High Performance Computing

2021UCS1513

Farhan Khan

CSE - 01

LAB 1

Aim:

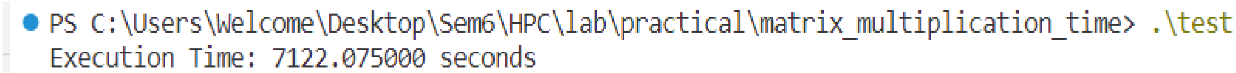
Write a program to multiply two matrices of size N * N, where N = 10000.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 10000
void multiplyMatrices(int firstMatrix[N][N], int secondMatrix[N][N],
                      int result[N][N])
{
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            int sum = 0;
            for (int k = 0; k < N; ++k)
            {
                sum += firstMatrix[i][k] * secondMatrix[k][j];
            }
            result[i][j] = sum;
        }
    }
}
int main()
{
    int(*firstMatrix)[N] = malloc(sizeof(int[N][N]));
    int(*secondMatrix)[N] = malloc(sizeof(int[N][N]));
    int(*result)[N] = malloc(sizeof(int[N][N]));
    clock_t start_time = clock();
    multiplyMatrices(firstMatrix, secondMatrix, result);
    clock_t end_time = clock();
    printf("Execution Time: %f seconds\n", ((double)(end_time -
                                                 start_time)) /
```

```
                                               CLOCKS_PER_SEC);
    free(firstMatrix);
    free(secondMatrix);
    free(result);
    return 0;
}
```

Output:



```
PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\practical\matrix_multiplication_time> .\test
Execution Time: 7122.075000 seconds
```

LAB 2

Aim:

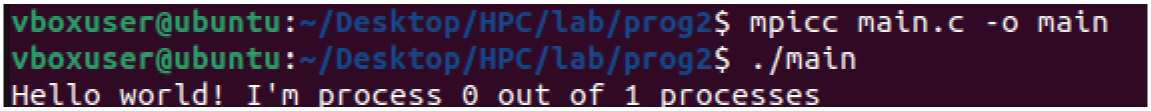
Write a parallel program to print “Hello World" using MPI

Definitions:

Code:

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

int main(int argc, char **argv)
{
    int ierr, num_procs, my_id;
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    printf("Hello world! I'm process %i out of %i processes\n", my_id, num_procs);
    ierr = MPI_Finalize();
}
```

Output:



```
vboxuser@ubuntu:~/Desktop/HPC/lab/prog2$ mpicc main.c -o main
vboxuser@ubuntu:~/Desktop/HPC/lab/prog2$ ./main
Hello world! I'm process 0 out of 1 processes
```

LAB 3

Aim:

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

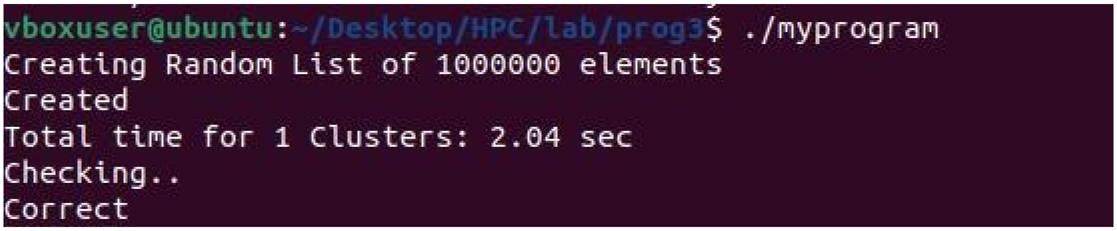
Code:

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <stdbool.h>
#include <stdlib.h>
#define SIZE 1000000

int partition(int *arr, int low, int high)
{
    int pivot = arr[low];
    int i = low + 1;
    int j = high;
    int temp;
    do
    {
        while (arr[i] <= pivot)
            i++;
        while (arr[j] > pivot)
            j--;

        if (j > i)
        {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    } while (j > i);

    temp = arr[low];
    arr[low] = arr[j];
    arr[j] = temp;

    return j;
}
```

```
int hoare_partition(int *arr, int low, int high)
{
    int mid = (low + high) / 2;
    int pivot = arr[mid];

    int j, temp;

    temp = arr[mid];
    arr[mid] = arr[high];
    arr[high] = temp;

    int i = low - 1;
    for (j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

void quicksort(int *number, int low, int high)
{
    if (low < high)
    {
        int pivot = partition(number, low, high);
        quicksort(number, low, pivot - 1);
        quicksort(number, pivot + 1, high);
    }
}

int quicksort_recursive(int *arr, int arrSize, int currProcRank, int maxRank, int
rankIndex)
{
    MPI_Status status;
    int shareProc = currProcRank + pow(2, rankIndex);
    rankIndex++;

    if (shareProc > maxRank)
    {
        MPI_Barrier(MPI_COMM_WORLD);
        quicksort(arr, 0, arrSize - 1);
        return 0;
    }

    int j = 0;
    int pivotIndex;
    pivotIndex = hoare_partition(arr, j, arrSize - 1);

    if (pivotIndex <= arrSize - pivotIndex)
    {
        MPI_Send(arr, pivotIndex, MPI_INT, shareProc, pivotIndex, MPI_COMM_WORLD);
        quicksort_recursive((arr + pivotIndex + 1), (arrSize - pivotIndex - 1),
currProcRank, maxRank, rankIndex);
        MPI_Recv(arr, pivotIndex, MPI_INT, shareProc, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
    }
    else
    {
        MPI_Send((arr + pivotIndex + 1), (arrSize - pivotIndex - 1), MPI_INT,
shareProc, pivotIndex + 1, MPI_COMM_WORLD);
        quicksort_recursive(arr, pivotIndex, currProcRank, maxRank, rankIndex);
        MPI_Recv((arr + pivotIndex + 1), (arrSize - pivotIndex - 1), MPI_INT,
shareProc, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    }
}

int main(int argc, char *argv[])
{
    int unsorted_array[SIZE];
    int array_size = SIZE;
    int size, rank;

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

    if (rank == 0)
    {
        printf("Creating Random List of %d elements\n", SIZE);
        for (int j = 0; j < SIZE; j++)
        {
            unsorted_array[j] = (int)rand() % 1000;
        }
        printf("Created\n");
    }

    int rankPower = 0;
    while (pow(2, rankPower) <= rank)
        rankPower++;

    MPI_Barrier(MPI_COMM_WORLD);
    double start_timer, finish_timer;

    if (rank == 0)
    {
        start_timer = MPI_Wtime();
        quicksort_recursive(unsorted_array, array_size, rank, size - 1,
rankPower);
    }
    else
    {
        MPI_Status status;
        int subarray_size;

        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        MPI_Get_count(&status, MPI_INT, &subarray_size);
        int source_process = status.MPI_SOURCE;
        int subarray[subarray_size];
        MPI_Recv(subarray, subarray_size, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        quicksort_recursive(subarray, subarray_size, rank, size - 1, rankPower);
        MPI_Send(subarray, subarray_size, MPI_INT, source_process, 0,
MPI_COMM_WORLD);
    }

    if (rank == 0)
    {
        finish_timer = MPI_Wtime();
        printf("Total time for %d Clusters: %2.2f sec \n", size, finish_timer -
start_timer);
```

```
        printf("Checking..\n");
        bool error = false;
        int i = 0;
        for (i = 0; i < SIZE - 1; i++)
        {
            if (unsorted_array[i] > unsorted_array[i + 1])
            {
                error = true;
                printf("error in i = %d \n", i);
            }
        }

        if (error)
        {
            printf("Error.. Not sorted correctly\n");
        }
        else
            printf("Correct\n");
    }

