

**CSE4001 - Parallel and Distributed
Computing**

Lab 21+22

Assessment-1

Submitted by: Alokam Nikhitha

Reg No:19BCE2555

Submitted to: HITESHWAR KUMAR AZAD

Question

1. Write an OpenMP code to find the Sum of Elements of a One-Dimensional Real Array using vector addition. Where the two vectors A and B are added into C by spawning a team of threads and assigning a chunk of work to each thread.

Note: Sets the environment variable `omp_num_threads` to 6.

Code:

```
#include <stdlib.h> //malloc and free
#include <stdio.h> //printf
#include <omp.h> //OpenMP

// Very small values for this simple illustrative example
#define ARRAY_SIZE 8
//Size of arrays whose elements will be added together.
#define NUM_THREADS 10 //Number of threads to use for vector addition.

/*
 * Classic vector addition using openMP default data decomposition.
 *
 * Compile using gcc like this:
 * gcc -o VectorAdd -fopenmp VectorAdd.c
 * or, g++ -fopenmp VectorAdd.c
 *
 * Execute:
 * ./VectorAdd
 * or, ./a.out
 */
```

```
int main (int argc, char *argv[])
{
// To pass command line arguments, we typically define main() with two arguments : first
argument is the number of command line arguments and second is list of command-line
arguments.
// int main(int argc, char *argv[]) { /* ... */ }
// argc (ARGument Count) is int and stores number of command-line arguments passed
by the user including the name of the program. So if we pass a value to a program,
value of argc would be 2 (one for argument and one for program name)
//The value of argc should be non negative.
//argv(ARGument Vector) is array of character pointers listing all the arguments.
//If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain
pointers to strings.
//Argv[0] is the name of the program , After that till argv[argc-1] every element is
command -line arguments.

// elements of arrays a and b will be added
// and placed in array c
int * a;
int * b;
int * c;

int n = ARRAY_SIZE; // number of array elements
int n_per_thread; // elements per thread
int total_threads = NUM_THREADS; // number of threads to use
int i; // loop index

// allocate spce for the arrays
a = (int *) malloc(sizeof(int)*n);
b = (int *) malloc(sizeof(int)*n);
c = (int *) malloc(sizeof(int)*n);
```

```

// initialize arrays a and b with consecutive integer values
// as a simple example
for(i=0; i<n; i++) {
    a[i] = 3*i;
}
for(i=0; i<n; i++) {
    b[i] = 2*i;
}

// Additional work to set the number of threads.
// We hard-code to 4 for illustration purposes only.
omp_set_num_threads(total_threads);

// determine how many elements each process will work on
n_per_thread = n/total_threads;

// Compute the vector addition
// Here is where the 4 threads are specifically 'forked' to
// execute in parallel. This is directed by the pragma and
// thread forking is compiled into the resulting executable.
// Here we use a 'static schedule' so each thread works on
// a 2-element chunk of the original 8-element arrays.
#pragma omp parallel for shared(a, b, c) private(i) schedule(static, n_per_thread)
for(i=0; i<n; i++) {
    c[i] = a[i]+b[i];
}

// Which thread am I? Show who works on what for this small example
printf("Thread %d works on element%d\n", omp_get_thread_num(), i);
}

// Check for correctness (only plausible for small vector size)
// A test we would eventually leave out
printf("i\t a[i]\t +\t b[i]\t =\t c[i]\n");

