Assignment 1: Basic Concepts

Introduction to Parallel Programming

Venkata Sriram Krishna Prasanth Kondapi

Pranav Vijayachandran Nair

Exercise 1

The program *non-determinism.cpp* executed using *'makeNonDeterminism'* makefile.

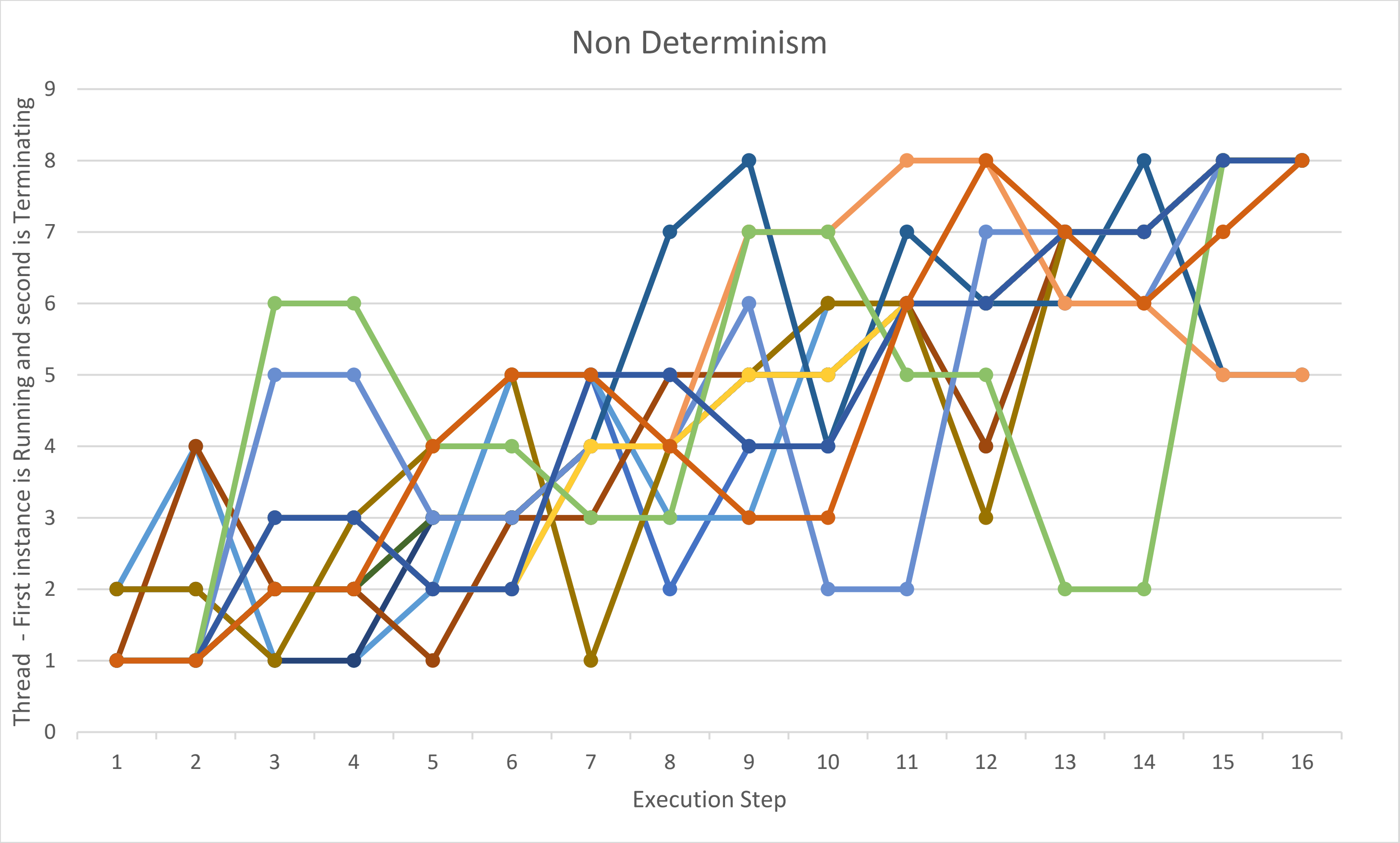

Figure 1: Non Determinism

We expect truly non-deterministic behaviour, but that is not observed in most cases. Ideally, we anticipate a different thread to catch control of the lock after the print "running" task of one thread, but the same thread tends to immediately terminate after running without yielding control to other threads. Percentages of the same thread immediately terminating after running: 89.3%. Out of the 20 iterations, control switches between threads are observed only 7 times. In those 7 iterations, it is mostly one or two threads which lost control before termination. Maximum number of observed control switches in a single iteration is 4 and that happened only twice.

They also tend to be more random at the middle of the program than at the beginning and the end.

Exercise 2

The program *shared-variable.cpp* executed using *'makeShareVariable'* makefile..

