Group-5

Md Tahseen Anam (19941202-T678)

Nafi Uz Zaman (19930919-T550)

Exercise 1: Concurrency and Non-Determinism

Our observations were as follows:

- We could not determine which thread would first start running, we could not determine in which order threads would execute and terminate.
- Most of the time, threads were executing and terminating sequentially. For example:

Output:

```
task 1 is running

task 1 is terminating

task 3 is running

task 3 is terminating

task 7 is running

task 7 is terminating
```

However, there were some cases where one thread started and did not terminate before other threads started and then terminated. For example:

```
task 1 is running

task 3 is running

task 3 is terminating

task 1 is terminating
```

This happens because when a thread acquires the first mutex lock and releases it, there is no guarantee that it will acquire the second mutex lock and print that it is terminating. It can be the case that before the thread acquires the second mutex lock some other thread acquires the first mutex lock and prints that it is running, then that other thread acquires the second mutex lock and prints that it is terminating and after that the first thread acquires the second mutex lock and terminates.

- Threads are not being called in order as shown in the previous point. Task 3 can be running before task 7 and vice versa.
- Main function will not finish it's execution until all the threads finish their execution because `thread.join()` function is called on all of the threads that are initiated.

What we did not observe:

- Threads did not run and then terminate in chronological order.
- We did not observe all threads start running one after another and then terminating one after the other.

Exercise 2: Shared-Memory Concurrency

Observed output:

- The function **`dec()`** (Decrement) is being called more frequently than the other functions.
- Only one thread acquires the lock and executes. Other threads need to wait inside their **`while`** loop to acquire the lock until that thread releases the lock.
- All 3 threads keep running until **`run=false`.**
- Other threads keep running even after the main thread is paused.
- The longer the main thread is paused, the more time the other threads get to execute.
- When main thread acquires the lock, it sets **`run=false`.** When **`run=false`**, all 3 threads execute their tasks exactly one more time before the program terminates.

Exercise 3: Race Conditions vs. Data Races

A **race condition** occurs when the result of a concurrent program depends on the timing of its execution (i.e., different tasks race to perform some operations or to access a shared resource).

A **data race** occurs when two (or more) tasks attempt to access the same shared memory location. At least one of the accesses is a write, and the accesses may happen simultaneously.

The program **`non-determinism.cpp`** does not have a data race however, it does have a race condition. Justification:

There are no shared variables in this program therefore a data race cannot occur. When the program is run, a thread executes the code where it acquires a lock and then prints something to the screen. But when it lets go of the lock, and tries to acquire it again, another thread may have acquired the lock and prints the first statement, subsequently those 2 threads could be racing to acquire the second lock, but only one does and this is where the race condition occurs.

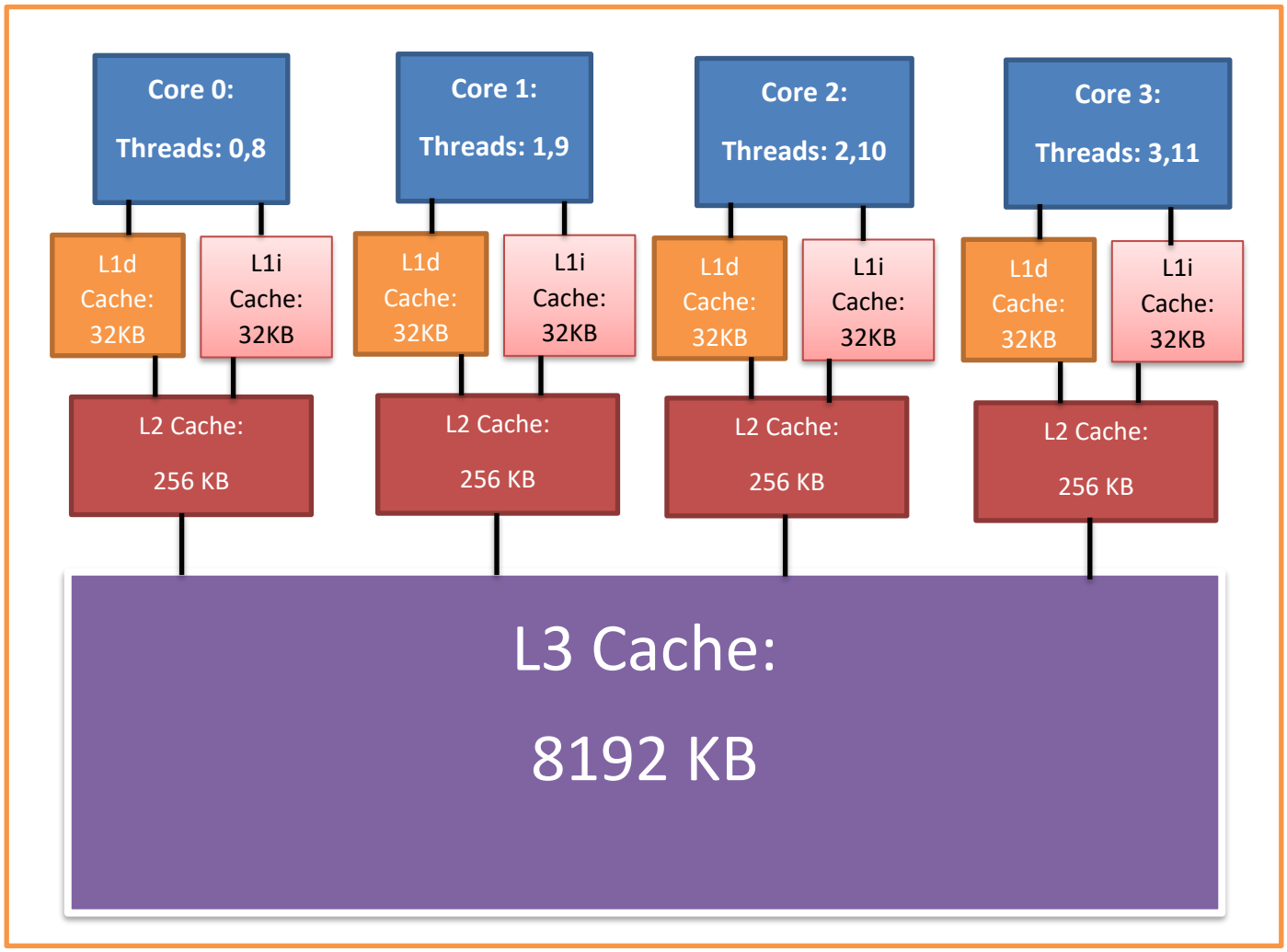
The program **`shared-variable.cpp`** does not have a data race nor does it have a race condition. Justification:

While there are shared variables in this program, none of them cause a data race as each thread must needs to acquire the lock first to work on the shared variable and when one thread is working on the shared variable, other threads can not operate on that memory location. Again

each thread is assigned a completely different task (function) to work on. This is the reason why a race condition does not occur as it is not the case that 2 or more threads are racing to execute the same particular task.

