



# Parallel Programming

## Homework 4

Student Name : ABID ALI  
Student ID : 2019380141  
Student Email : [abiduu354@gmail.com](mailto:abiduu354@gmail.com) / [3616990795@qq.com](mailto: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 parallel  
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  $0, 1, \dots, \text{thread\_count} - 1$ . The OpenMP library function `omp_get_thread_num` 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:**

The screenshot shows a VMware workstation window titled "Ubuntu 64-bit - VMware Workstation". Inside, a Visual Studio Code instance is open. The Explorer sidebar shows a folder named "hello-world-parallel" containing "hello-world-parallel.c" and a "Makefile". The code editor displays the following C code:

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(void) {
5
6     #pragma omp parallel
7     {
8         printf("Hello World\n");
9     }
10    return 0;
11
12 }
13
14
```

The terminal tab in VS Code shows the command-line interface of the host system (Ubuntu 22.04). It runs "make final" and then executes the resulting binary, showing multiple "Hello World!" messages indicating parallel execution.

- 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:**

The screenshot shows a VMware Workstation window titled "Ubuntu 64-bit - VMware Workstation". The desktop environment is Unity. A terminal window is open, showing the output of a parallel C program. The program, named `hello-world-parallel-id.c`, uses OpenMP to print "Hello World!" from multiple threads. The terminal output shows 8 threads, each printing "Hello World!". The code itself defines a main function that prints the thread ID and the total number of threads.

```
#include <stdio.h>
#include <omp.h>
int main(void) {
    int nThreads;
    #pragma omp parallel
    {
        int myid = omp_get_thread_num();
        printf("Thread %d says: Hello World!\n",myid);
        if(myid == 0) nThreads = omp_get_num_threads();
    }
    printf("The total number of threads used was %d.\n",nThreads);
    return 0;
}
```

Terminal Output:

```
abid@ubuntu:~/Parallel/Homework 4/hw8-code/2$ make final
gcc -o hello-world-parallel-id -fopenmp hello-world-parallel-id.c
abid@ubuntu:~/Parallel/Homework 4/hw8-code/2$ make execute
./hello-world-parallel-id
Thread 0 says: Hello World!
Thread 5 says: Hello World!
Thread 3 says: Hello World!
Thread 7 says: Hello World!
Thread 1 says: Hello World!
Thread 6 says: Hello World!
Thread 8 says: Hello World!
Thread 4 says: Hello World!
The total number of threads used was 8.
abid@ubuntu:~/Parallel/Homework 4/hw8-code/2$
```

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

## Solution:

The screenshot shows a terminal session within a Visual Studio Code window on an Ubuntu 64-bit VM. The terminal output is as follows:

```
● abid@ubuntu:~/Parallel/Homework/4/hub-code/3$ make final
gcc -o hello-world-parallel-id-func ./openmp hello-world-parallel-id-func.c
● abid@ubuntu:~/Parallel/Homework/4/hub-code/3$ make execute
./hello-world-parallel-id-func
Thread 6 says: Hello World!
Thread 4 says: Hello World!
Thread 2 says: Hello World!
Thread 5 says: Hello World!
Thread 1 says: Hello World!
Thread 7 says: Hello World!
Thread 9 says: Hello World!
Thread 8 says: Hello World!
The Total number of threads used was 8.
● abid@ubuntu:~/Parallel/Homework/4/hub-code/3$
```

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

### Solution:

The screenshot shows a VMware Workstation window titled "Ubuntu 64-bit - VMware Workstation". The desktop environment is Unity. A terminal window is open, showing the output of a parallel OpenMP program. The program prints "Hello World!" from 8 threads, and the total number of threads used was 8.

