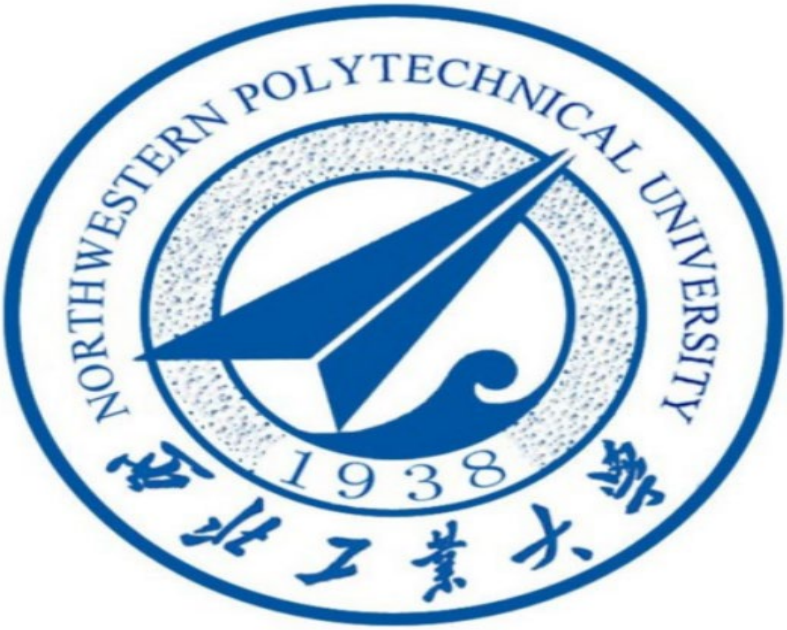


Parallel Programming

Homework 4

Student Name : ABID ALI
Student ID : 2019380141
Student Email : abiduu354@gmail.com / 3616990795@qq.com
Lecture : Professor Tianhai Zhao
Submission : 12/22/2022
Submitted Email : zhaoth@nwpu.edu.cn

1. Please read the Chapter 5 of An Introduction to Parallel Programming.

Solution:

Shared-memory programming with OpenMP

OpenMP is a standard for programming shared-memory MIMD systems. It uses both special functions and preprocessor directives called **pragmas**, so unlike Pthreads and MPI, OpenMP requires compiler support. One of the most important features of OpenMP is that it was designed so that developers could *incrementally* parallelize existing serial programs, rather than having to write parallel programs from scratch.

OpenMP programs start multiple **threads** rather than multiple processes. Threads can be much lighter weight than processes; they can share almost all the resources of a process, except each thread must have its own stack and program counter.

To get OpenMP's function prototypes and macros, we include the omp.h header in OpenMP programs. There are several OpenMP directives that start multiple threads; the most general is the parallel directive:

 # pragma omp p a r a l l e l
structured block

This directive tells the run-time system to execute the following structured block of code in parallel. It may **fork** or start several threads to execute the structured block. A **structured block** is a block of code with a single entry point and a single exit point, although calls to the C library function exit are allowed within a structured block. The number of threads started is system dependent, but most systems will start one thread for each available core. The collection of threads executing block of code is called a **team**. One of the threads in the team is the thread that was executing the code before the parallel directive. This thread is called the **parent**. The additional threads started by the parallel directive are called **child** threads. When all of the threads are finished, the child threads are terminated or **joined**, and the parent thread continues executing the code beyond the structured block.

Many OpenMP directives can be modified by **clauses**. We made frequent use of the num_threads clause. When we use an OpenMP directive that starts a team of threads, we can modify it with the num_threads clause so that the directive will start the number of threads we desire.

When OpenMP starts a team of threads, each of the threads is assigned a rank or ID in the range 0, 1, . . . , thread_count − 1. The OpenMP library function omp_get_thread_num the returns the calling thread's rank. The function omp_get_num_threads returns the number of threads in the current team. A major problem in the development of shared-memory programs is the possibility of **race conditions**. A race condition occurs when multiple threads attempt to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error. Code that is executed by multiple threads that update a shared resource that can only be updated by one thread at a time is called a **critical section**. Thus if multiple threads try to update a shared variable, the program has a race condition, and the code that updates the variable is a critical

section. OpenMP provides several mechanisms for ensuring **mutual exclusion** in critical sections. We examined four of them:

1. Critical directives ensure that only one thread at a time can execute the structured block. If multiple threads try to execute the code in the critical section, all but one of them will block before the critical section. When one thread finishes the critical section, another thread will be unblocked and enter the code.

2. Named critical directives can be used in programs having different critical sections that can be executed concurrently. Multiple threads trying to execute code in critical section(s) with the same name will be handled in the same way as multiple threads trying to execute an unnamed critical section. However, threads entering critical sections with different names can execute concurrently.

3. An atomic directive can only be used when the critical section has the form x <op> = <expression>, x++, ++x, x−−, or −−x. It's designed to exploit special hardware instructions, so it can be much faster than an ordinary critical section.

4. Simple locks are the most general form of mutual exclusion. They use function calls to restrict access to a critical section:

```
omp_set_lock(&lock ) ;
critical section
omp_unset_lock(&lock ) ;
```

When multiple threads call omp_set_lock, only one of them will proceed to the critical section. The others will block until the first thread calls omp_unset_lock.Then one of the blocked threads can proceed.

All of the mutual exclusion mechanisms can cause serious program problems, such as deadlock, so they need to be used with great care.

A **for** directive can be used to partition the iterations in a **for** loop among the threads. This directive doesn't start a team of threads; it divides the iterations in a **for** loop among the threads in an existing team. If we want also to start a team of threads, we can use the parallel **for** directive. There are a number of restrictions on the form of a **for** loop that can be parallelized; basically, the run-time system must be able to determine the total number of iterations through the loop body before the loop begins execution. For details, see Program 5.3.

