# 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:

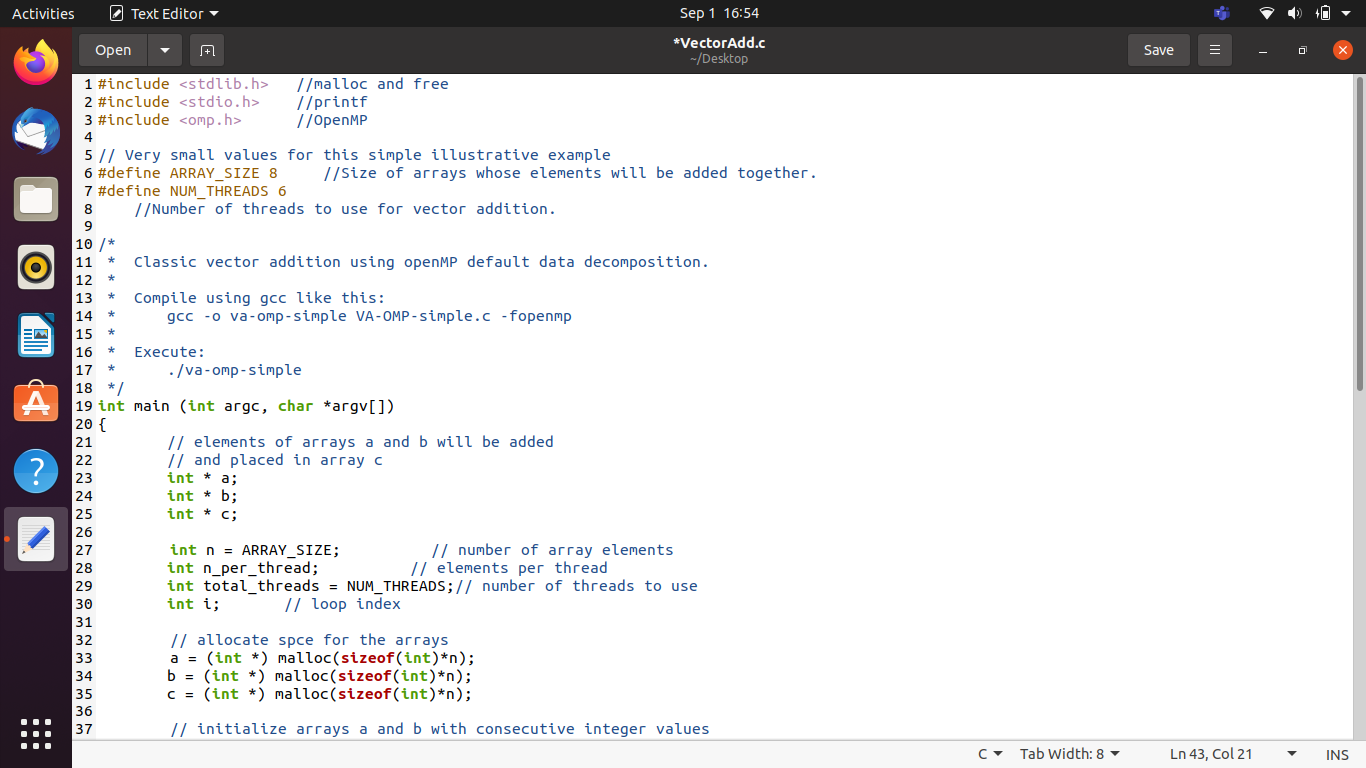
#include <stdlib.h> //malloc and free  
#include <stdio.h> //printf  
#include <omp.h> //OpenMP

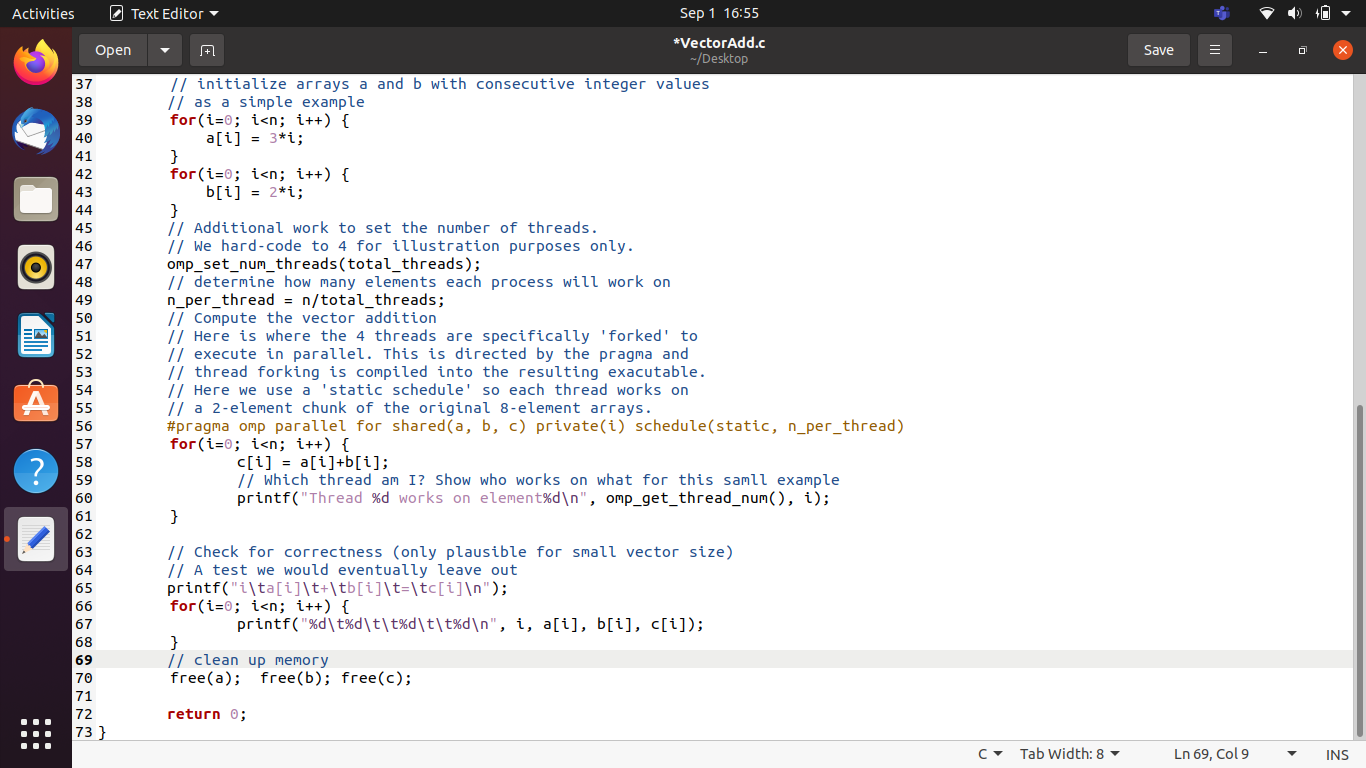
// Very small values for this simple illustrative example  
#define ARRAY\_SIZE 8  
//Size of arrays whose elements will be added together.  
#define NUM\_THREADS 10 //Number of threads to use for vector addition.

/\*  
\* 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 exacutable.  
// 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 samll 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\ta[i]\t+\tb[i]\t=\tc[i]\n");  
for(i=0; i<n; i++) {  
printf("%d\t%d\t\t%d\t\t%d\n", i, a[i], b[i], c[i]);  
}  
  
// clean up memory  
free(a); free(b); free(c);  
  
return 0;  
}





# OUTPUT:

