simple OpenMP program to demonstrate the use of reduction and critical clause.

- A. Sum of 'n' array Using Reduction Clause:**
- B. Product of 'n' array using Reduction Clause.**
- C. Show a suitable example for Critical Clause**

CODE:

```
#include<stdio.h>
#include<omp.h>

int main()
{
    int n=10;
    int a[10]={1,2,3,4,5,6,7,8,9,10};
    int i,sum=0;
    double t1,t2;
    t1=omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp for reduction(+:sum)
        for(i=0;i<10;i++)
            sum+=a[i];
    }
    t2=omp_get_wtime();
    printf("%d\n",sum);
    printf("Time taken:%f\n",(t2-t1));
}
```

REMARK:

Open MP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for Addition.

Each thread will then reduce into its local variable.

At the end of the loop, the local results are combined, again using the reduction operator, into the global variable.

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E:~$ nano sudo scheduling.c
arpit@LAPTOP-D1TGCI6E:~$ nano sudo sum.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp sum.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
55
Time taken:0.000495
arpit@LAPTOP-D1TGCI6E:~$
```

```
arpit@LAPTOP-D1TGCI6E: ~
[1/1] sum.c
#include<stdio.h>
#include<omp.h>

int main()
{
    int n=10;
    int a[10]={1,2,3,4,5,6,7,8,9,10};
    int i,sum=0;
    double t1,t2;
    t1=omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp for reduction(+:sum)
        for(i=0;i<10;i++)
            sum+=a[i];
    }
    t2=omp_get_wtime();
    printf("%d\n",sum);
    printf("Time taken:%f\n", (t2-t1));
}
```

B. Product of 'n' array using Reduction Clause.**CODE:**

```
#include <stdio.h>
#include <omp.h>

int main()
{
    int product = 1;
    int arr[] = {1,2,3,4,5,6,7,8,9};
    int n = 9;
```

```
//OpenMP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for multiplication.
```

```
#pragma omp parallel for reduction(*:product)
for(int i=0;i<n;i++)
product *= arr[i];
printf("The total product using reduction and critical clause is: \n %d \n",product);
return 0;
}
```

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E:~$ nano sudo pd.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp pd.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
The total product using reduction and critical clause is:
362880
arpit@LAPTOP-D1TGCI6E:~$
```

```
arpit@LAPTOP-D1TGCI6E: ~
[1/1] pd.c
#include <stdio.h>
#include <omp.h>
int main()
{
    int product = 1;
    int arr[] = {1,2,3,4,5,6,7,8,9};
    int n = 9;
//OpenMP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for multiplication.
#pragma omp parallel for reduction(*:product)
    for(int i=0;i<n;i++)
        product *= arr[i];
    printf("The total product using reduction and critical clause is: \n %d \n",product);
    return 0;
}
```

REMARK:

Open MP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for Multiplication (Product).

Each thread will then reduce into its local variable.

At the end of the loop, the local results are combined, again using the reduction operator, into the global variable.

(C)

SOURCE CODE:

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int arr[] = {11,100,1,27,21,200,32,87,310};
```

```

int n = 9;

int min_val = 999999;

int max_val = 0;

#pragma omp parallel for

for(int i=0;i<n;i++)

{

#pragma omp critical

{

if(arr[i] < min_val)

min_val = arr[i]; //Min value

}

#pragma omp critical

{

if(arr[i] > max_val)

max_val = arr[i]; //max element

}

}

printf("The minimum element in the array is %d\n",min_val);

printf("The maximum element in the array is %d\n",max_val);

return 0;

}

```

EXECUTION:

```

arpit@LAPTOP-D1TGCI6E:~$ nano sudo pd.c
arpit@LAPTOP-D1TGCI6E:~$ nano sudo cc.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp cc.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
The minimum element in the array is 1
The maximum element in the array is 310
arpit@LAPTOP-D1TGCI6E:~$
```

```
arpit@LAPTOP-D1TGCI6E: ~
[1/1] cc.c
#include <stdio.h>
#include <omp.h>
int main()
{
    int arr[] = {11,100,1,27,21,200,32,87,310};
    int n = 9;
    int min_val = 999999;
    int max_val = 0;
    #pragma omp parallel for
    for(int i=0;i<n;i++)
    {
        #pragma omp critical
        {
            if(arr[i] < min_val)
                min_val = arr[i]; //Min value
        }
        #pragma omp critical
        {
            if(arr[i] > max_val)
                max_val = arr[i]; //max element
        }
    }
    printf("The minimum element in the array is %d\n",min_val);
    printf("The maximum element in the array is %d\n",max_val);
    return 0;
}
```

REMARK:

OpenMP uses the critical clause to find the maximum and minimum element in the array using omp parallel using the for loop from the array of elements.

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 17th August 2020

Assessment No: 4

AIM:

Write a simple OpenMP program to demonstrate Arithmetic Operation using Section Clause

SEQUENTIAL FLOW

SOURCE CODE:

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>

void odd_elems_sum(int arr[], int N){
    int sum = 0;
    int prod = 1;
    for(int i=0; i<N; i++){
        if(i%2!=0)
        {
            sum += arr[i];
            prod *= arr[i];
            printf("Adding element %d: sum=%d \n", i, sum);
            printf("Multiplying element %d: product=%d \n", i , prod);
            sleep(1);
        }
    }
}

void even_elems_sum(int arr[], int N){
    int sum = 0;
    int prod = 1;
    for(int i=0; i<N; i++){

```

```

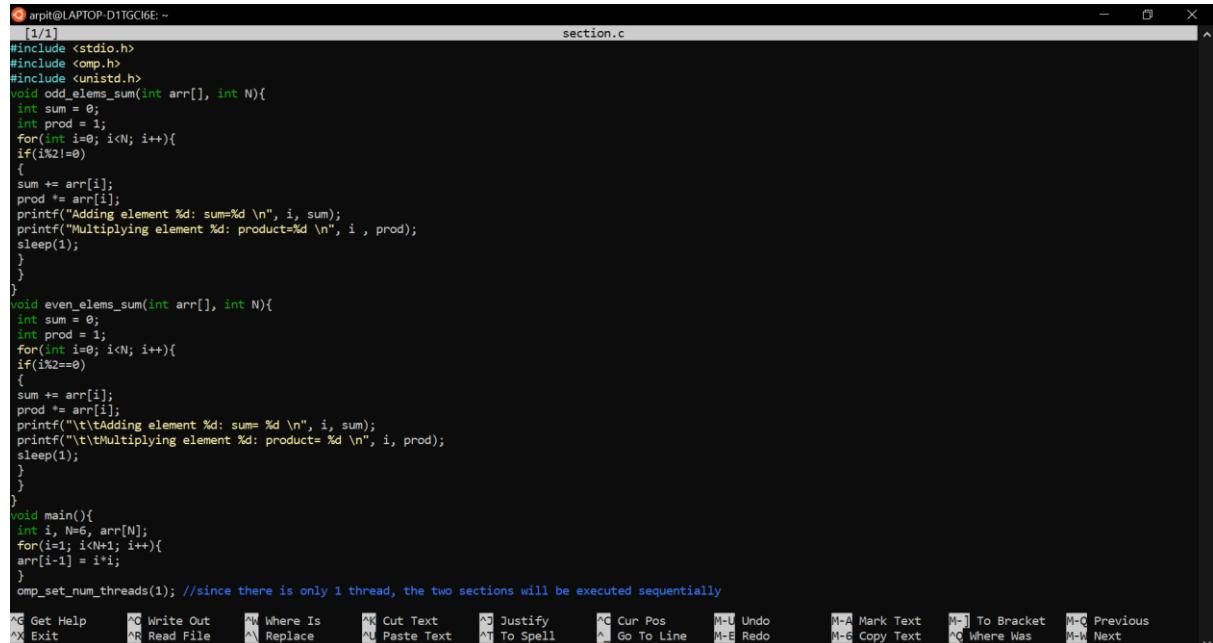
if(i%2==0)
{
    sum += arr[i];
    prod *= arr[i];
    printf("\t\tAdding element %d: sum= %d \n", i, sum);
    printf("\t\tMultiplying element %d: product= %d \n", i, prod);
    sleep(1);
}
}

void main(){
    int i, N=6, arr[N];
    for(i=1; i<N+1; i++){
        arr[i-1] = i*i;
    }
    omp_set_num_threads(1); //since there is only 1 thread, the two sections will be executed
    sequentially
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            odd_elems_sum(arr, N);
        }
        #pragma omp section
        {
            even_elems_sum(arr, N);
        }
    }
}

```

EXECUTION:

```
arpit@LAPTOP-D1TGC16E:~$ nano sudo section.c
arpit@LAPTOP-D1TGC16E:~$ gcc -fopenmp section.c -o H
arpit@LAPTOP-D1TGC16E:~$ ./H
Adding element 1: sum=4
Multiplying element 1: product=4
Adding element 3: sum=20
Multiplying element 3: product=64
Adding element 5: sum=56
Multiplying element 5: product=2304
          Adding element 0: sum= 1
          Multiplying element 0: product= 1
          Adding element 2: sum= 10
          Multiplying element 2: product= 9
          Adding element 4: sum= 35
          Multiplying element 4: product= 225
arpit@LAPTOP-D1TGC16E:~$
```



```
arpit@LAPTOP-D1TGC16E:~[1/1] section.c
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
void odd_elems_sum(int arr[], int N){
    int sum = 0;
    int prod = 1;
    for(int i=0; i<N; i++){
        if(i%2!=0)
        {
            sum += arr[i];
            prod *= arr[i];
            printf("Adding element %d: sum=%d \n", i, sum);
            printf("Multiplying element %d: product=%d \n", i , prod);
            sleep(1);
        }
    }
}
void even_elems_sum(int arr[], int N){
    int sum = 0;
    int prod = 1;
    for(int i=0; i<N; i++){
        if(i%2==0)
        {
            sum += arr[i];
            prod *= arr[i];
            printf("\t\tAdding element %d: sum= %d \n", i, sum);
            printf("\t\tMultiplying element %d: product=%d \n", i, prod);
            sleep(1);
        }
    }
}
void main(){
    int i, N=6, arr[N];
    for(i=1; i<N+1; i++){
        arr[i-1] = i*i;
    }
    omp_set_num_threads(1); //since there is only 1 thread, the two sections will be executed sequentially

```

PARALLEL FLOW

SOURCE CODE:

```
#include <stdio.h>

#include <omp.h>

#include <unistd.h>

void odd_elems_sum(int arr[], int N){

    int sum = 0;

    int prod = 1;

    for(int i=0; i<N; i++){

        if(i%2!=0)
```

```

{
    sum += arr[i];
    prod *= arr[i];
    printf("Adding element %d: sum=%d \n", i, sum);
    printf("Multiplying element %d: product=%d \n", i , prod);
    sleep(1);
}

}

}

void even_elems_sum(int arr[], int N){

int sum = 0;
int prod = 1;

for(int i=0; i<N; i++){
    if(i%2==0)

    {
        sum += arr[i];
        prod *= arr[i];
        printf("\t\tAdding element %d: sum= %d \n", i, sum);
        printf("\t\tMultiplying element %d: product= %d \n", i, prod);
        sleep(1);
    }
}
}

void main(){

int i, N=6, arr[N];

for(i=1; i<N+1; i++){
    arr[i-1] = i*i;
}

omp_set_num_threads(3); //since there is only 3 thread, the two sections will be executed
PARALLEL
}

```

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        odd_elems_sum(arr, N);
    }
    #pragma omp section
    {
        even_elems_sum(arr, N);
    }
}
```

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp parallel.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
            Adding element 0: sum= 1
            Multiplying element 0: product= 1
Adding element 1: sum=4
Multiplying element 1: product=4
Adding element 3: sum=20
Multiplying element 3: product=64
            Adding element 2: sum= 10
            Multiplying element 2: product= 9
            Adding element 4: sum= 35
            Multiplying element 4: product= 225
Adding element 5: sum=56
Multiplying element 5: product=2304
arpit@LAPTOP-D1TGCI6E:~$
```

```
arpit@LAPTOP-D1TGCJ6E: ~
[1/1]                                     parallel1.c
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
void odd_elems_sum(int arr[], int N){
    int sum = 0;
    int prod = 1;
    for(int i=0; i<N; i++){
        if(i%2!=0)
        {
            sum += arr[i];
            prod *= arr[i];
            printf("Adding element %d: sum=%d \n", i, sum);
            printf("Multiplying element %d: product=%d \n", i , prod);
            sleep(1);
        }
    }
}
void even_elems_sum(int arr[], int N){
    int sum = 0;
    int prod = 1;
    for(int i=0; i<N; i++){
        if(i%2==0)
        {
            sum += arr[i];
            prod *= arr[i];
            printf("\t\tAdding element %d: sum= %d \n", i, sum);
            printf("\t\tMultiplying element %d: product= %d \n", i , prod);
            sleep(1);
        }
    }
}
void main(){
    int i, N=6, arr[N];
    for(i=1; i<=N; i++){
        arr[i-1] = i*i;
    }
    omp_set_num_threads(3); //since there is only 3 thread, the two sections will be executed PARALLELLY
}

[ parallel1.c -- 50 lines ]
[G Get Help   ^G Write Out   ^W Where Is   ^K Cut Text   ^J Justify   ^C Cur Pos   M-U Undo   M-A Mark Text   M-] To Bracket   M-Q Previous
^X Exit      ^R Read File   ^\ Replace   ^U Paste Text   ^I To Spell   ^L Go To Line   M-E Redo   M-G Copy Text   ^Q Where Was   M-W Next
```

REMARK:

The sections clause is used to indicate that the particular parallel section will be having multiple subsections that are to be executed in parallel. The section clause identifies these subsections. In the experiment conducted above we split the addition of an array into a sum and product of odd elements and even elements respectively and conducted them in parallel. It was clearly depicted that when the processor had enough number of threads (\geq number of sections), the sections were processed parallelly (as seen in the printed lines), but when the number of threads $<$ number of sections, the two sections were executed sequentially. Section clause can be very useful when different parts of the same logic can be conducted in parallel, enhancing time efficiency. As seen in the experiment above, when the two sections were conducted sequentially, the processor was idle for 1 second between each print statement (due to the sleep(1) command), but when executed parallelly, when one section was idle, the other section took up the CPU time, thus decreasing the total time spent on the two sections together.

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 28th August 2020

Assessment No: 5

AIM:

Write a simple OpenMP program to demonstrate the use of pattern generation in schedule clause

* * * * *

* * * * *

* * * * *

* * * * *

* * * * *

A. Statically assign the loop iterations to threads

B. Dynamically assign one iteration to each threads

(A)

SOURCE CODE:

```
#include<omp.h>
#include<stdio.h>
void a(int i)
{
    printf("*");
}
void main()
{
    int k; int j;
#pragma omp parallel for schedule(static, 5)
    for(int k=0;k<5;k++)
    {
        for(int j=0;j<6;j++)
        {

```

```

    a(k);
}

printf("\n");
}

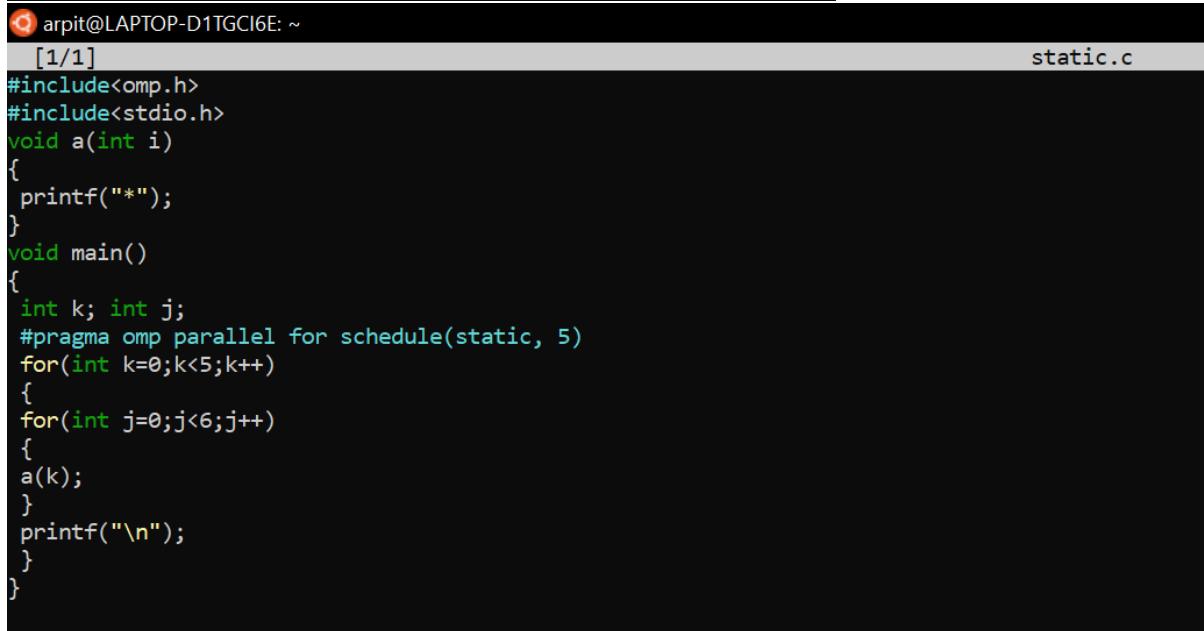
}

```

EXECUTION:

```

arpit@LAPTOP-D1TGCI6E:~$ nano sudo parallel.c
arpit@LAPTOP-D1TGCI6E:~$ nano sudo static.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp static.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
*****
*****
*****
*****
*****
*****
```



```

arpit@LAPTOP-D1TGCI6E: ~
[1/1]                                         static.c
#include<omp.h>
#include<stdio.h>
void a(int i)
{
    printf("*");
}
void main()
{
    int k; int j;
#pragma omp parallel for schedule(static, 5)
    for(int k=0;k<5;k++)
    {
        for(int j=0;j<6;j++)
        {
            a(k);
        }
        printf("\n");
    }
}
```

(B)

SOURCE CODE:

```

#include<omp.h>

#include<stdio.h>

void a(int i)

{
    printf("*");
}
```

```
void main()
{
    int k; int j;
    #pragma omp parallel for schedule(dynamic, 6)
    for(int k=0;k<5;k++)
    {
        for(int j=0;j<6;j++)
        {
            a(k);
        }
        printf("\n");
    }
}
```

EXECUTION:

```
arpit@LAPTOP-D1TGCI6E:~$ nano sudo static.c
arpit@LAPTOP-D1TGCI6E:~$ nano sudo scheduling.c
arpit@LAPTOP-D1TGCI6E:~$ gcc -fopenmp scheduling.c -o H
arpit@LAPTOP-D1TGCI6E:~$ ./H
*****
*****
*****
*****
*****
*****
```

```
arpit@LAPTOP-D1TGCI6E: ~
[1/1]                                     scheduling.c
#include<omp.h>
#include<stdio.h>
void a(int i)
{
    printf("*");
}
void main()
{
    int k; int j;
#pragma omp parallel for schedule(dynamic, 6)
    for(int k=0;k<5;k++)
    {
        for(int j=0;j<6;j++)
        {
            a(k);
        }
        printf("\n");
    }
}
```

REMARK:

Static:

Here the execution takes place row wise hence we specify 5 along with static clause to schedule 5 rows in the process of execution. It means that to each of the threads will be assigned 5 contiguous iterations. In this case the first thread will take 5 numbers. The second one will take another 5 and so on until there are no more data to process or the maximum number of threads is reached (typically equal to the number of cores). Sharing of workload is done during the compilation.

Dynamic:

Here the execution takes place column wise hence we specify 6 along with dynamic clause. The work will be shared amongst threads but this procedure will occur at a runtime. Thus involving more overhead. Second parameter specifies size of the chunk of the data. Not being very familiar to OpenMP I risk to assume that dynamic type is more appropriate when compiled code is going to run on the system that has a different configuration than the one on which code was compiled.

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 4th September 2020

Assessment No: 6

AIM:

Consider a suitable instance that has MPI routines to assign different tasks to different processors.

For example, parts of an input data set might be divided and processed by different processors, or a finite difference grid might be divided among the processors available. This means that the code needs to identify processors. In this example, processors are identified by rank - an integer from 0 to total number of processors.

- 1. Implement the logic using C**
- 2. Build the code**
- 3. Show the screenshots with proper justification**

SOURCE CODE:

```
#include<stdio.h>
#include<mpi.h>

int main(int argc, char **argv)
{
    int my_rank; int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello world! I'm rank (processor number) %d of size %d\n", my_rank, size);
    MPI_Finalize();
}
```

EXECUTION:

```

arpit@LAPTOP-D1TGC16E: ~
arpit@LAPTOP-D1TGC16E:~$ nano sudo mpi.c
arpit@LAPTOP-D1TGC16E:~$ nano sudo m1.c
arpit@LAPTOP-D1TGC16E:~$ mpicc m1.c -o H
arpit@LAPTOP-D1TGC16E:~$ mpirun -np 3 ./H
-----
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

Local host: LAPTOP-D1TGC16E
-----
Hello world! I'm rank (processor number) 0 of size 3
Hello world! I'm rank (processor number) 1 of size 3
Hello world! I'm rank (processor number) 2 of size 3
[LAPTOP-D1TGC16E:0156] 2 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[LAPTOP-D1TGC16E:0156] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
arpit@LAPTOP-D1TGC16E:~$ 


```

```

arpit@LAPTOP-D1TGC16E: ~
[1/1] m1.c
#include<stdio.h>
#include<mpi.h>
int main(int argc, char **argv)
{
    int my_rank; int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello world! I'm rank (processor number) %d of size %d\n", my_rank, size);
    MPI_Finalize();
}


```

REMARK:

An MPI process can query a communicator for information about the group, with MPI_COMM_SIZE and MPI_COMM_RANK. MPI_COMM_RANK (comm, rank) - MPI_COMM_RANK returns the rank of the calling process in the group associated with the communicator. MPI_COMM_SIZE (comm, size) - MPI_COMM_SIZE returns in size the number of processes in the group associated with the communicator. In the screenshot attached above, we see that “Hello World!” was printed on the screen using 4 different processors in parallel, and the rank of each of them being listed.

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 11th September 2020

Assessment No: 7

Aim: Consider the following program, called *mpi_sample1.c*. This program is written in C with MPI commands included.

The new MPI calls are to *MPI_Send* and *MPI_Recv* and to *MPI_Get_processor_name*. The latter is a convenient way to get the name of the processor on which a process is running. *MPI_Send* and *MPI_Recv* can be understood by stepping back and considering the two requirements that must be satisfied to communicate data between two processes:

1. Describe the data to be sent or the location in which to receive the data
2. Describe the destination (for a send) or the source (for a receive) of the data.

SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
#include"mpi.h"

int main(int argc, char* argv[]){
    int my_rank;
    int p;
    int source;
    int dest;
    int tag=0;
    char message[100];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if(my_rank != 0){
        sprintf(message,"Hello MPI World from process %d!",my_rank);
        dest = 0;
```

```

    MPI_Send(message,strlen(message)+1,MPI_CHAR,dest,tag,MPI_COMM_WORLD);

}

else{

    printf("Hello MPI World from process 0: Num processes : %d\n",p);

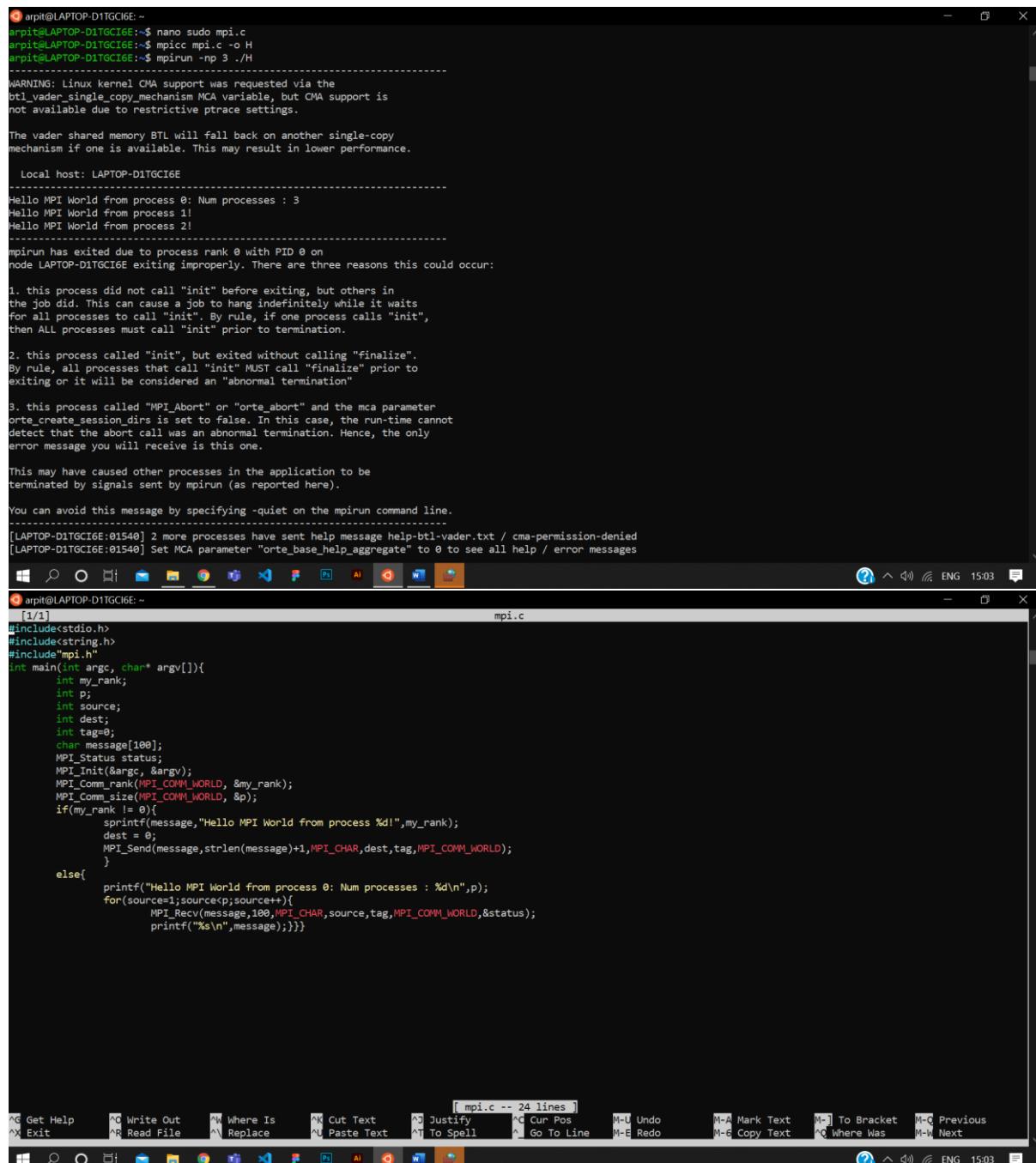
    for(source=1;source<p;source++){

        MPI_Recv(message,100,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);

        printf("%s\n",message);}}}

```

EXECUTION:



```

arpit@LAPTOP-DITGC16E: ~
arpit@LAPTOP-DITGC16E:~$ nano sudo mpi.c
arpit@LAPTOP-DITGC16E:~$ mpicc mpi.c -o H
arpit@LAPTOP-DITGC16E:~$ mpirun -np 3 ./H
-----
WARNING: Linux kernel OMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but OMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

Local host: LAPTOP-DITGC16E
-----
Hello MPI World from process 0: Num processes : 3
Hello MPI World from process 1!
Hello MPI World from process 2!
-----
mpirun has exited due to process rank 0 with PID 0 on
node LAPTOP-DITGC16E exiting improperly. There are three reasons this could occur:

1. this process did not call "init" before exiting, but others in
the job did. This can cause a job to hang indefinitely while it waits
for all processes to call "init". By rule, if one process calls "init",
then ALL processes must call "init" prior to termination.