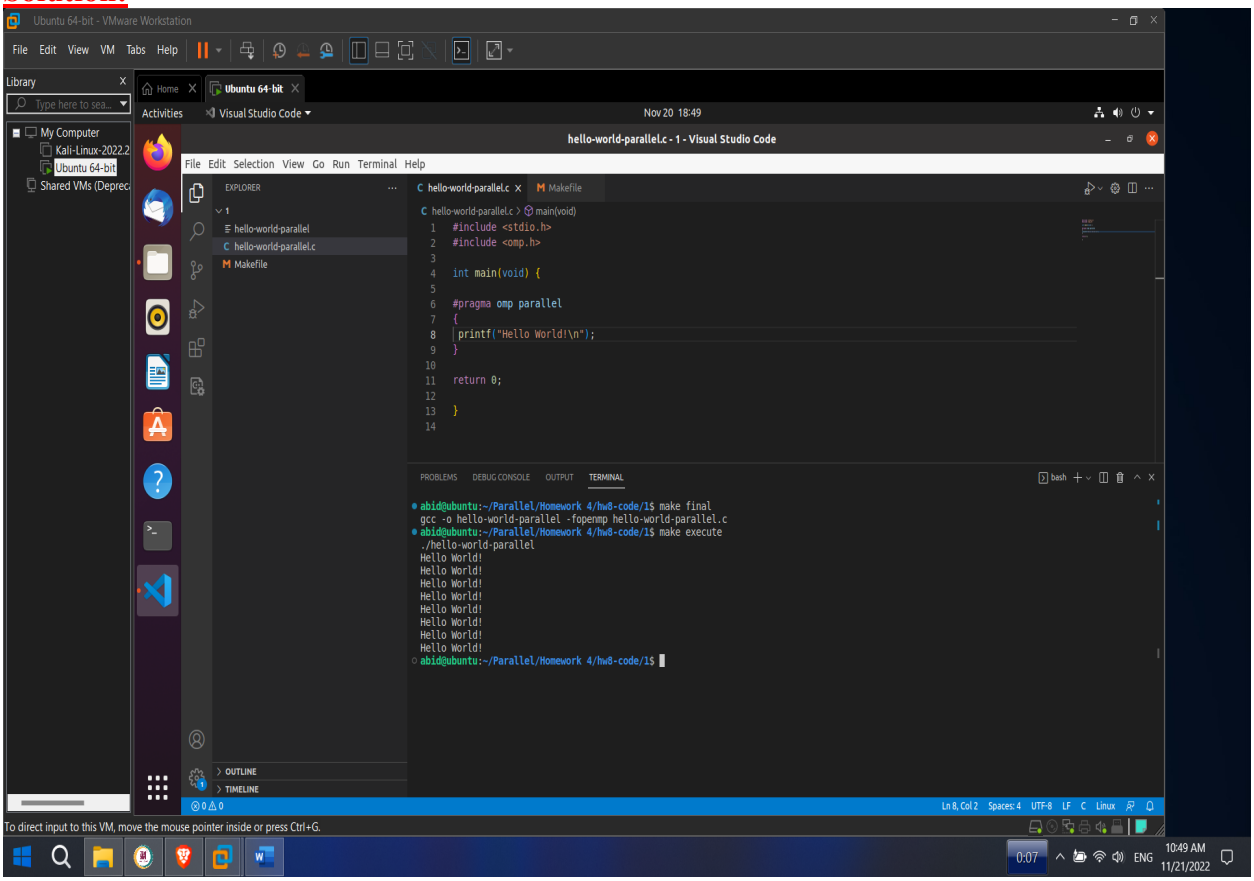
It's not enough, however, to ensure that our **for** loop has one of the canonical forms. It must also not have any **loop-carried dependences**. A loop-carried dependence occurs when a memory location is read or written in one iteration and written

2. Read and compile the OpenMP examples (if you use gcc, remember add -fopenmp flag.), and observe the output.

The following example programs introduce the main concepts of OpenMP step by step.

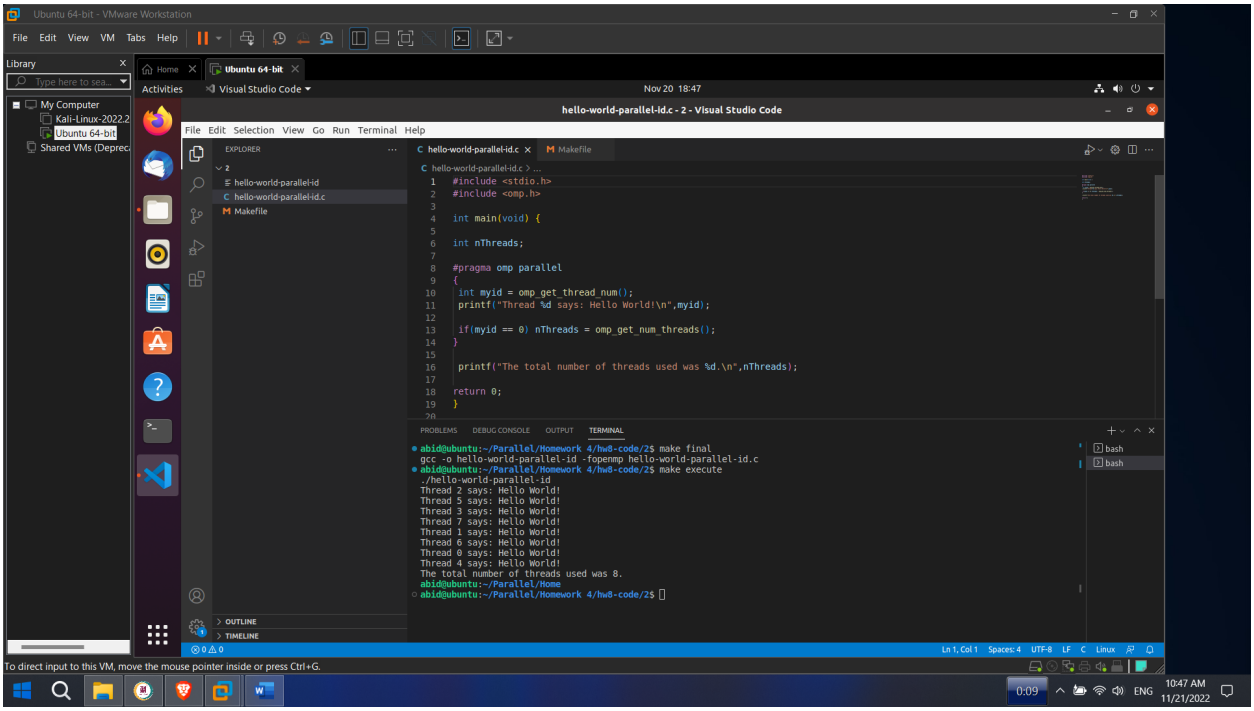
1) hello-world-parallel.c (the most basic hello world executed in parallel)

