Reg. No: 23361
MTCS-103(P)
Parallel Processing
Assignment 3

Parallel Loop

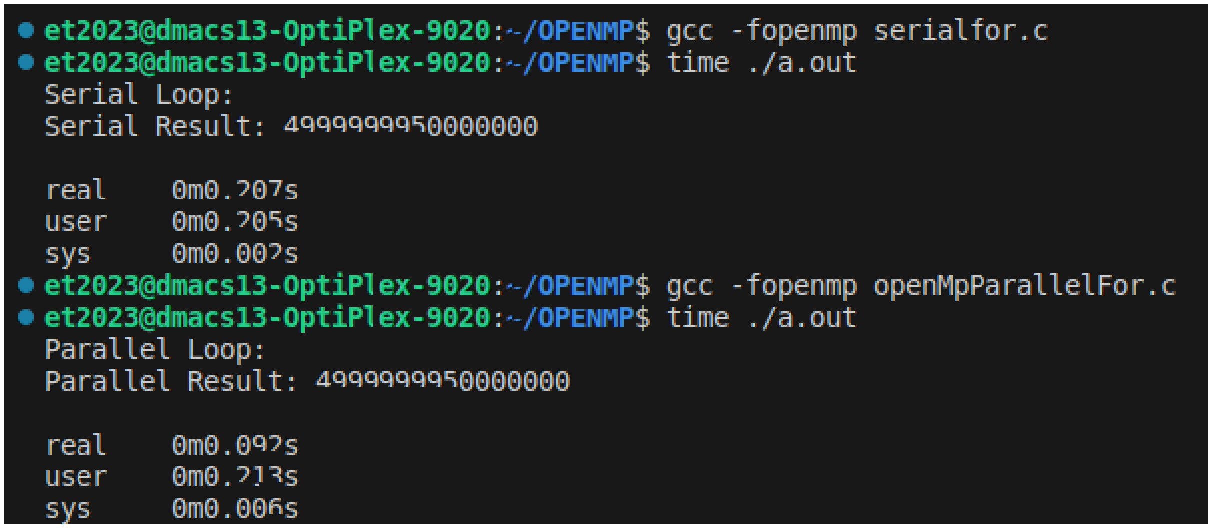
The parallel loop construct combines the benefits of both the parallel and loop constructs in OpenMP. It allows for easy parallelization of loops by distributing loop iterations among threads in a team. The construct is used with a `for` loop, and it employs clauses similar to those accepted by the `parallel` or `for` directives, except for the `nowait` clause. The parallel loop construct creates a team of threads and assigns each thread a portion of loop iterations to execute concurrently. It's particularly useful for exploiting parallelism in loops that can be divided into independent iterations.

#pragma omp parallel for [clause[[,]clause] ...]

```
//openMp Parallel
#include <stdio.h>
#include <omp.h>

#define N 100000000LL

long long int parallelLoop() {
   long long int result = 0;

   #pragma omp parallel for
   for (long long int i = 0; i < N; ++i) {
       result += i;
   }

   printf("Parallel Result: %lld\n", result);
```

```c
    return result;
}

int main() {
    printf("Parallel Loop:\n");
    long long int parallel_result = parallelLoop();

    return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ gcc -fopenmp serialfor.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ time ./a.out
  Serial Loop:
  Serial Result: 4999999950000000

  real    0m0.207s
  user    0m0.205s
  sys     0m0.002s
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ gcc -fopenmp openMpParallelFor.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ time ./a.out
  Parallel Loop:
  Parallel Result: 4999999950000000

  real    0m0.092s
  user    0m0.213s
  sys     0m0.006s
```

This program uses OpenMP to parallelize the loop calculation. It distributes the iterations of the loop among multiple threads, allowing them to compute partial sums concurrently. The partial sums are then combined to obtain the final result. The parallel version demonstrates a performance improvement due to the concurrent execution of loop iterations across multiple threads

The provided execution times suggest that the parallel version (openMpParallelFor.c) is faster than the serial version (SerialLoop.c). This speedup is expected since parallel execution divides the work among multiple threads, allowing for more efficient use of available CPU cores. The parallel version finishes in around 0.092 seconds, while the serial version takes around 0.207 seconds.

