Tek Acharya

ProjectA4

GSU Spring 2020

**Task 3a**

a) (68p) Foundation

- (15p) Race condition:

(2p) What is race condition?

* When a software, electronics, or other system depends on other event’s outputs as its inputs, a hazard can occur if timing or sequence do not happen properly. This is bug as the programmer has not intended.

(5p) Why race condition is difficult to reproduce and debug?

* Basically, race condition occurs in logic circuits, and multithreaded or distributed software programs. The race condition occurring in the logic circuit cannot be corrected through programming. And for bug due to relative timing between the intermediate threads in software, it is difficult to determine the end result as problems disappear in the production system when running in debug mode.

(8p) How can it be fixed? Provide an example from your Project\_A3 (see spmd2.c)

The spmd2.c program wasn’t working initially because, the race condition wasn’t handled in the program, however, later when we declare the variable to be private, the race condition got fixed. We noticed that this time four core of the raspberry pi got separate threads to handle without creating hazard.

The following code has a problem because the race condition has not been handled.

#include <stdio.h>

#include <omp.h>

#include <stdlib.h

int main(int argc, char\*\* argv) {

int id, numThreads;

printf("\n");

if (argc > 1) {

omp\_set\_num\_threads( atoi(argv[1]) );

}

#pragma omp parallel {

id = omp\_get\_thread\_num();

numThreads = omp\_get\_num\_threads();

printf("Hello from thread %d of %d\n", id, numThreads);

}

printf("\n");

return 0;

Once we declared the local variables in the above code, we give private arguments to the cores and the problem got fixed.

#include <stdio.h>

#include <omp.h>

#include <stdlib.h

int main(int argc, char\*\* argv) {

int id, numThreads;

printf("\n");

if (argc > 1) {

omp\_set\_num\_threads( atoi(argv[1]) );

}

#pragma omp parallel {

int id = omp\_get\_thread\_num();

int numThreads = omp\_get\_num\_threads();

printf("Hello from thread %d of %d\n", id, numThreads);

}

printf("\n");

return 0;

- (15p) Summaries the Parallel Programming Patterns section in the “Introduction to Parallel

Computing\_3.pdf” (two pages) in your own words (one paragraph, no more than 150

words).

As the developers come along their practice of parallel programming, they categorized the pattern found into two main areas

1. **Strategies**
2. **Concurrent Execution Mechanism** (CEM)

As we write a program, we must consider a strategy that can cover the following points

1. What kind of **algorithmic strategy** to use?
2. Considering a strategy, what **implementation strategy** to apply.

The algorithmic strategy primarily deals with choosing the type of task that can be run concurrently by multiple processors. The parallel program often runs with several of this pattern at a time and contributes to an overall structure while some other deal with the data that is used to compute.

The CEM has two major categories namely

1. **Process/Thread Control Pattern** which heavily relies on how the process of parallel execution is controlled during runtime.
2. **Coordination Pattern** which conveys how these controls are coordinated during runtime.

- (12p) In the section “Categorizing Patterns” in the “Introduction to Parallel

Computing\_3.pdf” compare the following:

o Collective synchronization (barrier) with Collective communication (reduction)

|  |  |
| --- | --- |
| *Collective Communication (reduction)* | *Collective Synchronization (reduction)* |
| All processes reach to a specific location before executing. | All processes are blocked until a specific synchronization point is reached |
| One process of the communicator collects data from all other processes and perform an operation to find the result | It blocks the processes until all other processes are being reached to synchronization point successfully. |
| It uses MPI reduce() function | It uses MPI barrier() function |
| This a type of parallel application of computing | This is a type of parallel application of computing |
| This uses concurrent execution mechanism for parallel execution | This uses concurrent execution mechanism for parallel execution |
| This is a type of coordination system of parallel computation |  |

o Master-worker with fork join

|  |  |
| --- | --- |
| Master-worker | Fork join |
| Main process is being divided into small chunks which in turn being distributed to several worker processes. | Pattern which is used to execute parallel light weight processes and threads. |
| The master handles one of the threads while rest is handled by the workers | Threads from a sequential mode is forked into several slave threads and are joined back once tasks for each is completed resulting into joined output. |

- (26p) Dependency: Using your own words and explanation, answer the following:

(3p) Where can we find parallelism in programming?

* Parallelism is found almost every modern computational machine. As parallelism yields the output faster yet accurate as in mobiles, server machines, virtual reality, database etc.

(6p) What is dependency and what are its types (provide one example for each)?

