Viktor Wallsten & Edvin Bruce

Excercise 1 The output indicates that the order each thread begins executing may vary from each run of the program, as well as the order they terminate. This is likely due to the OS scheduling the threads seemingly randomly. The program consists of multiple independent tasks, each managed by its own thread, making it concurrent. A concurrent program is often non-deterministic because its output frequently depends on the OS scheduling, which can explain the seemingly random results.Regarding wether there are other possible outputs that we did not observe, the answer is most likely yes (without having looked at the source code). If you consider it to be completely random which task starts running first, as well as which task terminates first, you would get 8! * 8! = 1625702400(!) different combinations of output (if my math is correct).

Excercise 2 Running the program the output starts at a seemingly random number. This number is the randomly modified, somethimes it is increased, sometimes it is decreased. The output is sometimes increased or decreased by more than one this is due to the fact that X is not incremented and printed on the same thread. Thus one the ”inc” thread X might be incremented by 3, thus when the print thread is executed X is increased by more than one. One thing we noticed was that very often the print thread would get to ”execute” many times in a row. We first thought of this as weird, thinking ”why does this thread get scheduled 10 times in a row” e.g. Later we realised by looking at the source code that this thread is in fact only scheduled once and the many prints are just the effect of a loop in the program, which mislead us initially.

Excercise 3 In program ”shared-variable” we almost have data race. Here we have a shared variable X between each thread. This variable is accessed by at least by two threads at the same time where atleased one is a write operation. However, as the writes are withing mutex locks the data is not accessed at the same time preventing data races. However there are race conditions in the program, in that which thread that is executed is decided by the OS and depending on which thread is executed first the output may differ.

In program ”non-determinism” we dont have any shared resources (apart from like the stdout if that is to be considered a resource but i doubt it). There are no data races in this program as a requirement for data races is to have multiple threads accessing the same data and one of these accesses being a write, which isnt the case in this program. However race conditions are prevalent as in what order the threads write to stdout is very much dependent on how the operating system decides to schedule the threads.

Excercise 4

a - How many (logical) CPUs does your machine have?

32 cpus

- How many (physical) processors (i.e., sockets) and processor cores?

2 physical processors (sockets) and 8 cores per processor (16 total).

- How many hardware threads are running on each core?

threads per core is 2.

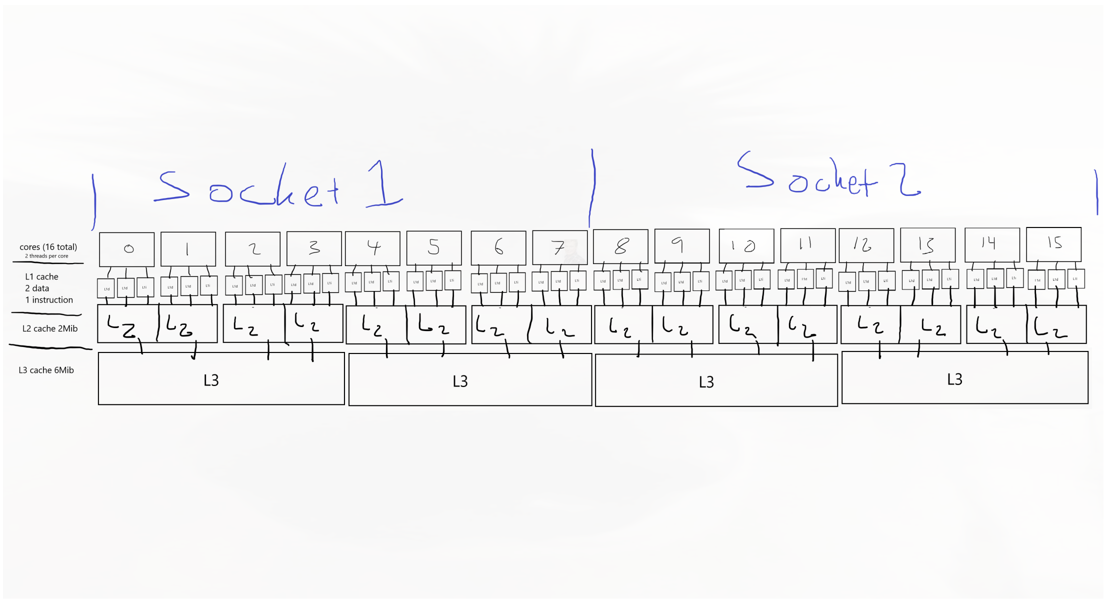

Figure 1: Excersice 4b

b The main difference between the intel chip and the AMD chips is the number of cores, the AMD chip has 16 cores while the intel chip only has 2. Another difference between the chips are that the chips is that the AMD has two L1d caches and the Intel chip only has 1. It is however worth noting that the total size of the L1d caches are the same for both the chips (32KB). Moreover the AMD chip has a larger L1i cache 64KB vs Intel 32 KB, as well as a larger L2 (2 MB vs 256 KB) and L3 (6 MB vs 3072 KB) cache.

