HIGH PERFORMANCE COMPUTING

PRACTICAL FILE

(Session: 2019-23)

Submitted by:

Ankit Kumar Yadav
Roll No: 2019UCO1712
COE (Semester VI)

NETAJI SUBHAS UNIVERSITY OF TEHNOLOGY

Sector-3, Dwarka, New Delhi – 110078

INDEX

Practical 1

Aim: Write a parallel program to print “Hello World" using MPI.

```
#include <iostream>
#include "mpi.h"

int main(int* argc, char* argv)
{
    int commsize;
    int rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    printf("Hello World from Process no. %d\n", rank);

    MPI_Finalize();
    return 0;
}
```

Output

```
C:\WINDOWS\System32\cmd.exe
Microsoft Windows [Version 10.0.22563.1]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Desktop\HPC Practical\MPI12\Debug>mpiexec -n 5 MPI12.exe
Hello World from Process no. 1
Hello World from Process no. 0
Hello World from Process no. 2
Hello World from Process no. 4
Hello World from Process no. 3

C:\Users\Administrator\Desktop\HPC Practical\MPI12\Debug>
```

Practical 2

Aim: Write a parallel program to find Sum of an array using MPI.

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

#define n 10

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };


int b[1000];

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

	int process_id, no_of_process,
		elements_per_process,
		n_elements_recieved;

	MPI_Status status;

	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &process_id);
	MPI_Comm_size(MPI_COMM_WORLD, &no_of_process);

		if (process_id == 0) {
		// printf("No of processes = %d\n", no_of_process);
		int index, i;
		elements_per_process = n / no_of_process;

		if (no_of_process > 1) {

			for (i = 1; i < no_of_process - 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);
			}


			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);
		}


		int sum = 0;
		for (i = 0; i < elements_per_process; i++)
			sum += a[i];

		printf("Sum by process %d = %d\n", process_id, sum);


		int tmp;
		for (i = 1; i < no_of_process; i++) {
			MPI_Recv(&tmp, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
			int sender = status.MPI_SOURCE;

			sum += tmp;
```

```
		}


		printf("Final Sum of array is : %d\n", sum);
	}

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

		MPI_Recv(&b, n_elements_recieved, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

		int partial_sum = 0;
		for (int i = 0; i < n_elements_recieved; i++)
			partial_sum += b[i];

		printf("Sum by process %d = %d\n", process_id, partial_sum);

		MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
	}

	MPI_Finalize();

	return 0;
}
```

Output

```
C:\WINDOWS\System32\cmd.exe
Microsoft Windows [Version 10.0.22563.1]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Desktop\HPC Practical\SumOfArray\Debug>mpiexec -n 5 SumOfArray.exe
Sum by process 3 = 15
Sum by process 4 = 19
Sum by process 1 = 7
Sum by process 2 = 11
No of processes = 5
Sum by process 0 = 3
Final Sum of array is : 55

C:\Users\Administrator\Desktop\HPC Practical\SumOfArray\Debug>
```

Practical 3

Aim: Write a C program for parallel implementation of Matrix Multiplication using MPI.

```
#include<stdio.h>
#include<iostream>
#include "mpi.h"
#define NUM_ROWS_A 8
#define NUM_COLUMNS_A 10
#define NUM_ROWS_B 10
#define NUM_COLUMNS_B 8

#define MASTER_TO_SLAVE_TAG 1

#define SLAVE_TO_MASTER_TAG 4
void create_matrix();

void printArray();

int rank;
int size;
int i, j, k;
double A[NUM_ROWS_A][NUM_COLUMNS_A];
double B[NUM_ROWS_B][NUM_COLUMNS_B];
double result[NUM_ROWS_A][NUM_COLUMNS_B];

int low_bound;
int upper_bound;
int portion;
MPI_Status status;
MPI_Request request;

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0)
    { // master process
        create_matrix();
        for (i = 1; i < size; i++)
        {
            portion = (NUM_ROWS_A / (size - 1));
            low_bound = (i - 1) * portion;
            if (((i + 1) == size) && ((NUM_ROWS_A % (size - 1)) != 0))
            {
                upper_bound = NUM_ROWS_A;
            }
            else {
                upper_bound = low_bound + portion;
            }
            MPI_Send(&low_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD);
            MPI_Send(&upper_bound, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG + 1, MPI_COMM_WORLD);
            MPI_Send(&A[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A, MPI_DOUBLE, i,
MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD);
        }
    }

