Class: Final Year (Computer Science and Engineering)

Year: 2023-24 **Semester:** 1

Course: High Performance Computing Lab

Practical No. 2

Exam Seat No: 2020BTECS00085

Title of practical: Study and implementation of basic OpenMP clauses

Implement following Programs using OpenMP with C:

1. Vector Scalar Addition
2. Calculation of value of Pi
   Analyse the performance of your programs for different number of threads and Data size.

Problem Statement 1: Implement Vector Scalar Addition using OpenMP.

Code:

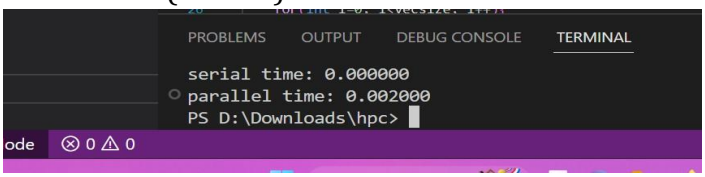
```c
#include<stdio.h>
#include<omp.h>
#define vecsize 10000
int main(){
    int s = 5;
    float vector[vecsize];
    for(int i=0; i<vecsize; i++){
        vector[i] = i;
    }

    double start_time = omp_get_wtime();
    for(int i=0; i<vecsize; i++){
          vector[i] += s;
    }
    double end_time = omp_get_wtime();
    printf("serial time: %f\n", end_time-start_time);

    // parallel
    for(int i=0; i<vecsize; i++){
        vector[i] = i;
    }

    double s_t = omp_get_wtime();
    #pragma omp parallel for private(s) num_threads(10000)
    for(int i=0; i<vecsize; i++){
        vector[i] += s;
    }
    double e_t = omp_get_wtime();

    printf("parallel time: %f\n",e_t-s_t);
```

```
    return 0;
}
```

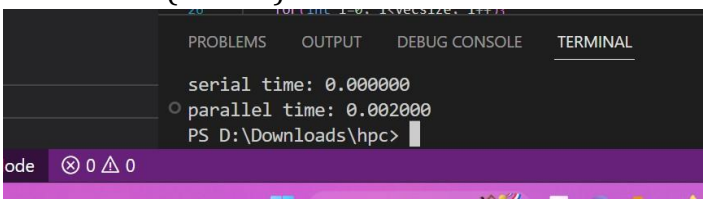
Screenshots:

Keeping number of threads constant and varying size of Data.

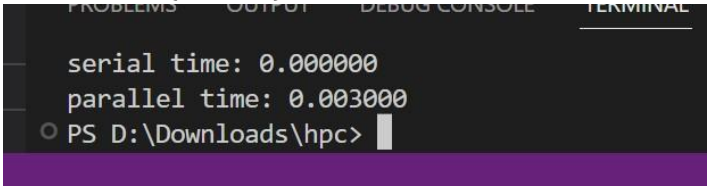
Threads = 8(default) Vector size = 100



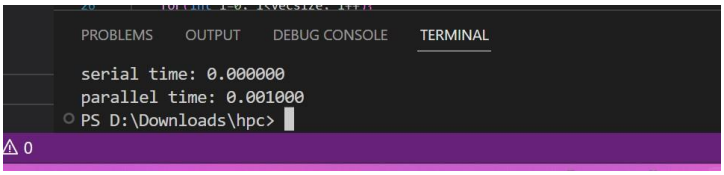
Threads = 8(default) Vector size = 1000



Threads = 8(default) Vector size = 10000

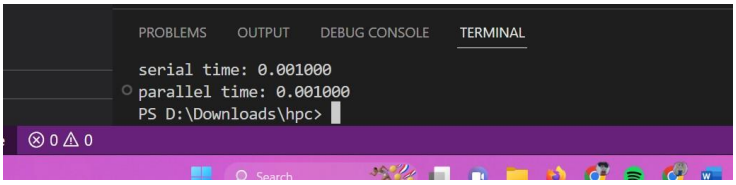


Threads = 8(default) Vector size = 100000

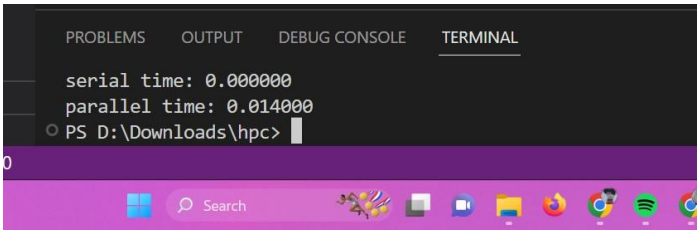


Keeping data constant and increasing number of threads.