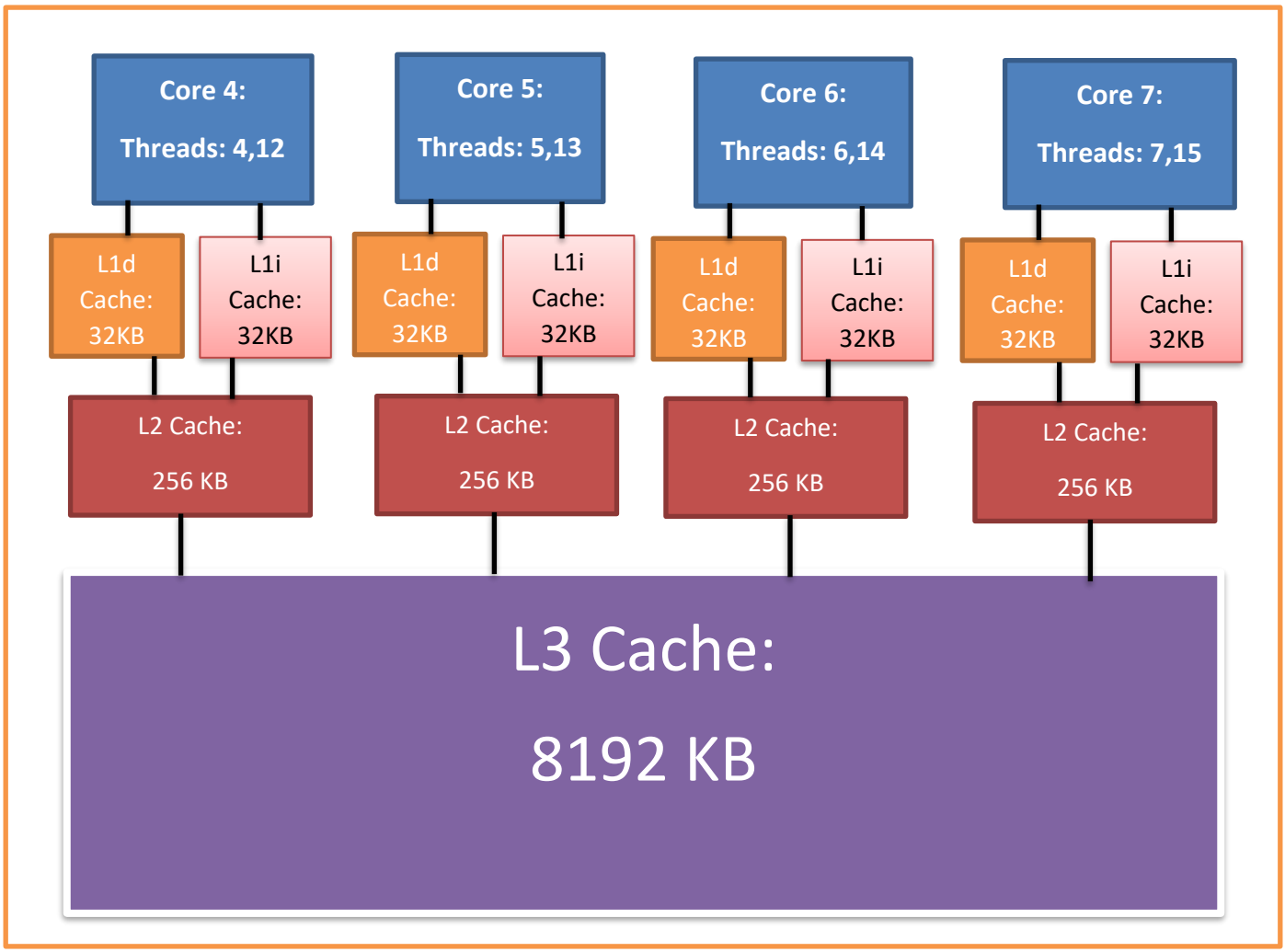
Exercise 4: Multicore Architectures

Socket-1



Exercise 4: Multicore Architectures (continued)