* When an input of one execution depends on the output or resource of another, such a scenario is called dependency in computing.
* There are three main categories

1. True Dependence

S1: x = 1

S2: y = x This can be avoided by directly passing a value of x to y i.e. y = 1

Here, the value of second is dependent on the value of first.

1. Output Dependence

S1: x = f(a)

S2: y = x This cannot be avoided as the value of x is to be evaluated before any

Value of y can be assigned

Here, the value of second is dependent on first

1. Anti-Dependence

S1: x =y

S2: y =1 This cannot be avoided as the value of first can only be accessed only once

Second is accessed.

Here, the value of first variable is dependent on the value of second variable

(3p) When a statement is dependent and when it is independent (Provide two

examples)?

* When the execution of

1. statement1;

statement2;

1. statement2;

statement1; if pattern A and B yields same result

then the two statement1 and statement2 are independent of each other. And if the they yield different result; they are dependent on each other.

For example,

S1: x = 1 | S2: y = 2

S2: y = x | S1: x =1

In the above example if the statement S1 and S2 are handled by two cores separately, the order does not matter and has the same output.

But if the same statements are handled by a single core, the order matters and the dependency occur.

(3p) When can two statements be executed in parallel?

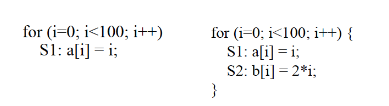
* If there are no dependency between two or more statements in a cycle, then the parallel execution can be applied.

(3p) How can dependency be removed?

* Some dependencies can be resolved by rearranging and eliminating the statements before execution.

(8p) How do we compute dependency for the following two loops and what type/s of

dependency?



We compare the statement sets of IN and OUT at each node to see the data dependence relations.

The statement S in a set of IN and OUT is defined as follows:

IN(S) = the S uses the set of memory location.

OUT(S) = The S modifies the set of memory location.

We use two basic strategies to find the dependencies of a statement, these include

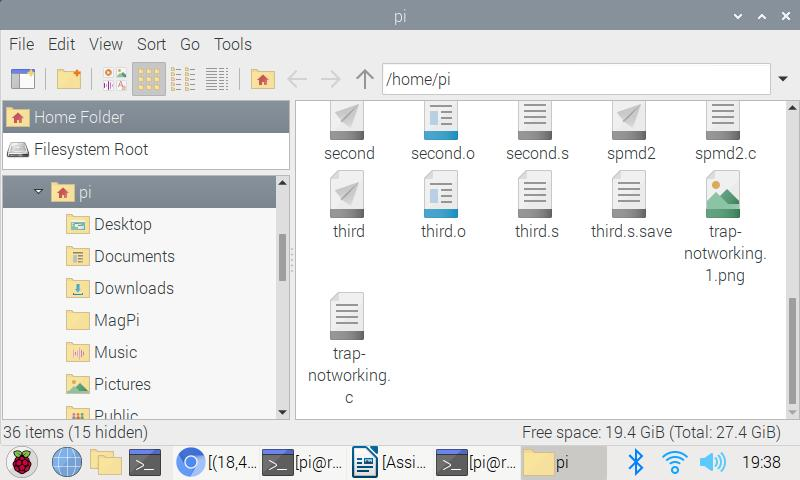
1. Unrolling of the loop (if any) into separate statements or iterations
2. Reveal or study the relationship between the statements

In our first loop above, S1(1), S1(2), ……, S1(99) are independent of each other.

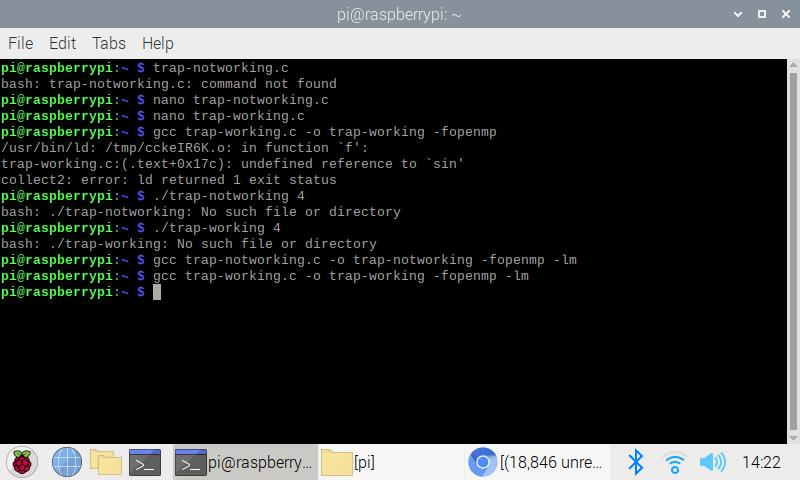
Whereas, in the second loop, S1(1), S1(2), …., S1(99) are independent of each other but statement S2(1), S2(2), …., S2(99) depends on S1 as we can see this after unrolling the above statements.

