ECSE 420 Assignment 1 Report
Group 19

Stuart Mashaal
260639962

Oliver Tse Sakkwun
260604362

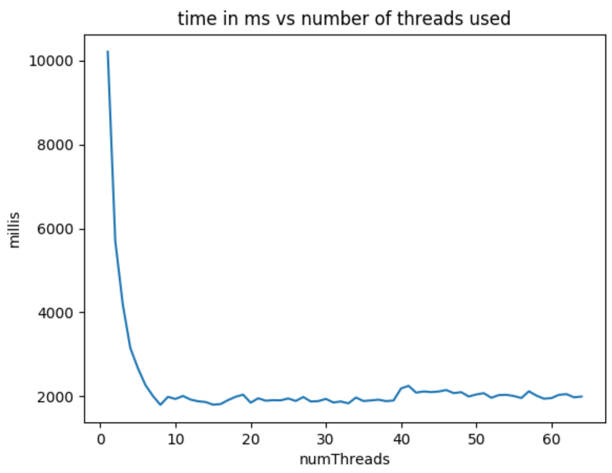
Due: October 10th, 2018

Question 1

1.2

Matrix multiplication was parallelized column-wise. That is, given a number T of threads and a size of matrix $N \times N$, the N columns of the matrix are divided up evenly among the T threads so that each thread must only calculate the matrix product for its corresponding columns.

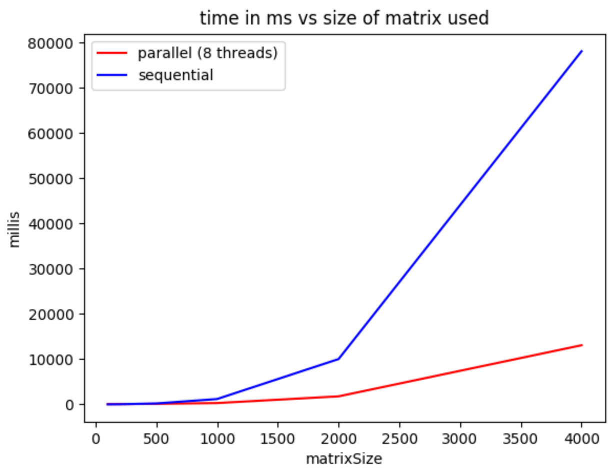
1.6 A



In the above figure we see that running time drops off sharply (hyperbolic) with the number of threads until 8 threads. This is sensible since the machine used has 8 cores and Ahmdal's law predicts that when the entire program is parallelizable, each extra core used increments the divisor of running time. Once more than 8 threads are used, parallelization is not improved and each new thread increases (by a small amount each time) the time spent on thread-switching overhead.

This explains why the curve appears to rise slowly and monotonically after 8 threads.

1.6 B



The above graphs depicts two function that rise at increasing rates. There is not enough data to model the functions exactly, but low-order polynomials like $O(n^3)$ are expected since the algorithm used for matrix multiplication runs in $O(n^3)$. The parallelized algorithm runs in $O(n^3/k)$ where the constant k is the number of threads, so the graphs above are reasonable since the parallelized graph looks like it is proportional to the sequential one with a rate of proportionality equal to $1/k$.

Moreover, we see that the two graphs begin to diverge at a matrix size of roughly 1000×1000. This can be explained by the fact that the thread-switching overhead and the parallelization speedup work against each other; for small matrices (1000×1000 and smaller), the parallelization speedup does not exceed the extra thread-switching overhead. However, parallelization speedup more than compensates for the thread-switching overhead for large matrices (it seems for any larger than 1000×1000).

Question 2

2.1

Deadlock can occur when these four conditions are meet:

1. Mutual exclusion: At least one resource must be held in a non-shareable mode. Otherwise, the processes would not be prevented from using the resource when necessary. Only one process can use the resource at any given instant of time
2. Hold and wait or resource holding: a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. No preemption: a resource can be released only voluntarily by the process holding it.
4. Circular wait: each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, $P = \{P_1, P_2, ..., P_N\}$, such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3 and so on until P_N is waiting for a resource held by P_1.