```

```
for(i=0; i<n; i++) {
printf("%d\t%d\t%d\t%d\t%d\n", i, a[i], b[i], c[i]);
}
```

```
// clean up memory
free(a); free(b); free(c);
```

```
return 0;
}
```

```

1#include <stdlib.h> //malloc and free
2#include <stdio.h> //printf
3#include <omp.h> //OpenMP
4
5// Very small values for this simple illustrative example
6#define ARRAY_SIZE 8 //Size of arrays whose elements will be added together.
7#define NUM_THREADS 6
8 //Number of threads to use for vector addition.
9
10/*
11 * Classic vector addition using openMP default data decomposition.
12 *
13 * Compile using gcc like this:
14 * gcc -o va-omp-simple VA-OMP-simple.c -fopenmp
15 *
16 * Execute:
17 * ./va-omp-simple
18 */
19int main (int argc, char *argv[])
20{
21    // elements of arrays a and b will be added
22    // and placed in array c
23    int * a;
24    int * b;
25    int * c;
26
27    int n = ARRAY_SIZE; // number of array elements
28    int n_per_thread; // elements per thread
29    int total_threads = NUM_THREADS; // number of threads to use
30    int i; // loop index
31
32    // allocate space for the arrays
33    a = (int *) malloc(sizeof(int)*n);
34    b = (int *) malloc(sizeof(int)*n);
35    c = (int *) malloc(sizeof(int)*n);
36
37    // initialize arrays a and b with consecutive integer values

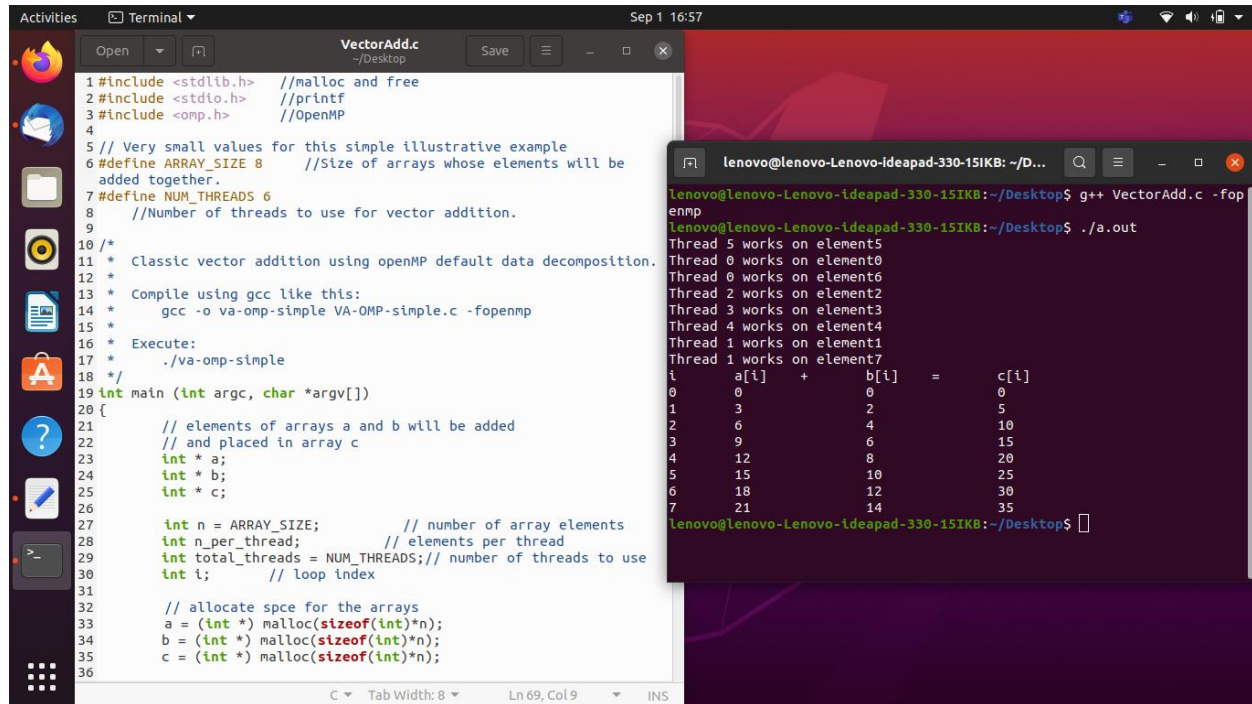
```

```
Activities Text Editor Sep 1 16:55
*VectorAdd.c ~/Desktop Save
37 // initialize arrays a and b with consecutive integer values
38 // as a simple example
39 for(i=0; i<n; i++) {
40     a[i] = 3*i;
41 }
42 for(i=0; i<n; i++) {
43     b[i] = 2*i;
44 }
45 // Additional work to set the number of threads.
46 // We hard-code to 4 for illustration purposes only.
47 omp_set_num_threads(total_threads);
48 // determine how many elements each process will work on
49 n_per_thread = n/total_threads;
50 // Compute the vector addition
51 // Here is where the 4 threads are specifically 'forked' to
52 // execute in parallel. This is directed by the pragma and
53 // thread forking is compiled into the resulting executable.
54 // Here we use a 'static schedule' so each thread works on
55 // a 2-element chunk of the original 8-element arrays.
56 #pragma omp parallel for shared(a, b, c) private(i) schedule(static, n_per_thread)
57 for(i=0; i<n; i++) {
58     c[i] = a[i]+b[i];
59     // Which thread am I? Show who works on what for this small example
60     printf("Thread %d works on element %d\n", omp_get_thread_num(), i);
61 }
62
63 // Check for correctness (only plausible for small vector size)
64 // A test we would eventually leave out
65 printf("i\ta[i]\t+b[i]\t=c[i]\n");
66 for(i=0; i<n; i++) {
67     printf("%d\t%d\t%d\t%d\n", i, a[i], b[i], c[i]);
68 }
69 // clean up memory
70 free(a); free(b); free(c);
71
72 return 0;
73 }
```