    MPI_Bcast(&B, NUM_ROWS_B * NUM_COLUMNS_B, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* Slave process*/
    if (rank > 0)
    {
        MPI_Recv(&low_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD, &status);
```

```c
        MPI_Recv(&upper_bound, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG + 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&A[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_A, MPI_DOUBLE, 0,
MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &status);

        printf("Process %d calculating for rows %d to %d of Matrix A\n", rank, low_bound,
upper_bound);
        for (i = low_bound; i < upper_bound; i++)
        {
            for (j = 0; j < NUM_COLUMNS_B; j++)
            {
                for (k = 0; k < NUM_ROWS_B; k++)
                {
                    result[i][j] += (A[i][k] * B[k][j]);
                }
            }
        }

        MPI_Send(&low_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD);
        MPI_Send(&upper_bound, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG + 1, MPI_COMM_WORLD);
        MPI_Send(&result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B, MPI_DOUBLE, 0,
SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD);
    }

    if (rank == 0) {
        for (i = 1; i < size; i++) {
            MPI_Recv(&low_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD, &status);
            MPI_Recv(&upper_bound, 1, MPI_INT, i, SLAVE_TO_MASTER_TAG + 1, MPI_COMM_WORLD, &status);
            MPI_Recv(&result[low_bound][0], (upper_bound - low_bound) * NUM_COLUMNS_B, MPI_DOUBLE,
i, SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &status);
        }
        printArray();
    }
    MPI_Finalize();
    return 0;
}

void create_matrix()
{
    for (i = 0; i < NUM_ROWS_A; i++) {
        for (j = 0; j < NUM_COLUMNS_A; j++) {
            A[i][j] = i + j;
        }
    }
    for (i = 0; i < NUM_ROWS_B; i++) {
        for (j = 0; j < NUM_COLUMNS_B; j++) {
            B[i][j] = i * j;
        }
    }
}

void printArray()
{
    printf("The matrix A is: \n");
    for (i = 0; i < NUM_ROWS_A; i++) {
        printf("\n");
        for (j = 0; j < NUM_COLUMNS_A; j++)
            printf("%8.2f  ", A[i][j]);
    }
    printf("\n\n\n");
    printf("The matrix B is: \n");
    for (i = 0; i < NUM_ROWS_B; i++) {
        printf("\n");
        for (j = 0; j < NUM_COLUMNS_B; j++)
            printf("%8.2f  ", B[i][j]);
    }
    printf("\n\n\n");
    printf("The result matrix is: \n");
    for (i = 0; i < NUM_ROWS_A; i++) {
        printf("\n");
```

```c
        for (j = 0; j < NUM_COLUMNS_B; j++)
            printf("%8.2f  ", result[i][j]);
    }
    printf("\n\n");
}
```

Output

C:\WINDOWS\System32\cmd.exe

```
C:\Users\Administrator\Desktop\HPC Practical\MatrixMultiplication\Debug>mpiexec -n 5 MatrixMultiplication.exe
Process 4 calculating for rows 6 to 8 of Matrix A
Process 2 calculating for rows 2 to 4 of Matrix A
Process 3 calculating for rows 4 to 6 of Matrix A
The matrix A is:

    0.00      1.00      2.00      3.00      4.00      5.00      6.00      7.00      8.00      9.00
    1.00      2.00      3.00      4.00      5.00      6.00      7.00      8.00      9.00     10.00
    2.00      3.00      4.00      5.00      6.00      7.00      8.00      9.00     10.00     11.00
    3.00      4.00      5.00      6.00      7.00      8.00      9.00     10.00     11.00     12.00
    4.00      5.00      6.00      7.00      8.00      9.00     10.00     11.00     12.00     13.00
    5.00      6.00      7.00      8.00      9.00     10.00     11.00     12.00     13.00     14.00
    6.00      7.00      8.00      9.00     10.00     11.00     12.00     13.00     14.00     15.00
    7.00      8.00      9.00     10.00     11.00     12.00     13.00     14.00     15.00     16.00


The matrix B is:

    0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
    0.00      1.00      2.00      3.00      4.00      5.00      6.00      7.00
    0.00      2.00      4.00      6.00      8.00     10.00     12.00     14.00
    0.00      3.00      6.00      9.00     12.00     15.00     18.00     21.00
    0.00      4.00      8.00     12.00     16.00     20.00     24.00     28.00
    0.00      5.00     10.00     15.00     20.00     25.00     30.00     35.00
    0.00      6.00     12.00     18.00     24.00     30.00     36.00     42.00
    0.00      7.00     14.00     21.00     28.00     35.00     42.00     49.00
    0.00      8.00     16.00     24.00     32.00     40.00     48.00     56.00
    0.00      9.00     18.00     27.00     36.00     45.00     54.00     63.00