Solution:



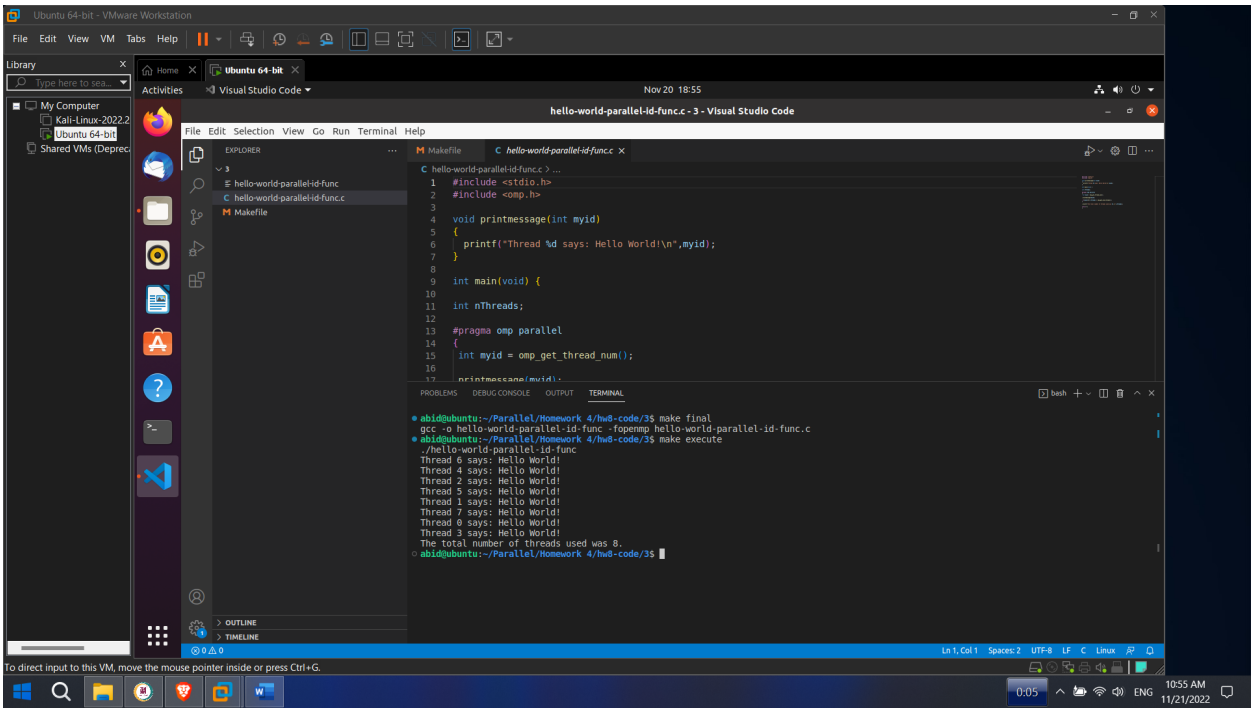
2) hello-world-parallel-id.c (similar to 1. but the id of each thread and the total number of threads are also printed)

Solution:



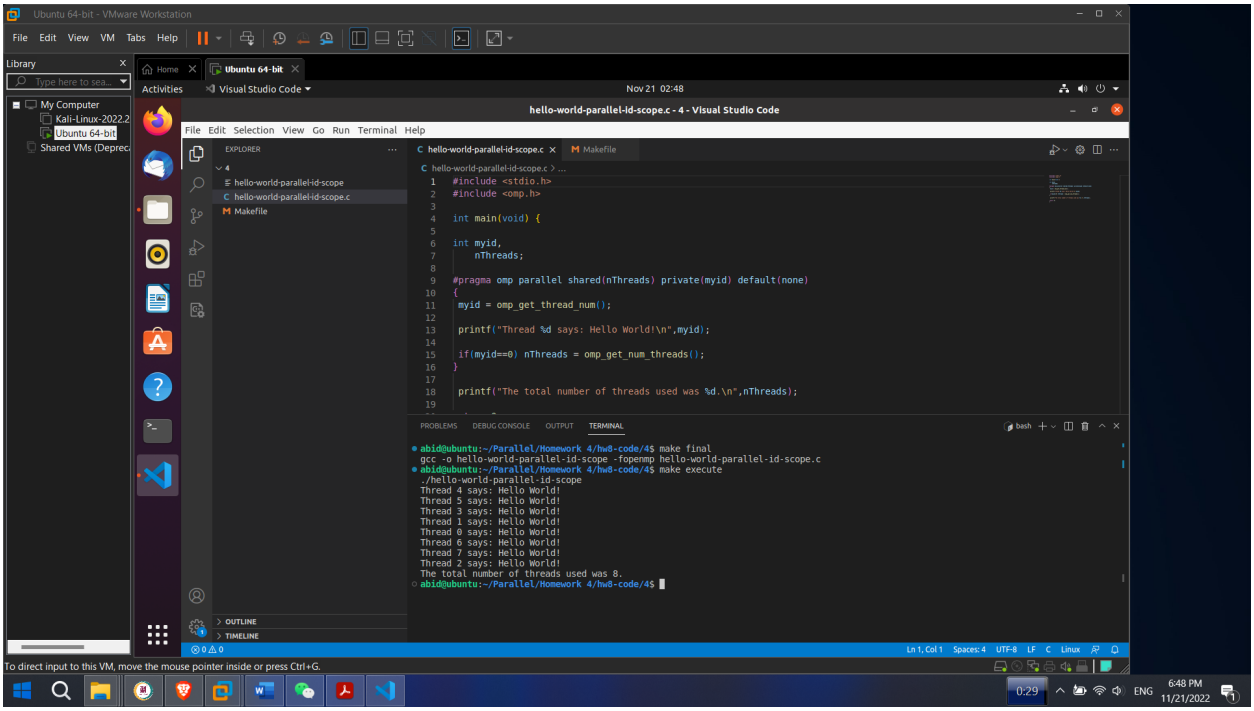
3) hello-world-parallel-id-func.c (same result as 2. but a function is called to print inside the parallel region)