```
ubuntu@ubid:~/Parallel/Homework 4/hw8-code/4$ make final
gcc -o hello-world-parallel-id-scope -fopenmp hello-world-parallel-id-scope.c
ubuntu@ubid:~/Parallel/Homework 4/hw8-code/4$ make execute
hello-world-parallel-id-scope
Thread 4 says: Hello World!
Thread 5 says: Hello World!
Thread 3 says: Hello World!
Thread 2 says: Hello World!
Thread 0 says: Hello World!
Thread 6 says: Hello World!
Thread 7 says: Hello World!
Thread 1 says: Hello World!
The total number of threads used was 8.
ubuntu@ubid:~/Parallel/Homework 4/hw8-code/4$
```

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

### Solution:

The screenshot shows a VMware Workstation window with an Ubuntu 64-bit guest operating system. The desktop environment is Unity. A Visual Studio Code window is open, displaying a C program named `table-add1-manual.c`. The code defines a macro `N` as 12 and contains a `main` function that prints array elements `A[0]` through `A[11]`. The terminal window shows the command `gcc -o table-add1-manual -fopenmp table-add1-manual.c` being run, followed by the output of the program itself, which lists the values from 1 to 12.

```
#include <stdio.h>
#include <omp.h>
#define N 12
int main(void) {
    int myid,
        nthreads,
        i,
        iStart,
        iEnd;
    /*****
     * parallel loop
     */
    myid = 0; iStart=0; iEnd=1;
    myid = 6; iStart=9; iEnd=10;
    myid = 5; iStart=8; iEnd=9;
    myid = 4; iStart=3; iEnd=4;
    myid = 4; iStart=6; iEnd=7;
    myid = 3; iStart=1; iEnd=2;
    myid = 1; iStart=10; iEnd=12;
    myid = 1; iStart=4; iEnd=3;
    A[0] = 1;
    A[1] = 2;
    A[2] = 3;
    A[3] = 4;
    A[4] = 5;
    A[5] = 6;
    A[6] = 7;
    A[7] = 8;
    A[8] = 9;
    A[9] = 10;
    A[10] = 11;
    A[11] = 12
}
```

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

### Solution:

The screenshot shows a Linux desktop environment with a terminal window open in a terminal emulator. The terminal window displays the following code and its execution:

```
#include <stdio.h>
#include <omp.h>
#define N 12
int main(void) {
    int i, A[N];
    for(i=0;i<N;i++) A[i] = i;
    #pragma omp parallel shared(A) private(i) default(none)
    {
        #pragma omp for
        for(i=0;i<N;i++) A[i] += i;
    }
}
```

The terminal output shows the results of the computation:

```
abid@ubuntu:~/Parallel/Homework 4/hw8-code/$ make final
gcc -o table-add1 -fopenmp table-add1.c
abid@ubuntu:~/Parallel/Homework 4/hw8-code/$ make execute
./table-add1
A[0] = 1
A[1] = 2
A[2] = 3
A[3] = 4
A[4] = 5
A[5] = 6
A[6] = 7
A[7] = 8
A[8] = 9
A[9] = 10
A[10] = 11
A[11] = 12
```

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

### Solution:

The screenshot shows a VMware Workstation interface with an Ubuntu 64-bit virtual machine running. The desktop environment is Unity. A Visual Studio Code window is open, displaying a C program named `table-addt-combined.c`. The code uses OpenMP to parallelize a loop that calculates a cumulative sum. The terminal window below shows the execution of the program and its output, which is a sequence of integers from 1 to 12.

```
#include <stdio.h>
#include <omp.h>
#define N 12
int main(void) {
    int i, A[N];
    for(i=0;i<N;i++) A[i] = i;
#pragma omp parallel for shared(A) private(i) default(none)
    for(i=0;i<N;i++) A[i] += i;
    for(i=0;i<N;i++) printf("A[%d] = %d\n",i,A[i]);
    return 0;
}
```

```
A[0] = 1
A[1] = 2
A[2] = 3
A[3] = 4
A[4] = 5
A[5] = 6
A[6] = 7
A[7] = 8
A[8] = 9
A[9] = 10
A[10] = 11
A[11] = 12
```

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

### Solution:

The screenshot shows a Visual Studio Code window running on an Ubuntu 64-bit VM in VMware Workstation. The code editor displays a file named `table-add1-wrong.c`. The code contains a parallel region where each thread increments an array element by 1 instead of adding 1 to it. The terminal output shows the resulting array values, which are all 1 greater than they should be.

