

CSE 4001: Lab Assignment #1

Due on Thursday, Jul 26, 2018

Prof. Deebak BD 5:30pm

Jacob John

Contents

Problem 1	3
Problem 2	6
Problem 3	9

Problem 1

Write a simple OpenMP program for array addition (sum). It is helpful to understand how threads are created in OpenMP. To examine the above scenario, we are going to take two one-dimensional arrays, each of size of 5. We will then create 5 threads. Each thread will be responsible for one addition operation.

Note: In OpenMP, pre-defined preprocessor directive `#pragma omp parallel` is used to create threads. We can mention the number of threads to be created as a parameter to `num_threads()`. If you are not mentioning `num_threads()`, then the number of threads to be created is equal to number of cores in processor. Thread id can be obtained by using predefined function `omp_get_thread_num()`.

Explanation

The scenario given below can be executed in two different ways - one with OpenMP and one without.

The simple addition method in Listing 1 using a *for* loop to add the two arrays and then store them in a third array. This is a fairly traditional approach.

Listing 2 utilizes thread and their ids 1 – 5 to index the arrays and add the corresponding digits together. Rather than sequentially executing the program, i.e., running a *for* loop and indexing the array, 5 threads are used. Using multi-threaded parallel processing, all the digits in the two arrays are added and stored parallelly.

Despite using a parallel approach, due to OpenMP's lower parallel efficiency, higher percentage of serial code and reliance on parallelizable loops. The parallel array addition does not run faster than sequential addition.

Listing 1: Scenario 1 without using OpenMP in C

```
//simple array addition
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
5 #include <time.h>

int main()
{
    int a[5], b[5], c[5];
10    int i;
    clock_t t, u;

    printf("-----16BCE2205-----\n");
    printf("Reading values for array A \n");
15    for (i = 0; i < 5; i++)
    {
        printf("Enter a value: ");
        scanf("%d", &a[i]);
    }

20    printf("Reading values for array B \n");
    for (i = 0; i < 5; i++)
    {
        printf("Enter a value: ");
25    scanf("%d", &b[i]);
    }
```

```

    }

    printf("Running program sequentially: \n");
    t = clock();
30  for (i = 0; i < 5; i++)
    {
        c[i] = a[i] + b[i];
    }
    t = clock() - t;
35  double time_taken_s = ((double)t) / CLOCKS_PER_SEC;

    printf("The program took %f seconds to execute \n", time_taken_s);

    for (i = 0; i < 5; i++)
40      printf("Sum of %d and %d is %d\n", a[i], b[i], c[i]);

    return 0;
}

```

Listing 2: Scenario 1 using OpenMP in C

```

//Write a simple OpenMP program for array addition (sum).
//we are going to take two one-dimensional arrays, each of size of 5.
//We will then create 5 threads.
//Each thread will be responsible for one addition operation.
5  #include <stdio.h>
    #include <stdlib.h>
    #include <omp.h>
    #include <time.h>

10  int main()
    {
        int a[5], b[5], c[5];
        int i;
        clock_t u;

15      printf("-----16BCE2205-----\n");
        printf("Reading values for array A \n");
        for (i = 0; i < 5; i++)
        {
20            printf("Enter a value: ");
            scanf("%d", &a[i]);
        }

        printf("Reading values for array B \n");
25        for (i = 0; i < 5; i++)
        {
            printf("Enter a value: ");
            scanf("%d", &b[i]);
        }

30        printf("Running program parallely: \n");
        omp_set_num_threads(5);
        #pragma omp parallel

```

```

35     {
        int tid = omp_get_thread_num();
        u = clock();
        c[tid] = a[tid] + b[tid];
        u = clock() - u;
        printf("There are %d nthreads in execution\n", omp_get_num_threads());
40     }

    for (i = 0; i < 5; i++)
        printf("Sum of %d and %d is %d\n", a[i], b[i], c[i]);

45    double time_taken_p = ((double)u) / CLOCKS_PER_SEC;
    printf("The program took %f seconds to execute \n", time_taken_p);

    return 0;
}

```

Output:

The screenshot displays the execution of a parallel array addition program. On the left, a terminal window shows the program being run sequentially and then in parallel. The sequential execution takes 0.000003 seconds, while the parallel execution takes 0.000003 seconds. The parallel execution shows 5 threads in execution. On the right, a code editor shows the source code for array_addition.c and array_addition_parallel.c. The code for array_addition.c is as follows:

```

1 //simple array addition
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <omp.h>
5 #include <time.h>
6
7 int main() {
8     int a[5], b[5], c[5];
9     int i;
10    clock_t t,u;
11
12    printf("-----16BCE2205-----\n");
13    printf("Reading values for array A \n");
14    for(i = 0; i < 5; i++){
15        printf("Enter a value: ");
16        scanf("%d", &a[i]);
17    }
18
19    printf("Reading values for array B \n");
20    for(i = 0; i < 5; i++){
21        printf("Enter a value: ");
22        scanf("%d", &b[i]);
23    }
24
25    printf("Running program sequentially:\n");
26    t = clock();
27    for (i = 0; i < 5; i++)
28    {
29        c[i] = a[i] + b[i];
30    }
31    t = clock() - t;
32    double time_taken_s = ((double)t) / CLOCKS_PER_SEC;
33
34    printf("Running the program sequentially took %f seconds to execute\n", time_taken_s);
35
36    for (i = 0; i < 5; i++)
37        printf("Sum of %d and %d is %d\n", a[i], b[i], c[i]);
38
39    return 0;
40 }

```

The code for array_addition_parallel.c is as follows:

```

1 //simple array addition
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <omp.h>
5 #include <time.h>
6
7 int main() {
8     int a[5], b[5], c[5];
9     int i;
10    clock_t t,u;
11
12    printf("-----16BCE2205-----\n");
13    printf("Reading values for array A \n");
14    for(i = 0; i < 5; i++){
15        printf("Enter a value: ");
16        scanf("%d", &a[i]);
17    }
18
19    printf("Reading values for array B \n");
20    for(i = 0; i < 5; i++){
21        printf("Enter a value: ");
22        scanf("%d", &b[i]);
23    }
24
25    printf("Running program sequentially:\n");
26    t = clock();
27    for (i = 0; i < 5; i++)
28    {
29        c[i] = a[i] + b[i];
30    }
31    t = clock() - t;
32    double time_taken_s = ((double)t) / CLOCKS_PER_SEC;
33
34    printf("Running the program sequentially took %f seconds to execute\n", time_taken_s);
35
36    for (i = 0; i < 5; i++)
37        printf("Sum of %d and %d is %d\n", a[i], b[i], c[i]);
38
39    return 0;
40 }

```

Problem 2

Write a simple OpenMP program for linear search. It is helpful to understand how threads can be executed in parallel to sequentially check each element of the list for the target value until a match is found or until all the elements have been searched.

To examine the above scenario, it is noteworthy to mention that with the parallel implementation, each and every element will be checked regardless of a match, though, parallelly. This is due to the fact that no thread can directly return after finding the element. So, our parallel implementation will be slower than the serial implementation if the element to be found is present in the range $[0, (n/p) - 1]$ where n is the length of the array and p is the number of parallel threads/sub-processes.

Note: In OpenMP, to parallelize the for loop, the openMP directive is: `#pragma omp parallel for`

Explanation

The first program below uses a simple for loop while the second one uses a pragma omp.

Logically, the first program traverses the array to find the element. While the second program deploys four threads to look for the element. Furthermore, the second program also uses a *private* variable *i*. This directive declares data to have a separate copy in the memory of each thread.

The reason for the inefficiency for parallel search is given above. Each element will be checked regardless of a match. This is due to the fact that no thread can directly return after finding the element. Hence the longer execution time.

Listing 3: Simple linear search using C

```
//Write a simple program for linear search
#include <stdio.h>
#include <omp.h>
#include <time.h>

5  int main()
    {
        int i, key = 85, tid;
        int a[100];
10     clock_t t;

        for (i = 1; i <= 100; i++)
            a[i - 1] = i;

15     printf("-----16BCE2205-----");

        t = clock();
        for (i = 0; i < 100; i++)
        {
20             if (a[i] == key)
                {
                    printf("\nKey found. Position = %d by thread %d \n", i, tid);
                }
            }
25     t = clock() - t;
```

```
double time_taken_s = ((double)t) / CLOCKS_PER_SEC;
printf("Program took %f seconds to execute \n", time_taken_s);
return 0;
30 }
```

Listing 4: Simple linear search using OpenMP

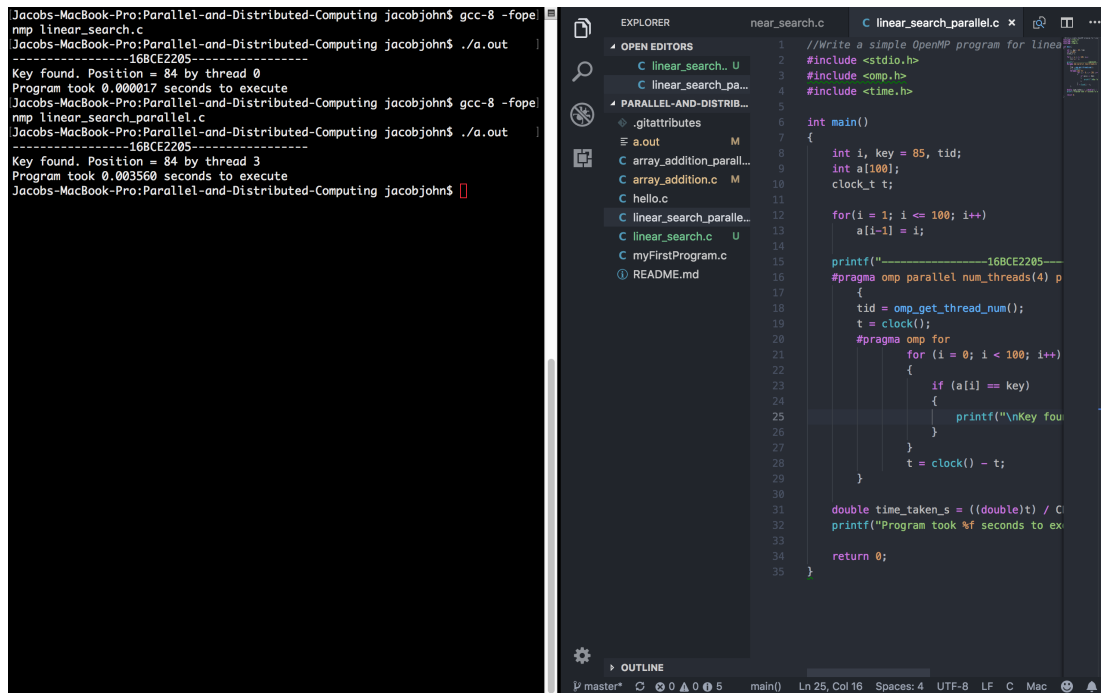
```
//Write a simple OpenMP program for linear search
#include <stdio.h>
#include <omp.h>
#include <time.h>
5
int main()
{
    int i, key = 85, tid;
    int a[100];
10    clock_t t;

    for (i = 1; i <= 100; i++)
        a[i - 1] = i;

15    printf("-----16BCE2205-----");
    #pragma omp parallel num_threads(4) private(i)
    {
        tid = omp_get_thread_num();
        t = clock();
20    #pragma omp for
        for (i = 0; i < 100; i++)
        {
            if (a[i] == key)
            {
25                printf("\nKey found. Position = %d by thread %d \n", i, tid);
            }
        }
        t = clock() - t;
    }
30

    double time_taken_s = ((double)t) / CLOCKS_PER_SEC;
    printf("Program took %f seconds to execute \n", time_taken_s);

    return 0;
35 }
```

Output:

The screenshot displays a code editor with two panes. The left pane shows the terminal output of a C program, and the right pane shows the source code of the program.

Terminal Output (Left Pane):

```
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ gcc-8 -fopenmp linear_search.c
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ ./a.out
-----16BCE2205-----
Key found. Position = 84 by thread 0
Program took 0.000017 seconds to execute
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ gcc-8 -fopenmp linear_search_parallel.c
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ ./a.out
-----16BCE2205-----
Key found. Position = 84 by thread 3
Program took 0.003560 seconds to execute
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$
```

Source Code (Right Pane):

```
near_search.c
C linear_search_parallel.c
//Write a simple OpenMP program for linear search
#include <stdio.h>
#include <omp.h>
#include <time.h>

int main()
{
    int i, key = 85, tid;
    int a[100];
    clock_t t;

    for(i = 1; i <= 100; i++)
        a[i-1] = i;

    printf("-----16BCE2205-----\n");
    #pragma omp parallel num_threads(4) private(tid)
    {
        tid = omp_get_thread_num();
        t = clock();

        #pragma omp for
        for (i = 0; i < 100; i++)
        {
            if (a[i] == key)
            {
                printf("\nKey found at position %d by thread %d\n", i, tid);
            }
        }

        t = clock() - t;
    }

    double time_taken_s = ((double)t) / CLOCKS_PER_SEC;
    printf("Program took %f seconds to execute\n", time_taken_s);

    return 0;
}
```


Problem 3

Write a simple OpenMP program for Matrix Multiplication. It is helpful to understand how threads make use of the cores to enable coarse-grain parallelism. Note that different threads will write different parts of the result in the array *multiply*, so we don't get any problems during the parallel execution. Note that accesses to matrices *first* and *second* are read-only and do not introduce any problems either.

Note: In OpenMP, to parallelize the for loop, the openMP directive is: `#pragma omp parallel for`

Explanation

The first program below uses a simple for loop while the second one uses a pragma omp.

In the parallel program, we use the access modifier, *shared* on the array elements of *first*, *second* and *multiply*. Each outer loop computes the scalar product of one row of first by vector from second. This allows the threads to access each element parallelly.

Furthermore, the static schedule is, as according to Intel, "Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size." This enables us to run nested for loops to allow matrix multiplication of arrays of different sizes.