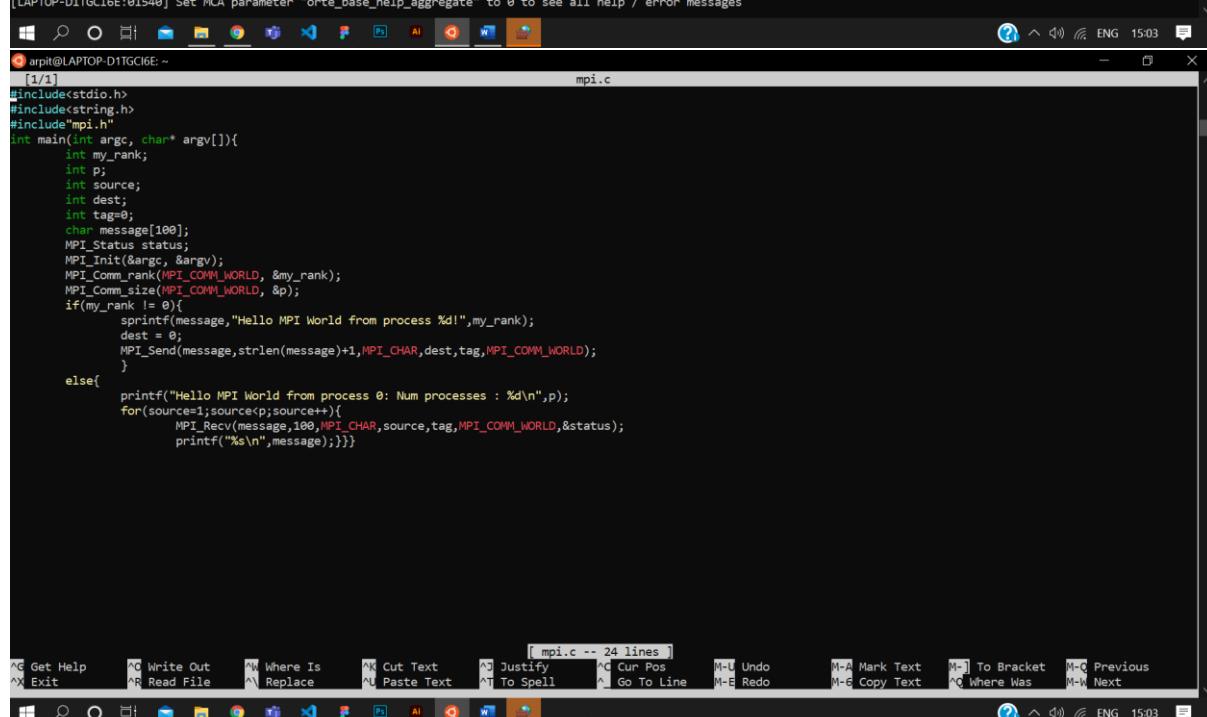
2. this process called "init", but exited without calling "finalize".
By rule, all processes that call "init" MUST call "finalize" prior to
exiting or it will be considered an "abnormal termination"

3. this process called "MPI_Abort" or "orte_abort" and the mca parameter
orte_create_session_dirs is set to false. In this case, the run-time cannot
detect that the abort call was an abnormal termination. Hence, the only
error message you will receive is this one.

This may have caused other processes in the application to be
terminated by signals sent by mpirun (as reported here).

You can avoid this message by specifying -quiet on the mpirun command line.
-----
[LAPTOP-DITGC16E:01540] 2 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[LAPTOP-DITGC16E:01540] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages

```



```

arpit@LAPTOP-DITGC16E: ~
[1/1] mpi.c
#include<stdio.h>
#include<string.h>
#include<mpi.h>
int main(int argc, char* argv[]){
    int my_rank;
    int p;
    int source;
    int dest;
    int tag=0;
    char message[100];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if(my_rank != 0){
        sprintf(message,"Hello MPI World from process %d!",my_rank);
        dest = 0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
    }
    else{
        printf("Hello MPI World from process 0: Num processes : %d\n",p);
        for(source=1;source<p;source++){
            MPI_Recv(message,100,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
            printf("%s\n",message);}}}

```

REMARKS:

- MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.
- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.
- MPI's send and receive calls operate in the following manner. First, process A decides a message needs to be sent to process B. Process A then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as envelopes since the data is being packed into a single message before transmission (similar to how letters are packed into envelopes before transmission to the post office). After the data is packed into a buffer, the communication device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank.
- Even though the message is routed to B, process B still has to acknowledge that it wants to receive A's data. Once it does this, the data has been transmitted. Process A is acknowledged that the data has been transmitted and may go back to work.
- Sometimes there are cases when A might have to send many different types of messages to B. Instead of B having to go through extra measures to differentiate all these messages, MPI allows senders and receivers to also specify message IDs with the message (known as tags). When process B only requests a message with a certain tag number, messages with different tags will be buffered by the network until B is ready for them.

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 18th September 2020

Assessment No: 8

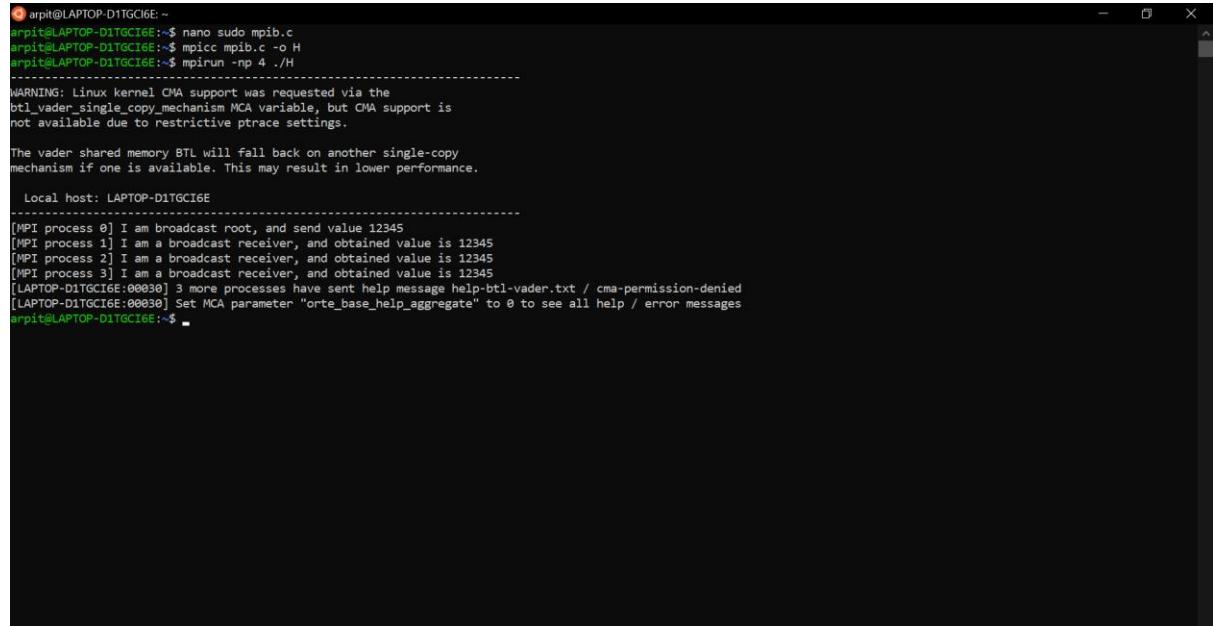
Aim: Write a C program to demonstrate the use of MPI broadcasting

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>

int main(int argc,char* argv[])
{
    MPI_Init(&argc,&argv);
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    int broadcast_root=0;
    int buffer;
    if(my_rank==broadcast_root){
        buffer=12345;
        printf("[MPI process %d] I am broadcast root, and send value %d\n",my_rank,buffer);
    }
    MPI_Bcast(&buffer,1,MPI_INT,broadcast_root,MPI_COMM_WORLD);
    if(my_rank!=broadcast_root){
        printf("[MPI process %d] I am a broadcast receiver, and obtained value is %d\n",my_rank,buffer);
    }
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

EXECUTION:



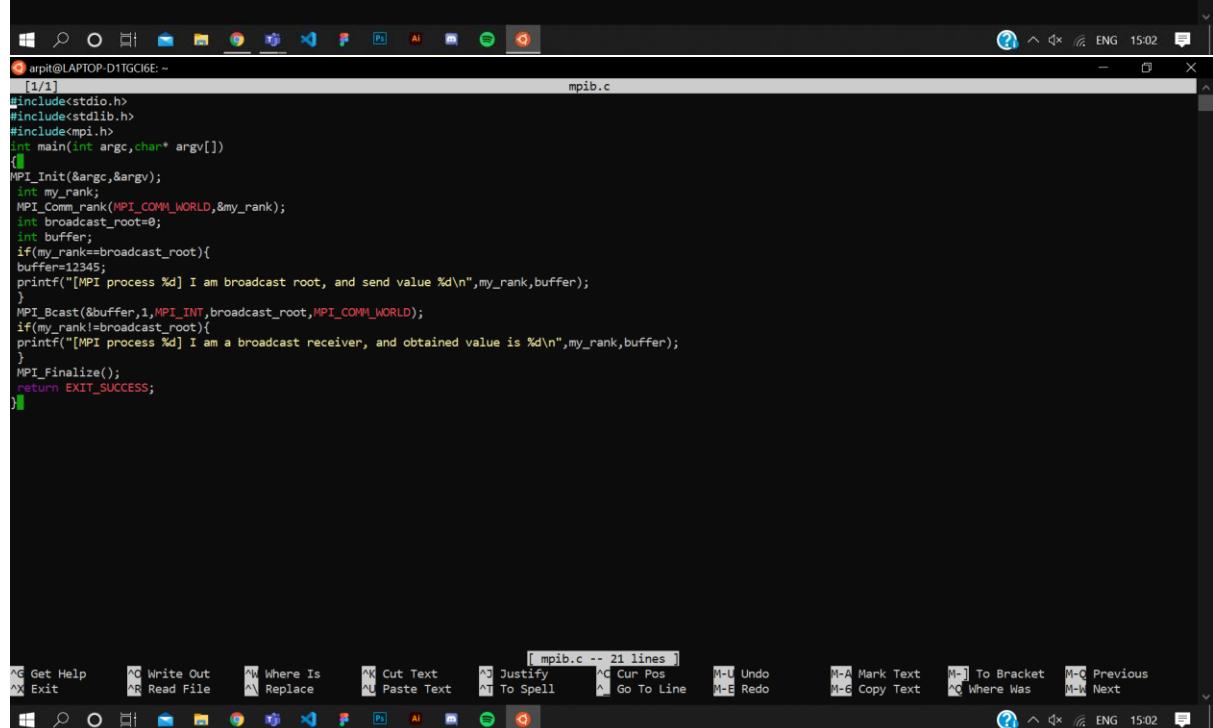
arpit@LAPTOP-D1TGC16E: ~

```
arpit@LAPTOP-D1TGC16E:~$ nano sudo mpib.c
arpit@LAPTOP-D1TGC16E:~$ mpicc mpib.c -o H
arpit@LAPTOP-D1TGC16E:~$ mpirun -np 4 ./H

WARNING: Linux kernel OMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but OMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

Local host: LAPTOP-D1TGC16E
-----
[MPI process 0] I am broadcast root, and send value 12345
[MPI process 1] I am a broadcast receiver, and obtained value is 12345
[MPI process 2] I am a broadcast receiver, and obtained value is 12345
[MPI process 3] I am a broadcast receiver, and obtained value is 12345
[MPI process 0] 3 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[LAPTOP-D1TGC16E:0003@] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
arpit@LAPTOP-D1TGC16E:~$
```