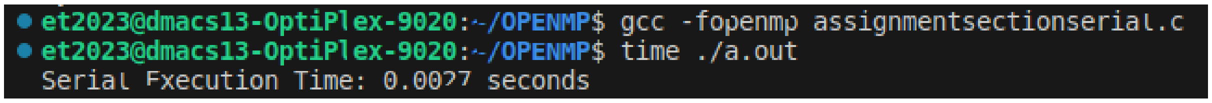
Parallel Sections:

The parallel sections construct in OpenMP offers a convenient way to parallelize sections of code by distributing them among threads in a team. It is used to create a parallel execution environment for multiple sections of code that can be executed concurrently. Each section is enclosed within a `section` directive, and the parallel sections construct creates a team of threads to execute these sections in parallel. This construct is particularly useful when there are distinct tasks that can be performed independently, allowing for efficient utilization of multiple threads.

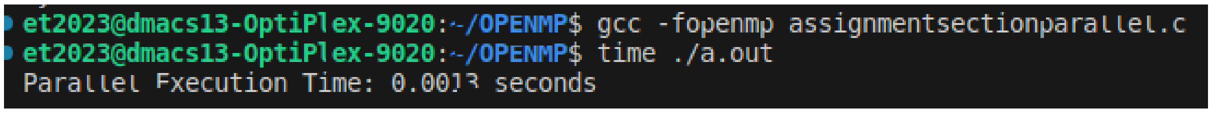
```
#include <stdio.h>
#include <omp.h>
#define ARRAY_SIZE 1000000
void calculateSquaresParallel(long long int array[]) {
   #pragma omp parallel sections
   {
       #pragma omp section
       for (long long int i = 0; i < ARRAY_SIZE / 5; ++i) {
           array[i] = array[i] * array[i];
       }
       #pragma omp section
       for (long long int i = ARRAY_SIZE / 5; i < 2 * ARRAY_SIZE / 5; ++i)
{
           array[i] = array[i] * array[i];
       }
       #pragma omp section
       for (long long int i = 2 * ARRAY_SIZE / 5; i < 3 * ARRAY_SIZE / 5;
++i) {
           array[i] = array[i] * array[i];
       }
       #pragma omp section
       for (long long int i = 3 * ARRAY_SIZE / 5; i < 4 * ARRAY_SIZE / 5;
++i) {
           array[i] = array[i] * array[i];
       }
       #pragma omp section
```

```
        for (long long int i = 4 * ARRAY_SIZE / 5; i < ARRAY_SIZE; ++i) {
            array[i] = array[i] * array[i];
        }
    }
}
int main() {
    long long int array[ARRAY_SIZE];

    for (long long int i = 0; i < ARRAY_SIZE; ++i) {
        array[i] = i + 1;
    }
    double start_time = omp_get_wtime();
    calculateSquaresParallel(array);
    double end_time = omp_get_wtime();
    printf("Parallel Execution Time: %.4f seconds\n", end_time -
start_time);
    return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ gcc -fopenmp assignmentsectionserial.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP$ time ./a.out
  Serial Execution Time: 0.0027 seconds
```



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP$ gcc -fopenmp assignmentsectionparallel.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP$ time ./a.out
Parallel Execution Time: 0.0013 seconds
```

Explanations:

Here we demonstrated the advantages of parallel processing for a simple computation task, such as calculating the square of array elements. In the above code the parallel implementation divided the task of calculating squared values among multiple sections using OpenMP parallel sections. Each section independently operated on a subset of the array elements, thereby utilising multiple threads to achieve parallelism. The execution time for the parallel code was measured to be approximately 0.0013 seconds whereas the serial execution measured to be approximately 0.0027 seconds