Excercise 5

a

Num threads	Array size (MB)	wall time (seconds)
1	1	0.911696
1	2	1.8218
1	4	3.64919
1	8	7.29951
1	16	14.5811
1	32	29.1984
2	1	0.456694
2	2	0.913164
2	4	1.82704
2	8	3.65598
2	16	7.31357
2	32	14.5909
4	1	0.229687
4	2	0.458869
4	4	0.917118
4	8	1.83676
4	16	3.67342
4	32	7.34345
8	1	0.115926
8	2	0.231372
8	4	0.462006
8	8	0.92304
8	16	1.84768
8	32	3.69759
16	1	0.0604367
16	2	0.120475
16	4	0.241072
16	8	0.481187
16	16	0.962096
16	32	1.92824
32	1	0.047389
32	2	0.0956358
32	4	0.187238
32	8	0.361498
32	16	0.746166
32	32	1.36948

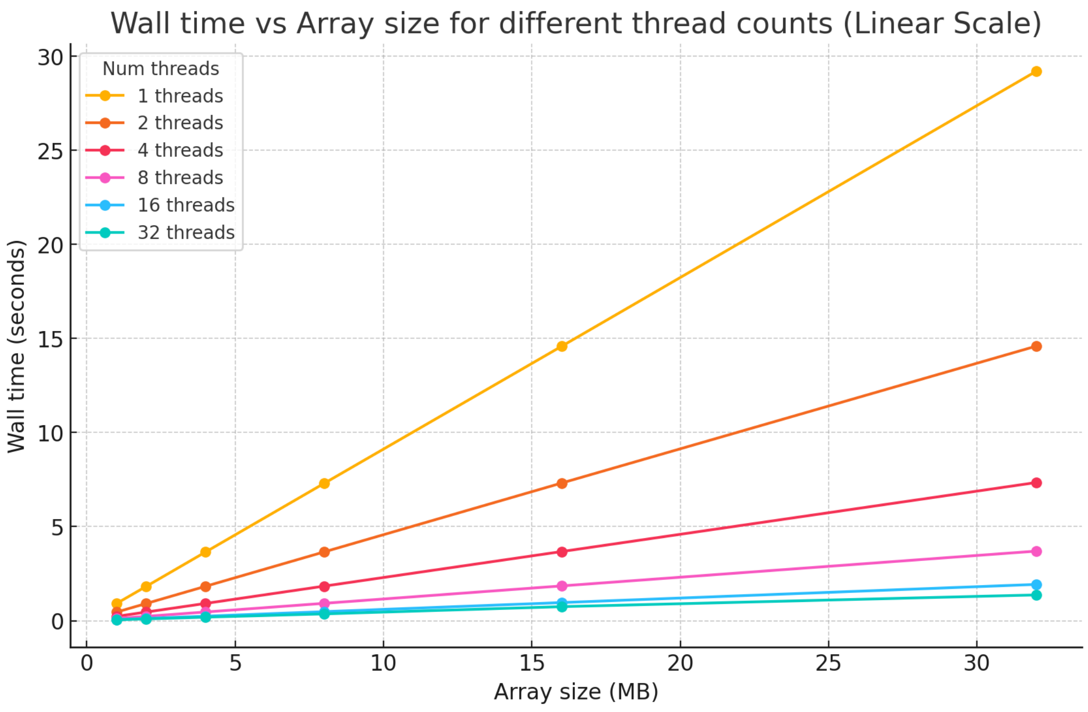

Figure 2: Excercise 5b

b

c

The measurements from the table above show that when doubling the amount of threads, the execution time (wall time) approximately gets cut in half. However, this was not the case for when going from 16 to 32 threads, where the execution time decreased but not by 50% but decreased less. According to Amdahl's law it is also not possible to infinetly speedup a program by just using more processors, as there is a limit to how much of a program is parallelizable. Insted if we want the program to run faster more of the code needs to be parallelizable before we increaste the amount of threads. For this program it seems that the speed up is decrementing after 16 threads.

Excercise 6 a It appears that deadlocks occur eventually no matter the amount of philosophers that are dining. This is occurs when all philosophers are waiting for their right fork to be vacant and are holding their left fork (i think), meaning that everone is waiting for the philosopher to their right to drop their fork. A deadlock occurs when four conditions are met: mutual exclusion, resource holding, non-preemption and circular wait. Inorder to brake a deadlock one of these conditions must be negated.

b See the provided source code in ”dining.cpp”. Basically the change we made was that if a philosopher has their left fork but cant take their right, they will drop their left fork. This will give another philosopher the opportunity to take their right fork (the first philosophers left fork) and begin eating!