    MPI_Finalize();
    return 0;
}
```

Output:



```
vboxuser@ubuntu:~/Desktop/HPC/lab/prog3$ ./myprogram
Creating Random List of 1000000 elements
Created
Total time for 1 Clusters: 2.04 sec
Checking..
Correct
```

LAB 4

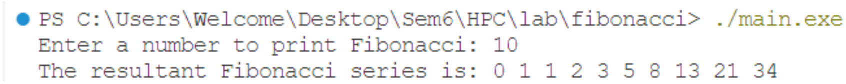
Aim:

Write a multithreaded program to generate Fibonacci series using pThreads

Code:

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

struct ThreadArgs
{
    int n;
    int *arr;
};

void *fib(void *arg)
{
    struct ThreadArgs *args = (struct ThreadArgs *)arg;
    int n = args->n;
    int *arr = args->arr;
    arr[0] = 0;
    arr[1] = 1;

    if (n > 1)
    {
        for (int i = 2; i < n; i++)
        {
            arr[i] = arr[i - 1] + arr[i - 2];
        }
    }

    pthread_exit(NULL);
}

int main()
{
    int n;
    printf("Enter a number to print Fibonacci: ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));

    pthread_t tid;
    struct ThreadArgs args;
    args.n = n;
    args.arr = arr;

    pthread_create(&tid, NULL, fib, (void *)&args);

    pthread_join(tid, NULL);

    printf("The resultant Fibonacci series is: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
    free(arr);

    return 0;
}
```

Output:



```
● PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\fibonacci> ./main.exe
  Enter a number to print Fibonacci: 10
  The resultant Fibonacci series is: 0 1 1 2 3 5 8 13 21 34
```

LAB 5

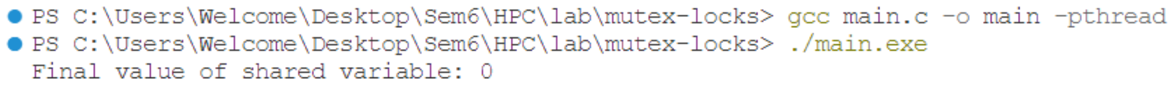
Aim:

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

Code:

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

#define NUM_THREADS 2
#define ITERATIONS 10000

int shared_variable = 0;
pthread_mutex_t mutex;

void *increment(void *arg)
{
    for (int i = 0; i < ITERATIONS; ++i)
    {
        pthread_mutex_lock(&mutex);
        shared_variable++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

void *decrement(void *arg)
{
    for (int i = 0; i < ITERATIONS; ++i)
    {
        pthread_mutex_lock(&mutex);
        shared_variable--;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&threads[0], NULL, increment, NULL);
    pthread_create(&threads[1], NULL, decrement, NULL);

    for (int i = 0; i < NUM_THREADS; ++i)
    {
        pthread_join(threads[i], NULL);
    }

    printf("Final value of shared variable: %d\n", shared_variable);

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Output:



```
● PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\mutex-locks> gcc main.c -o main -pthread
● PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\mutex-locks> ./main.exe
  Final value of shared variable: 0
```

LAB 6

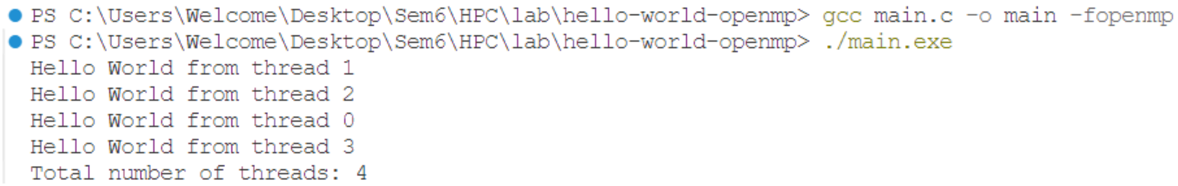
Aim:

Write a "Hello World" program using OpenMP library also display number of threads created during execution.