```
#include <stdio.h>
#include <omp.h>
#define N 12
int main(void) {
    int i, A[N];
    for(i=0;i<N;i++) A[i] = i;
#pragma omp parallel shared(A) private(i) default(none)
{
    for(i=0;i<N;i++) A[i] += 1;
}
    for(i=0;i<N;i++) printf("A[%d] = %d\n",i,A[i]);
}
```

```
* abid@ubuntu:~/Parallel/Homework 4/hw8-code/$ make final
gcc -o table-add1-wrong -fopenmp table-add1-wrong.c
* abid@ubuntu:~/Parallel/Homework 4/hw8-code/$ make execute
./table-add1-wrong
A[0] = 9
A[1] = 9
A[2] = 10
A[3] = 11
A[4] = 12
A[5] = 13
A[6] = 14
A[7] = 15
A[8] = 16
A[9] = 17
A[10] = 18
A[11] = 19
* abid@ubuntu:~/Parallel/Homework 4/hw8-code/$
```

Figure : Wrong Answer

The screenshot shows the same Visual Studio Code window after the bug was fixed. The code editor now correctly adds 1 to each array element within the parallel region. The terminal output shows the expected array values.

```
#include <stdio.h>
#include <omp.h>
#define N 12
int main(void) {
    int i, A[N];
    for(i=0;i<N;i++) A[i] = i;
#pragma omp parallel shared(A) private(i) default(none)
{
#pragma omp for
    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;
}
```

```
* abid@ubuntu:~/Parallel/Homework 4/hw8-code/$ make final
gcc -o table-add1-wrong -fopenmp table-add1-wrong.c
* abid@ubuntu:~/Parallel/Homework 4/hw8-code/$ make execute
./table-add1-wrong
A[0] = 1
A[1] = 2
A[2] = 3
A[3] = 4
A[4] = 5
A[5] = 6
A[6] = 7
A[7] = 8
A[8] = 9
A[9] = 10
A[10] = 11
A[11] = 12
* abid@ubuntu:~/Parallel/Homework 4/hw8-code/$
```

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:

```

Ubuntu 64-bit - VMware Workstation
File Edit View VM Tabs Help || Type here to search ...
Activities Visual Studio Code
File Edit Selection View Go Run Terminal Help
LIBRARY
My Computer
Kali-Linux-2022.2
Ubuntu 64-bit
Shared VMS (Deprec)
ACTIVITIES
VS Code
File Edit Selection View Go Run Terminal Help
EXPLORER
table-implicit-notpar.c
Makefile
table-implicit-notpar.c
table-implicit-notpar.c
1 #include <stdio.h>
2 #include <omp.h>
3
4 #define N 12
5
6 int main(void) {
7
8     int i,
9         A[N];
10
11    for(i=0;i<N;i++) A[i] = i;
12
13    #pragma omp parallel shared(A) private(i) default(none)
14    {
15        #pragma omp for
16        for(i=1;i<N;i++) A[i] = A[i-1];
17    }
18
19    for(i=0;i<N;i++) printf("A[%d] = %d\n",i,A[i]);

```

TERMINAL

