Class: Final Year (Computer Science and Engineering)
Year: 2023-24
Semester: 1
Course: High Performance Computing Lab

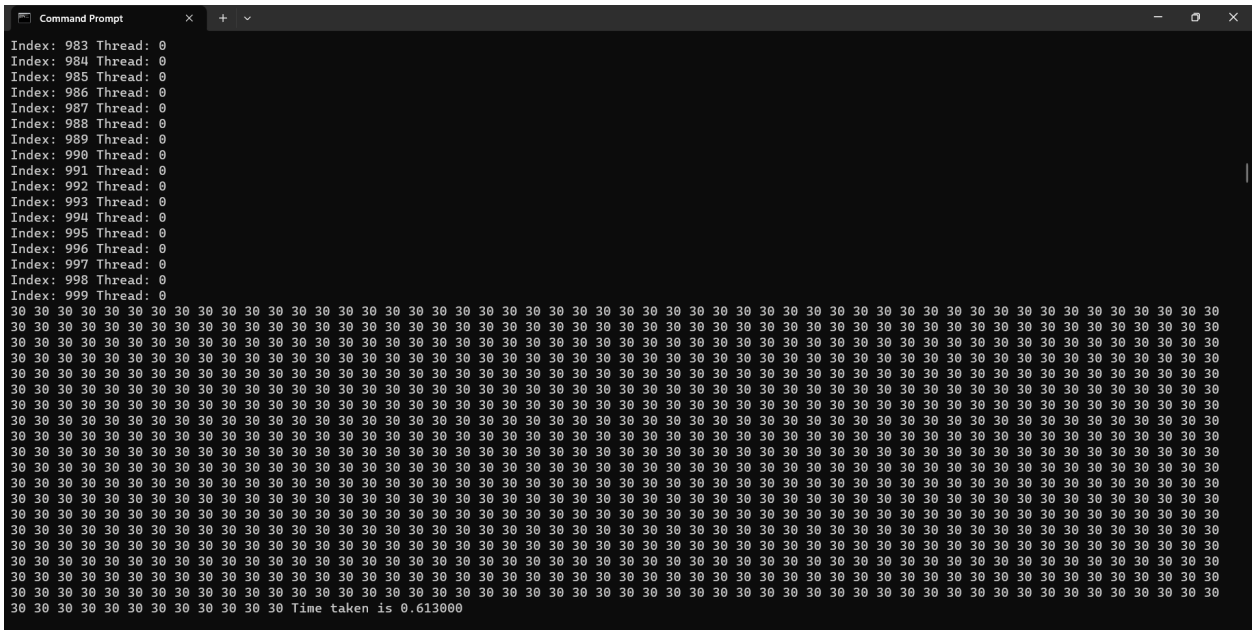
Practical No. 2

Exam Seat No.: 2020BTECS00029
Name: Akshata Gawande
Title: Study and implementation of basic OpenMP clauses

1. Vector Scalar Addition
   Sequential (Source Code):

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

int main() {
    int N = 1000;
    int A[1000];
    for (int i = 0; i < N; i++)
        A[i] = 10;

    int B[1000];
    for (int i = 0; i < N; i++)
        B[i] = 20;

    int C[1000] = {0};
    double stime = omp_get_wtime();
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
        printf("Index: %d Thread: %d\n", i, omp_get_thread_num());
    }

    for (int i = 0; i < N; i++) {
        printf("%d ", C[i]);
    }

    double etime = omp_get_wtime();
    double time = etime - stime;
    printf("Time taken is %f\n", time);
    return 0;
}
```

Output:



We have performs element-wise addition of two arrays `A` and `B`, storing the result in array `C`. It also measures and prints the time taken for the computation. The program utilizes OpenMP for parallelism, enabling multiple threads to execute the addition concurrently.

Information:

The code utilizes OpenMP to create a parallel region where multiple threads can execute the addition of array elements concurrently.

It defines three arrays: `A` and `B` of size 1000, both initialized with constant values, and `C` of the same size to store the result.

A timer (`omp_get_wtime()`) is used to measure the execution time of the parallel computation.

After the addition of elements, the code prints the index of the element being processed and the thread number executing that addition.

Finally, it prints the elements of array `C` and the time taken for the computation.

Analysis:

This code demonstrates a simple parallel computation using OpenMP, where the addition of each element in arrays `A` and `B` is performed concurrently by multiple threads.

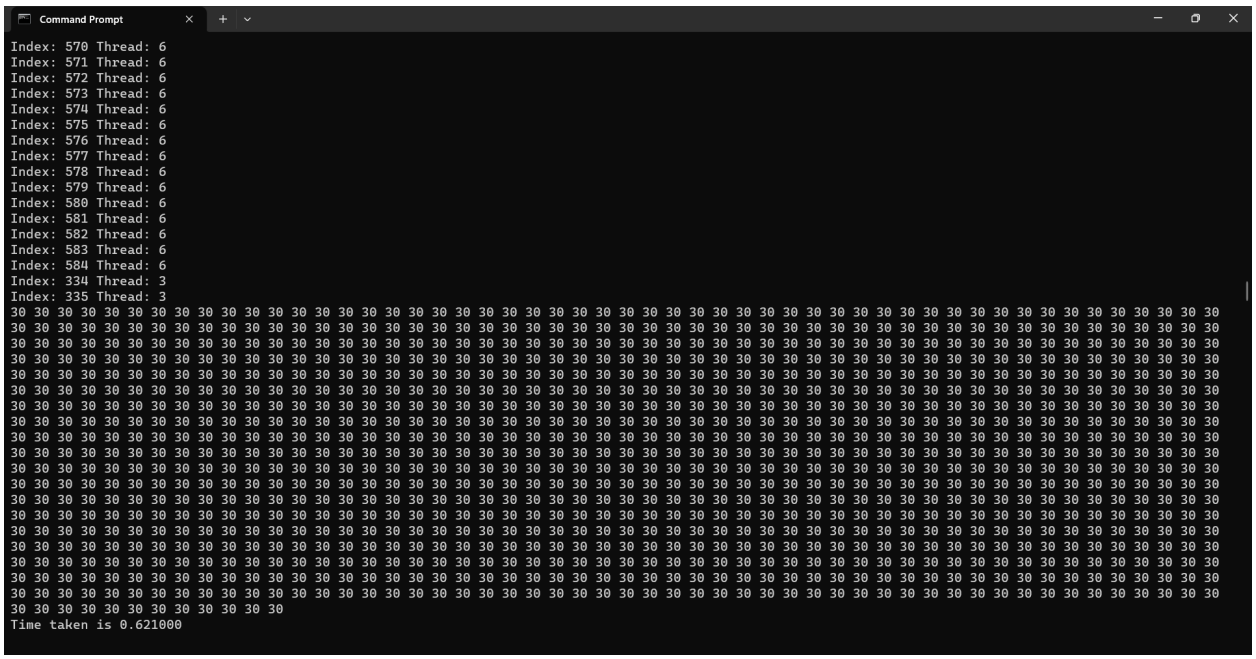
- The parallel region is created using `#pragma omp parallel`, and each thread works on a separate element of the arrays within a loop. The thread number is retrieved using `omp_get_thread_num()`.
- The use of OpenMP parallelism can significantly improve performance for large arrays, as each thread can work on different elements simultaneously, leveraging multi-core processors.
- However, for small arrays like the ones in this example (size 1000), the overhead of thread creation and synchronization might offset the benefits of parallelism. Parallelism is generally more advantageous for larger data sets.
- The code also measures and prints the execution time, which can be helpful for performance analysis and optimization.
- Note that this code does not explicitly specify the number of threads, so the number of threads used depends on the system's default settings or any environment variable that may have been set to control the number of threads.

Source Code(Parallel):

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

int main() {
    int N = 1000;
    int A[1000];
    for (int i = 0; i < N; i++)
        A[i] = 10;

    int B[1000];
    for (int i = 0; i < N; i++)
        B[i] = 20;

    int C[1000] = {0};
    double stime = omp_get_wtime();
#pragma omp parallel for reduction(+ \
                                   : C)
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
        printf("Index: %d Thread: %d\n", i, omp_get_thread_num());
    }

    for (int i = 0; i < N; i++) {
        printf("%d ", C[i]);
    }

    double etime = omp_get_wtime();
```

```
    double time = etime - stime;
    printf("\nTime taken is %f\n", time);
    printf("\n");
    return 0;
}
```

Output:



We performs element-wise addition of two arrays `A` and `B`, storing the result in array `C`. It also measures and prints the time taken for the computation. The program utilizes OpenMP for parallelism, enabling multiple threads to execute the addition concurrently. Here's a short information and analysis of the code with a focus on the OpenMP `reduction` clause:

Information:

The code is similar to the previous version but with the addition of the `reduction` clause in the OpenMP parallel for loop.