Solution:



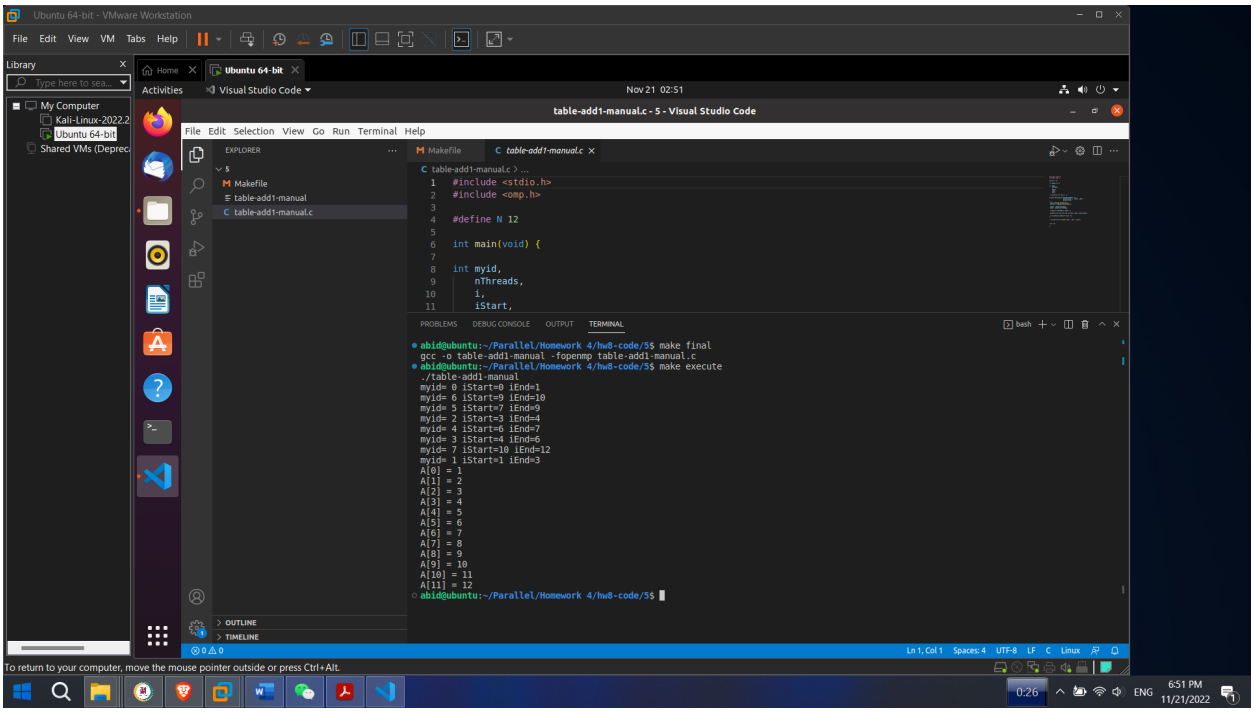
4) hello-world-parallel-id-scope.c (same result as 2. but data scope is used in the parallel region)

Solution:



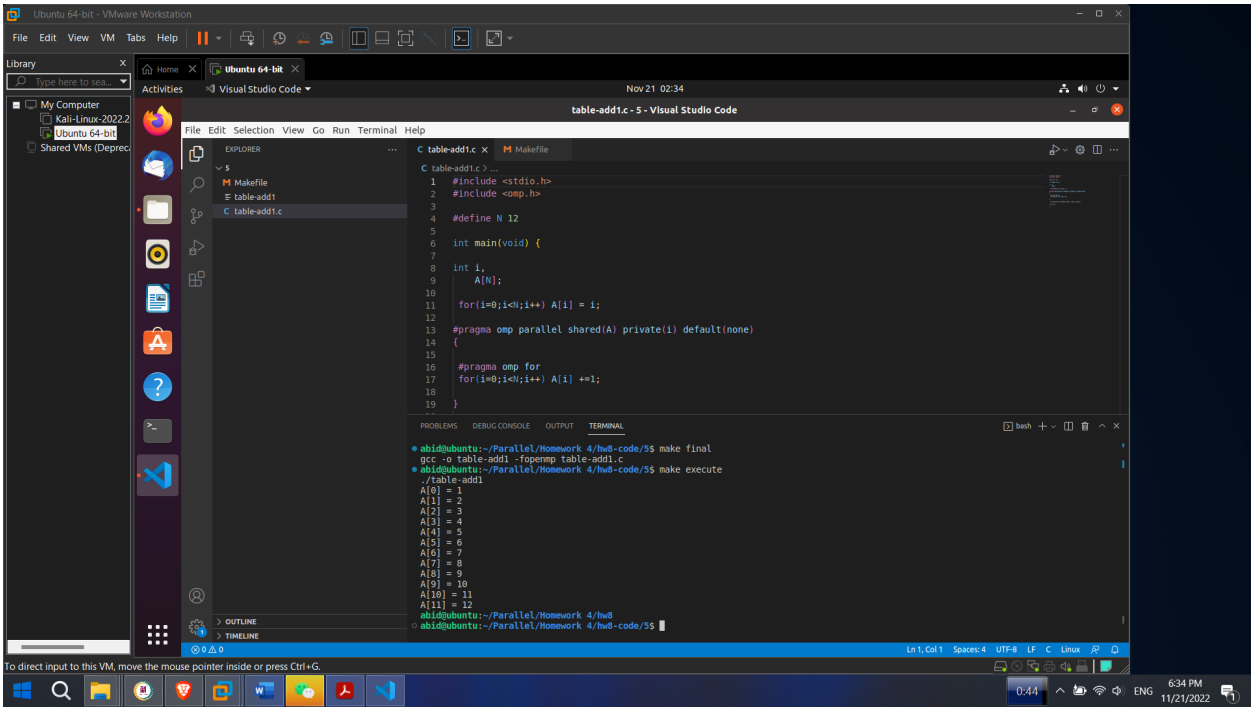
5) table-add1-manual.c (add a number to each element of a table, using manual parallelization)

Solution:



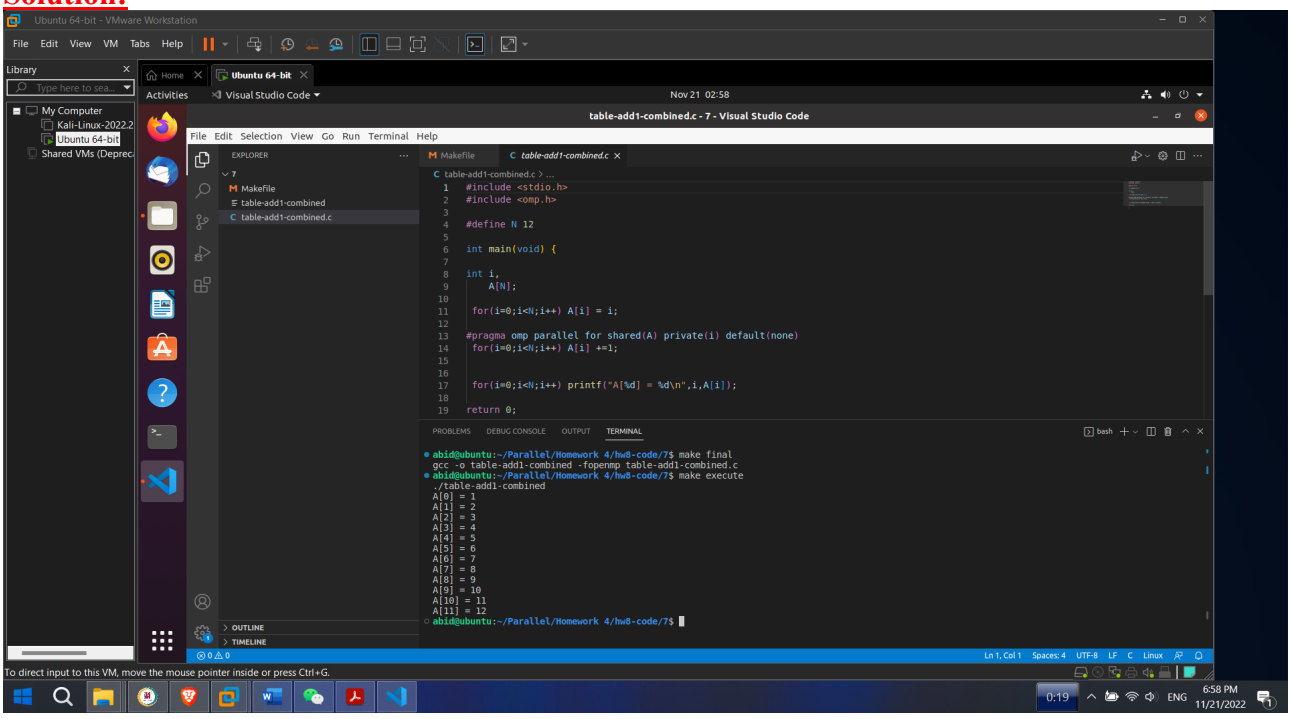
6) table-add1.c (same result as 5. but with automatic work scheduling)

Solution:



7) table-add1-combined.c (same result as 6. but with combined parallel region and for construct)

Solution:



8) table-add1-wrong.c (similar to 6. and 7. but giving wrong answer - find the error in the code!)

Solution:

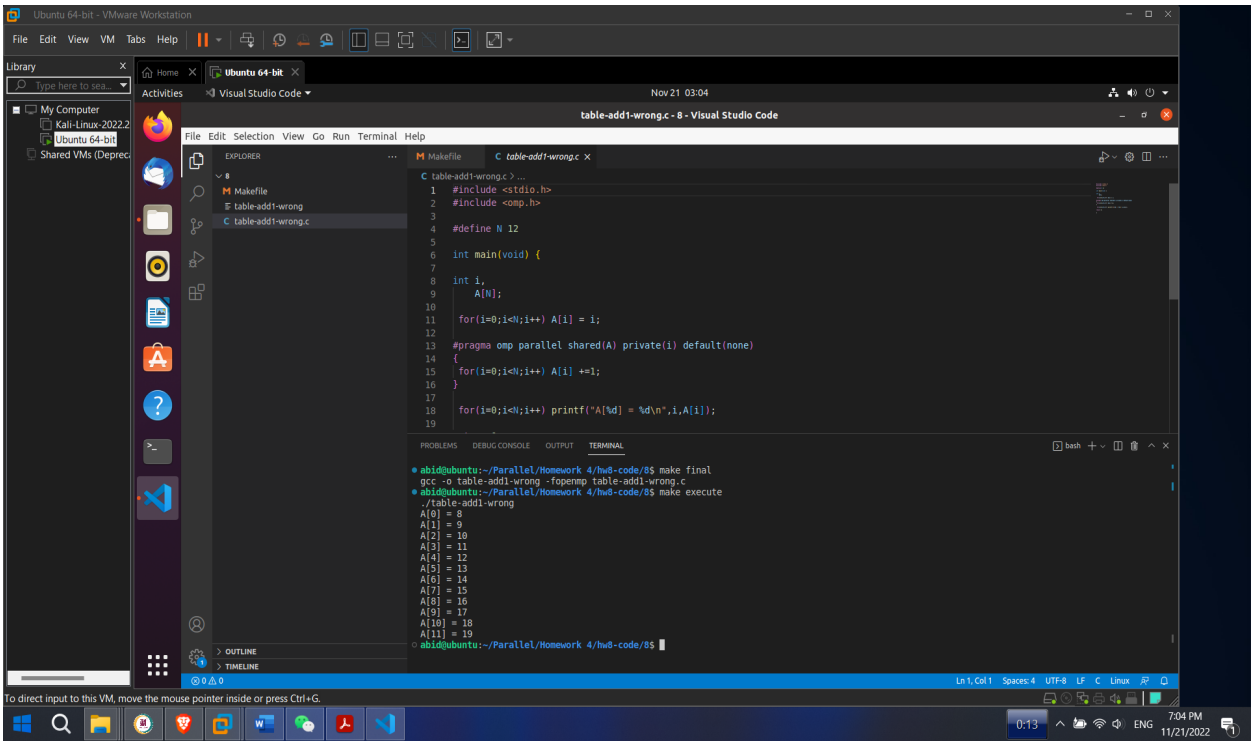


Figure : Wrong Answer

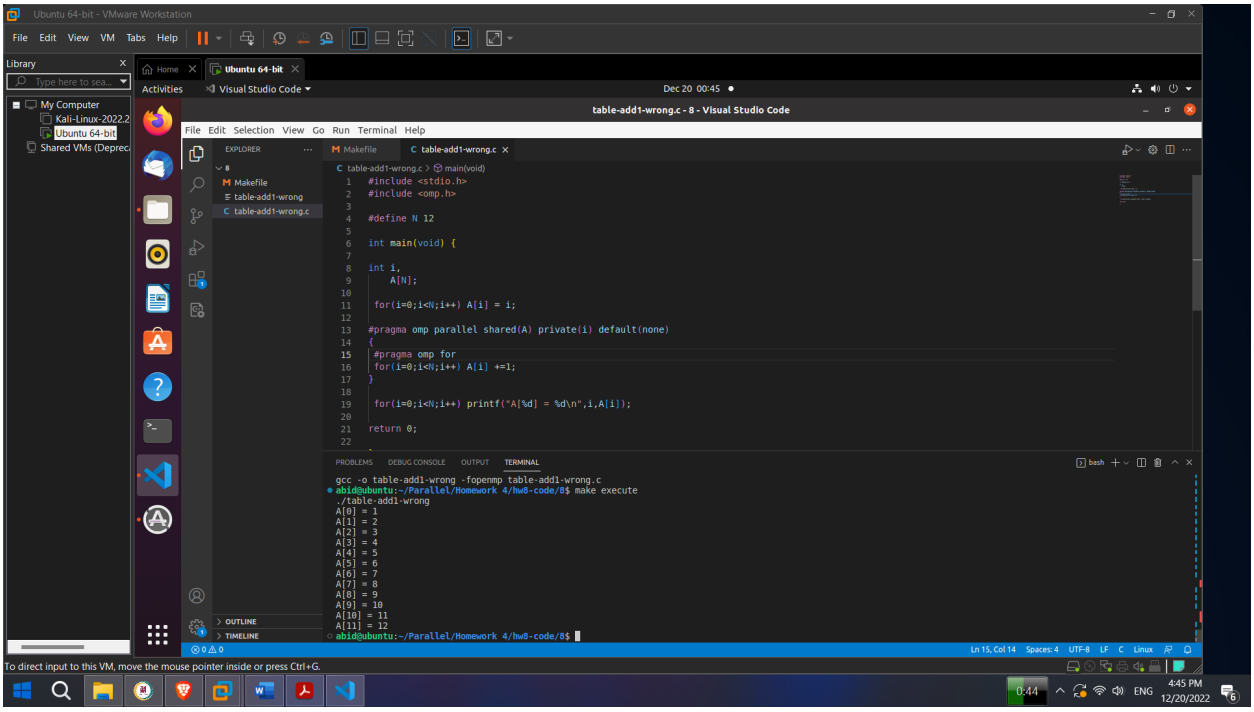


Figure : Correct Answer

```
#pragma omp parallel shared(A) private(i) default(none)
{
```

```
#pragma omp for //This paralyze the program
 for(i=0;i<N;i++) A[i] +=1;
}

 for(i=0;i<N;i++) printf("A[%d] = %d\n",i,A[i]);

return 0;

}
```

9) table-implicit-notpar.c (parallel execution gives wrong answer - find out why!)

Solution:

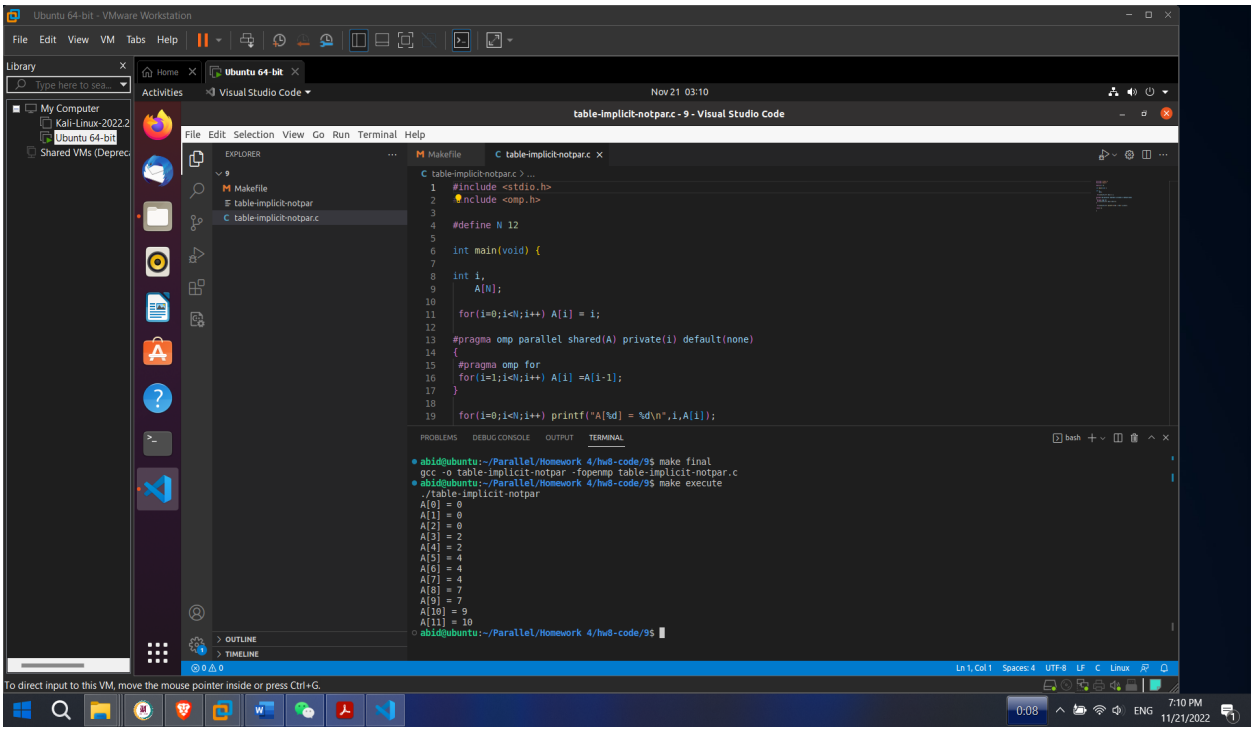


Figure : Wrong Answer

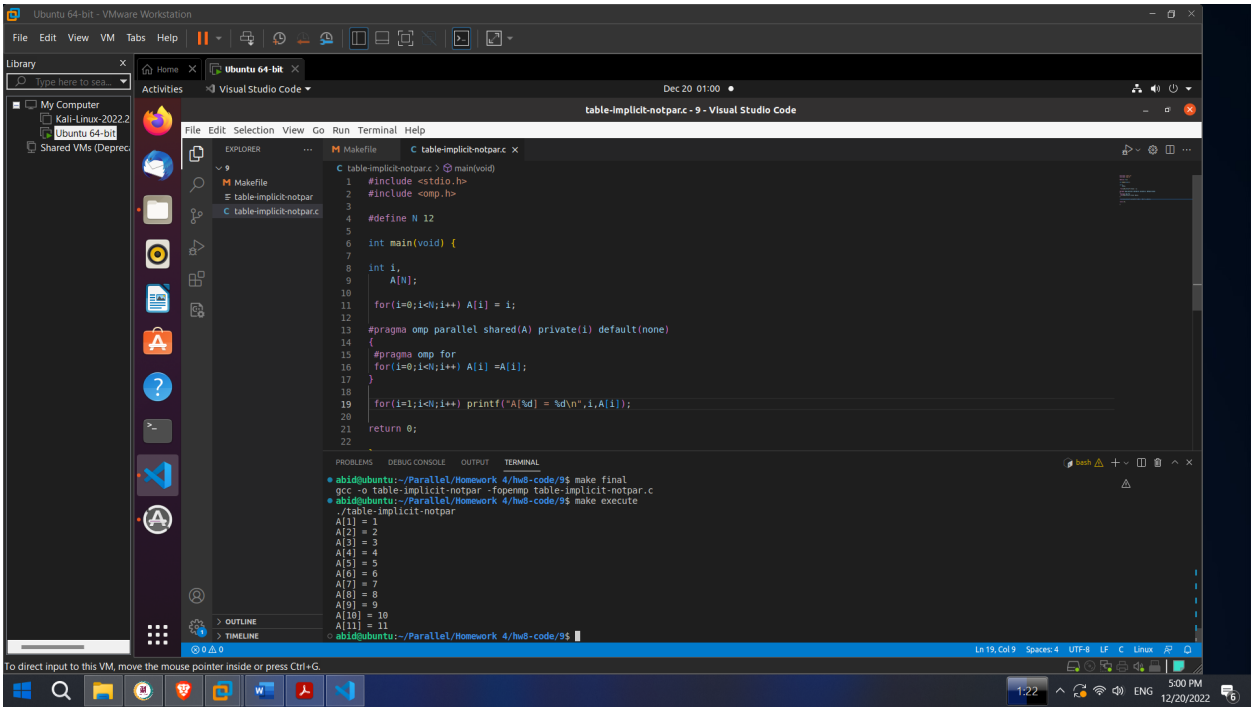


Figure : Correct Answer

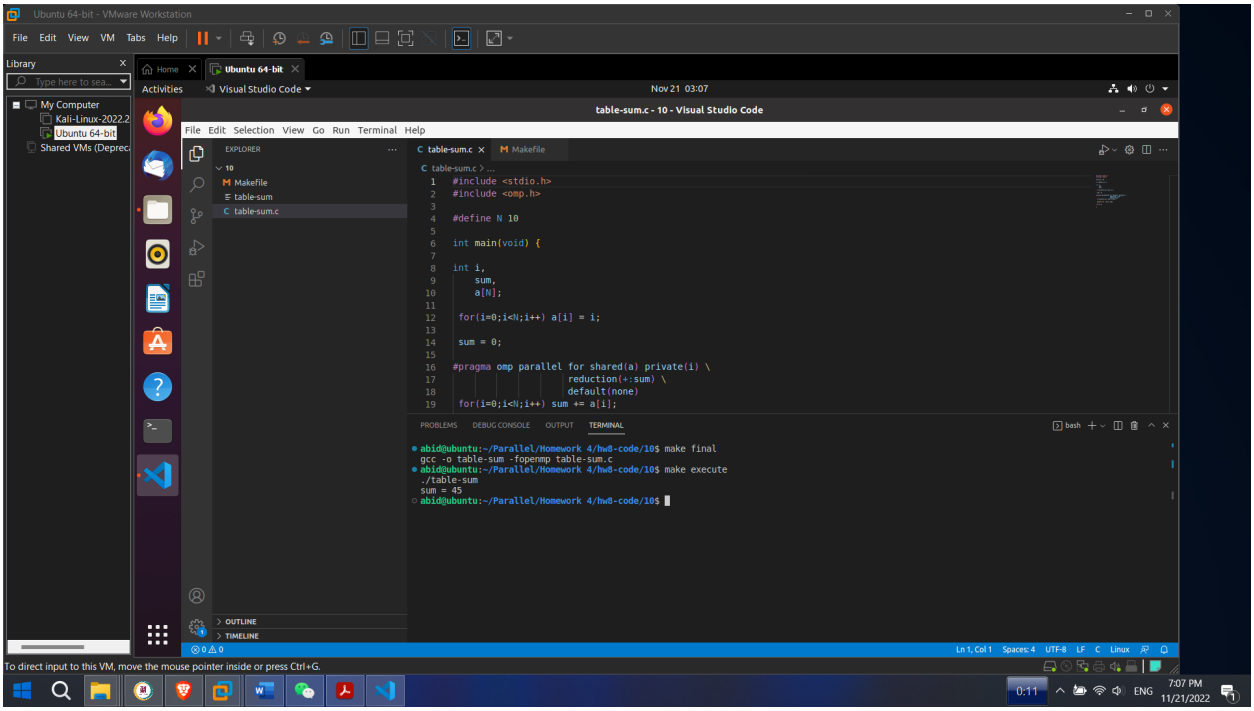
```
#pragma omp parallel shared(A) private(i) default(none)
{
 #pragma omp for
 for(i=0;i<N;i++) A[i] =A[i]; // Here index was reduced by 1 then we got correct value
}

 for(i=1;i<N;i++) printf("A[%d] = %d\n",i,A[i]);

return 0;
```

10) table-sum.c (computing the sum of all elements in a table)

Solution:



11) table-sum-wrong.c (similar to 10. but gives wrong answer - find the error in the code!)