Code:

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

int main()
{
    int num_threads;

    omp_set_num_threads(4);

#pragma omp parallel
    {
        num_threads = omp_get_num_threads();

        int tid = omp_get_thread_num();

        printf("Hello World from thread %d\n", tid);
    }

    printf("Total number of threads: %d\n", num_threads);

    return 0;
}
```

Output:



```
PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\hello-world-openmp> gcc main.c -o main -fopenmp
PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\hello-world-openmp> ./main.exe
Hello World from thread 1
Hello World from thread 2
Hello World from thread 0
Hello World from thread 3
Total number of threads: 4
```

LAB 7

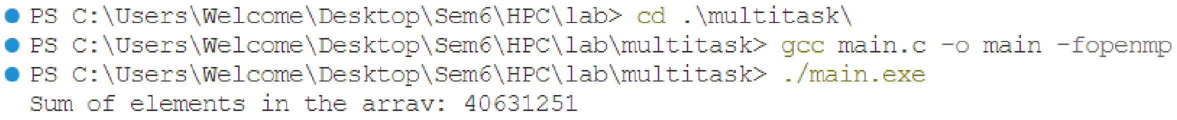
Aim:

Write a C program to demonstrate multitask using OpenMP

Code:

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

#define ARRAY_SIZE 10000
#define NUM_THREADS 4

int main()
{
    int i, sum = 0;
    int arr[ARRAY_SIZE];

    for (i = 0; i < ARRAY_SIZE; i++)
    {
        arr[i] = i + 1;
    }

#pragma omp parallel num_threads(NUM_THREADS) reduction(+ : sum)
    {
        int thread_id = omp_get_thread_num();
        int chunk_size = ARRAY_SIZE / omp_get_num_threads();
        int start = thread_id * chunk_size;
        int end = (thread_id == omp_get_num_threads() - 1) ? ARRAY_SIZE : start +
chunk_size;

        for (i = start; i < end; i++)
        {
            sum += arr[i];
        }
    }

    printf("Sum of elements in the array: %d\n", sum);
    return 0;
}
```

Output:



```
PS C:\Users\Welcome\Desktop\Sem6\HPC\lab> cd .\multitask\
PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\multitask> gcc main.c -o main -fopenmp
PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\multitask> ./main.exe
Sum of elements in the array: 40631251
```

LAB 8

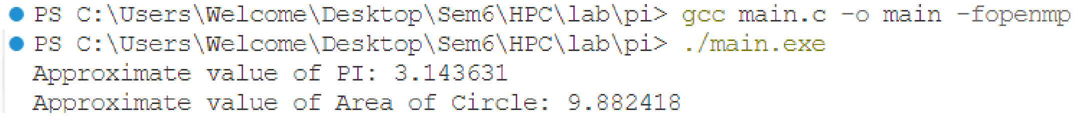
Aim:

Write a parallel program to calculate the value of PI/Area of Circle using OpenMP library.

Code:

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

#define NUM_STEPS 1000000000

int main()
{
    double step = 1.0 / NUM_STEPS;
    double sum = 0.0;
    double x;

#pragma omp parallel for reduction(+ : sum)
    for (int i = 0; i < NUM_STEPS; i++)
    {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }

    double pi = sum * step;

    printf("Approximate value of PI: %f\n", pi);
    printf("Approximate value of Area of Circle: %f\n", pi * pi);

    return 0;
}
```

Output:



```
● PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\pi> gcc main.c -o main -fopenmp
● PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\pi> ./main.exe
  Approximate value of PI: 3.143631
  Approximate value of Area of Circle: 9.882418
```

LAB 9

Aim:

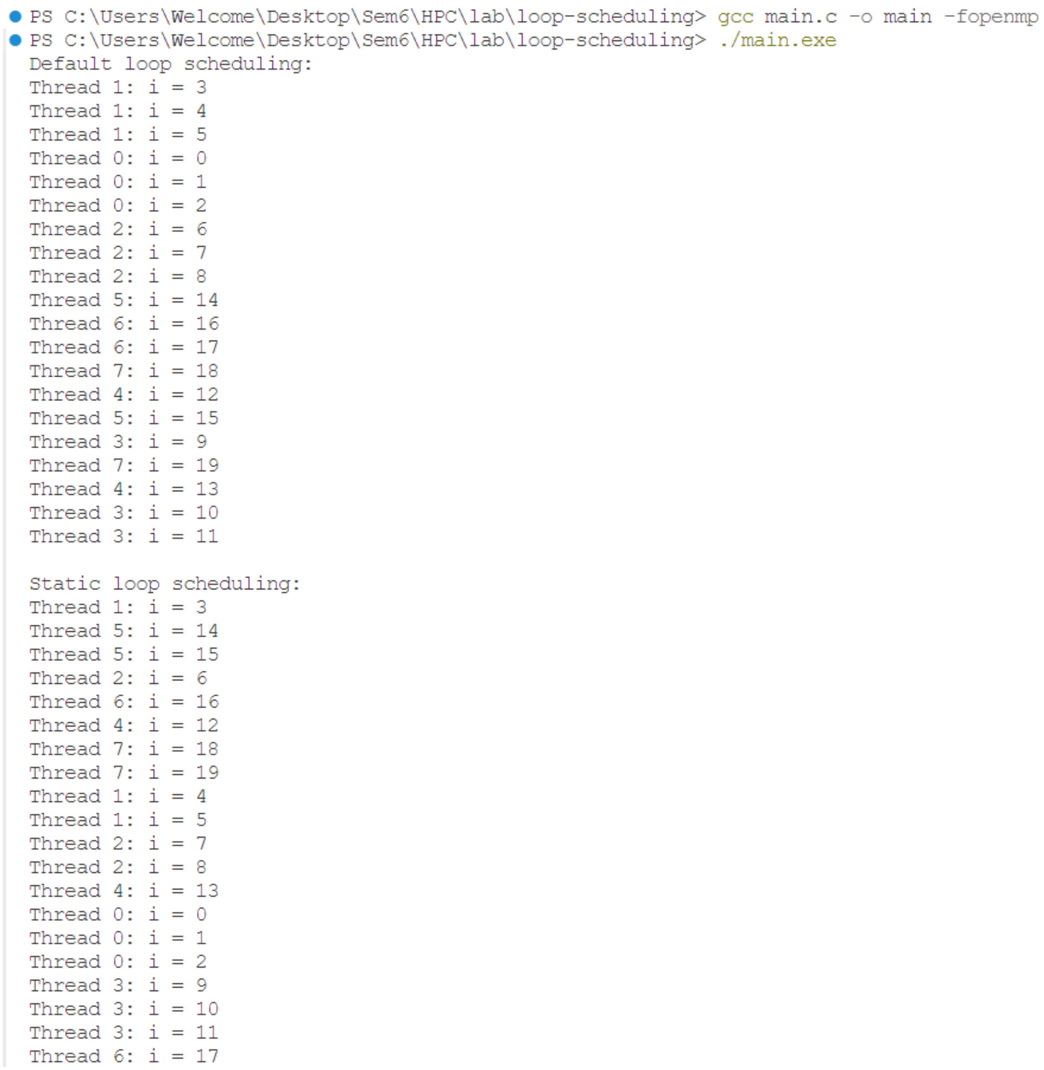
Write a C program to demonstrate default, static and dynamic loop scheduling using OpenMP.

Code:

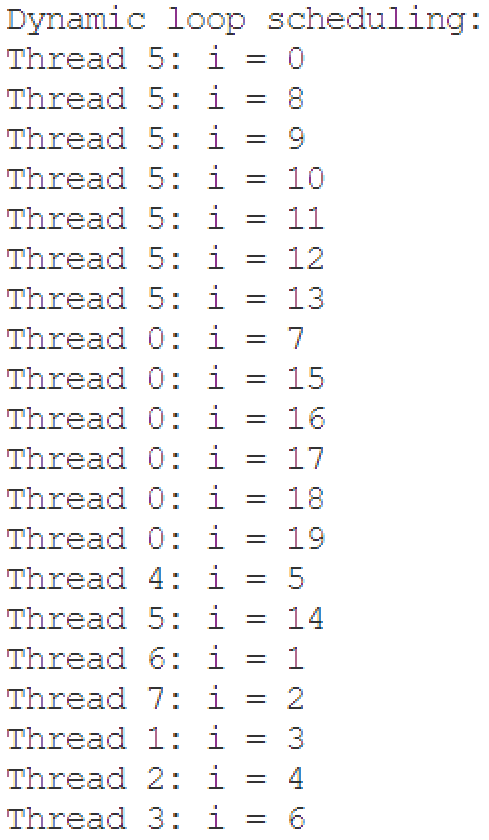
```
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define NUM_ITERATIONS 20