OUTPUT:

The screenshot shows a Linux desktop with a red background. A terminal window is open, displaying the execution of a C++ program. The program uses OpenMP for parallelization. The output shows the program running on 8 threads (0-7) and calculating the sum of two arrays, a and b, to produce array c. The results are as follows:

i	a[i]	b[i]	c[i]
0	0	0	0
1	3	2	5
2	6	4	10
3	9	6	15
4	12	8	20
5	15	10	25
6	18	12	30
7	21	14	35



```

1 #include <stdlib.h> //malloc and free
2 #include <stdio.h> //printf
3 #include <omp.h> //OpenMP
4
5 // Very small values for this simple illustrative example
6 #define ARRAY_SIZE 8 //Size of arrays whose elements will be
7 //added together.
8 #define NUM_THREADS 6 //Number of threads to use for vector addition.
9
10 /*
11 * Classic vector addition using openMP default data decomposition.
12 *
13 * Compile using gcc like this:
14 * gcc -o va-omp-simple VA-OMP-simple.c -fopenmp
15 *
16 * Execute:
17 * ./va-omp-simple
18 */
19 int main (int argc, char *argv[])
20 {
21     // elements of arrays a and b will be added
22     // and placed in array c
23     int * a;
24     int * b;
25     int * c;
26
27     int n = ARRAY_SIZE; // number of array elements
28     int n_per_thread; // elements per thread
29     int total_threads = NUM_THREADS; // number of threads to use
30     int i; // loop index
31
32     // allocate space for the arrays
33     a = (int *) malloc(sizeof(int)*n);
34     b = (int *) malloc(sizeof(int)*n);
35     c = (int *) malloc(sizeof(int)*n);
36
37     // OpenMP parallel region
38     #pragma omp parallel
39     {
40         // Loop over array elements
41         for (i = 0; i < n; i++)
42             c[i] = a[i] + b[i];
43     }
44
45     // Print the result
46     for (i = 0; i < n; i++)
47         printf("a[%d] + b[%d] = c[%d]\n", i, i, i);
48
49     return 0;
50 }

```

```

lenovo@lenovo-Lenovo-Ideapad-330-15IK8: ~/Desktop$ gcc -fopenmp VectorAdd.c -fopenmp
lenovo@lenovo-Lenovo-Ideapad-330-15IK8: ~/Desktop$ ./a.out
Thread 5 works on element5
Thread 0 works on element0
Thread 0 works on element6
Thread 2 works on element2
Thread 3 works on element3
Thread 4 works on element4
Thread 1 works on element1
Thread 1 works on element7
i  a[i]  +  b[i]  =  c[i]
0  0      +  0      =  0
1  3      +  2      =  5
2  6      +  4      =  10
3  9      +  6      =  15
4  12     +  8      =  20
5  15     +  10     =  25
6  18     +  12     =  30
7  21     +  14     =  35
lenovo@lenovo-Lenovo-Ideapad-330-15IK8: ~/Desktop$

```