Solution:

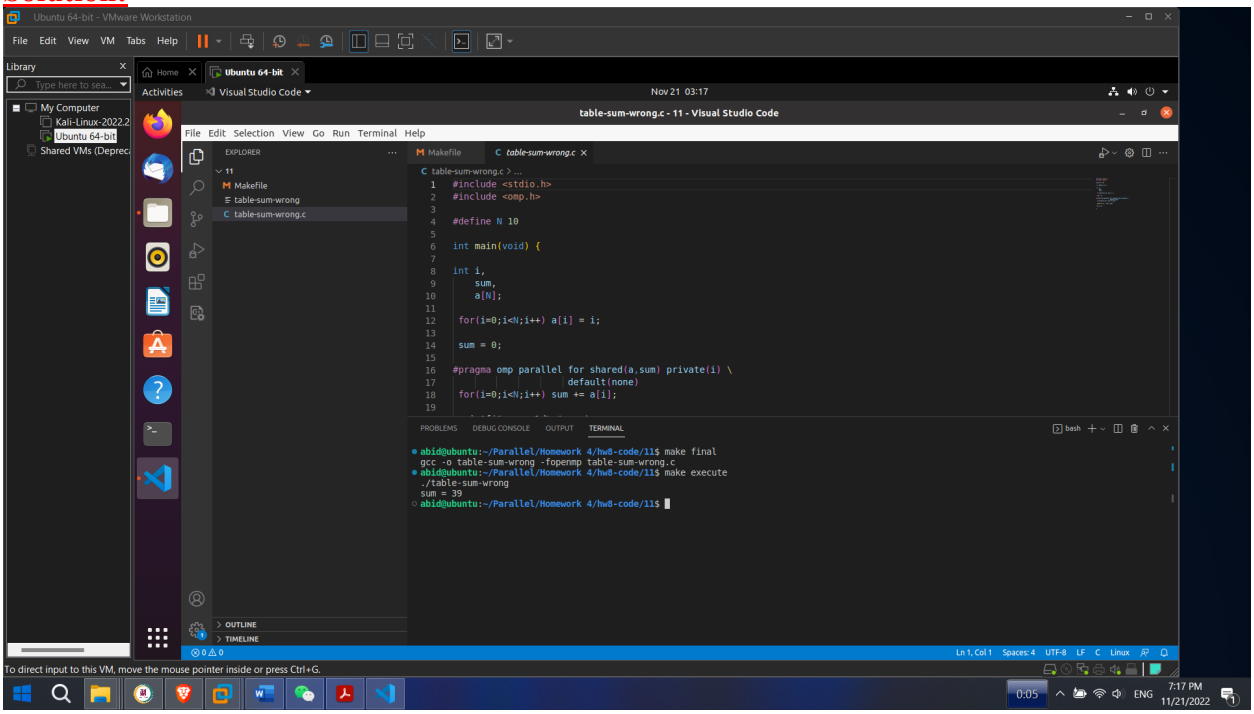


Figure : Wrong Answer

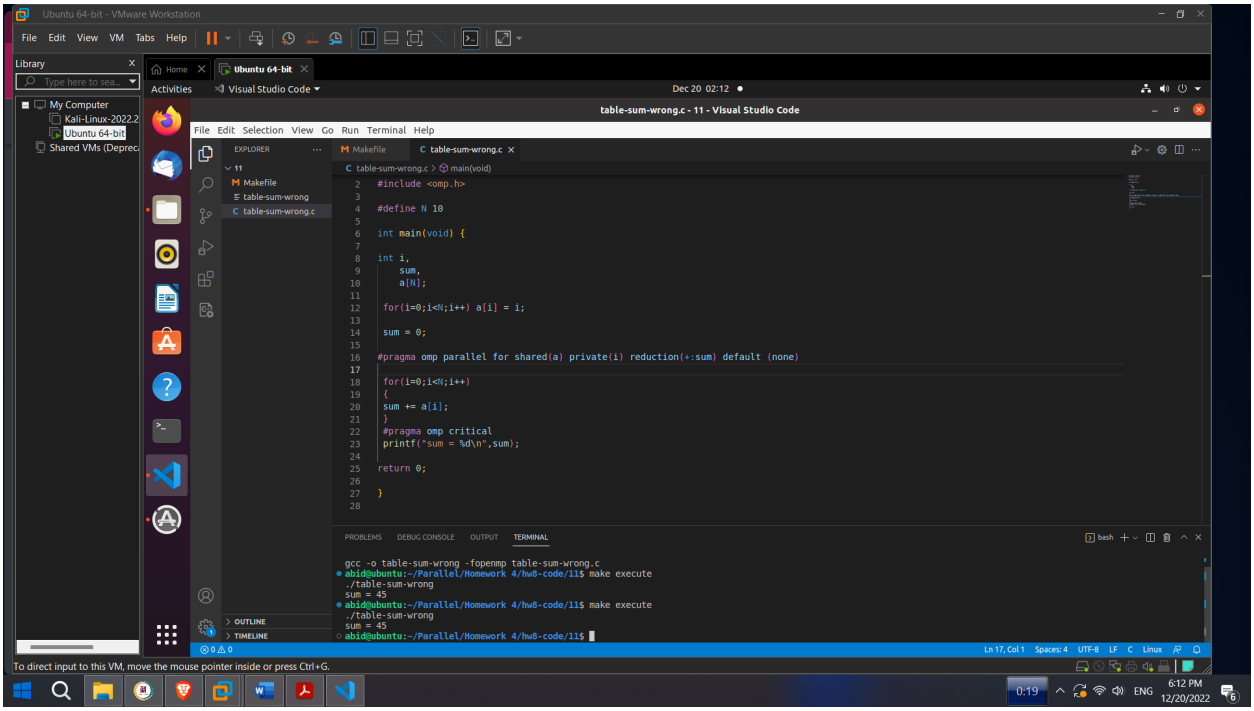


Figure : Correct Answer

#pragma omp parallel for shared(a) private(i) reduction(+:sum) default (none)

```
 for(i=0;i<N;i++)
 {
 sum += a[i];
 }
 #pragma omp critical
 printf("sum = %d\n",sum);

return 0;
```