IT 301

Parallel Computing

Lab 2

Jason Krithik Kumar S

191IT123

---

1. Program 1

```c
/*shared.c*/
#include<stdio.h>
#include<omp.h>

int main()
{
    int x=20;
    #pragma omp parallel shared(x)
    {
        int tid=omp_get_thread_num();
        x=x+1;
        printf("Thread [%d]: value of x is %d\n",tid,x);
    }
}
```

```
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ gcc -fopenmp shared.c -o shared
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ export OMP_NUM_THREADS=3
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./shared
Thread [0]: value of x is 21
Thread [1]: value of x is 21
Thread [2]: value of x is 22
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ export OMP_NUM_THREADS=5
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./shared
Thread [1]: value of x is 23
Thread [0]: value of x is 21
Thread [4]: value of x is 24
Thread [3]: value of x is 22
Thread [2]: value of x is 22
```

```
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ export OMP_NUM_THREADS=7
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./shared
Thread [1]: value of x is 21
Thread [5]: value of x is 24
Thread [0]: value of x is 22
Thread [4]: value of x is 22
Thread [3]: value of x is 22
Thread [2]: value of x is 22
Thread [6]: value of x is 23
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./shared
Thread [4]: value of x is 22
Thread [1]: value of x is 22
Thread [0]: value of x is 22
Thread [2]: value of x is 22
Thread [6]: value of x is 23
Thread [5]: value of x is 22
Thread [3]: value of x is 21
```

```
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ export OMP_NUM_THREADS=12
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./shared
Thread [5]: value of x is 22
Thread [11]: value of x is 23
Thread [0]: value of x is 21
Thread [9]: value of x is 21
Thread [4]: value of x is 22
Thread [7]: value of x is 21
Thread [8]: value of x is 21
Thread [1]: value of x is 21
Thread [6]: value of x is 21
Thread [10]: value of x is 22
Thread [2]: value of x is 22
Thread [3]: value of x is 21
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$
```

It could be seen here that the values of x printed by the various threads are different, some have the same values and the others increments of the threads which executed before the given thread. Since the variable is shared and there is no synchronization between the read and write we have values which are same.

2. Program 2

learn.c > main()

```c
/*learn.c*/
#include<stdio.h>
#include<omp.h>

int main()
{
    int i=20;
    printf("Value of i before pragma i=%d\n",i);
    #pragma omp parallel num_threads(4) private(i)
    {
        printf("Value after entering pragma i=%d tid=%d\n",i, omp_get_thread_num());
        i=i+omp_get_thread_num(); //adds thread_id to i
        printf("Value after changing value i=%d tid=%d\n",i, omp_get_thread_num());
    }
    printf("Value after having pragma i=%d tid=%d\n",i, omp_get_thread_num());
}
```

```
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ gcc -fopenmp learn.c -o learn
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./learn
Value of i before pragma i=20
Value after entering pragma i=0 tid=0
Value after changing value i=0 tid=0
Value after entering pragma i=0 tid=1
Value after changing value i=1 tid=1
Value after entering pragma i=0 tid=2
Value after changing value i=2 tid=2
Value after entering pragma i=0 tid=3
Value after changing value i=3 tid=3
Value after having pragma i=20 tid=0
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$
```

C learn.c > main()

```c
/*learn.c*/
#include<stdio.h>
#include<omp.h>

int main()
{
    int i=20;
    printf("Value of i before pragma i=%d\n",i);
    #pragma omp parallel num_threads(4) firstprivate(i)
    {
        printf("Value after entering pragma i=%d tid=%d\n",i, omp_get_thread_num());
        i=i+omp_get_thread_num(); //adds thread_id to i
        printf("Value after changing value i=%d tid=%d\n",i, omp_get_thread_num());
    }
    printf("Value after having pragma i=%d tid=%d\n",i, omp_get_thread_num());
}
```

```
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ gcc -fopenmp learn.c -o learn
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./learn
Value of i before pragma i=20
Value after entering pragma i=20 tid=0
Value after changing value i=20 tid=0
Value after entering pragma i=20 tid=2
Value after changing value i=22 tid=2
Value after entering pragma i=20 tid=3
Value after changing value i=23 tid=3
Value after entering pragma i=20 tid=1
Value after changing value i=21 tid=1
Value after having pragma i=20 tid=0
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$
```

It could be observed that private(x) initializes the variable randomly (here 0) for each thread and performs the subsequent operations using its instance of x. firstprivate(x) initialises x to its value outside the parallel block. Since all the variable are private, changes made are not reflected in the original variable.