2.2

The solution to resolve deadlock is to remove the above conditions by employing the following two strategies:

1. Prevention
   1. Mutual exclusion: In general, this condition cannot be disallowed.
   2. Hold-and-wait: The hold and-wait condition can be prevented by requiring that a process request all its required resources at one time, and blocking the process until all requests can be granted simultaneously.
   3. No preemption: One solution is that if a process holding certain resources is denied a further request, that process must release its unused resources and request them again, together with the additional resource.
   4. Circular Wait: The circular wait condition can be prevented by defining a global ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.
2. Detection and reallocation of resources
   - The system constantly monitors processes for deadlocks/unsafe states (for example using the Banker's Algorithm) and when it detects them, it will restart and/or delays all or some of the offending processes.

Question 3

Deadlock and Starvation in the Dining Philosophers Problem.

3.2

To avoid deadlock in the Dining Philosophers Problem, we had our philosophers follow one rule: even-numbered philosophers are ‘lefty’ (pick up their left chopstick first) and odd-numbered ones are ‘righty’. This prevents deadlock by prevent circular wait. By alternating righty and lefty philosophers, philosophers are considered to be grouped into pairs where both members of the pair try to pick up the same chopstick first. Because they share a first chopstick, it is impossible for them to hold the other‘s second chopstick while waiting on the first.

Note that the blocking lock acquisition (which doesn't return until the lock can be and is acquired) was used intead of non-blocking lock acquisition (which traditonally returns false if the lock is not acquired).

3.3

To avoid the possibility of starvation, one must guarantee that any philosopher that attempts to acquire a chopstick's lock **will eventually acquire it**. We provide this guarantee by designing the choptstick acquisition to be ‘FIFO fair‘. That is, when a philosopher attempts to acquire a lock (on a chopstick) and must wait for said lock, the philosopher waits in a FIFO queue. Because the queue is FIFO, and deadlock is already prevented in the same manner as question 3.2, any philosopher that enters the queue will eventually get the chopstick.

Note that we manually implemented the fair chopstick using a monitor with a queue of semaphores. Please have a look at the implementation, as readability was a goal.

Question 4

Amdahl's Law:

$$\text{Speed-Up} = \frac{1}{S + \frac{P}{N}}$$

where

- S and P are the sequential and parallel time percentages of the program, respectively. $S + P = 1$.
- N is the number of processors that can be used to paralellize the parallel fraction of the program

4.1

The maximum speed-up of a program occurs when the program is executed on an *infinite* number of processors. So, for a program where the sequential portion is 40% of the program, the maximum speed-up is:

$$\lim_{N \to \infty} \frac{1}{0.4 + \frac{0.6}{N}} = \frac{1}{0.4} = 2.5$$

4.2

Given a fixed number of processors, N, we wish to make a 20% sequential program twice as fast. We wish to do this by decreasing the program's sequential time percentage, S, by a multiplicative factor, k. We can find k by isolating it in the following equation:

$$\begin{aligned}
& 0.2 + \frac{0.8}{N} &&= 2 \cdot \left(0.2k + \frac{1 - 0.2k}{N}\right) \\
\Longrightarrow\ & 0.2N + 0.8 &&= 0.4kN + 2 - 0.4k \\
\Longrightarrow\ & 0.2N - 1.2 &&= 0.4k \cdot (N - 1) \\
\Longrightarrow\ & \frac{0.2 \cdot (N - 6)}{0.4} &&= k \cdot (N - 1) \\
\Longrightarrow\ & \frac{N - 6}{2 \cdot (N - 1)} &&= k
\end{aligned}$$

4.3

Given a fixed number of processors, N, and a program that runs twice as fast when the sequential time percentage, S, is divided by 3, the original sequential time percentage found by isolating S in the following equation:

$$\begin{aligned}
& S + \frac{1-S}{N} &&= 2 \cdot \left(\frac{S}{3} + \frac{1 - \frac{S}{3}}{N} \right) \\
\implies & SN - S + 1 &&= \frac{2SN}{3} + 2 - \frac{2S}{3} \\
\implies & 3SN - 3S + 3 &&= 2SN + 6 - 2S \\
\implies & SN - S &&= 3 \\
\implies & S &&= \frac{3}{N-1}
\end{aligned}$$