The result matrix is:

    0.00    285.00    570.00    855.00   1140.00   1425.00   1710.00   1995.00
    0.00    330.00    660.00    990.00   1320.00   1650.00   1980.00   2310.00
    0.00    375.00    750.00   1125.00   1500.00   1875.00   2250.00   2625.00
    0.00    420.00    840.00   1260.00   1680.00   2100.00   2520.00   2940.00
    0.00    465.00    930.00   1395.00   1860.00   2325.00   2790.00   3255.00
    0.00    510.00   1020.00   1530.00   2040.00   2550.00   3060.00   3570.00
    0.00    555.00   1110.00   1665.00   2220.00   2775.00   3330.00   3885.00
    0.00    600.00   1200.00   1800.00   2400.00   3000.00   3600.00   4200.00

Process 1 calculating for rows 0 to 2 of Matrix A
```

Practical 4

Aim: Write a C program to implement the Quick Sort Algorithm using MPI.

```
#include "mpi.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
using namespace std;
#define ARRAY_SIZE 20

void swap(int* arr, int i, int j) {
	int t = arr[i];
	arr[i] = arr[j];
	arr[j] = t;
}

void quicksort(int* arr, int start, int end)
{
	int pivot, index;

	if (end <= 1)
		return;

	pivot = arr[start + end / 2];
	swap(arr, start, start + end / 2);

	index = start;

	for (int i = start + 1; i < start + end; i++) {

		if (arr[i] < pivot) {
			index++;
			swap(arr, i, index);
		}
	}

	swap(arr, start, index);
	quicksort(arr, start, index - start);
	quicksort(arr, index + 1, start + end - index - 1);
}

int* merge(int* arr1, int n1, int* arr2, int n2)
{
	int* result = (int*)malloc((n1 + n2) * sizeof(int));
	int i = 0;
	int j = 0;
	int k;

	for (k = 0; k < n1 + n2; k++) {
		if (i >= n1) {
			result[k] = arr2[j];
			j++;
		}
		else if (j >= n2) {
			result[k] = arr1[i];
			i++;
		}

		else if (arr1[i] < arr2[j]) {
			result[k] = arr1[i];
			i++;
		}

		else {
```

```
                        result[k] = arr2[j];
                        j++;
                }
        }
        return result;
}

int main(int argc, char* argv[])
{
        int number_of_elements;
        int* data = NULL;
        int chunk_size, own_chunk_size;
        int* chunk;
        MPI_Status status;

        int number_of_process, rank_of_process;
        MPI_Init(&argc, &argv);

        MPI_Comm_size(MPI_COMM_WORLD, &number_of_process);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank_of_process);

        if (rank_of_process == 0)
        {
                number_of_elements = ARRAY_SIZE;

                data = (int*)malloc(number_of_elements * sizeof(int));

                for (int i = 0; i < number_of_elements; i++)
                        data[i] = ARRAY_SIZE - i;

                printf("The initial array is : \n");
                for (int i = 0; i < number_of_elements; i++)
                        printf("%d ", data[i]);

                printf("\n");
        }


        MPI_Barrier(MPI_COMM_WORLD);


        MPI_Bcast(&number_of_elements, 1, MPI_INT, 0, MPI_COMM_WORLD);


        if (number_of_elements % number_of_process == 0)
                chunk_size = number_of_elements / number_of_process;

        else
                chunk_size = (number_of_elements / (number_of_process - 1));


        chunk = (int*)malloc(chunk_size * sizeof(int));


        MPI_Scatter(data, chunk_size, MPI_INT, chunk, chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

        free(data);
        data = NULL;


        own_chunk_size = (number_of_elements >= chunk_size * (rank_of_process + 1)) ?
                chunk_size : (number_of_elements -
                        chunk_size * rank_of_process);

        printf("The process %d sorted the following array: \n", rank_of_process);
        for (int i = 0;i < own_chunk_size;i++)
                printf("%d ", chunk[i]);
        printf("\n");
```