```
[1/1] mpib.c
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
int main(int argc,char* argv[])
{
MPI_Init(&argc,&argv);
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
int broadcast_root=0;
int buffer;
if(my_rank==broadcast_root){
buffer=12345;
printf("[MPI process %d] I am broadcast root, and send value %d\n",my_rank,buffer);
}
MPI_Bcast(&buffer,1,MPI_INT,broadcast_root,MPI_COMM_WORLD);
if(my_rank!=broadcast_root){
printf("[MPI process %d] I am a broadcast receiver, and obtained value is %d\n",my_rank,buffer);
}
MPI_Finalize();
return EXIT_SUCCESS;
}
```

REMARKS:

A broadcast is one among the quality collective communication techniques. During a broadcast, one process sends an equivalent data to all or any processes during a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes. In MPI, broadcasting can be accomplished by using MPI_Bcast. Although the basis process and receiver processes do different jobs, all of them call an equivalent MPI_Bcast function. When the basis process (in our example, it had been process zero) calls MPI_Bcast, the info variable are going to be sent to all or any other processes. When all of the receiver processes call MPI_Bcast, the info variable are going to be filled in with the info from the basis process.

Name: Arpit Bhattar

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 25th September 2020

Assessment No: 9

Aim: Write a C program to use MPI_Reduce that divides the processors into the group to find the addition independently.

SOURCE CODE:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
    int arr[] = {1,2,3,4,5,6,7,8}, n=8, sum, local_sum, numprocs, myid, i;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    local_sum = 0;
    printf("Proc: %d\n", myid);
    for(i=0;i<2;i++){
        printf("Adding... arr[%d]=%d\n", (myid*2)+i, arr[(myid*2)+i]);
        local_sum += arr[(myid*2)+i];
    }
    printf("local sum for proc:%d is %d\n\n", myid, local_sum);
    MPI_Reduce(&local_sum, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if(myid == 0){
        printf("final sum is %d\n", sum);
    }
    MPI_Finalize();
```

```
return 0;
```

```
}
```

EXECUTION:

```
arpit@LAPTOP-DITGC16E: ~
[1/1]                               mpirreduce.c
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main(int argc, char *argv[])
{
    int arr[] = {1,2,3,4,5,6,7,8}, n=8, sum, local_sum, numprocs, myid, i;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    local_sum = 0;
    printf("Proc: %d\n", myid);
    for(i=0;i<2;i++){
        printf("Adding... arr[%d]=%d\n", (myid*2)+i, arr[(myid*2)+i]);
        local_sum += arr[(myid*2)+i];
    }
    printf("local sum for proc:%d is %d\n", myid, local_sum);
    MPI_Reduce(&local_sum, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if(myid == 0){
        printf("Final sum is %d\n", sum);
    }
    MPI_Finalize();
    return 0;
}

[ mpirreduce.c -- 24 lines ]
arpit@LAPTOP-DITGC16E: ~
To check for new updates run: sudo apt update

This message is shown once every day. To disable it please create the
/home/arpit/.hushlogin file.
arpit@LAPTOP-DITGC16E: $ nano sudo mpirreduce.c
arpit@LAPTOP-DITGC16E: $ mpicc mpirreduce.c -o H
arpit@LAPTOP-DITGC16E: $ mpirun -np 4 ./H
-----
WARNING: Linux kernel DMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but DMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

Local host: LAPTOP-DITGC16E
-----
Proc: 0
Adding... arr[0]=1
Adding... arr[1]=2
local sum for proc:0 is 3

Proc: 1
Adding... arr[2]=3
Adding... arr[3]=4
local sum for proc:1 is 7

Proc: 2
Adding... arr[4]=5
Adding... arr[5]=6
local sum for proc:2 is 11

Proc: 3
Adding... arr[6]=7
Adding... arr[7]=8
local sum for proc:3 is 15

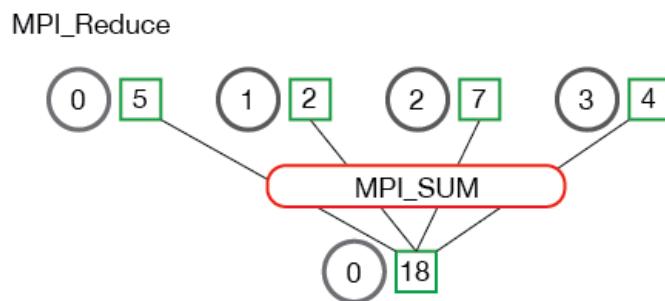
Final sum is 30
[LAPTOP-DITGC16E:00078] 3 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[LAPTOP-DITGC16E:00078] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
```

REMARKS:

MPI_Reduce takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result. The prototype for MPI_Reduce looks like this:

```
MPI_Reduce(  
void* send_data,  
void* recv_data,  
int count,  
MPI_Datatype datatype,  
MPI_Op op,  
int root,  
MPI_Comm communicator)
```

The send_data parameter is an array of elements of type datatype that each process wants to reduce. The recv_data is only relevant on the process with a rank of root. The recv_data array contains the reduced result and has a size of sizeof(datatype) * count. The op parameter is the operation that you wish to apply to your data. MPI contains a set of common reduction operations that can be used. Although custom reduction operations can be defined, it is beyond the scope of this lesson.



Name: Arpit Bhattacharya

Registration Number: 18BCE0124

Course: Parallel and Distributed Computing

Dated: 2nd October 2020

Assessment No: 10

Aim: Assume the variable rank contains the process rank and root is 3. What will be stored in array b [] on each of four processes if each executes the following code fragment?

SOURCE CODE:

```
#include<stdio.h>
#include "mpi.h"

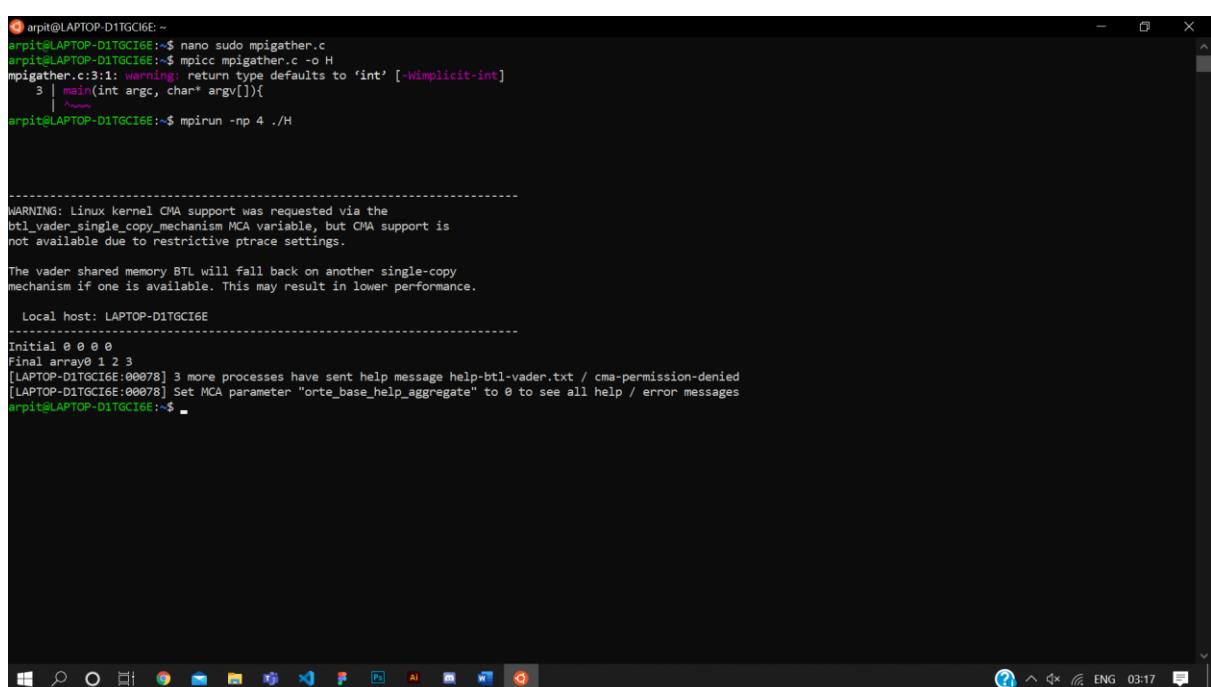
main(int argc, char* argv[]){
    int rank;
    int p;
    int b[4]={0, 0, 0 ,0};
    int root=3;
    printf("\n");
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    if(rank==0)
    {
        printf("Initial ");
        for(int i=0;i<4;i++)
        {
            printf("%d ",b[i]);
        }
        printf("\n");
    }
    MPI_Gather(&rank ,1, MPI_INT,&b,1,MPI_INT, root, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

```

if(rank==3)
{
    printf("Final array");
    for(int i=0;i<4;i++)
    {
        printf("%d ",b[i]);
    }
    printf("\n");
}

```

EXECUTION:



```

arpit@LAPTOP-D1TGC16E: ~
arpit@LAPTOP-D1TGC16E:~$ nano sudo mpigather.c
arpit@LAPTOP-D1TGC16E:~$ mpicc mpigather.c -o H
mpigather.c:3:1: warning: return type defaults to 'int' [-Wimplicit-int]
  3 | main(int argc, char* argv[]){
   | ^~~
arpit@LAPTOP-D1TGC16E:~$ mpirun -np 4 ./H

-----
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

Local host: LAPTOP-D1TGC16E
-----
Initial 0 0 0 0
Final array 0 1 2 3
[LAPTOP-D1TGC16E:00078] 3 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[LAPTOP-D1TGC16E:00078] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
arpit@LAPTOP-D1TGC16E:~$ 

```

```

arpit@LAPTOP-D1TGC16E: ~
[1/1] mpigather.c
#include<stdio.h>
#include "mpi.h"
main(int argc, char* argv[]){
    int rank;
    int p;
    int b[4]={0, 0, 0 ,0};
    int root=3;
    printf("\n");
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    if(rank==0)
    {
        printf("Initial ");
        for(int i=0;i<4;i++)
        {
            printf("%d ",b[i]);
        }
        printf("\n");
    }
    MPI_Gather(&rank ,1, MPI_INT,&b,1,MPI_INT, root, MPI_COMM_WORLD);
    MPI_Finalize();
    if(rank==3)
    {
        printf("Final array");
        for(int i=0;i<4;i++)
        {
            printf("%d ",b[i]);
        }
        printf("\n");
    }
}

```

[mpigather.c -- 32 lines]

Get Help Write Out Where Is Cut Text Justify Cur Pos Undo Mark Text To Bracket Previous Exit Read File Replace Paste Text To Spell Go To Line Redo Copy Text Where Was Next

ENG 03:17

REMARKS:

MPI_Gather is the inverse of **MPI_Scatter**. Instead of spreading elements from one process to many processes, **MPI_Gather** takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching. Below is a simple illustration of this algorithm.

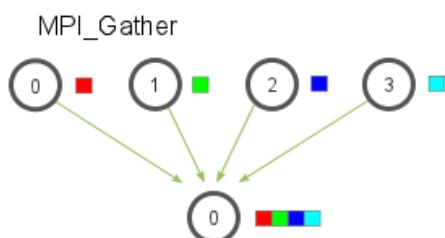
Similar to **MPI_Scatter**, **MPI_Gather** takes elements from each process and gathers them to the root process. The elements are ordered by the rank of the process from which they were received. The function prototype for **MPI_Gather** is identical to that of **MPI_Scatter**.