Similar to the first exercise, there is a race condition between the incrementing, decrementing and printing tasks. If the executions are truly non-deterministic, the value should remain steady with minor fluctuations as the incrementing and decrementing threads balance out each other. Out of 17 observations, this non-deterministic output is observed most of the time with some sudden changes in value. These sudden changes indicate that a thread held on to the control longer.

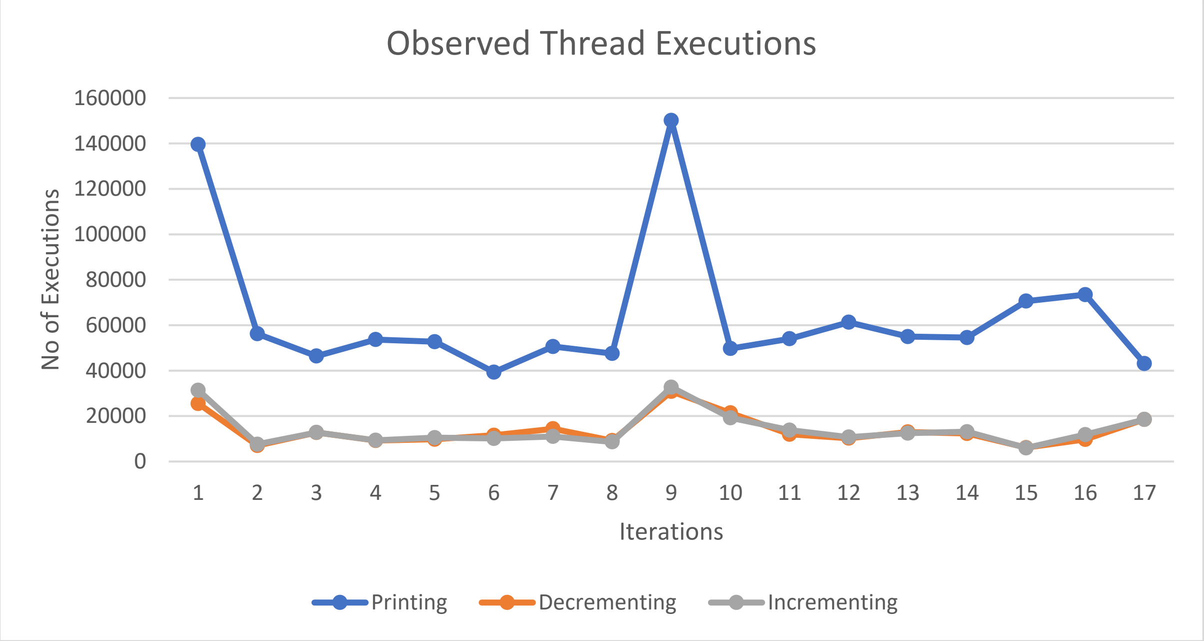

Figure 2: No Of Executions Observed

Figure 2 shows that there are more instances of the printing thread compared to decrementing or incrementing thread. It also shows that there are similar number of instances of both incrementing and decrementing threads. But these observations are deceptive. As the observations are only made while printing thread is executed, large number of increment or decrement operations can be masked as a single instance. The same can be observed in Figure 3 with sudden spikes in the value which shows the incrementing or decrementing threads holding on to the control for a longer period.