```

● abid@ubuntu:~/Parallel/Homework 4/hw8-code/$ make final
gcc -o table-implicit-notpar ./table-implicit-notpar
● abid@ubuntu:~/Parallel/Homework 4/hw8-code/$ make execute
./table-implicit-notpar
A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 2
A[4] = 4
A[5] = 4
A[6] = 4
A[7] = 4
A[8] = 7
A[9] = 7
A[10] = 9
A[11] = 10

```

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

Figure : Wrong Answer

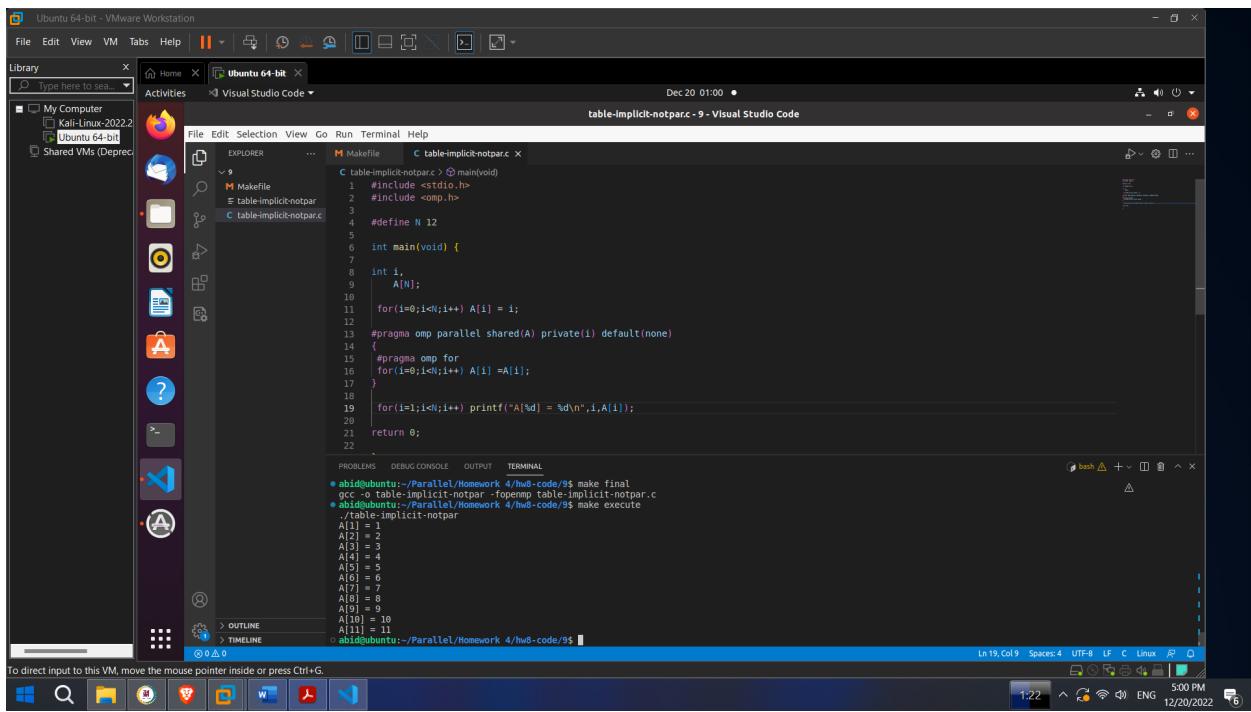


Figure : Correct Answer

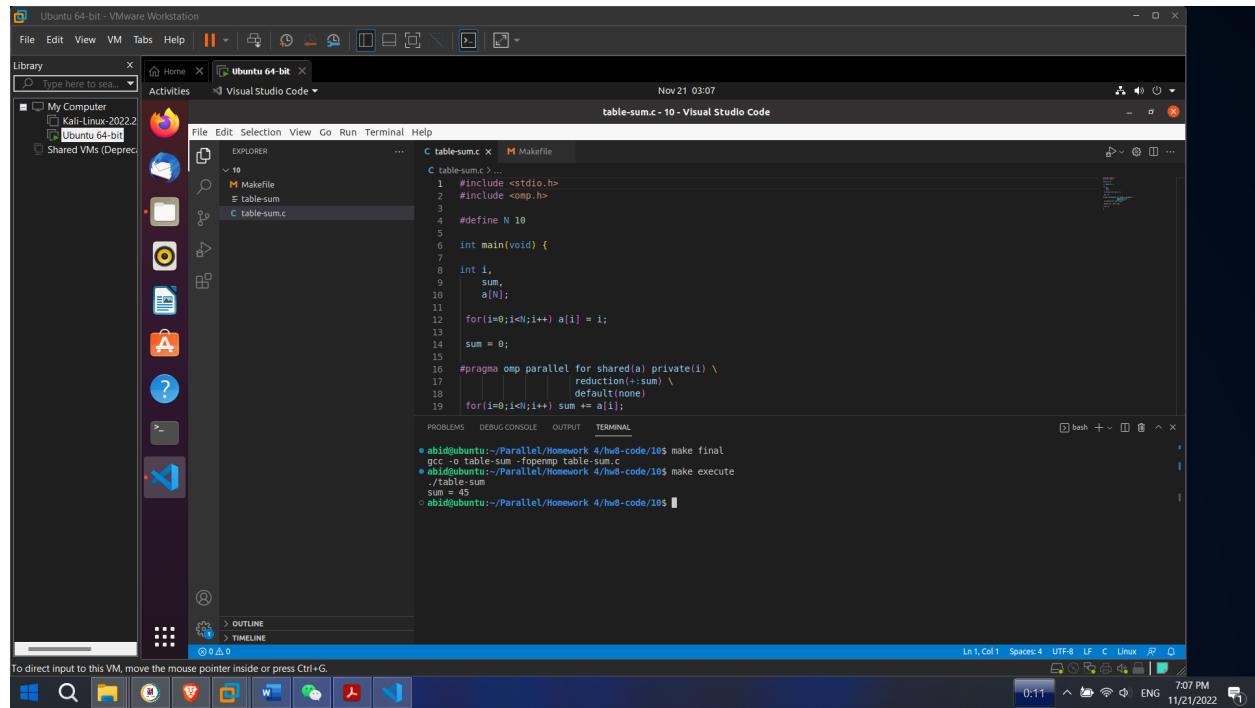
```
#pragma omp parallel shared(A) private(i) default(none)
{
#pragma omp for
for(i=0;i<N;i++) A[i] = A[i]; // Here index was reduced by 1 then we got correct value
}