```
        quicksort(chunk, 0, own_chunk_size);

        for (int step = 1; step < number_of_process; step = 2 * step)
        {
                if (rank_of_process % (2 * step) != 0)
                {
                        MPI_Send(chunk, own_chunk_size, MPI_INT, rank_of_process - step, 0,
MPI_COMM_WORLD);
                        break;
                }

                if (rank_of_process + step < number_of_process)
                {
                        int received_chunk_size = (number_of_elements >= chunk_size * (rank_of_process
+ 2 * step)) ? (chunk_size * step)
                                : (number_of_elements - chunk_size * (rank_of_process + step));

                        int* chunk_received;
                        chunk_received = (int*)malloc(received_chunk_size * sizeof(int));

                        MPI_Recv(chunk_received, received_chunk_size, MPI_INT, rank_of_process + step,
0,
                                MPI_COMM_WORLD, &status);

                        data = merge(chunk, own_chunk_size, chunk_received, received_chunk_size);

                        free(chunk);
                        free(chunk_received);
                        chunk = data;
                        own_chunk_size = own_chunk_size + received_chunk_size;
                }
        }


        if (rank_of_process == 0)
        {
                printf("The Sorted array is: \n");

                for (int i = 0; i < number_of_elements; i++)
                        printf("%d ", chunk[i]);

        }

        MPI_Finalize();
        return 0;
}
```

Output

```
C:\WINDOWS\System32\cmd.exe
Microsoft Windows [Version 10.0.22563.1]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Desktop\HPC Practical\QuickSort\Debug>mpiexec -n 5 QuickSort
The process 2 sorted the following array:
12 11 10 9
The initial array is :
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
The process 0 sorted the following array:
20 19 18 17
The Sorted array is:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
The process 1 sorted the following array:
16 15 14 13
The process 3 sorted the following array:
8 7 6 5
The process 4 sorted the following array:
4 3 2 1
```

Practical 5

Aim: Write a multithreaded program to generate Fibonacci series using pThreads.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<signal.h>
#include<pthread.h>

int data[200];

void * ThreadFunction(void *num) {
	data[1] = 0;
	data[2] = 1;
	int n = *((int *)num);
	if (n > 2) {
		for (int i = 3; i <= n; i++) {
			data[i] = data[i - 1] + data[i - 2];
		}
	}
	pthread_detach(pthread_self());
	pthread_exit(NULL);
}


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

	if (argc != 2) {
		printf("ERROR!! INCORRECT PARAMETERS \n");
	}

	int n = atoi(argv[1]);


	pthread_t ptid;

	printf("Creating fibonacci series till %d th element! \n", n);

	pthread_create(&ptid, NULL, &ThreadFunction, &n);

	pthread_join(ptid, NULL);

	printf("The series is :\n");

	for (int i = 1; i <= n; i++) {
		printf("%d ", data[i]);
	}

	printf("\n");

	pthread_exit(NULL);

	return (0);
```

}

Output

```
yash@yash-VirtualBox:~/HPC Practical$ gcc -pthread fibonacci.c
yash@yash-VirtualBox:~/HPC Practical$ ./a.out 8
Creating fibonacci series till 8 th element!
The series is :
0 1 1 2 3 5 8 13
yash@yash-VirtualBox:~/HPC Practical$
```

Practical 6

Aim: Write a program to implement Process Synchronization by mutex locks using pThreads.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<pthread.h>

int shared = 1;
pthread_mutex_t lock;

void * ThreadFunction1(void *num) {
        int x;

        printf("Acquiring lock (thread 1)...\n");

        pthread_mutex_lock(&lock);

        printf("Lock acquired  by thread 1...\n");

        printf("Thread1 read : %d\n", shared);

        shared++;

        printf("Thread1 updates the variable to : %d\n", shared);

        pthread_mutex_unlock(&lock);

        printf("Relinquishing lock (thread 1)...\n");

}


void * ThreadFunction2(void *num) {
        int x;

        printf("Acquiring lock (thread 2)...\n");

        pthread_mutex_lock(&lock);

        printf("Lock acquired  by thread 2...\n");

        printf("Thread2 read : %d\n", shared);

        shared--;

        printf("Thread 2 updates the variable to : %d\n", shared);

        pthread_mutex_unlock(&lock);

        printf("Relinquishing lock (thread 2)...\n");
}
```

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

    pthread_mutex_init(&lock, NULL);

    pthread_t ptid1, ptid2;


    pthread_create(&ptid1, NULL, &ThreadFunction1, NULL);

    pthread_create(&ptid2, NULL, &ThreadFunction2, NULL);

    pthread_join(ptid1, NULL);
    pthread_join(ptid2, NULL);

    printf("The final updated value of shared variable is: %d\n", shared);

    pthread_exit(NULL);

    return (0);

}
```