```

MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)

```

In **MPI_Gather**, only the root process needs to have a valid receive buffer. All other calling processes can pass NULL for **recv_data**. Also, don't forget that the **recv_count** parameter is the count of elements received per process, not the total summation of counts from all processes. This can often confuse beginning MPI programmers.





NAME:RITIK GUPTA

REG.NO:18BCE0154

SCHOOL OF COMPUTER SCIENCE ENGINEERING
FALL SEMESTER 2020-2021
CSE4001-PARALLEL AND DISTRIBUTED COMPUTING LAB
ASSIGNMENT-1 L1+L2
Dr. K. Murugan

ASSIGNMENT – 1

- 1. Create a parallel Hello World Program using OpenMp**
- 2. Compute vector addition using OpenMp**
- 3. Compute Vector multiplication using OpenMp**
- 4. Compute the sum of Two array in parallel using OpenMp**

CODE(MENU DRIVEN):

```
#include<stdio.h>
#include<omp.h>
void input1(int a[], int n)
{
int i;
for(i=0;i<n;i++)
{
scanf("%d", &a[i]);
}
}
void input2(int *f, int n, int m)
{
int i,j;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
scanf("%d", &*((f+i*m) + j));
```

```

}

}

void display1(int a[], int n)
{
int i;
for(i=0;i<n;i++)
printf("%d ", a[i]);
printf("\n");
}
void display2(int *g,int n, int m)
{
int i,j;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
printf("%d ",*(g+i*m) + j));
printf("\n");
}
}

int main()
{ int ch;
do
{
printf("\nREG. NO- 18BCE0154");
printf("\n *****MENU*****\n");
printf("1. Vector Addition using OpenMP\n");
printf("2. Vector Multiplication using OpenMP\n");
printf("3. Sum of Two n-sized Array in parallel using OpenMP\n");
printf("4. Hello World Program using OpenMP\n");
printf("5.Exit\n");
printf("\nEnter your choice : ");
int n,m,o,i,j,k,l,p,q,it,no,u,s=0;
scanf("%d",&ch);
switch(ch)
{
case 1 : printf("1. Single dimension\n");
printf("2. Two Dimensional Array or 2D Matrix\n");
printf("\nEnter your choice : ");
int f;
scanf("%d",&f);
if(f==1)
{
printf("Enter the size of the first vector : ");
scanf("%d",&n);
printf("Enter the size of the second vector : ");
scanf("%d",&m);
int f[n],g[m],h[n];
printf("\nEnter the elements of first vector : ");
input1(f,n);
printf("Enter the elements of second vector : ");
input1(g,m);
if(n==m)

```

```

{
#pragma omp parallel default(none) shared(f,g,h,i,n)
for(i=0;i<n;i++)
h[i]=f[i]+g[i];
printf("\nAddition Vector \n");
display1(h,n);
}
else
printf("The size should be same for addition! \n");
}
else if(f==2)
{
printf("Enter the dimensions of the first matrix : ");
scanf("%d",&n);
scanf("%d",&m);
printf("Enter the dimensions of the second matrix : ");
scanf("%d",&p);
scanf("%d",&q);
int a[n][m],b[p][q],c[n][m],d[n][m],e[n][q];
printf("\nEnter the values of the first matrix : ");
input2((int *)a,n,m);
printf("Enter the values of the second matrix : ");
input2((int *)b,p,q);
if(n==p && m==q)
{
#pragma omp parallel for default(none) shared(a,b,c,d,n,m,i,j)
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
c[i][j]=a[i][j]+b[i][j];
}
}
printf("\nAddition Matrix \n");
display2((int *)c,n,m);
}
else
printf("Wrong Dimensions! Addition and Subtraction not possible. \n");
}
else
printf("Wrong choice! Try Again.");
break;
case 2 : printf("1. Single dimension\n");
printf("2. Two Dimensional Array or 2D Matrix\n");
printf("\nEnter your choice : ");
int g;
scanf("%d",&g);
if(g==1)
{
printf("Enter the size of the vector : ");
scanf("%d",&n);
int a[n];
}

```

```

printf("Enter the elements of vector : ");
input1(a,n);
printf("Enter the scalar to multiply with the vector : ");
scanf("%d",&m);
#pragma omp parallel default(none) shared(a,n,m,i)
for(i=0;i<n;i++)
a[i] *= m;
printf("\nMultiplication Vector \n");
display1(a,n);
}
else if(g==2)
{
printf("Enter the dimensions of the first matrix : ");
scanf("%d",&n);
scanf("%d",&m);
printf("Enter the dimensions of the second matrix : ");
scanf("%d",&p);
scanf("%d",&q);
int a[n][m],b[p][q],c[n][m],d[n][m],e[n][q];
printf("\nEnter the values of the first matrix : ");
input2((int *)a,n,m);
printf("Enter the values of the second matrix : ");
input2((int *)b,p,q);
if(m==p)
{
#pragma omp parallel for default(none) shared(a,b,e,n,q,i,j,k)
for(i=0;i<n;i++)
{
for(j=0;j<q;j++)
{
e[i][j]=0;
for(k = 0; k < q; ++k)
{
e[i][j] += a[i][k] * b[k][j];
}
}
}
printf("\nMultiplication Matrix \n");
display2((int *)e,n,q);
}
else
printf("\nWrong Dimensions! Cannot Multiply\n");
}
else
printf("Wrong choice! Try Again.");
break;
case 3 : {
printf("Enter the size of the first array : ");
scanf("%d",&n);
printf("Enter the size of the second array : ");
scanf("%d",&m);
int f[n],g[m],h[n];

```

```

printf("\nEnter the elements of first array : ");
input1(f,n);
printf("Enter the elements of second array : ");
input1(g,m);
if(n==m)
{
#pragma omp parallel default(none) shared(f,g,h,i,n)
for(i=0;i<n;i++)
h[i]=f[i]+g[i];
printf("\nAddition Array \n");
display1(h,n);
}
else
printf("The size should be same for addition! \n");
}break;
case 4 : printf("\nThread Execution\n");
#pragma omp parallel
{
it=omp_get_thread_num();
no=omp_get_num_threads();
printf("Hello World!! from thread = %d\n",it);
}
printf("End Execution\n");
break;
case 5 : break;
default : printf("Wrong choice! Try Again.");
break;
}
}while(ch!=5);
}

```

OUTPUT:

Vector Addition Using OpenMp

```

ritik@ritik-Predator-PH315-51:~$ gcc program.c -o obj -fopenmp
ritik@ritik-Predator-PH315-51:~$ ./obj

REG. NO- 18BCE0154
*****MENU*****
1. Vector Addition using OpenMP
2. Vector Multiplication using OpenMP
3. Sum of Two n-sized Array in parallel using OpenMP
4. Hello World Program using OpenMP
5.Exit

Enter your choice : 1
1. Single dimension
2. Two Dimensional Array or 2D Matrix

Enter your choice : 1
Enter the size of the first vector : 4
Enter the size of the second vector : 4

Enter the elements of first vector : 5 3 2 2
Enter the elements of second vector : 2 4 5 6

Addition Vector
7 7 7 8

```

```
REG. NO- 18BCE0154
*****MENU*****
1. Vector Addition using OpenMP
2. Vector Multiplication using OpenMP
3. Sum of Two n-sized Array in parallel using OpenMP
4. Hello World Program using OpenMP
5.Exit

Enter your choice : 1
1. Single dimension
2. Two Dimensional Array or 2D Matrix

Enter your choice : 2
Enter the dimensions of the first matrix : 2 2
Enter the dimensions of the second matrix : 2 2

Enter the values of the first matrix : 3 4 3 4
Enter the values of the second matrix : 1 2 1 2

Addition Matrix
4 6
4 6
```

Compute Vector multiplication using OpenMp

```
REG. NO- 18BCE0154
*****MENU*****
1. Vector Addition using OpenMP
2. Vector Multiplication using OpenMP
3. Sum of Two n-sized Array in parallel using OpenMP
4. Hello World Program using OpenMP
5.Exit

Enter your choice : 2
1. Single dimension
2. Two Dimensional Array or 2D Matrix

Enter your choice : 1
Enter the size of the vector : 2
Enter the elements of vector : 2 1
Enter the scalar to multiply with the vector : 1

Multiplication Vector
2 1
```

```
REG. NO- 18BCE0154
*****MENU*****
1. Vector Addition using OpenMP
2. Vector Multiplication using OpenMP
3. Sum of Two n-sized Array in parallel using OpenMP
4. Hello World Program using OpenMP
5.Exit

Enter your choice : 2
1. Single dimension
2. Two Dimensional Array or 2D Matrix

Enter your choice : 2
Enter the dimensions of the first matrix : 2 2
Enter the dimensions of the second matrix : 2 2

Enter the values of the first matrix : 1 2 3 4
Enter the values of the second matrix : 1 2 3 4

Multiplication Matrix
7 10
15 22
```

Compute the sum of Two array in parallel using OpenMp

```
REG. NO- 18BCE0154
*****MENU*****
1. Vector Addition using OpenMP
2. Vector Multiplication using OpenMP
3. Sum of Two n-sized Array in parallel using OpenMP
4. Hello World Program using OpenMP
5.Exit

Enter your choice : 3
Enter the size of the first array : 4
Enter the size of the second array : 4

Enter the elements of first array : 1 2 3 4
Enter the elements of second array : 5 6 7 8

Addition Array
6 8 10 12
```

Create a parallel Hello World Program using OpenMp

```
REG. NO- 18BCE0154
*****MENU*****
1. Vector Addition using OpenMP
2. Vector Multiplication using OpenMP
3. Sum of Two n-sized Array in parallel using OpenMP
4. Hello World Program using OpenMP
5.Exit

Enter your choice : 4

Thread Execution
Hello World!! from thread = 8
Hello World!! from thread = 1
Hello World!! from thread = 7
Hello World!! from thread = 0
Hello World!! from thread = 4
Hello World!! from thread = 5
Hello World!! from thread = 5
Hello World!! from thread = 10
Hello World!! from thread = 3
Hello World!! from thread = 6
Hello World!! from thread = 9
Hello World!! from thread = 2
End Execution
```



Name : RITIK GUPTA

Reg. No. : 18BCE0154

Course : Parallel Distributed Computing Lab

Slot : L1+L2

ASSIGNMENT – 2

1) Orphan Parallel Loop Reduction

Code :

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define VECLEN 100
float a[VECLEN], b[VECLEN], sum;
float dotprod ()
{
int i,tid;
tid = omp_get_thread_num();
#pragma omp for reduction(+:sum)
for (i=0; i < VECLEN; i++)
{
    sum = sum + (a[i]*b[i]);
    printf(" tid= %d i=%d\n",tid,i);
}
int main (int argc, char *argv[])
{
int i;
for (i=0; i < VECLEN; i++)
    a[i] = b[i] = 1.0 * i;
sum = 0.0;
#pragma omp parallel
    dotprod();
printf("Sum = %f\n",sum);
}
```

Output :

```
ritik@ritik-Predator-PH315-51:~$ gedit orphan.c
ritik@ritik-Predator-PH315-51:~$ gcc orphan.c -fopenmp
ritik@ritik-Predator-PH315-51:~$ ./a.out
Ritik Gupta 18BCE0154 tid= 0 i=0
tid= 0 i=1
tid= 0 i=2
tid= 0 i=3
tid= 0 i=4
tid= 0 i=5
tid= 0 i=6
tid= 0 i=7
tid= 0 i=8
tid= 4 i=36
tid= 4 i=37
tid= 4 i=38
tid= 4 i=39
tid= 4 i=40
tid= 4 i=41
tid= 4 i=42
tid= 4 i=43
tid= 6 i=52
tid= 6 i=53
tid= 6 i=54
tid= 6 i=55
tid= 6 i=56
tid= 6 i=57
tid= 6 i=58
tid= 6 i=59
tid= 6 i=84
tid= 10 i=85
tid= 10 i=86
tid= 10 i=87
tid= 10 i=88
tid= 10 i=89
tid= 10 i=90
tid= 10 i=91
tid= 9 i=76
tid= 9 i=77
tid= 3 i=27
tid= 3 i=28
tid= 3 i=29
tid= 3 i=30
tid= 7 i=60
tid= 7 i=61
tid= 7 i=62
tid= 9 i=78
tid= 9 i=79
tid= 9 i=80
tid= 9 i=81
tid= 9 i=82
tid= 9 i=83
tid= 7 i=63
tid= 7 i=64
tid= 7 i=65
tid= 7 i=66
```

```
tid= 9 i=83
tid= 7 i=63
tid= 7 i=64
tid= 7 i=65
tid= 7 i=66
tid= 7 i=67
tid= 3 i=31
tid= 3 i=32
tid= 3 i=33
tid= 3 i=34
tid= 3 i=35
tid= 5 i=44
tid= 5 i=45
tid= 5 i=46
tid= 5 i=47
tid= 5 i=48
tid= 5 i=49
tid= 5 i=50
tid= 5 i=51
tid= 2 i=18
tid= 2 i=19
tid= 2 i=20
tid= 2 i=21
tid= 2 i=22
tid= 2 i=23
tid= 2 i=24
tid= 2 i=25
tid= 2 i=26
tid= 1 i=9
tid= 1 i=10
tid= 1 i=11
tid= 1 i=12
tid= 1 i=13
tid= 1 i=14
tid= 1 i=15
tid= 1 i=16
tid= 1 i=17
tid= 11 i=92
tid= 11 i=93
tid= 11 i=94
tid= 11 i=95
tid= 11 i=96
tid= 11 i=97
tid= 11 i=98
tid= 11 i=99
tid= 8 i=68
tid= 8 i=69
tid= 8 i=70
tid= 8 i=71
tid= 8 i=72
tid= 8 i=73
tid= 8 i=74
tid= 8 i=75
Sum = 328350.000000
```

```
rittk@rittk-Predator-PH315-51:~$
```

2) Combined Parallel Loop Reduction

Code :

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int i, n;
    float a[100], b[100], sum;
    n = 100;
    for (i=0; i < n; i++)
        a[i] = b[i] = i * 1.0;
    sum = 0.0;

#pragma omp parallel for reduction(+:sum)
    for (i=0; i < n; i++)
        sum = sum + (a[i] * b[i]);

    printf("Sum = %f",sum);
}
```

Output :

```
ritik@ritik-Predator-PH315-51:~$ gedit combined.c
ritik@ritik-Predator-PH315-51:~$ gcc combined.c -fopenmp
ritik@ritik-Predator-PH315-51:~$ ./a.out
Ritik Gupta 18BCE0154
Sum = 328350.000000ritik@ritik-Predator-PH315-51:~$
```

3) Loop Sharing

Code :

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100

int main (int argc, char *argv[])
{
    int nthreads, tid, i, chunk;
```

```

float a[N], b[N], c[N];

for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0)
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);

#pragma omp for schedule(dynamic,chunk)
for (i=0; i<N; i++)
{
    {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }
}
}