3.

addArray.c > main()

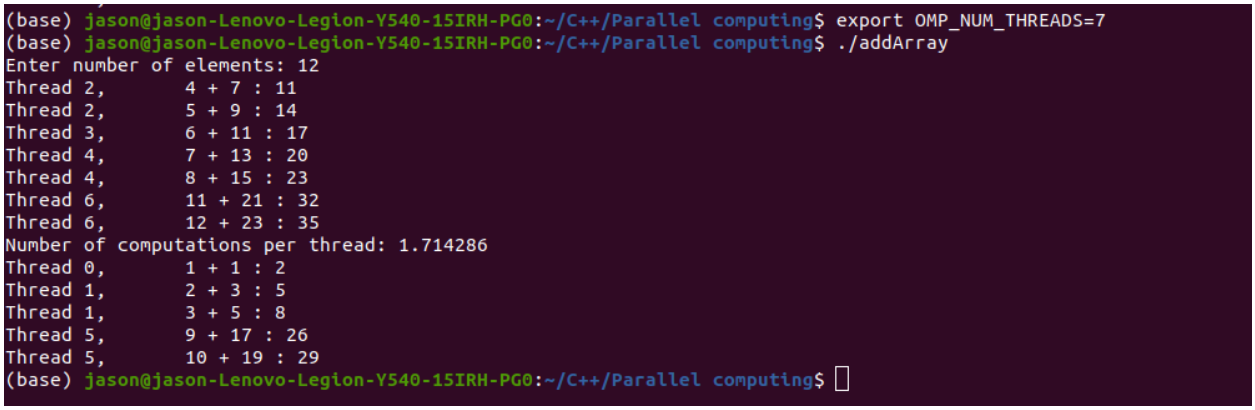
```c
#include<stdio.h>
#include<omp.h>

int main()
{
    int N, i, threadnum, numthreads, low, high;
    printf("Enter number of elements: ");
    scanf("%d", &N);
    int a[N], b[N], c[N];
    for (i = 0; i < N; i++) {
        a[i] = i+1;
        b[i] = i*2+1;
    }

    #pragma omp parallel default(shared) private(threadnum, numthreads, low, high, i)
    {
        threadnum = omp_get_thread_num();
        numthreads = omp_get_num_threads();
        if (threadnum == 0) printf("Number of computations per thread: %f\n", (float)N/numthreads);
        low = N*threadnum/numthreads;
        high = N*(threadnum+1)/numthreads;
        for (i=low; i<high; i++) {
            c[i]=b[i]+a[i];
            printf("Thread %d,\t%d + %d : %d\n", threadnum, a[i], b[i], c[i]);
        }
    }
}
```

```
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ gcc -fopenmp addArray.c -o addArray
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ export OMP_NUM_THREADS=4
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./addArray
Enter number of elements: 10
Thread 2,       6 + 11 : 17
Thread 2,       7 + 13 : 20
Thread 1,       3 + 5 : 8
Thread 1,       4 + 7 : 11
Thread 1,       5 + 9 : 14
Thread 3,       8 + 15 : 23
Thread 3,       9 + 17 : 26
Thread 3,       10 + 19 : 29
Number of computations per thread: 2.500000
Thread 0,       1 + 1 : 2
Thread 0,       2 + 3 : 5
```

```
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ export OMP_NUM_THREADS=5
(base) jason@jason-Lenovo-Legion-Y540-15IRH-PG0:~/C++/Parallel computing$ ./addArray
Enter number of elements: 12
Number of computations per thread: 2.400000
Thread 0,       1 + 1 : 2
Thread 0,       2 + 3 : 5
Thread 2,       5 + 9 : 14
Thread 2,       6 + 11 : 17
Thread 2,       7 + 13 : 20
Thread 1,       3 + 5 : 8
Thread 1,       4 + 7 : 11
Thread 3,       8 + 15 : 23
Thread 3,       9 + 17 : 26
Thread 4,       10 + 19 : 29
Thread 4,       11 + 21 : 32
Thread 4,       12 + 23 : 35
```



In this program we compute the sum of a[i] & b[i] and store it in c[i]. But do this parallelly.

We do this by first setting the number of threads(numThreads) and asking the user to enter the number of elements(N). After this we split the array into chunks to be processed by each thread. Since threadIDs start from 0 to numThreads-1, we can assign the start and end indices for each thread as start = N*threadID/numThread and end = N*(threadID+1)/numThread. If N/NumThreads is not an integer, some threads are assigned, ceil(N/numThreads) and others, floor(N/numThreads) such that the sum is N.