Output

```
yash@yash-VirtualBox:~/HPC Practical$ gcc -pthread lock.c
yash@yash-VirtualBox:~/HPC Practical$ ./a.out
Acquiring lock (thread 2)...
Lock acquired  by thread 2...
Thread2 read : 1
Thread 2 updates the variable to : 0
Relinquishing lock (thread 2)...
Acquiring lock (thread 1)...
Lock acquired  by thread 1...
Thread1 read : 0
Thread1 updates the variable to : 1
Relinquishing lock (thread 1)...
The final updated value of shared variable is: 1
yash@yash-VirtualBox:~/HPC Practical$
```

Practical 7

Aim: Write a C program to demonstrate multitask using OpenMP.

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

int fib(int n)
{
	int res;
	if (n == 0 or n == 1)
		res = n;
	else
	{
		int a, b;
		#pragma omp task shared(a)
		a = fib(n - 1);
		#pragma omp task shared(b)
		b = fib(n - 2);

		#pragma omp taskwait
		res = a + b;
	}

	printf("%d th Fibonacci task calculated by process %d\n", n, omp_get_thread_num());
	return res;
}

int main(int argc, char *argv[])
{
	#pragma omp parallel
	#pragma omp single
	{
		int n = atoi(argv[1]);
		printf("\nThe %d th Fibonacci Number = %d\n", n, fib(n));
	}
}
```

Output

```
yash@yash-VirtualBox:~/HPC Practical$ g++ -fopenmp multitask.c
yash@yash-VirtualBox:~/HPC Practical$ ./a.out 3
1 th Fibonacci task calculated by process 0
0 th Fibonacci task calculated by process 0
1 th Fibonacci task calculated by process 0
2 th Fibonacci task calculated by process 0
3 th Fibonacci task calculated by process 0

The 3 th Fibonacci Number = 2
yash@yash-VirtualBox:~/HPC Practical$
```

Practical 8

Aim: Write a C program to demonstrate default, static and dynamic loop scheduling using OpenMP.

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

int main()
{
	int i, N = 10, THREAD_COUNT = 3, CHUNK_SIZE = 3;
	printf("Default Scheduling\n");
	#pragma omp parallel for num_threads(THREAD_COUNT)
	for (i = 0; i < N; i++)
		printf("ThreadID: %d, iteration: %d\n", omp_get_thread_num(), i);

	printf("\nStatic Scheduling\n");
	#pragma omp parallel for num_threads(THREAD_COUNT) schedule(static, CHUNK_SIZE)
	for (i = 0; i < N; i++)
		printf("ThreadID: %d, iteration: %d\n", omp_get_thread_num(), i);

	printf("\nDynamic Scheduling\n");
	#pragma omp parallel for num_threads(THREAD_COUNT) schedule(dynamic, CHUNK_SIZE)
	for (i = 0; i < N; i++)
		printf("ThreadID: %d, iteration: %d\n", omp_get_thread_num(), i);
	return 0;
}
```

Output

```
yash@yash-VirtualBox:~/HPC Practical$ g++ -fopenmp loop.c
yash@yash-VirtualBox:~/HPC Practical$ ./a.out
Default Scheduling
ThreadID: 1, iteration: 4
ThreadID: 1, iteration: 5
ThreadID: 1, iteration: 6
ThreadID: 0, iteration: 0
ThreadID: 0, iteration: 1
ThreadID: 0, iteration: 2
ThreadID: 0, iteration: 3
ThreadID: 2, iteration: 7
ThreadID: 2, iteration: 8
ThreadID: 2, iteration: 9

Static Scheduling
ThreadID: 1, iteration: 3
ThreadID: 1, iteration: 4
ThreadID: 1, iteration: 5
ThreadID: 0, iteration: 0
ThreadID: 0, iteration: 1
ThreadID: 0, iteration: 2
ThreadID: 0, iteration: 9
ThreadID: 2, iteration: 6
ThreadID: 2, iteration: 7
ThreadID: 2, iteration: 8

Dynamic Scheduling
ThreadID: 1, iteration: 0
ThreadID: 1, iteration: 1
ThreadID: 1, iteration: 2
ThreadID: 1, iteration: 9
ThreadID: 2, iteration: 6
ThreadID: 2, iteration: 7
ThreadID: 2, iteration: 8
ThreadID: 0, iteration: 3
ThreadID: 0, iteration: 4
ThreadID: 0, iteration: 5
yash@yash-VirtualBox:~/HPC Practical$
```