```

Output :

```

ritik@ritik-Predator-PH315-51:~$ gcc loop.c -fopenmp
ritik@ritik-Predator-PH315-51:~$ ./a.out
Ritik Gupta 18BCE0154
Number of threads = 12
Thread 5 starting...
Thread 5: c[0]= 0.000000
Thread 5: c[1]= 2.000000
Thread 5: c[2]= 4.000000
Thread 5: c[3]= 6.000000
Thread 5: c[4]= 8.000000
Thread 5: c[5]= 10.000000
Thread 5: c[6]= 12.000000
Thread 5: c[7]= 14.000000
Thread 5: c[8]= 16.000000
Thread 5: c[9]= 18.000000
Thread 5: c[10]= 20.000000
Thread 5: c[11]= 22.000000
Thread 5: c[12]= 24.000000
Thread 5: c[13]= 26.000000
Thread 5: c[14]= 28.000000
Thread 5: c[15]= 30.000000
Thread 5: c[16]= 32.000000
Thread 5: c[17]= 34.000000
Thread 5: c[18]= 36.000000
Thread 5: c[19]= 38.000000
Thread 5: c[20]= 40.000000
Thread 5: c[21]= 42.000000
Thread 5: c[22]= 44.000000
Thread 5: c[23]= 46.000000
Thread 5: c[24]= 48.000000
Thread 5: c[25]= 50.000000
Thread 5: c[26]= 52.000000
Thread 5: c[27]= 54.000000
Thread 5: c[28]= 56.000000
Thread 5: c[29]= 58.000000
Thread 5: c[30]= 60.000000
Thread 5: c[31]= 62.000000
Thread 5: c[32]= 64.000000
Thread 5: c[33]= 66.000000
Thread 5: c[34]= 68.000000
Thread 5: c[35]= 70.000000
Thread 5: c[36]= 72.000000
Thread 5: c[37]= 74.000000
Thread 5: c[38]= 76.000000
Thread 5: c[39]= 78.000000
Thread 5: c[40]= 80.000000
Thread 5: c[41]= 82.000000
Thread 5: c[42]= 84.000000
Thread 5: c[43]= 86.000000
Thread 5: c[44]= 88.000000
Thread 5: c[45]= 90.000000
Thread 5: c[46]= 92.000000
Thread 5: c[47]= 94.000000
Thread 5: c[48]= 96.000000
Thread 5: c[49]= 98.000000

```

```

Thread 5: c[57]= 114.000000
Thread 5: c[58]= 116.000000
Thread 5: c[59]= 118.000000
Thread 5: c[60]= 120.000000
Thread 5: c[61]= 122.000000
Thread 5: c[62]= 124.000000
Thread 5: c[63]= 126.000000
Thread 5: c[64]= 128.000000
Thread 5: c[65]= 130.000000
Thread 5: c[66]= 132.000000
Thread 5: c[67]= 134.000000
Thread 5: c[68]= 136.000000
Thread 5: c[69]= 138.000000
Thread 5: c[70]= 140.000000
Thread 5: c[71]= 142.000000
Thread 5: c[72]= 144.000000
Thread 5: c[73]= 146.000000
Thread 5: c[74]= 148.000000
Thread 5: c[75]= 150.000000
Thread 5: c[76]= 152.000000
Thread 5: c[77]= 154.000000
Thread 5: c[78]= 156.000000
Thread 5: c[79]= 158.000000
Thread 5: c[80]= 160.000000
Thread 5: c[81]= 162.000000
Thread 5: c[82]= 164.000000
Thread 5: c[83]= 166.000000
Thread 5: c[84]= 168.000000
Thread 5: c[85]= 170.000000
Thread 5: c[86]= 172.000000
Thread 5: c[87]= 174.000000
Thread 5: c[88]= 176.000000
Thread 5: c[89]= 178.000000
Thread 5: c[90]= 180.000000
Thread 5: c[91]= 182.000000
Thread 5: c[92]= 184.000000
Thread 2 starting...
Thread 11 starting...
Thread 6 starting...
Thread 0 starting...
Thread 9 starting...
Thread 7 starting...
Thread 8 starting...
Thread 4 starting...
Thread 3 starting...
Thread 1 starting...
Thread 5: c[93]= 186.000000
Thread 5: c[94]= 188.000000
Thread 5: c[95]= 190.000000
Thread 5: c[96]= 192.000000
Thread 5: c[97]= 194.000000
Thread 5: c[98]= 196.000000
Thread 5: c[99]= 198.000000
Thread 10 starting...
ritik@ritik-Predator-PH315-51:~$ 
```

4) Section Sharing

Code :

```

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 50

int main (int argc, char *argv[])
{
    int i, nthreads, tid;
    float a[N], b[N], c[N], d[N];

    for (i=0; i<N; i++) {
        a[i] = i * 1.5;
    }
} 
```

```

b[i] = i + 22.35;
c[i] = d[i] = 0.0;
}

#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
tid = omp_get_thread_num();
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
printf("Thread %d starting...\n",tid);

#pragma omp sections nowait
{
#pragma omp section
{
printf("Thread %d doing section 1\n",tid);
for (i=0; i<N; i++)
{
c[i] = a[i] + b[i];
printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
}

#pragma omp section
{
printf("Thread %d doing section 2\n",tid);
for (i=0; i<N; i++)
{
d[i] = a[i] * b[i];
printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
}
}
printf("Thread %d done.\n",tid);
}
}

```

Output :

```
ritik@ritik-Predator-PH315-51:~$ gedit section.c
ritik@ritik-Predator-PH315-51:~$ gcc section.c -fopenmp
ritik@ritik-Predator-PH315-51:~$ ./a.out
Ritik Gupta 18BCE0154Number of threads = 12
Thread 0 starting...
Thread 0 doing section 1
Thread 0: c[0]= 22.350000
Thread 0: c[1]= 24.850000
Thread 0: c[2]= 27.350000
Thread 0: c[3]= 29.850000
Thread 0: c[4]= 32.349998
Thread 0: c[5]= 34.849998
Thread 0: c[6]= 37.349998
Thread 0: c[7]= 39.849998
Thread 0: c[8]= 42.349998
Thread 0: c[9]= 44.849998
Thread 0: c[10]= 47.349998
Thread 0: c[11]= 49.849998
Thread 0: c[12]= 52.349998
Thread 0: c[13]= 54.849998
Thread 5 starting...
Thread 5 doing section 2
Thread 5: d[0]= 0.000000
Thread 5: d[1]= 35.025002
Thread 5: d[2]= 73.050003
Thread 5: d[3]= 114.075005
Thread 5: d[4]= 158.100006
Thread 5: d[5]= 205.125000
Thread 5: d[6]= 255.150009
Thread 5: d[7]= 308.175018
Thread 5: d[8]= 364.200012
Thread 5: d[9]= 423.225006
Thread 9 starting...
Thread 9 done.
Thread 2 starting...
Thread 2 done.
Thread 0: c[14]= 57.349998
Thread 0: c[15]= 59.849998
Thread 0: c[16]= 62.349998
Thread 0: c[17]= 64.849998
Thread 0: c[18]= 67.349998
Thread 0: c[19]= 69.849998
Thread 0: c[20]= 72.349998
Thread 0: c[21]= 74.849998
Thread 0: c[22]= 77.349998
Thread 0: c[23]= 79.849998
Thread 0: c[24]= 82.349998
Thread 7 starting...
Thread 7 done.
Thread 3 starting...
Thread 3 done.
Thread 6 starting...
Thread 6 done.
Thread 4 starting...
Thread 4 done.
```

```
File Edit View Search Terminal Help
Thread 10 starting...
Thread 10 done.
Thread 8 starting...
Thread 8 done.
Thread 0: c[25]= 84.849998
Thread 0: c[26]= 87.349998
Thread 0: c[27]= 89.849998
Thread 0: c[28]= 92.349998
Thread 0: c[29]= 94.849998
Thread 0: c[30]= 97.349998
Thread 0: c[31]= 99.849998
Thread 0: c[32]= 102.349998
Thread 0: c[33]= 104.849998
Thread 0: c[34]= 107.349998
Thread 0: c[35]= 109.849998
Thread 0: c[36]= 112.349998
Thread 0: c[37]= 114.849998
Thread 0: c[38]= 117.349998
Thread 0: c[39]= 119.849998
Thread 0: c[40]= 122.349998
Thread 0: c[41]= 124.849998
Thread 0: c[42]= 127.349998
Thread 0: c[43]= 129.850006
Thread 0: c[44]= 132.350006
Thread 0: c[45]= 134.850006
Thread 0: c[46]= 137.350006
Thread 0: c[47]= 139.850006
Thread 0: c[48]= 142.350006
Thread 0: c[49]= 144.850006
Thread 0 done.
Thread 5: d[10]= 485.249969
Thread 5: d[11]= 550.274963
Thread 5: d[12]= 618.299988
Thread 5: d[13]= 689.324951
Thread 5: d[14]= 763.349976
Thread 5: d[15]= 840.374939
Thread 5: d[16]= 920.399963
Thread 5: d[17]= 1083.424988
Thread 5: d[18]= 1089.449951
Thread 5: d[19]= 1178.474976
Thread 5: d[20]= 1270.500000
Thread 5: d[21]= 1365.524902
Thread 5: d[22]= 1463.549927
Thread 5: d[23]= 1564.574951
Thread 5: d[24]= 1668.599976
Thread 5: d[25]= 1775.625000
Thread 5: d[26]= 1885.649902
Thread 5: d[27]= 1998.674927
Thread 5: d[28]= 2114.699951
Thread 5: d[29]= 2233.724854
Thread 5: d[30]= 2355.750000
Thread 5: d[31]= 2480.774902
Thread 5: d[32]= 2608.799805
Thread 5: d[33]= 2739.824951
Thread 5: d[34]= 2873.849854
```

```
Thread 5: d[34]= 2873.849854
Thread 5: d[35]= 3010.875000
Thread 5: d[36]= 3150.899982
Thread 5: d[37]= 3293.924805
Thread 5: d[38]= 3439.949951
Thread 5: d[39]= 3588.974854
Thread 5: d[40]= 3741.000000
Thread 5: d[41]= 3896.024902
Thread 5: d[42]= 4054.849885
Thread 5: d[43]= 4215.874787
Thread 5: d[44]= 4379.100098
Thread 5: d[45]= 4546.125000
Thread 5: d[46]= 4716.149982
Thread 5: d[47]= 4889.174805
Thread 5: d[48]= 5065.199787
Thread 5: d[49]= 5244.225098
Thread 5 done.
Thread 11 starting...
Thread 11 done.
Thread 1 starting...
Thread 1 done.
ritik@ritik-Predator-PH315-51:~$
```

NAME – RITIK GUPTA

REG. NO. – 18BCE0154



VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE ENGINEERING

FALL SEMESTER 2020-2021

CSE4001-PARALLEL AND DISTRIBUTED COMPUTING LAB

ASSESSMENT-3

L1+L2

Dr. K. Murugan

1. Develop a MPI program to perform Merge sort with MPI_Scatter and MPI_Gather.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <time.h>
#include <mpi.h>
using namespace std;
//Merge
void merge(int *a, int *b, int l, int m, int r);
//main function for mearge
void mergeSort(int *a, int *b, int l, int r);
int main(int argc, char** argv)
{
//intilize Array :
int ArraySize=6;
int *original_array = new int[ArraySize] ;
cout<<"#####MPI MERGE SORT USING MPI GATHER AND MPI
SCATTER--- 18BCE0154#####\n";
//start intialize MPI services:
int Process_rank;
```

```

int ProcessSize;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &Process_rank);
MPI_Comm_size(MPI_COMM_WORLD, &ProcessSize);
//Print array
if (Process_rank==0)
{
cout << "Enter Array to sort\n ";
for (int c = 0; c < ArraySize; c++)
{

original_array[c] = rand() % ArraySize;
if(c==ArraySize-1){
cout << original_array[c];
}
else{
cout << original_array[c]<<", ";
}
cout << "\n";
}
// Divide the array
int SizeForSub = ArraySize / ProcessSize;
//send sub array
int *sub_array = new int[SizeForSub];
MPI_Scatter(original_array, SizeForSub, MPI_INT, sub_array, SizeForSub, MPI_INT, 0,
MPI_COMM_WORLD);
//show sub array
for (int i = 0; i < ProcessSize; i++)
{

cout << "sub array (Sub Parallel Process): " << Process_rank << "\n";
for (int c = 0; c < SizeForSub; c++)
{
if(c==SizeForSub-1){
cout << sub_array[c];
}
else{
cout << sub_array[c]<<", ";
}
}
}

```

```

}

printf("\n");
break;
}

//apply merge sort for each process
int *tmp_array = new int[SizeForSub];
mergeSort(sub_array, tmp_array, 0, (SizeForSub - 1));

// Gather the sorted subarrays
int *sorted=NULL;
if(Process_rank== 0)
{
    sorted = new int[ArraySize];

}

MPI_Gather(sub_array, SizeForSub, MPI_INT, sorted, SizeForSub, MPI_INT, 0,
MPI_COMM_WORLD);

//call final mearge function
if(Process_rank == 0)
{
    int *other_array =new int[ArraySize] ;
    mergeSort(sorted, other_array, 0, (ArraySize - 1));