Socket-2



Exercise 5: Performance Measurements

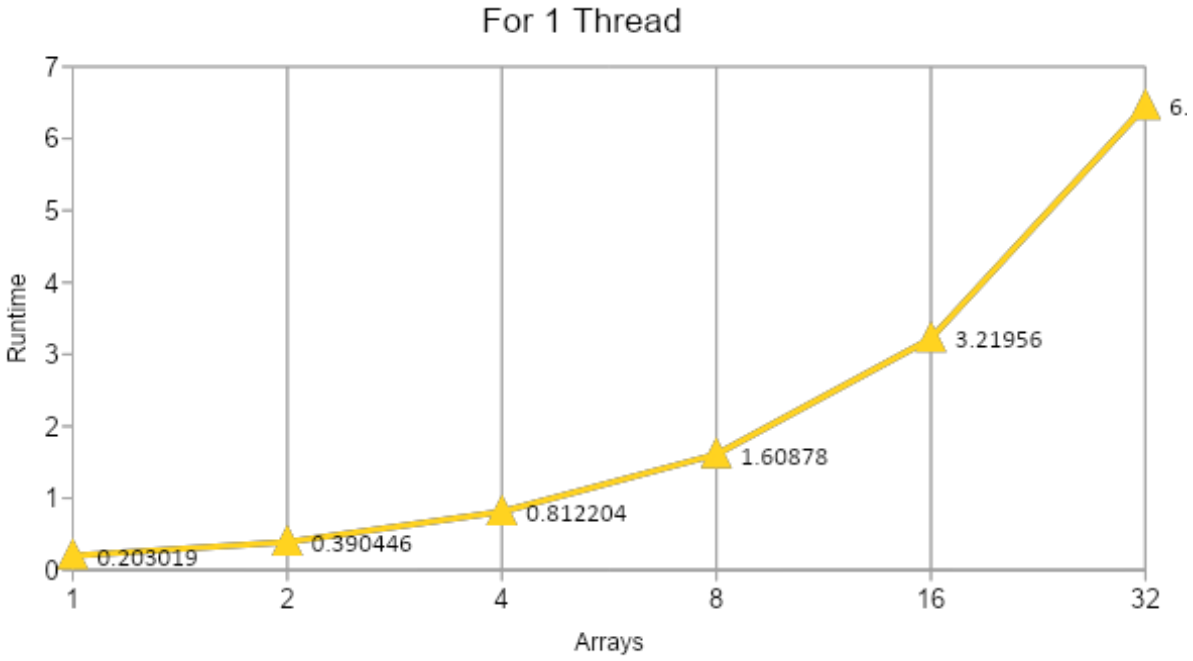
Part-a

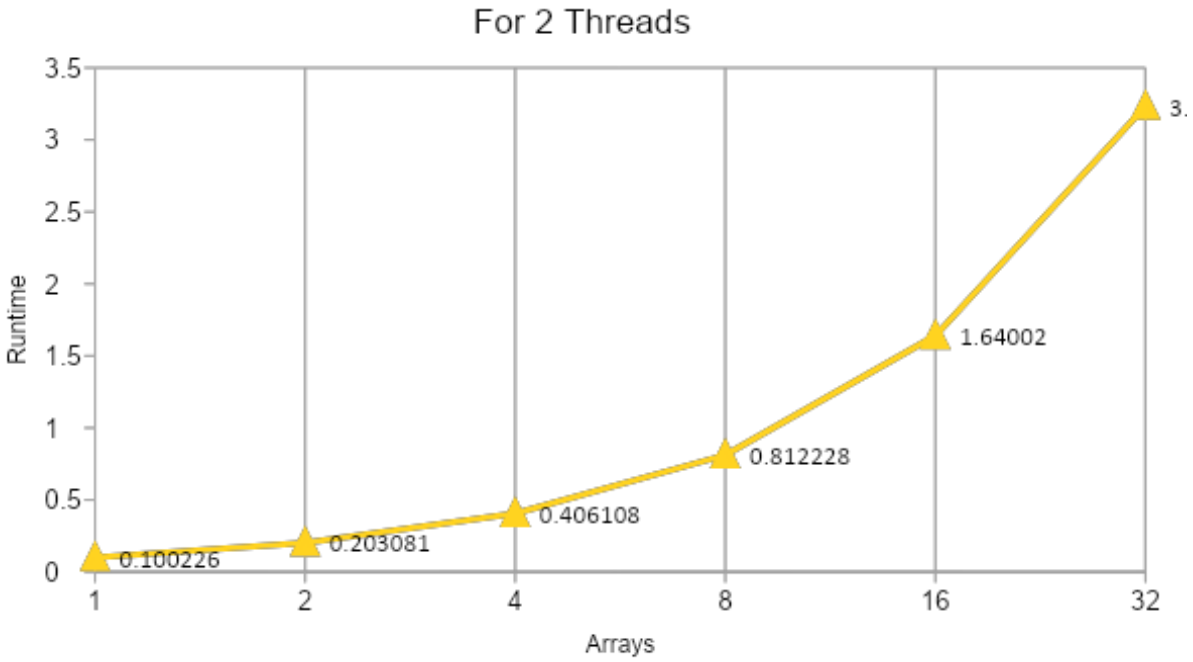
We have run the program for each combination of T ∈ 1, 2, 4, 8, 16, 32 and N ∈ 1, 2, 4, 8, 16, 32. The run times that we have observed are given below:

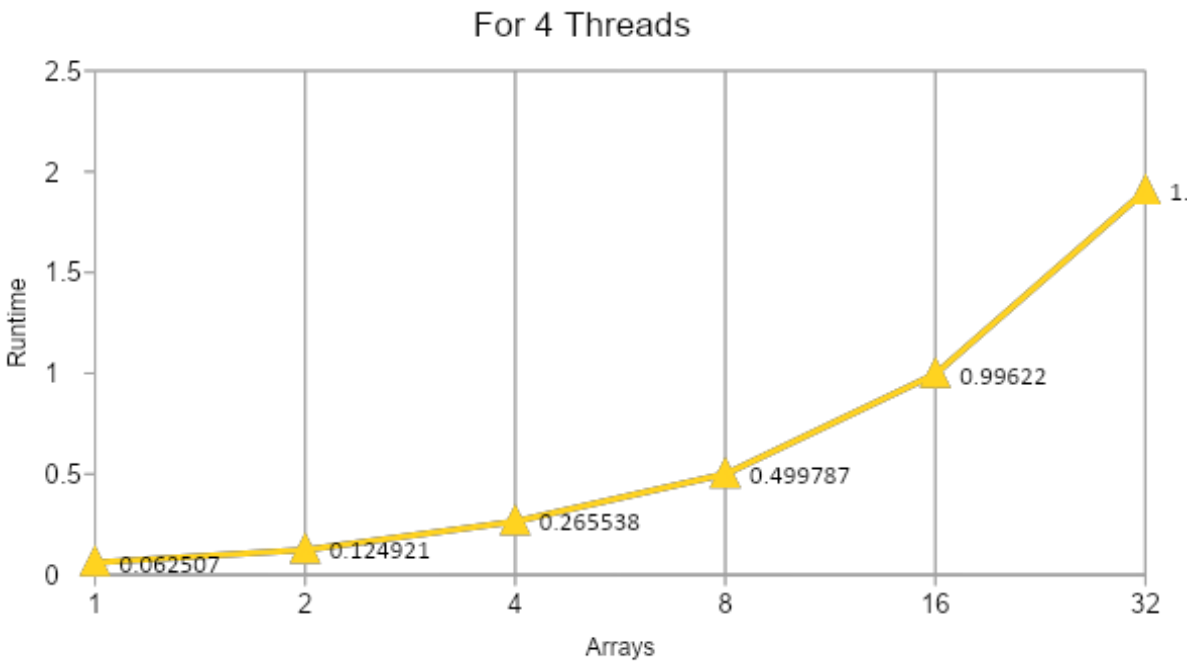
Threads (T)	Arrays (N)	Run Time
1	1	0.203019
1	2	0.390446
1	4	0.812204
1	8	1.60878
1	16	3.21956
1	32	6.45588
2	1	0.100226
2	2	0.203081
2	4	0.406108
2	8	0.812228
2	16	1.64002
2	32	3.23319
4	1	0.062507
4	2	0.124921
4	4	0.265538
4	8	0.499787
4	16	0.99622