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)

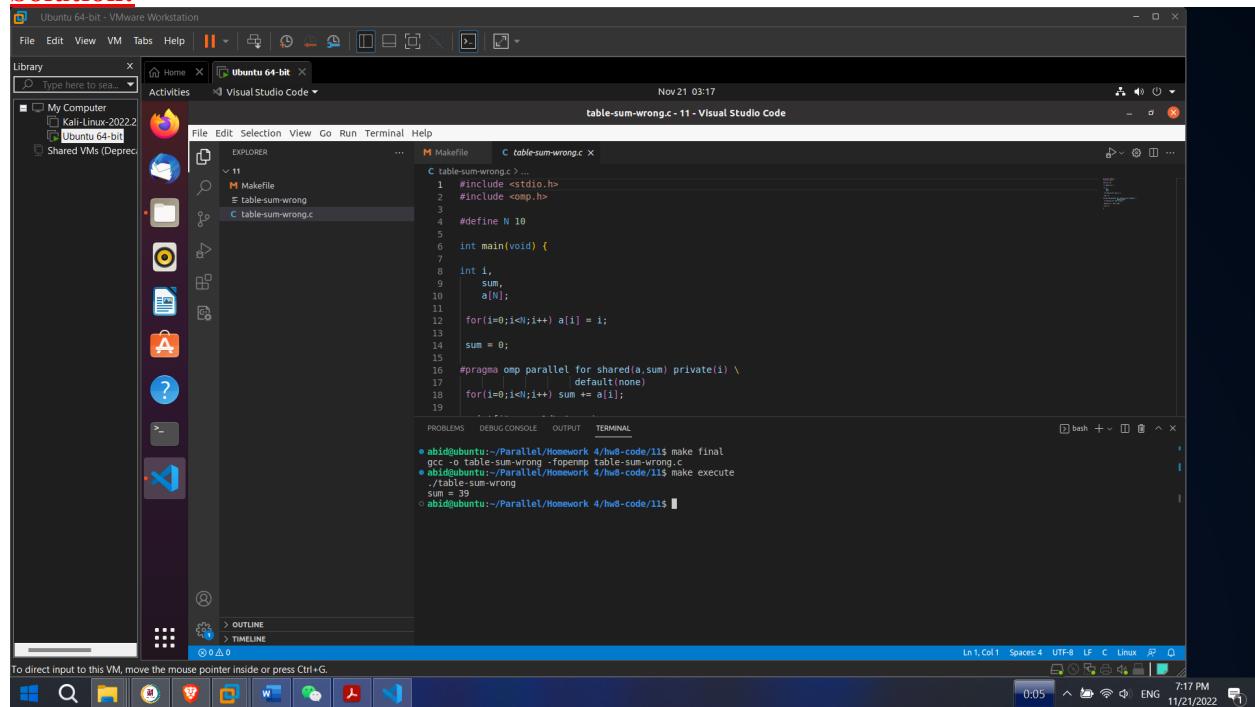
**Solution:**



```
#include <stdio.h>
#include <omp.h>
#define N 10
int main(void) {
    int i,
    sum,
    a[N];
    for(i=0;i<N;i++) a[i] = i;
    sum = 0;
#pragma omp parallel for shared(a) private(i) \
    reduction(+:sum) \
    default(none)
    for(i=0;i<N;i++) sum += a[i];
}
abid@ubuntu:~/Parallel/Homework 4/hw8-code/10$ make final
gcc -o table-sum -fopenmp table-sum.c
abid@ubuntu:~/Parallel/Homework 4/hw8-code/10$ make execute
./table-sum
sum = 45
abid@ubuntu:~/Parallel/Homework 4/hw8-code/10$
```