    //show stored array
    cout<<"The sorted array is:\n";
    for(int c = 0; c <ArraySize; c++)
    {
        if(c==ArraySize-1){
            cout << sorted[c];
        }
        else{
            cout << sorted[c]<<", ";
        }
    }
}

```

```
}

cout<<"\n\n\n";
free(sorted);
free(other_array);
}

free(original_array);
free(sub_array);
free(tmp_array);
// Finalize MPI
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
}

//merge Function
void merge(int *a, int *b, int l, int m, int r)
{
int h, i, j, k;
h = l;
i = l;
j = m + 1;
while ((h <= m) && (j <= r))
{
if (a[h] <= a[j])
{
b[i] = a[h];
h++;
}
else
{
b[i] = a[j];
j++;
}
i++;
}
if (m < h)
{
for (k = j; k <= r; k++)
{
```

```
b[i] = a[k];
i++;
}
}
else
{
for (k = h; k <= m; k++)
{
b[i] = a[k];
i++;
}
}

for (k = l; k <= r; k++)
{
a[k] = b[k];
}
}

//Merge Sort Function:
void mergeSort(int *a, int *b, int l, int r)
{
int m;
if (l < r) {
m = (l + r) / 2;
mergeSort(a, b, l, m);
mergeSort(a, b, (m + 1), r);
merge(a, b, l, m, r);
}
}
```

OUTPUT:

```

ritik@ritik-Predator-PH315-51:~$ gedit merge.cpp
ritik@ritik-Predator-PH315-51:~$ mpic++ merge.cpp -o MergeSort
ritik@ritik-Predator-PH315-51:~$ mpirun -np 2 ./MergeSort
#####
MPI MERGE SORT USING MPI GATHER AND MPI SCATTER--- 18BCE0154#####
#####
MPI MERGE SORT USING MPI GATHER AND MPI SCATTER--- 18BCE0154#####
Enter Array to sort
1, 4, 3, 1, 5, 1
sub array (Sub Parallel Process): 0
1, 4, 3
sub array (Sub Parallel Process): 1
1, 5, 1
The sorted array is:
1, 1, 1, 3, 4, 5

```

2. Write a C program using MPI to implement Dot product.

CODE:

```

#include <stdio.h>
#include "mpi.h"
#define MAX_LOCAL_ORDER 100
void main(int argc, char* argv[]) {
printf("##### 18BCE0154- DOT PRODUCT USING MPI #####\n");
float local_x[MAX_LOCAL_ORDER];
float local_y[MAX_LOCAL_ORDER];
int n,n_bar; /* = n/p */
float dot;
int p,my_rank;
void Read_vector(char* prompt, float local_v[], int n_bar, int p,int my_rank);
float Parallel_dot(float local_x[], float local_y[], int n_bar);
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank == 0) {
printf("Enter the order of the vectors\n");
scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

```

```

n_bar = n/p;

Read_vector("the first vector", local_x, n_bar, p, my_rank);

Read_vector("the second vector", local_y, n_bar, p, my_rank);

dot = Parallel_dot(local_x, local_y, n_bar);

if (my_rank == 0) printf("The dot product is %f\n", dot);

MPI_Finalize();

} /* main */

void Read_vector(
char* prompt /* in */,
float local_v[] /* out */,
int n_bar /* in */,
int p /* in */,
int my_rank /* in */) {
int i, q;

float temp[MAX_LOCAL_ORDER];

MPI_Status status;

if (my_rank == 0) {
printf("Enter %s\n", prompt);

for (i = 0; i < n_bar; i++)
scanf("%f", &local_v[i]);

for (q = 1; q < p; q++) {
for (i = 0; i < n_bar; i++)
scanf("%f", &temp[i]);

MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
}

} else {

MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
&status);

}

} /* Read_vector */

```

```
float Serial_dot(
    float x[] /* in */,
    float y[] /* in */,
    int n /* in */) {
    int i;
    float sum = 0.0;
    for (i = 0; i < n; i++)
        sum = sum + x[i]*y[i];
    return sum;
}

float Parallel_dot(
    float local_x[] /* in */,
    float local_y[] /* in */,
    int n_bar /* in */) {
    float local_dot;
    float dot = 0.0;
    float Serial_dot(float x[], float y[], int m);
    local_dot = Serial_dot(local_x, local_y, n_bar);
    MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
               MPI_SUM, 0, MPI_COMM_WORLD);
    return dot;
} /* Parallel_dot */
```

OUTPUT:

```

ritik@ritik-Predator-PH315-51:~$ gedit dot.c
ritik@ritik-Predator-PH315-51:~$ mpicc dot.c -o object_file
ritik@ritik-Predator-PH315-51:~$ mpirun -np 1 ./object_file
##### 18BCE0154 - DOT PRODUCT USING MPI #####
Enter the order of the vectors
12
Enter the first vector
1 2 3 4 5 6 7 8 9 10 11 12
Enter the second vector
2 4 6 8 10 12 14 16 18 20 22 24
The dot product is 1300.000000
ritik@ritik-Predator-PH315-51:~$ █

```

3. Write a MPI program to print pi value.

CODE:

```

#include <stdio.h>
#include <stdlib.h>
#include "omp.h"
#include "mpi.h"

int threads;

static long num_steps;

double step;

void main(int argc, char** argv)
{
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    printf("#####18BCE0154- Approximate Value of PI Using MPI#####\n");
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process

```

```
int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

num_steps = atol(argv[2]);

if (argc>3) threads = atol(argv[4]);

double pi,sum=0.,sum_inter = 0.;

step = 1.0/(double) num_steps;

omp_set_num_threads(threads);

#pragma omp parallel shared(sum) private(sum_inter)

{

double x;

int ID = omp_get_thread_num();

int workers = omp_get_num_threads()*world_size;

for (int i=ID+world_rank*omp_get_num_threads();i<num_steps;i+=workers)

{

x = (i+0.5)*step;

sum_inter = sum_inter + 4.0/(1.0+x*x);

}

#pragma omp critical

sum += sum_inter;

}

//MPI_Barrier(MPI_COMM_WORLD);

double global_sum;

MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (world_rank==0)

{

pi = step*global_sum;
```

```
    printf("PI = %f\n",pi);
}

// Finalize the MPI environment.

    MPI_Finalize();

}
```

OUTPUT:

```
ritik@ritik-Predator-PH315-51:~$ gedit pi.c
ritik@ritik-Predator-PH315-51:~$ mpicc pi.c -o pi_omp_mpi -fopenmp
ritik@ritik-Predator-PH315-51:~$ mpiexec -n 2 ./pi_omp_mpi -steps 100000 -threads 4
#####
#18BCE0154- Approximate Value of PI Using MPI#####
#18BCE0154- Approximate Value of PI Using MPI#####
PI = 3.141593
ritik@ritik-Predator-PH315-51:~$
```

4. Compute sum of array using MPI.

CODE:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// size of array
#define n 10

int a[10];
int a2[1000];

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

```
printf("\n 18BCE0154-SUM OF ARRAY USING MPI\nEnter the array elements to get the
Array Sum: \n");

for(int i=0;i<10;i++){
    scanf("%d",&a[i]);
}

int pid, np, elements_per_process, n_elements_recieved;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &np);
if (pid == 0) {
    int index, i;
    elements_per_process = n / np;
    if (np > 1) {
        for (i = 1; i < np - 1; i++) {
            index = i * elements_per_process;
            MPI_Send(&elements_per_process, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
        // last process adds remaining elements
        index = i * elements_per_process;
        int elements_left = n - index;
        MPI_Send(&elements_left, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        MPI_Send(&a[index], elements_left, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
    // master process add its own sub array
    int sum = 0;
```

```
for (i = 0; i < elements_per_process; i++)
sum += a[i];

// collects partial sums from other processes

int tmp;

for (i = 1; i < np; i++) {
MPI_Recv(&tmp, 1, MPI_INT,
MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD,
&status);

int sender = status.MPI_SOURCE;
sum += tmp;
}

// prints the final sum of array
printf("Sum of array is : %d\n", sum);
}

// slave processes
else {
MPI_Recv(&n_elements_recieved,
1, MPI_INT, 0, 0,
MPI_COMM_WORLD,
&status);

// stores the received array segment
// in local array a2
MPI_Recv(&a2, n_elements_recieved,
MPI_INT, 0, 0,
MPI_COMM_WORLD,
&status);
```

```
// calculates its partial sum

int partial_sum = 0;

for (int i = 0; i < n_elements_recieved; i++)

partial_sum += a2[i];

// sends the partial sum to the root process

MPI_Send(&partial_sum, 1, MPI_INT,

0, 0, MPI_COMM_WORLD);

}

// cleans up all MPI state before exit of process

MPI_Finalize();

return 0;

}
```

OUTPUT:

```
ritik@ritik-Predator-PH315-51:~$ gedit array_sum.c
ritik@ritik-Predator-PH315-51:~$ mpicc array_sum.c -o object_file
ritik@ritik-Predator-PH315-51:~$ mpirun -np 1 ./object_file
#####
18BCE0154 - SUM OF ARRAY USING MPI #####
Enter The array elements to get the Array Sum:
1 2 3 4 5 6 7 8 9 10
Sum of array is:55
ritik@ritik-Predator-PH315-51:~$
```

NAME – RITIK GUPTA

REG. NO. – 18BCE0154



**SCHOOL OF COMPUTER SCIENCE ENGINEERING
FALL SEMESTER 2020-2021
ESE4001-PARALLEL AND DISTRIBUTED COMPUTING LAB
ASSESSMENT-4
L1+L2
Dr. K. Murugan**

1. Write a C Program with OpenMP to compute the value of “pi” function by numerical integration of a function $f(x) = 4/(1+x^2)$ between the limits 0 and 1

CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<math.h>
#include<omp.h>

#define PI 3.1415926538837211

/* Main Program */
void main(int argc,char **argv)
{
    int      Noofintervals,
    i,Noofthreads,threadid; double
    totalsum, h;
    struct timeval TimeValue_Start;
    struct timezone TimeZone_Start;

    struct timeval TimeValue_Final;
    struct          timezone
TimeZone_Final; long time_start,
time_end;
    double time_overhead;
```

```

printf("\n\t-----");
printf("\n\t 18BCE0154");
printf("\n\t-----");
printf("\n\t PDC LAB DA-4 ");
printf("\n\t Write a C Program with OpenMP to compute the value of “pi” function by numerical
integration of a function f(x) = 4/(1+x*x ) between the limits 0 and 1");
printf("\n\t.....\n");

/* Checking for command line arguments */
if( argc != 3 ){

    printf("\t Very Few Arguments\n ");
    printf("\t Syntax : exec <Threads> <no. of interval>\n");
    exit(-1);
}

Noofthreads=atoi(argv[1]);
if ((Noofthreads!=1) && (Noofthreads!=2) && (Noofthreads!=4) && (Noofthreads!=8) &&
(Noofthreads!= 16) ) {
    printf("\n Number of threads should be 1,2,4,8 or 16 for the execution of program. \n\n");
    exit(-1);
}

Noofintervals=atoi(argv[2]);

printf("\n\t Threads : %d ",Noofthreads);

/* No. of intervals should be positive integer */
if (Noofintervals <= 0) {
    printf("\n\t Number of intervals should be positive integer\n");
    exit(1);
}
totalsum = 0.0;

gettimeofday(&TimeValue_Start, &TimeZone_Start);

h = 1.0 / Noofintervals;

/* set the number of threads */
omp_set_num_threads(Noofthreads);
/*
 * OpenMP Parallel Directive With Private Clauses And Critical
 * Section
 */

```

```

#pragma omp parallel for private(x)
for (i = 1; i < Noofintervals + 1; i = i+1)
{ x = h * (i + 0.5);
/*printf("the thread id is %d with iteration %d ",omp_get_thread_num(),i);*/
#pragma omp critical
totalsum = totalsum + 4.0/(1.0 + x * x);
} /* All thread join Master thread */
totalsum = totalsum * h;
gettimeofday(&TimeValue_Final, &TimeZone_Final);

time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;

time_overhead = (time_end - time_start)/1000000.0;

/*printf("\n\t\t Calculated PI : \t%1.15lf \n\t\t Error : \t%1.16lf\n", totalsum, fabs(totalsum - PI));*/
printf("\n\t\t Calculated PI : %1.15lf",totalsum );
printf("\n\t\t Time in Seconds (T)%lf",time_overhead);
printf("\n\n\t\t ( T represents the Time taken for computation )");
printf("\n\t.....\n");
}

```

OUTPUT:

```

ritik@ritik-Predator-PH315-51:~$ gedit integration.c
ritik@ritik-Predator-PH315-51:~$ gcc -o omp_helloc -fopenmp integration.c
ritik@ritik-Predator-PH315-51:~$ ./omp_helloc 4 1000000000

18BCE0154

PDC LAB DA-4
Write a C Program with OpenMP to compute the value of "pi" function by numerical integration of a function f(x) = 4/(1+x*x ) between the limits 0 and 1
.....
Threads : 4
Calculated PI : 3.141592651589906
Time in Seconds (T)57.768548

( T represents the Time taken for computation )
.....
ritik@ritik-Predator-PH315-51:~$ 