We are also not able to witness the number of increment and decrement operations happening between the print statements.

Exercise 3

Race conditions are observed in both the *non-determinism.cpp* and *shared-variable.cpp* exercises. Both the programs have implemented mutex locks to apply mutual exclusion. Therefore, the race between the threads is to attain control of the shared lock, but not for data.

Mutex locks also prevented data inconsistencies at the shared data, and thereby avoided data races and maintain synchronization between threads.

Exercise 4

The following details were obtained after logging on to the server *gullviva.it.uu.se* and using the *lscpu* and *lscpu -p* commands:

Number of logical CPUs – 32

Number of (physical processors) -2

Number of processor cores – 16

Number of hardware threads running on each core - 2

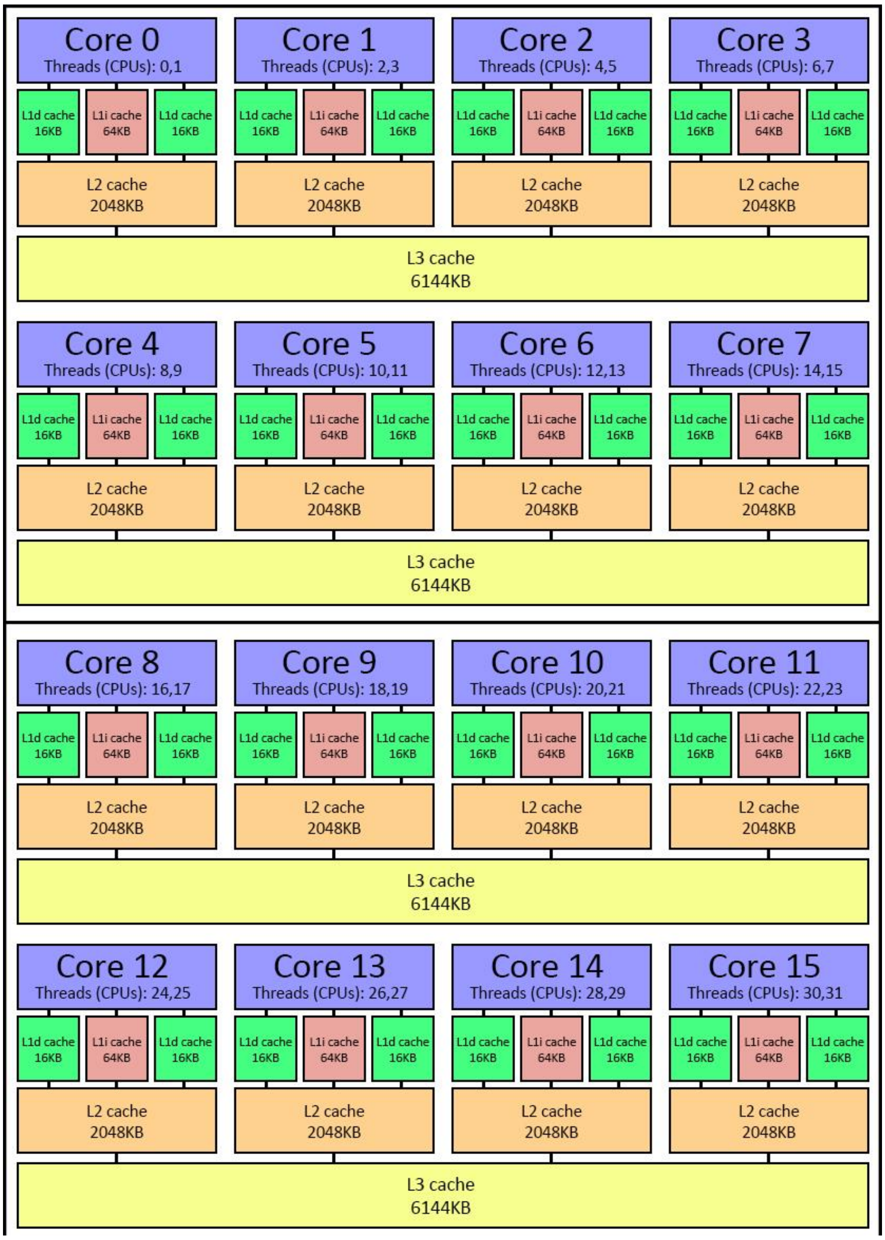

Figure 4: Core Architecture

The AMD x86_64 architecture obtained from the server has 16 cores with two threads of execution per core compared to the Intel i5-450M processor. With respect to Intel, AMD has 2 L1d caches for a single core. The layout and size of the caches varies between the two too.