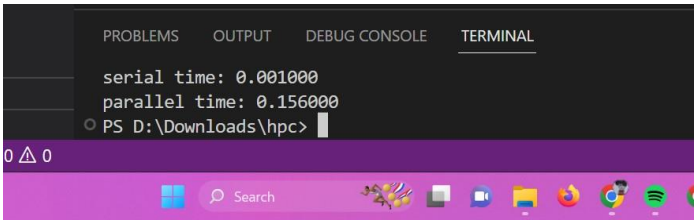
Threads = 10 Vector size = 10000



Threads = 100 Vector size = 10000



Threads = 1000 Vector size = 10000



Information:

Vector and scaler addition is to be performed using sequential and parallel approach. We have to analyse the time both approaches. For parallel approach analysis can be done in two ways, first by keeping data constant and varying number of threads and secondly by keeping number of threads constant and varying size of data.

Analysis:

1) As we go on increasing the size of data the time it takes to execute in parallel also increases.
2) By keeping data constant and increasing number for threads gradually increase the execution time due to increase in logical thread causes extra mapping time.
3) Here Serial time is less than parallel because insufficient data for parallelism which causes extra overhead of communication time.

Number of Threads	Data Size	Sequential Time	Parallel Time
8	100	0.00000	0.00200
8	1000	0.00000	0.00200
8	10000	0.00000	0.00300
8	100000	0.00000	0.00100
10	10000	0.00100	0.00100
100	10000	0.00000	0.01400
1000	10000	0.00100	0.15600

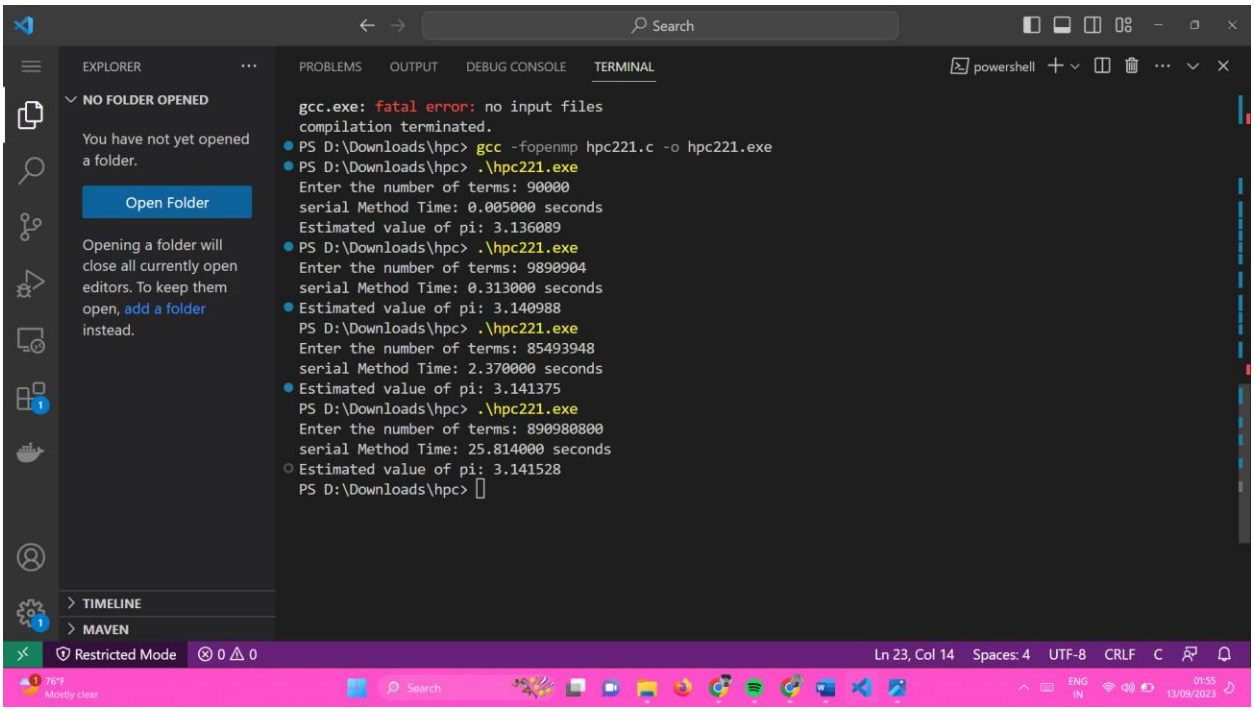
Problem Statement 2: Calculation of value of Pi using OpenMP

Code:

Serial code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
int main() {
    int totalPoints = 10000000;
    int pointsInsideCircle = 0;
    double x, y;
    printf("Enter the number of terms: ");
    scanf("%d", &totalPoints);
    double start_time_serial = omp_get_wtime();
    for (int i = 0; i < totalPoints; ++i) {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        if (x * x + y * y <= 1.0) {
            pointsInsideCircle++;
        }
    }
    double pi = 4.0 * pointsInsideCircle / totalPoints;
    double end_time_serial = omp_get_wtime();
    printf("serial Method Time: %f seconds\n", (end_time_serial -
start_time_serial));
    printf("Estimated value of pi: %f\n", pi);
    return 0;
}
```

Output:

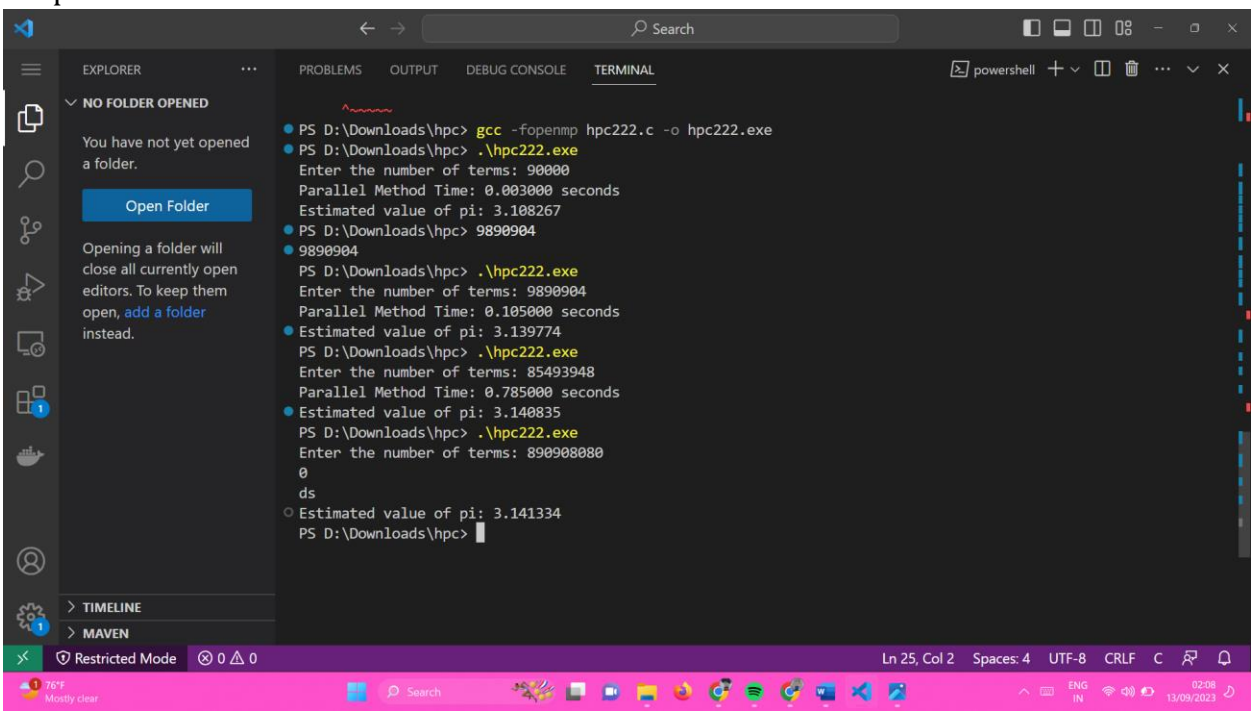


Parallel code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
int main() {
    int totalPoints = 10000000;
    int pointsInsideCircle = 0;
    double x, y;
    printf("Enter the number of terms: ");
    scanf("%d", &totalPoints);
    double start_time_parallel = omp_get_wtime();
    #pragma omp parallel for private(x, y) reduction(+:pointsInsideCircle)
    for (int i = 0; i < totalPoints; ++i) {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        if (x * x + y * y <= 1.0) {
            pointsInsideCircle++;
        }
    }
    double pi = 4.0 * pointsInsideCircle / totalPoints;
    double end_time_parallel = omp_get_wtime();
    printf("Parallel Method Time: %f seconds\n", (end_time_parallel -
start_time_parallel));
    printf("Estimated value of pi: %f\n", pi);
    return 0;
```

```
}
```

Output:



Information:

1) **Private Clause:** The private clause in an OpenMP parallel for loop specifies that each thread should have its own private copy of the specified variable. In the given program, the loop variable 'x' and 'y' are declared as private. This means that each thread will have its own separate x and y variables, avoiding conflicts between threads trying to modify the same memory location
2) **Reduction Clause:** The reduction clause in an OpenMP parallel construct allows you to perform a reduction operation on a specified variable, such as summing the values of that variable across all threads. In the given program, the 'pointsInsideCircle' variable is declared to be reduced with the '+' operator.

Analysis:

To calculate value of Pi, private and reduction clauses are used. OpenMP parallel for is used to iterate and calculate the Pi value. It concludes that parallel program has less time of execution than that of serial.

Number of Threads	Input term	Sequential Time	Parallel Time
8 – default	90000	0.016000	0.000000
8 – default	9890904	0.204000	0.066000
8 – default	85493948	1.605000	0.383000
8 – default	890980800	18.467000	4.048000

GitHub Link:
https://github.com/manjiri-chandure/HPC