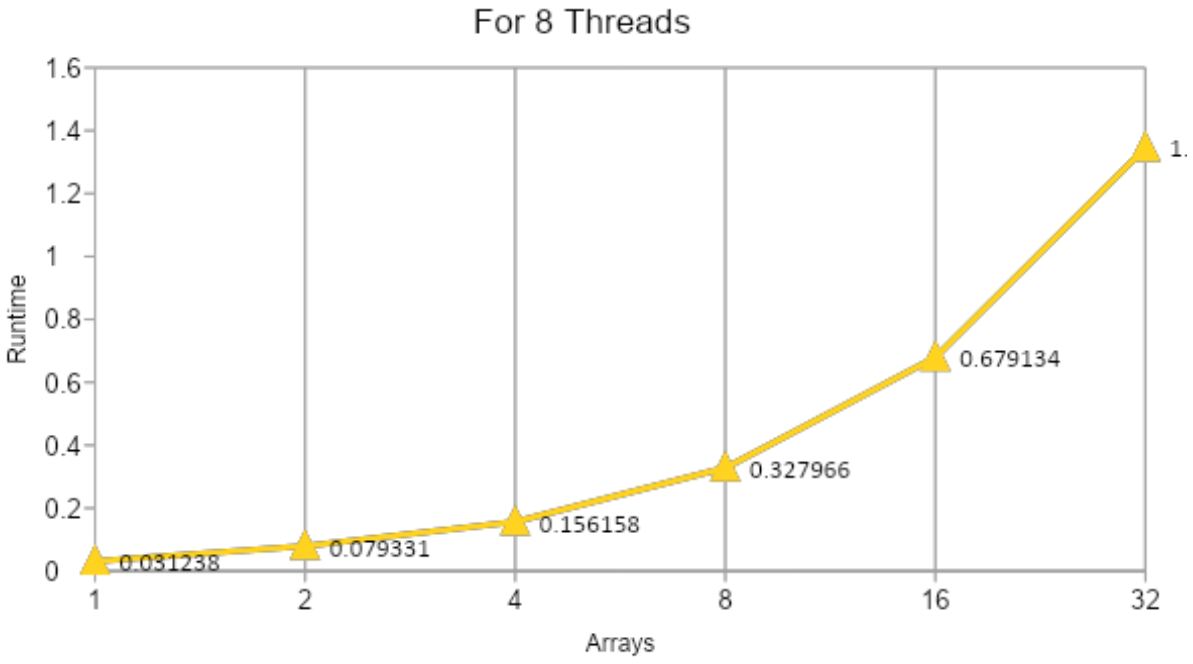
4	32	1.90554
8	1	0.031238
8	2	0.079331
8	4	0.156158
8	8	0.327966
8	16	0.679134
8	32	1.34568
16	1	0.046859
16	2	0.093721
16	4	0.187449
16	8	0.359301
16	16	0.702949
16	32	1.35897
32	1	0.04686
32	2	0.089751
32	4	0.189509
32	8	0.343613
32	16	0.687296
32	32	1.35895