Exercise 5

The *performance.cpp* program was executed using *'makePerformance'* makefile.

As expected, the performance increased exponentially as the number of threads increased, thereby delivering lesser runtimes.

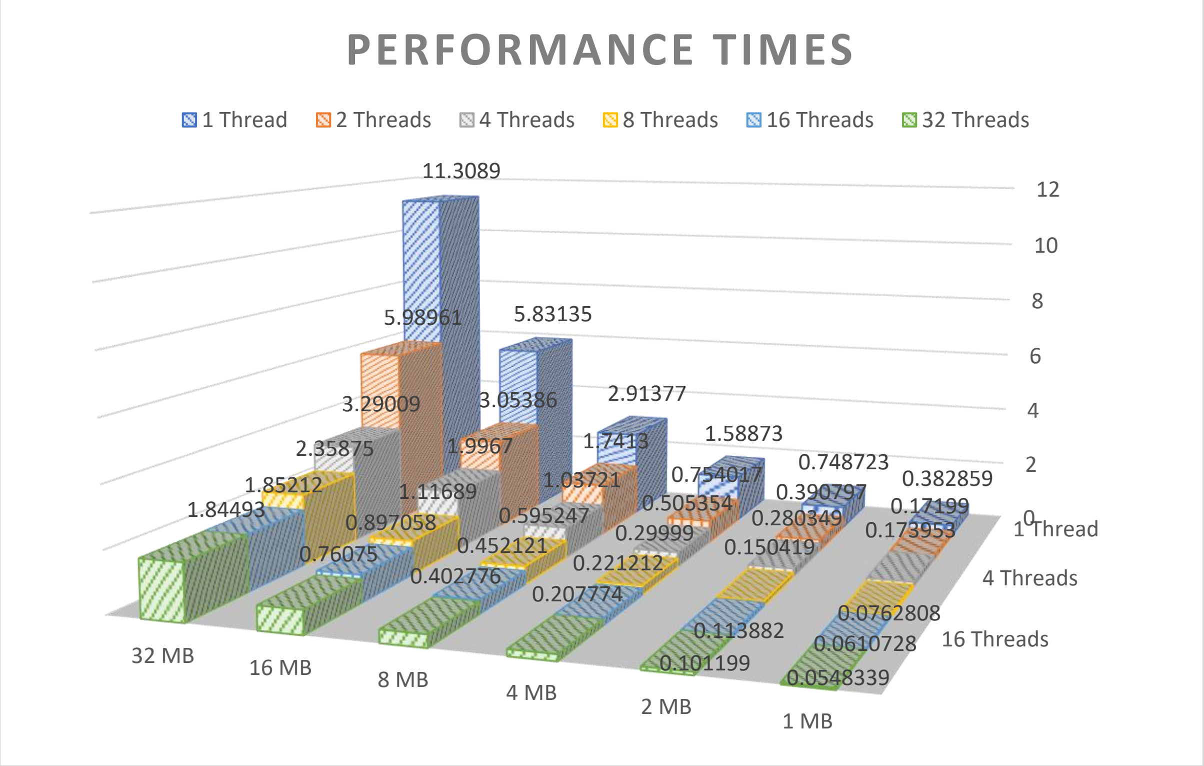

Figure 5: Performance Times

Figure 5 also shows that there isn't much improvement between the 16 and 32 threads. A contributing factor for this observation is that the program is run on a 16 core system (trigger.it.uu.se) and can thus support a maximum of only 16 threads. This implies that 2 threads are sharing a single CPU when run with 32 threads.

