PRN No: 2019BTECS00010

Full name: Kajal Jitendra Pawar

Batch: B1

Assignment: 2

Study on Parallel Programming Concepts

SPMD: Single Program Multiple Data

It is a parallel programming style. In SPMD style tasks are split up and run simultaneously on multiple processors with different data. This is done to achieve results faster.

Worksharing

Threads are assigned, an independent subset of the total workload For example, different chunks of an iteration are distributed among the threads

OpenMP provides various constructs for worksharing.

OpenMP loop worksharing construct

OpenMP's loop worksharing construct splits loop iterations among all active threads

#pragma omp for

Types of Variables

1. **Shared Variables:**
   There exist one instance of this variable which is shared among all threads

2. **Private Variables:**
   Each thread in a team of threads has its own local copy of the private variable

Two ways to assign variables as private or shared are:

1. Implicit:
   All the variables declared outside of the pragma are by default shared and all the variables declared inside pragma are private

2. Explicit:

   Shared Clause
   eg. #pragma omp parallel for shared(n, a) => n and a are declared as shared variables

   Private Clause
   eg. #pragma omp parallel for shared(n, a) private(c) => here c is private variable

   Default Clause
   eg. #pragma omp parallel for default(shared) => now all variables are shared
   #pragma omp parallel for default(private) => now all variables are private
   FirstPrivate
   Firstprivate make the variable private but that variable is initialised with the value that it has before the parallel region

   LastPrivate
   Lastprivate make the variable private but it retain the last value of that private variable outside of the private region

Schedule

- a specification of how iterations of associated loops are divided into contiguous
- non-empty subsets.

- syntax: #pragma omp parallel for schedule([modifier [modifier]:]kind[,chunk_size])

  Five kinds of schedules for OpenMP loop:
  - static
  - dynamic
  - guided
  - auto
  - runtime

Static Schedule

In static scheduling openMP assingn interations to threads in a cyclic order

eg: #pragma omp parallel for schedule(static,1)
=> Number “1” indicates that we assign one iteration to each thread before switching to the next thread — we use chunks of size 1.

Dynamic Schedule

In dynamic scheduling, each thread will take one iteration, process it, and then see what is the next iteration that is currently not being processed by anyone. This way it will never happen that one thread finishes while other threads have still lots of work to do:

eg: #pragma omp parallel for schedule(dynamic,1)

nowait

When we use a parallel region, OpenMP will automatically wait for all threads to finish before execution continues. There is also a synchronization point after each omp for loop
However, if we do not need synchronization after the loop, we can disable it with nowait
eg: #pragma omp for nowait

Reduction

The OpenMP reduction clause lets you specify one or more thread-private variables that are subject to a reduction operation at the end of the parallel region.

Problems:

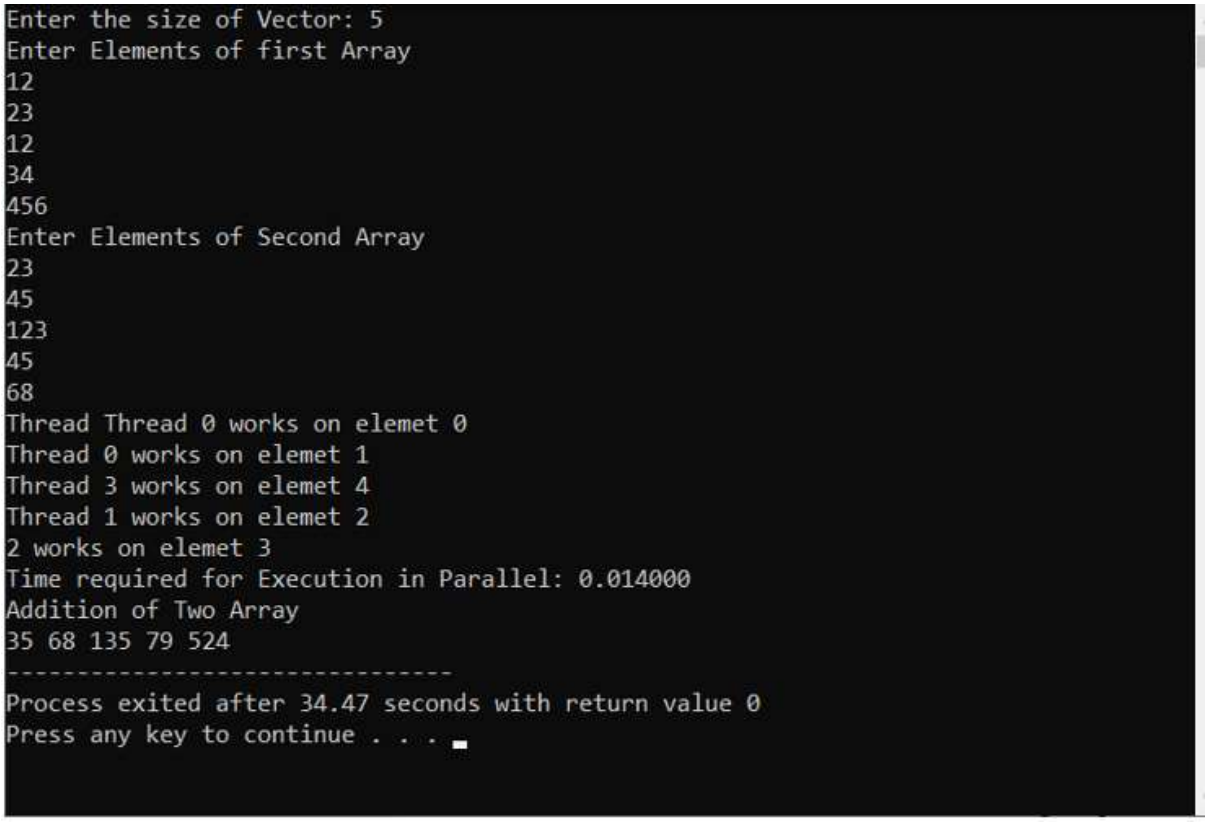
1. **Vector Vector Addition**

Code:

```
#include<bits/stdc++.h>
#include<omp.h>

using namespace std;

int main()
{
	int n,i;
	cout<<"Enter the size of Vector: ";
	cin>>n;
	vector<int> a(n),b(n),c(n);

	cout<<"Enter Elements of first Array\n";
	for(i=0;i<n;i++)
		cin>>a[i];

	cout<<"Enter Elements of Second Array\n";
	for(i=0;i<n;i++)
		cin>>b[i];

	double getInTime = omp_get_wtime();

	#pragma omp parallel for shared(a, b, c) private(i) schedule(static)

	for(i=0;i<n;i++)
```

```
        {
                c[i] = a[i] + b[i];
                printf("Thread %d works on elemet %d \n",
omp_get_thread_num(), i);
        }

        double getOutTime = omp_get_wtime();

        double exptTime= getOutTime - getInTime;

        printf("Time required for Execution in Parallel: %f\n", exptTime);

        cout<<"Addition of Two Array\n";
        for(i=0;i<n;i++)
        {
                printf("%d ", c[i]);
        }

        return 0;
}
```

Output:



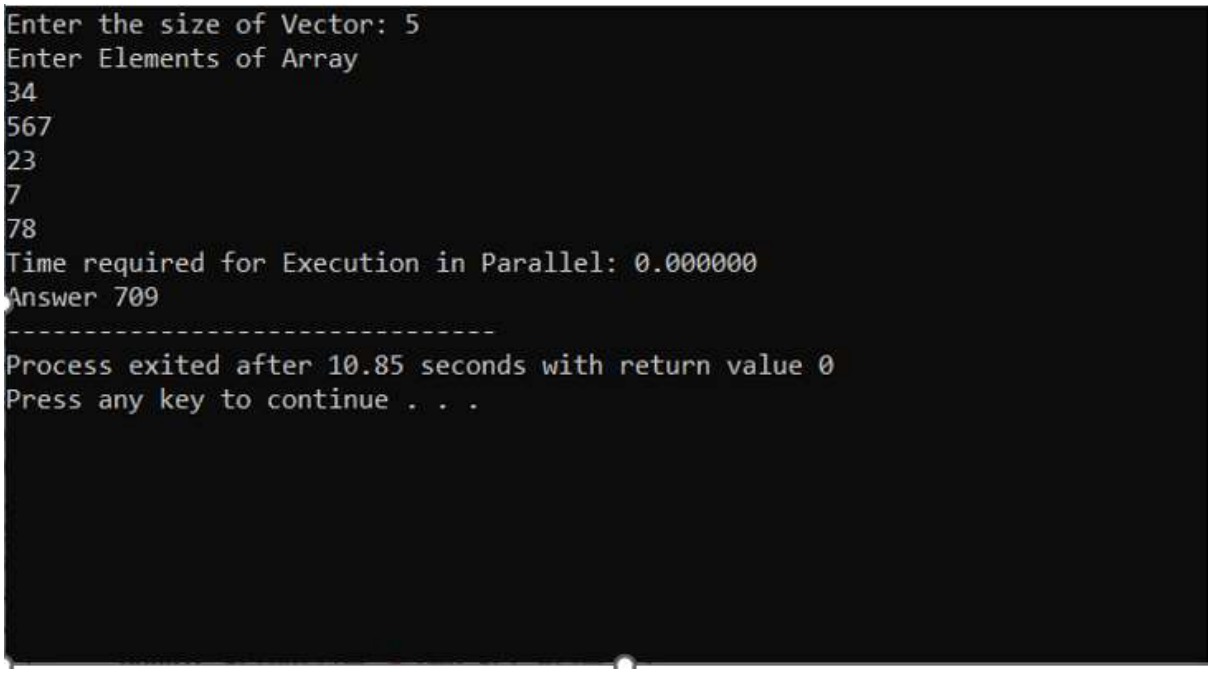
2. Vector Scalar Addition

Code:

```
#include<bits/stdc++.h>
#include<omp.h>

using namespace std;

int main()
{
	int n,i;
	cout<<"Enter the size of Vector: ";
	cin>>n;
	vector<int> a(n);

	cout<<"Enter Elements of Array\n";
	for(i=0;i<n;i++)
```

```
                cin>>a[i];
    int answer = 0;

        double getInTime = omp_get_wtime();

    #pragma omp parallel for
    for (i = 0; i < 5; i++)
    {
       answer += a[i];
    }

    double getOutTime = omp_get_wtime();

        double exptTime= getOutTime - getInTime;

        printf("Time required for Execution in Parallel: %f\n", exptTime);

        printf("Answer %d ", answer);

        return 0;
}
```

Output:


Enter the size of Vector: 5
Enter Elements of Array
34
567
23
7
78
Time required for Execution in Parallel: 0.000000
Answer 709

Process exited after 10.85 seconds with return value 0
Press any key to continue . . .