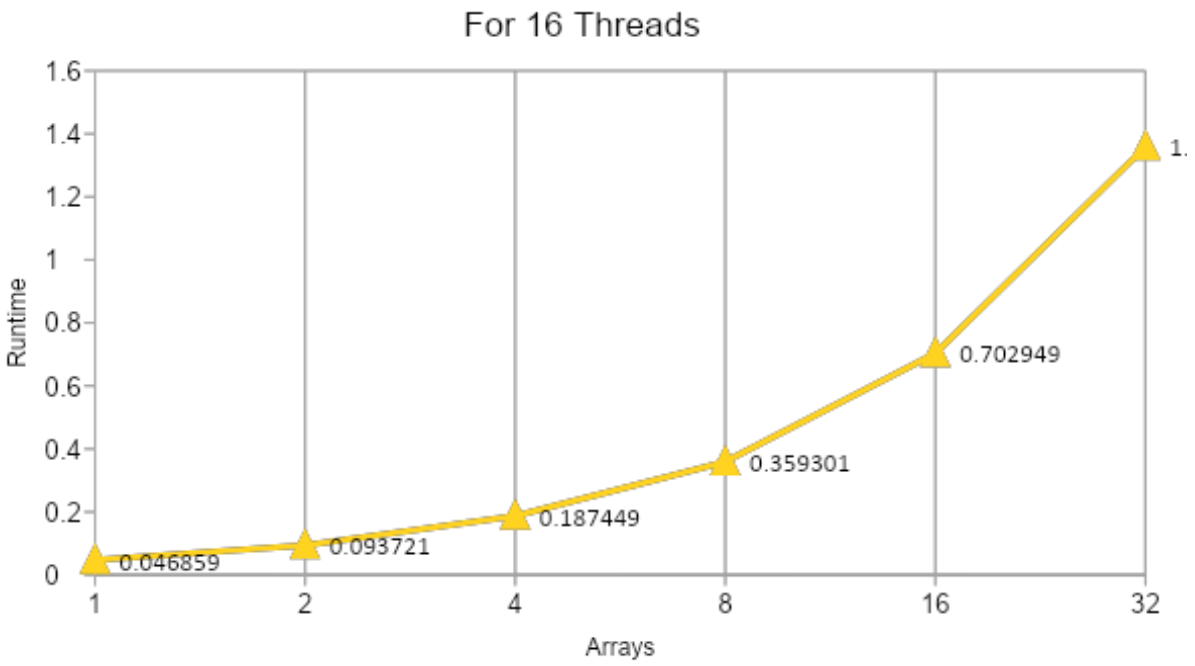
Part-b

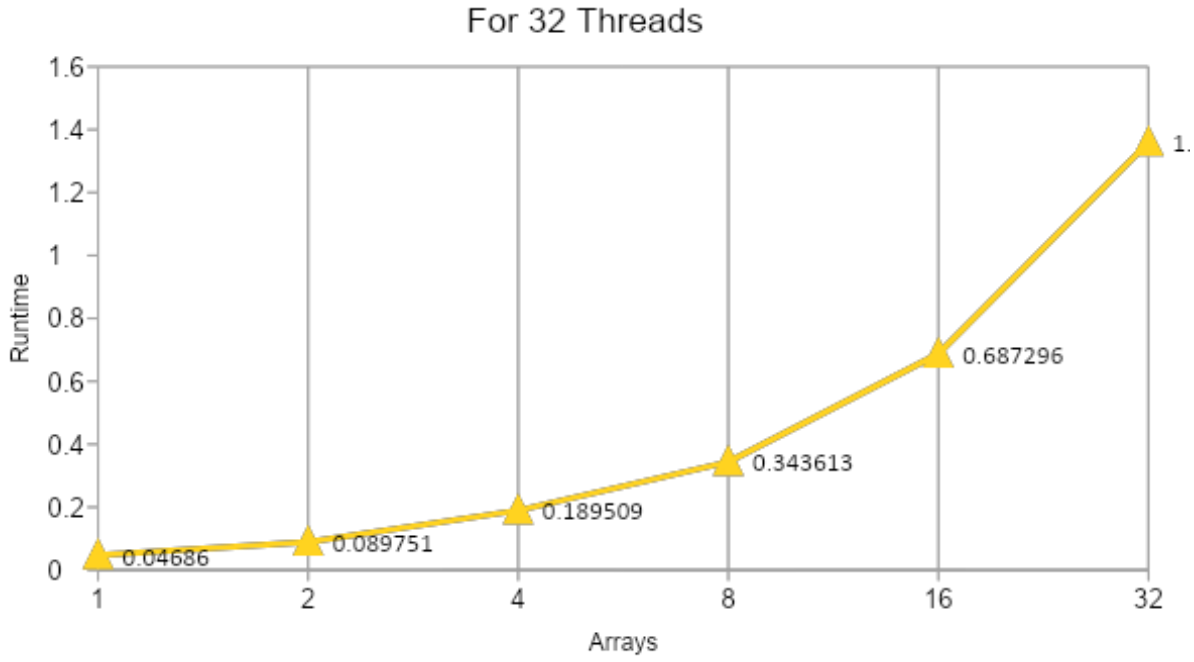

For 1 Thread
Runtime
Arrays
0.203019
0.390446
0.812204
1.60878
3.21956
6.
0
1
2
3
4
5
6
7
1
2
4
8
16
32


For 2 Threads
Runtime
Arrays
0.100226
0.203081
0.406108
0.812228
1.64002
3.
0
0.5
1
1.5
2
2.5
3
3.5
1
2
4
8
16
32


For 4 Threads
Runtime
Arrays
2.5
2
1.5
1
0.5
0
1
2
4
8
16
32
0.062507
0.124921
0.265538
0.499787
0.99622
1.


For 8 Threads
Runtime
Arrays
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
4
8
16
32
0.031238
0.079331
0.156158
0.327966
0.679134
1.


For 16 Threads
Runtime
Arrays
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
4
8
16
32
0.046859
0.093721
0.187449
0.359301
0.702949
1.


For 32 Threads
Runtime
Arrays
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
4
8
16
32
0.04686
0.089751
0.189509
0.343613
0.687296
1.