```

2. Implement merge sort and parallelize it. (can use any programming language)

CODE:

```
#include<iostream>
#include<cstdlib>
#include<omp.h>
#include<pthread.h>
#include<time.h>
using namespace std;
#define MAX 20
#define THREAD_MAX 4
int a[MAX];
int part = 0;
void merge(int low, int mid, int high)
{
    int* left = new int[mid - low + 1];
    int* right = new int[high - mid];
    int n1 = mid - low + 1, n2 = high - mid, i, j;
    for (i = 0; i < n1; i++)
        left[i] = a[i + low];
    for (i = 0; i < n2; i++)
        right[i] = a[i + mid + 1];
    int k = low;
    i = j = 0;
    while (i < n1 && j < n2)
    { if (left[i] <= right[j])
        a[k++] = left[i++];
        else
        a[k++] = right[j++];
    }
    while (i < n1) { a[k+
    +] = left[i++];
    }
    while (j < n2) { a[k+
    +] = right[j++];
    }
}
void merge_sort(int low, int high)
{
    int mid = low + (high - low) / 2;
    if (low < high)
    { merge_sort(low, mid);
        merge_sort(mid + 1, high);
        merge(low, mid, high);
    }
}
```

```

}

void* merge_sort(void* arg)
{
int thread_part = part++;
int low = thread_part * (MAX / 4);
int high = (thread_part + 1) * (MAX / 4) - 1;
int mid = low + (high - low) / 2;
if (low < high)
{ merge_sort(low, mid);
merge_sort(mid + 1, high);
merge(low, mid, high);
}
return 0;
}

int main()
{
cout<<"-----18BCE0154-----\n";
cout<<"-----PDC LAB DA-4 (parallelized merge sort)-----\n";
for (int i = 0; i < MAX; i++)
a[i] = rand() % 100;
cout<<"Original Array : \t";
for (int i = 0; i < MAX; i++)
cout<<a[i]<< " ";
cout<<endl;
clock_t t1, t2;
t1 = clock();
pthread_t threads[THREAD_MAX];
for (int i = 0; i < THREAD_MAX; i++)
pthread_create(&threads[i], NULL, merge_sort,
(void*)NULL);
for (int i = 0; i < 4; i++)
pthread_join(threads[i], NULL);
merge(0, (MAX / 2 - 1) / 2, MAX / 2 - 1);
merge(MAX / 2, MAX/2 + (MAX-1-MAX/2)/2, MAX - 1);
merge(0, (MAX - 1)/2, MAX - 1);
t2 = clock();
cout << "Sorted array: \t\t";
for (int i = 0; i < MAX; i++)
cout<<a[i]<< " ";
cout<<endl << "\nTime taken : "<<(t2 - t1) /
(double)CLOCKS_PER_SEC << endl;
return 0;
}

```

OUTPUT:

```
ritik@ritik-Predator-PH315-51:~$ gedit merge.cpp
ritik@ritik-Predator-PH315-51:~$ g++ -o omp_helloc -fopenmp merge.cpp
ritik@ritik-Predator-PH315-51:~$ ./omp_helloc 4
-----18BCE0154
-----PDC LAB DA-4 (parallalized merge sort)
Original Array :      83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36
Sorted array:        15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93
Time taken : 0.000338
ritik@ritik-Predator-PH315-51:~$
```

NAME – RITIK GUPTA

REG. NO. – 18BCE0154



VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

**SCHOOL OF COMPUTER SCIENCE ENGINEERING
FALL SEMESTER 2020-2021
CSE4001-PARALLEL AND DISTRIBUTED COMPUTING LAB
ASSESSMENT-5
L1+L2
Dr. K. Murugan**

1. Write MPI code for point to point communication

CODE:

```
#include "mpi.h"

#include<stdio.h>

#include<string.h>

#include<iostream>

using namespace std;

int main(int argc, char** argv)

{

    char reply[100];

    char buff[128];

    int numprocs;
```

```
int myid;  
  
int i;  
  
MPI_Status stat;  
  
MPI_Init(&argc, &argv);  
  
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);  
  
MPI_Comm_rank(MPI_COMM_WORLD, &myid);  
  
if (myid == 0)  
{  
  
cout<<"##### POINT TO POINT COMMUNICATION USING MPI  
#####\n";  
  
cout<<"##### 18BCE0154 #####\n";  
  
printf("WE have %d processors\n", numprocs);  
  
for (i = 1; i < numprocs; i++)  
{  
  
sprintf(buff, "Hello %d", i);  
  
MPI_Send(buff, 128, MPI_CHAR, i, 1234, MPI_COMM_WORLD);  
}  
  
for (i = 1; i < numprocs; i++)  
{  
  
MPI_Recv(buff, 128, MPI_CHAR, i, 4444, MPI_COMM_WORLD, &stat);  
  
cout << buff << endl;  
}  
}
```

```
else
{
MPI_Recv(buff, 128, MPI_CHAR, 0, 1234, MPI_COMM_WORLD, &stat);

sprintf(reply,
" |--> Hello 0, Processor %d is present and accounted for !",
myid);

strcat(buff, reply);

MPI_Send(buff, 128, MPI_CHAR, 0, 4444, MPI_COMM_WORLD);

}

MPI_Finalize();

}
```

OUTPUT:

```
ritik@ritik-Predator-PH315-51:~$ gedit point2point.cpp
ritik@ritik-Predator-PH315-51:~$ mpic++ point2point.cpp -o communication
ritik@ritik-Predator-PH315-51:~$ mpirun -np 4 ./communication

#####
# POINT TO POINT COMMUNICATION USING MPI #####
#####
# 18BCE0154 #####
WE have 4 processors
Hello 1 |--> Hello 0, Processor 1 is present and accounted for !
Hello 2 |--> Hello 0, Processor 2 is present and accounted for !
Hello 3 |--> Hello 0, Processor 3 is present and accounted for !
ritik@ritik-Predator-PH315-51:~$
```

2. Write MPI code for group communication

CODE:

```
#include "mpi.h"
#include <iostream>
#include <string.h>
using namespace std;

int main(int argc, char** argv)
{
    char buff[128];
    int secret_num;
    int numprocs;
    int myid;
    int i;
    MPI_Status stat;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (argc == 1)
    {
        if (myid == 0)
            cout << "Usage: mpirun -np 4 ./BCast SECRET-CODE" << endl;
        MPI_Finalize();
        exit(1);
    }
    if (myid == 0)
    {
        cout << "Usage: mpirun -np 4 ./BCast SECRET-CODE" << endl;
        cout<<"\n##### MPI code for group communication\n";
        #####\n";
```

```
cout<<"##### 18BCE0154 #####\n";
cout << "WE have " << numprocs << " processors" << endl;
secret_num = atoi(argv[1]);
}
MPI_Bcast(&secret_num, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (myid == 0)
{
for (i = 1; i < numprocs; i++)
{
MPI_Recv(buff, 128, MPI_CHAR, i, 0, MPI_COMM_WORLD, &stat);
cout << buff << endl;
}
}
else
{
sprintf(buff, "Processor %d knows the secret code: %d",
myid, secret_num);
MPI_Send(buff, 128, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

OUTPUT:

```
ritik@ritik-Predator-PH315-51:~$ gedit mpigroup.cpp
ritik@ritik-Predator-PH315-51:~$ mpic++ mpigroup.cpp -o communication2
ritik@ritik-Predator-PH315-51:~$ mpirun -np 4 ./communication 9898

##### POINT TO POINT COMMUNICATION USING MPI #####
##### 18BCE0154 #####
WE have 4 processors
Hello 1 |--> Hello 0, Processor 1 is present and accounted for !
Hello 2 |--> Hello 0, Processor 2 is present and accounted for !
Hello 3 |--> Hello 0, Processor 3 is present and accounted for !
ritik@ritik-Predator-PH315-51:~$
```

3. Implement a MPI program to calculate the size of the incoming message.

CODE:

```
#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

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

MPI_Init(NULL, NULL);

int world_size;

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

if (world_size != 2) {

fprintf(stderr, "Must use two processes for this example\n");

MPI_Abort(MPI_COMM_WORLD, 1);

}

int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

```
int number_amount;

if (world_rank == 0) {

const int MAX_NUMBERS = 100;

int numbers[MAX_NUMBERS];

srand(time(NULL));

number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

MPI_Send(numbers, number_amount, MPI_INT, 1, 0,

MPI_COMM_WORLD);

printf("\n##### MPI program to calculate the size of the incoming message

#####\n");

printf("\n##### 18BCE0154 #####\n");

printf("\nProcessor 0 sent %d numbers(size) of messages to Processor 1\n", number_amount);

} else if (world_rank == 1) {

MPI_Status status;

MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

MPI_Get_count(&status, MPI_INT, &number_amount);

int* number_buf = (int*)malloc(sizeof(int) * number_amount);

MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0,

MPI_COMM_WORLD, MPI_STATUS_IGNORE);

printf("Processor 1 dynamically received %d numbers(size) of messages from Processor 0.\n",

number_amount);

free(number_buf);

}
```

```
MPI_Finalize();
```

```
}
```

OUTPUT

```
ritik@ritik-Predator-PH315-51:~$ gedit msgSize.cpp
ritik@ritik-Predator-PH315-51:~$ mpic++ msgSize.cpp -o MessageCount
ritik@ritik-Predator-PH315-51:~$ mpirun -np 2 ./MessageCount

##### MPI program to calculate the size of the incoming message #####
#####
Processor 0 sent 11 numbers(size) of messages to Processor 1
Processor 1 dynamically received 11 numbers(size) of messages from Processor 0.
ritik@ritik-Predator-PH315-51:~$ █
```

NAME – RITIK GUPTA

REG. NO. – 18BCE0154



VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE ENGINEERING

FALL SEMESTER 2020-2021

CSE4001-PARALLEL AND DISTRIBUTED COMPUTING LAB

ASSESSMENT-6

L1+L2

Dr. K. Murugan

1) Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there is a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and its area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{Number in circle}}{\text{Total number of tosses}} = \pi/ 4$$

since the ratio of the area of the circle to the area of the square is $\pi/ 4$

We can use this formula to estimate the value of π with a random number generator:
number_in_circle = 0;

```
for (toss = 0; toss < number_of_tosses; toss++)
{x = random double between -1 and 1;
y = random double between -1 and 1;
distance_squared = x * x + y * y;
if(distance_squared<=1) number_in_circle++;
}
pi_estimate=4*number_in_circle/(double)number_of_tosses;
```

This is called a “Monte Carlo” method, since it uses randomness. Write a program that uses the above Monte Carlo method to estimate π (MPI / Pthreads/OpenMP).

CODE:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <float.h>
#include <string.h>

double square_distance(double x, double y){
    return (x * x) + (y * y);
}

int random_toss_generator(int num_tosses, int rank){
    int number_in_circle = 0;
    int toss;
    double x, y, distance_squared;

    srand((unsigned)time(NULL));
    for (toss = 0; toss < num_tosses; toss++) {
        x = (((double)rand() / RAND_MAX) * 2 ) - 1;
        y = (((double)rand() / RAND_MAX) * 2 ) - 1;
        distance_squared = square_distance(x,y);
        if(distance_squared <= 1) {
            number_in_circle++;
        }
    }
    float pi = 4 * number_in_circle / ((double) num_tosses);
    printf("\nEstimated pi Value: %f\n",pi);
    return number_in_circle;
}

int main (int argc, char *argv[])
{
    int numtasks, rank;
    int num_tosses, number_in_circle;
    double global_toss_sum;

    /* Setup MPI */

```

```

MPI_Init(&argc,&argv);

/* Determine number of tasks and rank */
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if( rank == 0 ) {
    /* Master reads number of tosses */
    printf("\n");
    printf("\n##### PDC ASSIGNMENT 6 #####\n");
    printf("\n NAME: RITIK GUPTA");
    printf("\n REGISTRATION NO - 18BCE0154\n");
    printf("\n Enter the number of tosses: ");
    fflush(stdout);
    scanf("%d", &num_tosses);

    /* Master broadcasts the number to the other processes */
    MPI_Bcast(&num_tosses, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* Reduce to find global sum */
    MPI_Reduce(&number_in_circle, &global_toss_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
    printf("\n Global sum of tosses: %d\n", number_in_circle);

}

else {
    /* Slaves receive the number of tosses */
    MPI_Bcast(&num_tosses, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* Find number of tosses in the circle generated randomly */
    number_in_circle = random_toss_generator(num_tosses, rank);
    printf(" Rank %d has %d tosses in the circle \n ",rank, number_in_circle);

    /* Reduce to find global sum */
    MPI_Reduce(&number_in_circle, &global_toss_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
}

/* Master prints result of pi */

```

```
    MPI_Finalize();  
    return 0;  
}
```

OUTPUT:

```
ritik@ritik-Predator-PH315-51:~$ gedit 18BCE0154.c  
ritik@ritik-Predator-PH315-51:~$ mpicc 18BCE0154.c -o output  
ritik@ritik-Predator-PH315-51:~$ mpirun -np 4 ./output  
  
##### PDC ASSIGNMENT 6 #####  
NAME: RITIK GUPTA  
REGISTRATION NO - 18BCE0154  
  
Enter the number of tosses: 123456  
  
Esimated pi Value:3.138867  
Rank 1 has 96878 tosses in the circle  
  
Esimated pi Value:3.138867  
Rank 3 has 96878 tosses in the circle  
  
Esimated pi Value:3.138867  
Rank 2 has 96878 tosses in the circle
```