**TASK 3b**

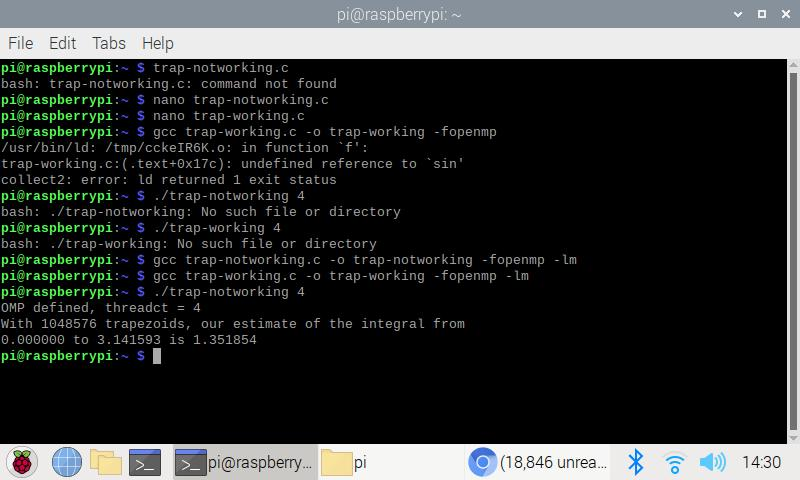
Copied and pasted the code, and created an executable as bellow



The executable created shows that there was no programming error.

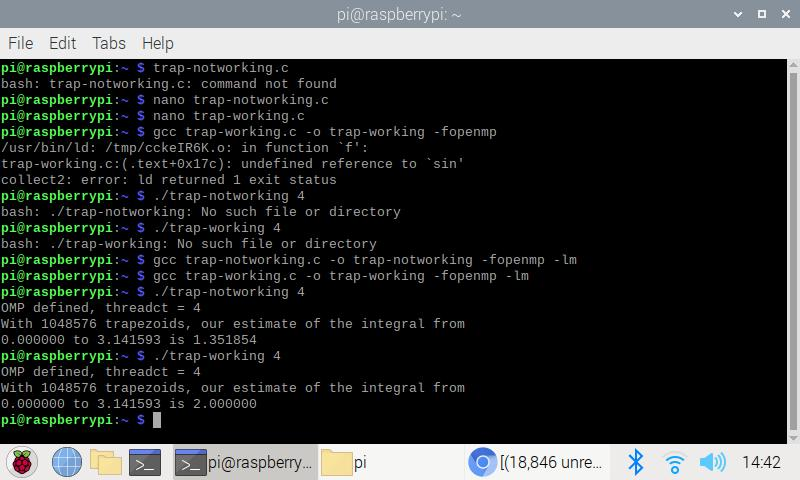


Trap-notworking successfully compiled and ran



The result is not correct as we can see from the above value of our integral (2).

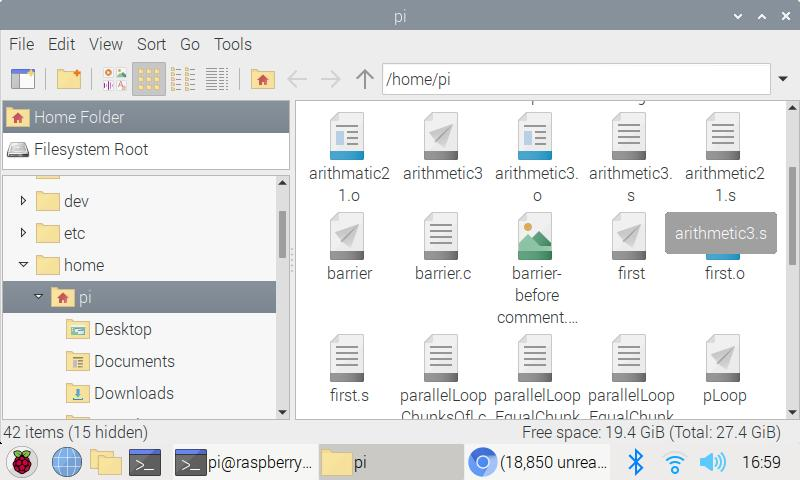
Then, compiled and ran with trap-working version and got the correct expected value of our integral as shown below.