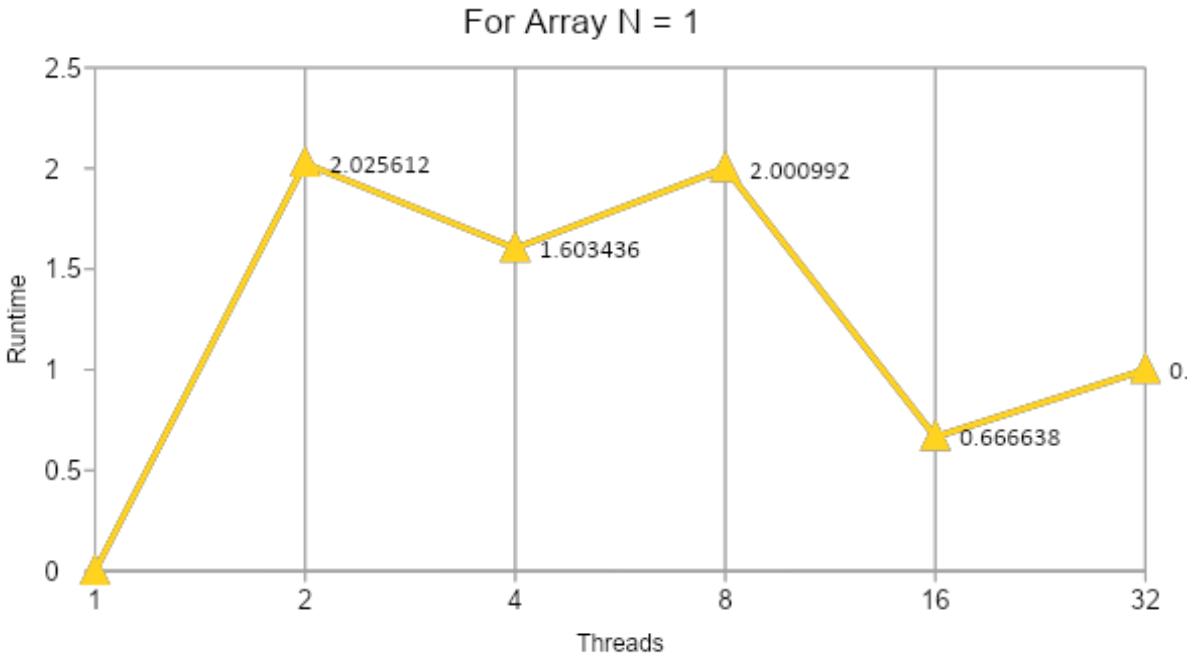
Part-c

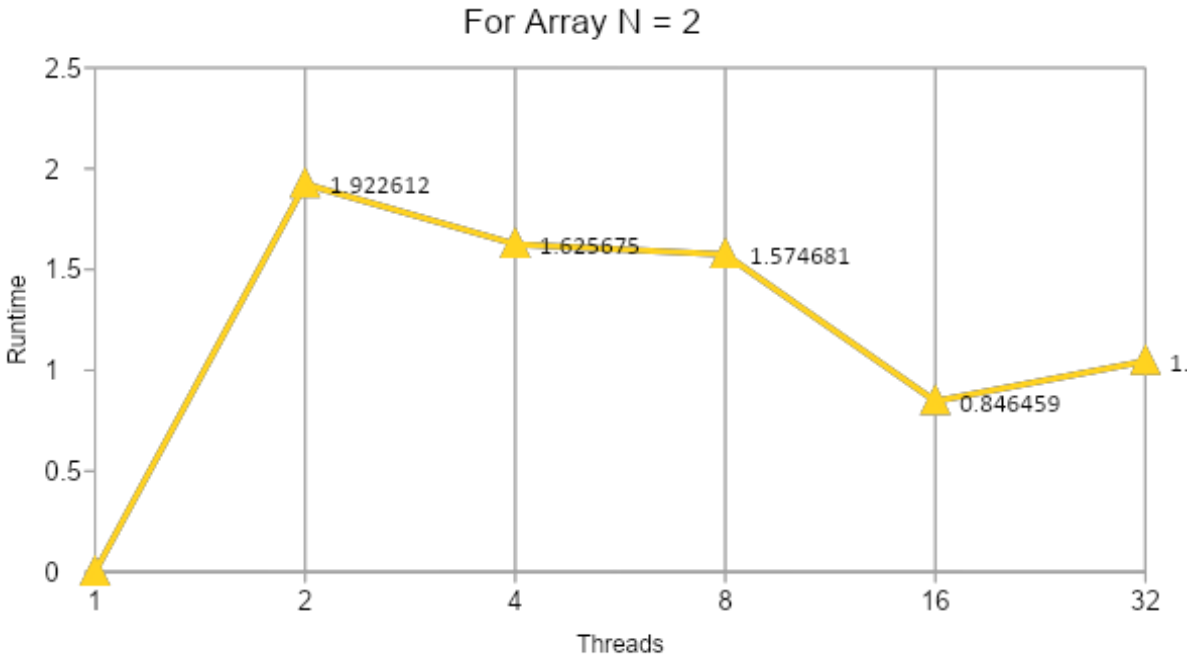
From part b, we can observe that for T independent threads, as we increase array N, run time increases. This happens because each independent thread performs some work on a separate part of the array, of size (approximately) N/T MB. So, as the value of N increases, N/T also increases which results in the increment of run time of the program.

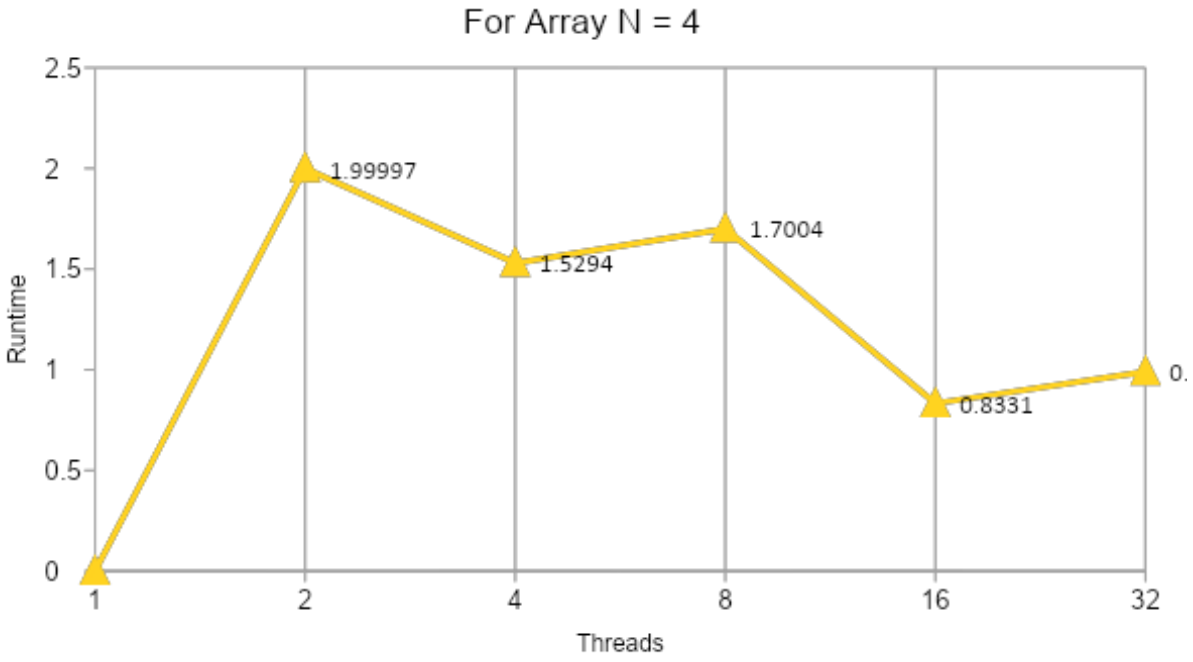
We know that, given the old execution time T_{old} and the new execution time T_{new} for a program, the speedup is:

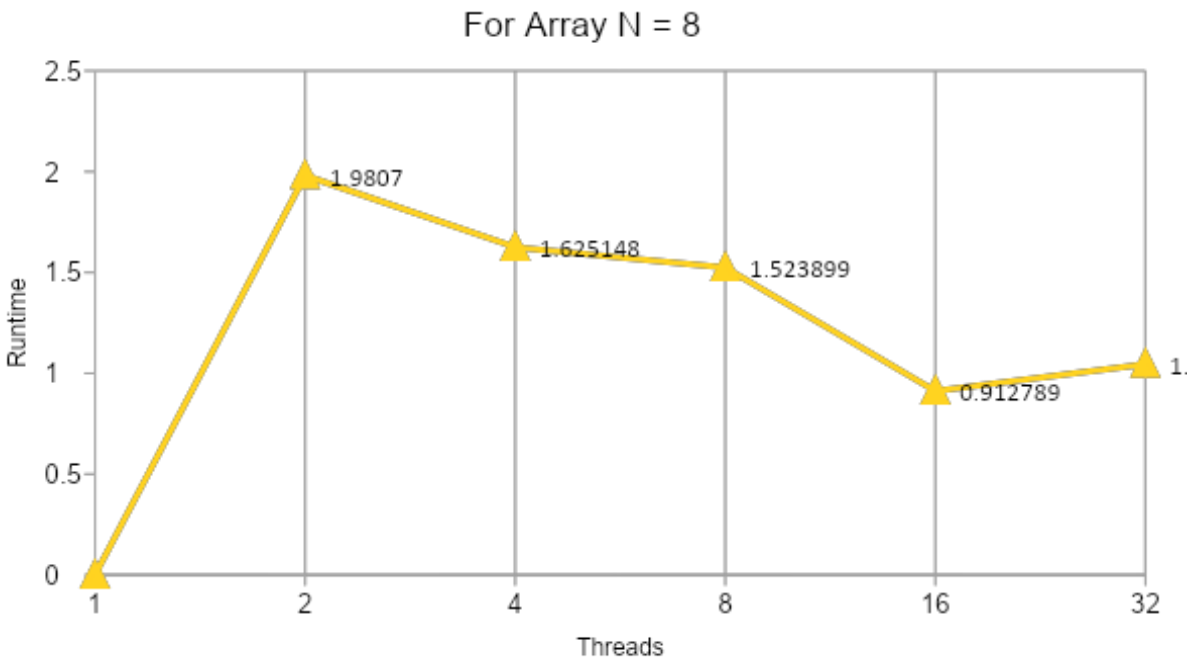
$$S = T_{old} / T_{new}$$

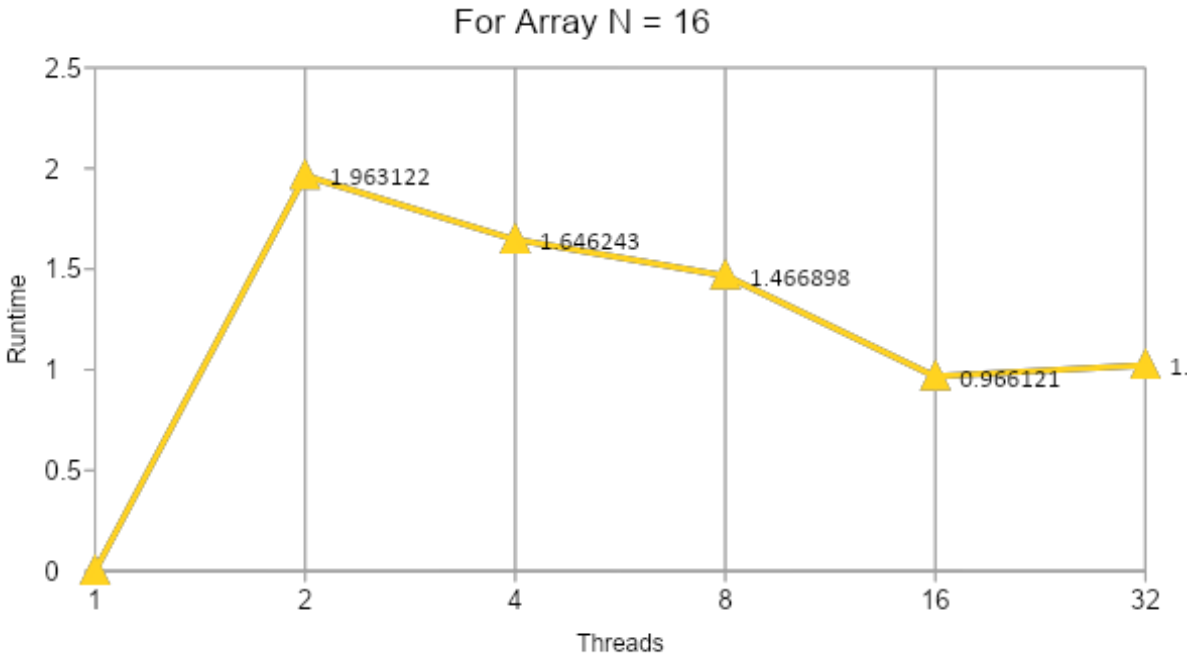
Let us now observe the speed up for different values of N:

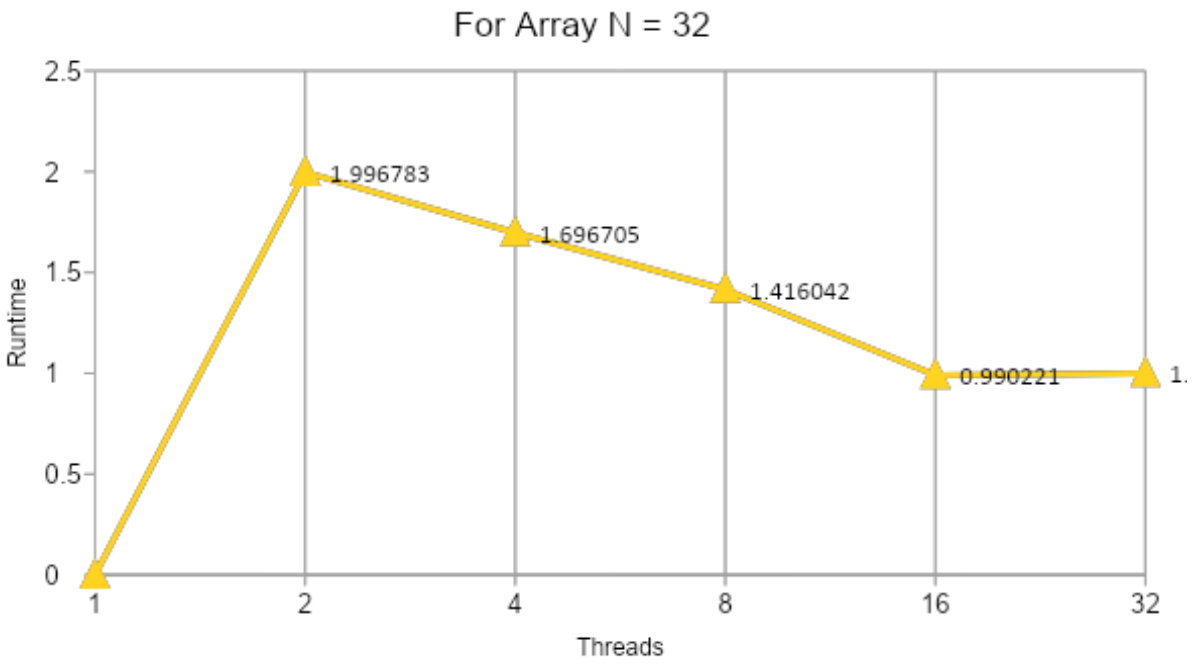



For Array N = 2
2.5
2
1.5
1
0.5
0
Runtime
1.922612
1.625675
1.574681
0.846459
1
2
4
8
16
32
Threads


For Array N = 4
2.5
2
1.5
1
0.5
0
Runtime
1.99997
1.5294
1.7004
0.8331
1
2
4
8
16
32
Threads


For Array N = 8
2.5
2
1.5
1
0.5
0
Runtime
1.9807
1.625148
1.523899
0.912789
1
2
4
8
16
32
Threads


For Array N = 16
2.5
2
1.5
1
0.5
0
Runtime
1.963122
1.646243
1.466898
0.966121
1
2
4
8
16
32
Threads



From this observation, it is also noticeable that we did not observe any super linear speed up of run time.