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

**Solution:**



```
#include <stdio.h>
#include <omp.h>
#define N 10
int main(void) {
    int i,
    sum,
    a[N];
    for(i=0;i<N;i++) a[i] = i;
    sum = 0;
#pragma omp parallel for shared(a,sum) private(i) \
    reduction(+:sum) \
    default(none)
    for(i=0;i<N;i++) sum += a[i];
}
abid@ubuntu:~/Parallel/Homework 4/hw8-code/11$ make final
gcc -o table-sum-wrong -fopenmp table-sum-wrong.c
abid@ubuntu:~/Parallel/Homework 4/hw8-code/11$ make execute
./table-sum-wrong
sum = 39
abid@ubuntu:~/Parallel/Homework 4/hw8-code/11$
```

**Figure : Wrong Answer**

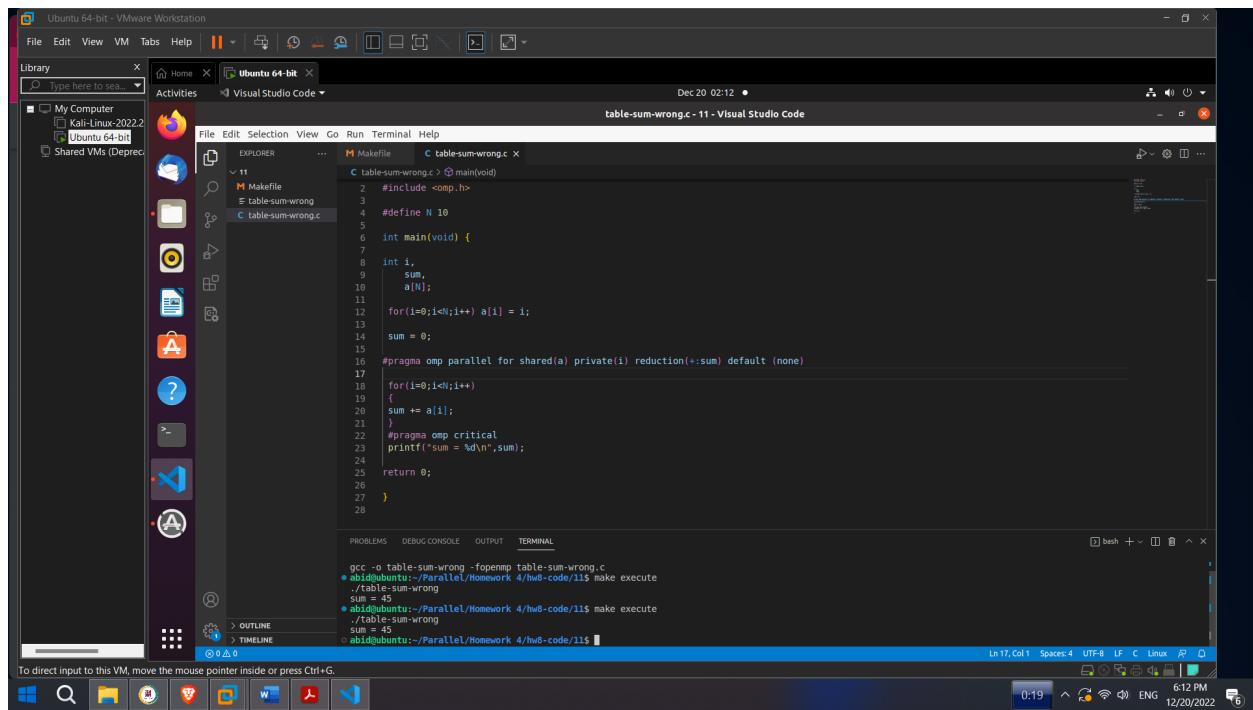


Figure : Correct Answer

```
#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;
```