The same efficiency as mentioned earlier persists. The reason for this inefficiency is due to OpenMP's lower parallel efficiency, higher percentage of serial code and reliance on parallelizable loops. The parallel matrix multiplication does not run faster than sequential matrix multiplication.

Listing 5: Simple matrix multiplication using C

```
#include <stdio.h>
#include <time.h>

int main()
5 {
    int m, n, p, q, c, d, k, sum = 0;
    int first[10][10], second[10][10], multiply[10][10];
    clock_t t;

10    printf("-----16BCE2205-----\n");

    printf("Enter number of rows and columns of first matrix\n");
    scanf("%d%d", &m, &n);
    printf("Enter elements of first matrix\n");

15    for (c = 0; c < m; c++)
        for (d = 0; d < n; d++)
            scanf("%d", &first[c][d]);

20    printf("Enter number of rows and columns of second matrix\n");
    scanf("%d%d", &p, &q);

    if (n != p)
        printf("The matrices can't be multiplied with each other.\n");

25    else
    {
```

```

        printf("Enter elements of second matrix\n");

        for (c = 0; c < p; c++)
30         for (d = 0; d < q; d++)
            scanf("%d", &second[c][d]);

        t = clock();
        for (c = 0; c < m; c++)
35        {
            for (d = 0; d < q; d++)
            {
                for (k = 0; k < p; k++)
                {
40                    sum = sum + first[c][k] * second[k][d];
                }

                multiply[c][d] = sum;
                sum = 0;
45            }
        }
        t = clock() - t;

        printf("Product of the matrices:\n");
50        for (c = 0; c < m; c++)
        {
            for (d = 0; d < q; d++)
                printf("%d\t", multiply[c][d]);
55            printf("\n");
        }
    }
    double time_taken_s = ((double)t) / CLOCKS_PER_SEC;
60    printf("Program took %f seconds to execute \n", time_taken_s);

    return 0;
}

```

Listing 6: Simple matrix multiplication using OpenMP

```

#include <stdio.h>
#include <time.h>

int main()
5 {
    int m, n, p, q, c, d, k, sum = 0;
    int first[10][10], second[10][10], multiply[10][10];
    clock_t t;

10    printf("-----16BCE2205-----\n");

    printf("Enter number of rows and columns of first matrix\n");
    scanf("%d%d", &m, &n);
    printf("Enter elements of first matrix\n");

```

```
15     for (c = 0; c < m; c++)
        for (d = 0; d < n; d++)
            scanf("%d", &first[c][d]);

20     printf("Enter number of rows and columns of second matrix\n");
    scanf("%d%d", &p, &q);

    if (n != p)
        printf("The matrices can't be multiplied with each other.\n");
25     else
    {
        printf("Enter elements of second matrix\n");

        for (c = 0; c < p; c++)
30         for (d = 0; d < q; d++)
            scanf("%d", &second[c][d]);

#pragma omp parallel shared(first, second, multiply) private(c, d, k)
    {
35         t = clock();
#pragma omp for schedule(static)
        for (c = 0; c < m; c++)
        {
            for (d = 0; d < q; d++)
40            {
                multiply[c][d] = 0;
                for (k = 0; k < p; k++)
                {
                    multiply[c][d] = multiply[c][d] + first[c][k] * second[k][d];
45                }
            }
        }

        t = clock() - t;

50        printf("Product of the matrices:\n");

        for (c = 0; c < m; c++)
        {
55            for (d = 0; d < q; d++)
                printf("%d\t", multiply[c][d]);

            printf("\n");
        }
60    }

    double time_taken_s = ((double)t) / CLOCKS_PER_SEC;
    printf("Program took %f seconds to execute \n", time_taken_s);

    return 0;
65 }
```

Output:

```
3 3 2
3 4 5
Program took 0.000347 seconds to execute
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ clear
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ gcc-8 -fopenmp matrix_multiplication.c
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ ./a.out
-----16BCE2205-----
Enter number of rows and columns of first matrix
3 3
Enter elements of first matrix
1 2 0
0 1 1
2 0 1
Enter number of rows and columns of second matrix
3 3
Enter elements of second matrix
1 1 2
2 1 1
1 2 1
Product of the matrices:
5 3 4
3 3 2
3 4 5
Program took 0.000005 seconds to execute
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ gcc-8 -fopenmp matrix_multiplication_parallel.c
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ ./a.out
-----16BCE2205-----
Enter number of rows and columns of first matrix
3 3
Enter elements of first matrix
1 2 0
0 1 1
2 0 1
Enter number of rows and columns of second matrix
3 3
Enter elements of second matrix
1 1 2
2 1 1
1 2 1
Product of the matrices:
5 3 4
3 3 2
3 4 5
Program took 0.000183 seconds to execute
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$
```