Parallel slowdown occurs when parallelization beyond a certain point causes the program to run slower. From the above observation of speed up, we can conclude that parallel slowdown occurs when we take more than 2 threads.

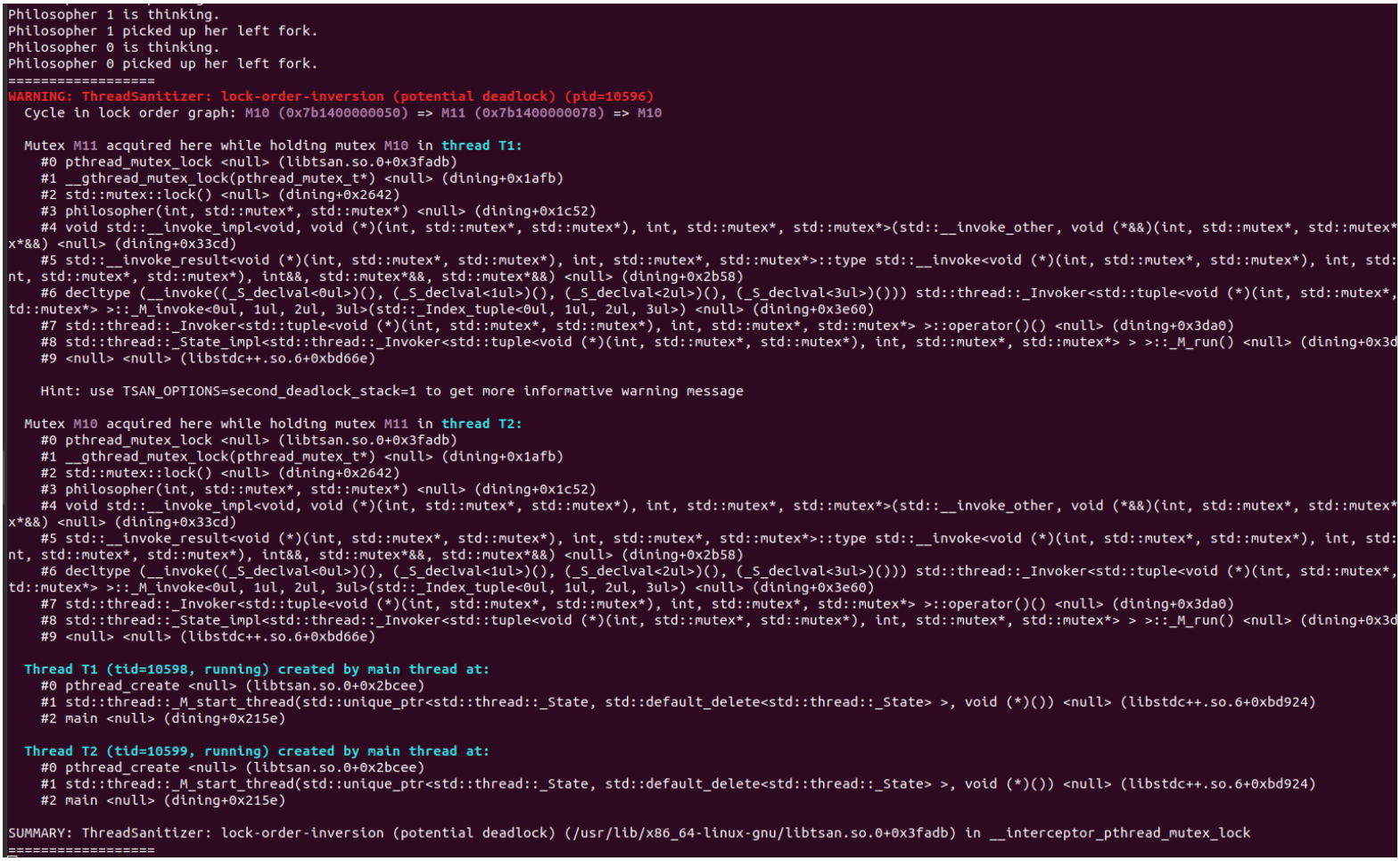
Exercise 6: Dining Philosophers

Part-a

We know that a deadlock occurs in a program when two or more tasks are waiting for each other.

In this Dining Philosophers exercise, most of the time we observed deadlocks. For example, when we took 2 philosophers, we saw that after some time, both philosophers picked up their left forks. So, both philosophers were waiting to acquire the right lock in our program. Which means, each philosopher was waiting to pick up his right fork, but as one's right fork is another philosopher's left fork sitting next to him, the program went to a deadlock situation. We observed this kind of deadlock not only for 2 philosophers but also for up to 10 philosophers. Each time, all philosophers picked up their left forks and started waiting to pick up their right forks and deadlock occurred.

We have examined the problem with Thread Sanitizer tool. Below is a screen shot of what we have found for 2 philosophers.



Picture: Examining deadlock in Dining Philosophers problem for 2 philosophers using Thread Sanitizer.

In the above image, we can see that thread T1 has acquired Mutex M11 and holding mutex M10 which is acquired by thread T2 and thread T2 is holding mutex M11 which is acquired by thread T1. This situation caused a deadlock to occur.

Part-b

To overcome the problem that we have observed in part a, we have come up with a solution. In this solution, we have taken two global variables named **'leftcounter'** and **'total'** . Variable 'total' is assigned to the value of the total number of philosophers. We increment 'leftcounter' variable each time a philosopher picks up his left fork and decrement the value of 'leftcounter' whenever a philosopher puts down his left fork. When the value of 'leftcounter' is equal to

‘total’ or more than ‘total’, no philosopher can pick up his left fork. In this way, we are ensuring that if there are n philosophers then n left forks do not get picked up and at least one philosopher’s right fork remains available to pick up and eat. This prevents the deadlock that we have discussed in part a. In our solution, we have also introduced another boolean variable **‘pickedupleftfork’** to ensure that a philosopher can pick up his right fork only after he has picked up his left fork.

To observe the output of our solution, compile the program **dining.cpp** with: **g++ -std=c++11 -Wall -pthread -fsanitize=thread** and Run the program several times for different values of N ($2 \leq N \leq 10$) .