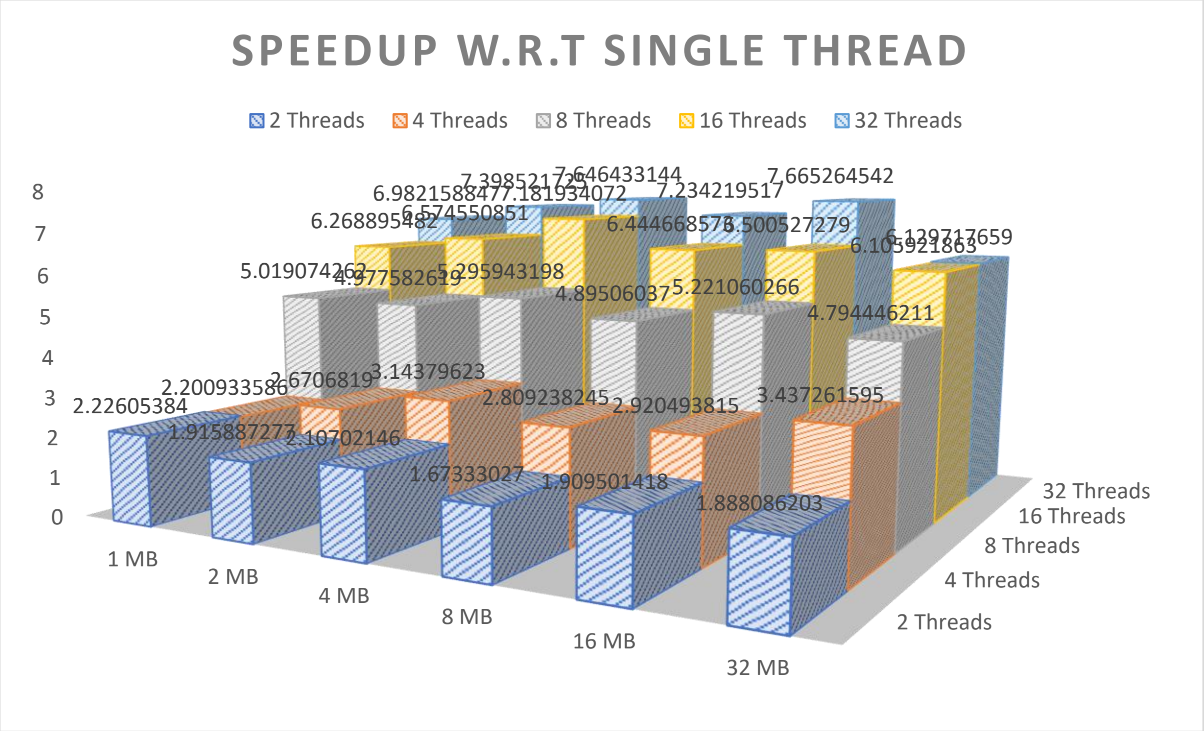

Figure 6: SPEEDUP

Figure 6 shows the speedup with respect to a single thread. It is interesting to note from this chart that the speedup is less for 32 MB compared to 1 MB.

Exercise 6

The *dining.cpp* program was downloaded and executed using *'makeDining'* makefile.

The program expects two kinds of problems:

Starvation: This results when one or more philosophers get greedy with the forks continuously and other philosophers might not get to eat.

Deadlock: This results when every one of the philosophers have one of the forks up which then does not let any of them to eat anymore.

To better understand the program a thread sanitizer was used, which reported a lock-order-inversion crash which implies a potential deadlock.

One of the solutions that can solve for avoiding deadlock without incurring starvation is by introducing an altruistic philosopher. This is a philosopher who drops the fork that it holds when he realises that no one can eat because there is a deadlock situation.
Given N philosophers, a round robin policy guarantees that each philosopher becomes altruistic exactly once in a sequence of N deadlocks. Hence, each philosopher is guaranteed to eat (at least) once every N deadlocks (because it is the predecessor of an altruistic philosopher exactly once every N deadlocks).