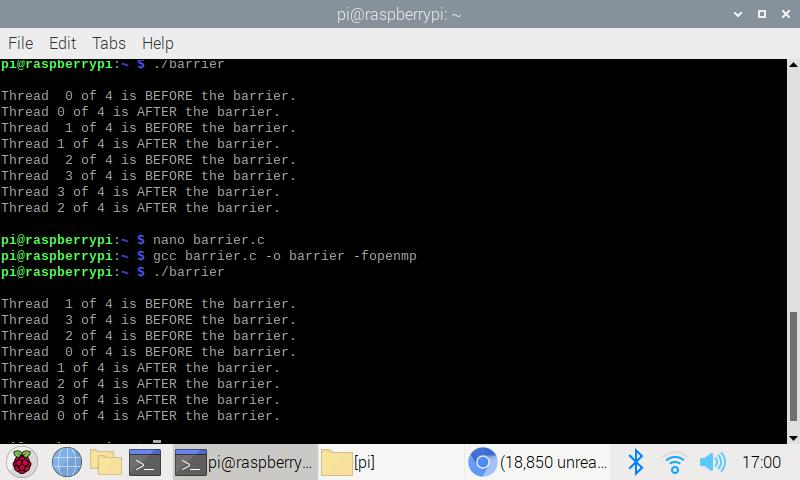
The reason behind the trap-notworking version giving wrong answer is due to race condition. The program is trying to read a variable that is changed by different threads. Using \ at the end of #pragma will save the last iteration data for use outside the #pragma clause. Here, the threads are not independent of each other.

**Barrier**

**(without barrier condition)**

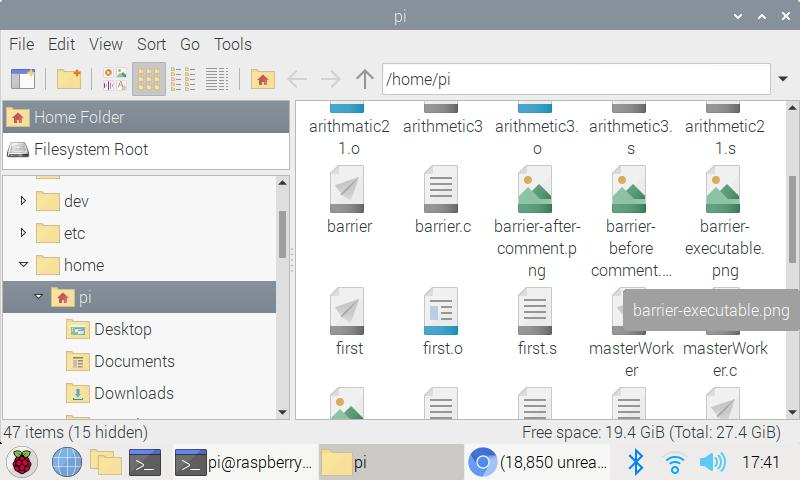


Executable created for barrier.c



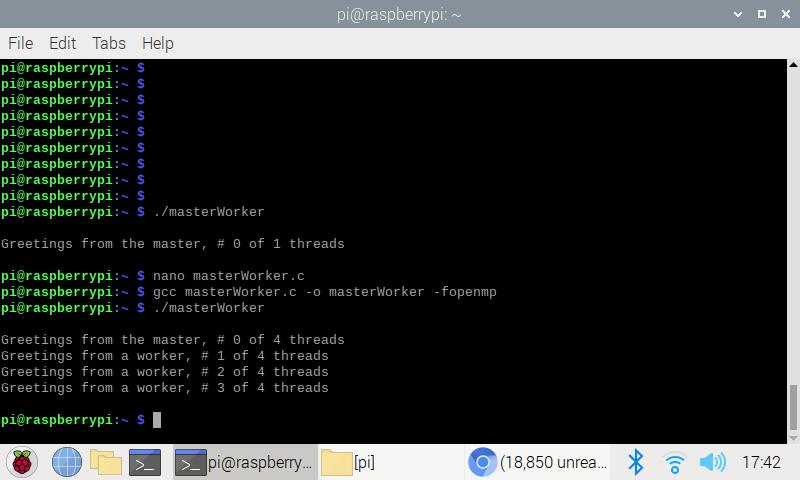
The difference between with and without barrier is that the one without barrier, each thread completes their tasks whereas with the barrier, all threads recombined at a point and complete their tasks only upon all tasks has been completed by each thread. Here, threads will wait until all threads meet at a point and then proceeds further execution.

**MasterWorker**



masterWorker executable created

Worker-master without pragma and with pragma



In the one with commented pragmas, no threads are being used, so just sequential computation.

But once the comment is lifted, the threads has been activated to four different worker-threads and one still handled by the master thread.

This is an example of masterWorker computation.