The `reduction` clause is used to specify a reduction operation (in this case, addition) that is applied to a shared variable (`C` in this case) within the parallel loop.

The `reduction(+ : C)` clause ensures that each thread maintains a private copy of `C`, performs the addition operation on its private copy, and then combines the results of all threads using the specified reduction operation (addition) to update the shared variable `C`.

Analysis:

- The `reduction` clause improves code performance by eliminating data races and efficiently aggregating the results of parallel computations.
- The code structure remains the same as the previous version, with a loop that distributes the work among multiple threads using OpenMP.
- Each thread independently computes the element-wise addition of arrays `A` and `B` and stores the result in its private copy of array `C`.
- After the loop, the reduction operation sums up the private copies of `C` from all threads and updates the shared `C` array with the final result.
- The code measures and prints the execution time, which can be helpful for performance analysis and optimization.
- The use of the `reduction` clause simplifies the code by automatically handling the aggregation of results in a thread-safe manner.
- Like the previous version, the code doesn't explicitly specify the number of threads, so the number of threads used depends on the system's default settings or any environment variable that may have been set to control the number of threads.

2. Calculation of value of Pi (Source Code):

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

#define NUM_POINTS 1000000

int main() {
    int i, num_inside_circle = 0;
    double x, y;

    // Set the seed for random number generation
    srand(12345);

    #pragma omp parallel for private(x, y) reduction(+:num_inside_circle)
    for (i = 0; i < NUM_POINTS; i++) {
        // Generate random points within the unit square
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;

        // Check if the point is inside the quarter-circle (radius <= 1)
        if (x * x + y * y <= 1.0) {
            num_inside_circle++;
        }
    }
```

```
    // Estimate Pi using the Monte Carlo method
    double pi_estimate = 4.0 * num_inside_circle / NUM_POINTS;

    printf("Estimated Pi: %f\n", pi_estimate);

    return 0;
}
```

Output:

```
D:\AKSHU\Academics\7th Sem (Final Year-1st Sem)\HPC_Lab\Experiments\Assign_2>g++ -o Pi Pi.cpp -fopenmp -lgomp

D:\AKSHU\Academics\7th Sem (Final Year-1st Sem)\HPC_Lab\Experiments\Assign_2>Pi.exe
Estimated Pi: 3.135664
```

The code estimates the value of Pi using a Monte Carlo simulation method. This code employs OpenMP for parallelism, allowing multiple threads to participate in the computation. Here's a short information and analysis of the code:

Information:

The code aims to estimate the value of Pi by simulating random points within a unit square and calculating the ratio of points falling inside a quarter-circle to the total points generated.

It uses OpenMP to parallelize the generation of random points and the calculation of the ratio.

`NUM_POINTS` is defined as 1,000,000, representing the number of random points to generate.

Random points are generated within the unit square by generating random `x` and `y` coordinates using `rand()`.

The `private` clause in the OpenMP `parallel for` directive ensures that each thread has its private copy of `x` and `y` variables to prevent data races.

The `reduction` clause is used to accumulate the count of points inside the quarter-circle (`num_inside_circle`) across all threads.

The estimated value of Pi is calculated using the Monte Carlo method, and the result is printed.

Analysis:

- The Monte Carlo method is a statistical approach to estimating Pi by randomly sampling points. As the number of points generated (`NUM_POINTS`) increases, the accuracy of the estimation improves.
- OpenMP is used to distribute the work among multiple threads, which can significantly speed up the computation, especially when a large number of

points are involved. Each thread works independently on its portion of the points.

- The code ensures thread safety by using the `private` clause to give each thread its private copy of variables and by using the `reduction` clause to safely accumulate the counts.
- The accuracy of the estimated Pi value depends on the number of points generated. You can increase `NUM_POINTS` for a more accurate estimation.
- Keep in mind that the Monte Carlo method provides an approximation, and the accuracy increases with more iterations. The estimated Pi value will be close to the actual Pi value, but it won't be exact.
- This code serves as a basic example of parallelism with OpenMP and illustrates how to perform parallel computations for Monte Carlo simulations, making it useful for educational and introductory purposes.