int main()
{
    int i, tid;

    printf("Default loop scheduling:\n");
#pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
#pragma omp for
        for (i = 0; i < NUM_ITERATIONS; i++)
        {
            printf("Thread %d: i = %d\n", tid, i);
        }
    }

    printf("\nStatic loop scheduling:\n");
#pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
#pragma omp for schedule(static)
        for (i = 0; i < NUM_ITERATIONS; i++)
        {
            printf("Thread %d: i = %d\n", tid, i);
        }
    }

    printf("\nDynamic loop scheduling:\n");
```

```c
#pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
#pragma omp for schedule(dynamic)
        for (i = 0; i < NUM_ITERATIONS; i++)
        {
            printf("Thread %d: i = %d\n", tid, i);
        }
    }

    return 0;
}
```

Output:



```
PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\loop-scheduling> gcc main.c -o main -fopenmp
PS C:\Users\Welcome\Desktop\Sem6\HPC\lab\loop-scheduling> ./main.exe
Default loop scheduling:
Thread 1: i = 3
Thread 1: i = 4
Thread 1: i = 5
Thread 0: i = 0
Thread 0: i = 1
Thread 0: i = 2
Thread 2: i = 6
Thread 2: i = 7
Thread 2: i = 8
Thread 5: i = 14
Thread 6: i = 16
Thread 6: i = 17
Thread 7: i = 18
Thread 4: i = 12
Thread 5: i = 15
Thread 3: i = 9
Thread 7: i = 19
Thread 4: i = 13
Thread 3: i = 10
Thread 3: i = 11

Static loop scheduling:
Thread 1: i = 3
Thread 5: i = 14
Thread 5: i = 15
Thread 2: i = 6
Thread 6: i = 16
Thread 4: i = 12
Thread 7: i = 18
Thread 7: i = 19
Thread 1: i = 4
Thread 1: i = 5
Thread 2: i = 7
Thread 2: i = 8
Thread 4: i = 13
Thread 0: i = 0
Thread 0: i = 1
Thread 0: i = 2
Thread 3: i = 9
Thread 3: i = 10
Thread 3: i = 11
Thread 6: i = 17
```



```
Dynamic loop scheduling:
Thread 5: i = 0
Thread 5: i = 8
Thread 5: i = 9
Thread 5: i = 10
Thread 5: i = 11
Thread 5: i = 12
Thread 5: i = 13
Thread 0: i = 7
Thread 0: i = 15
Thread 0: i = 16
Thread 0: i = 17
Thread 0: i = 18
Thread 0: i = 19
Thread 4: i = 5
Thread 5: i = 14
Thread 6: i = 1
Thread 7: i = 2
Thread 1: i = 3
Thread 2: i = 4
Thread 3: i = 6
```