Writing Assignment 1

Name: Aditya Agrawal

Roll No.: CS20S026

Subject: Analysis of Parallel Programs

A1a) The given program greets a thread by printing Hello there, followed by the thread name (expected) on the console. The class Greeter contains a method greet which takes a name as a parameter to greet the thread with the name as specified in name. In this case, as we know the System.out is an instance of the PrintStream class and instances of a class are stored in the heap. So, the threads share the PrintStream instance and print to the output buffer (which is the shared resource). The main function (in the main thread) spawns the threads and the corresponding run() of the thread calls the greet method with the name as returned from the syntax Thread.currentThread().getName() which returns the name of the thread. Since the greeter object is stored in the heap, so the threads also share the object on the heap. In this case, two threads can first print there respective System.out statement followed by writing the name to the console.

A1b) The given program simulates a game of chance where a person plays the game where he, with a probability p wins a gamble every hour and loses the gamble with a probability q (1-p). He plays the game for 10 hours. The given program utilizes the Math class random() to generate a random number between 0 and 1 and accordingly decide whether the gambler won the gamble or lost and accordingly update the shared variable. The class Game creates four members, currAmount

which stores the current amount the gambler has after the end of any draw and two double variables p and q to store the respective probabilities. The constructor initializes the Game object to the appropriate values. The class GameRunner takes as member an object of type Game which is stored in the heap (shared by the threads). The run() designates a thread (whose name is Decrementer) to execute the decreaseAmount function and designates a thread to execute the incrementAmount function to decrease and increase the amount according to the gamble result respectively. The main thread initializes a game object and for correctness purpose, simulates the game on its own as well (without using threading) by utilizing the variable trueAmount. Now, the for loop inside the main function which is executed by the main thread generates a random number p (between 0 and 1) and accordingly spawns a thread to perform the required updation. Now, since we know the threads share the heap space, the threads share the Game Object and its respective members. The function printCurrentAmount prints the status of the game (the current amount the gambler has). Now, let us see how the execution proceeds when a thread executes the increaseAmount function. The increase amount function first performs an atomic read to a shared variable(named currAmount) and updates its local thread variable temp with the value returned from the function getAmount(). It then performs the update to its local variable temp to increase it by 10 as the person had won the current gamble. After this, the thread performs an atomic write to the shared variable currAmount. So, the reads and writes are ensured to be performed by only one thread at a time due to the presence of synchronized block in the methods respectively. Similarly, the decreaseAmount() performs an atomic read to retrieve the amount and update its local variable to decrease it by 10 followed by atomically updating the shared variable currAmount. Even though

we are ensuring only one thread is writing to the shared variable or reading from a shared variable, still memory inconsistencies can occur. To understand it further, let us simulate the execution and see how inconsistency occurs. Let us say in the Hour = 0 (Iteration I = 0), the gambler wins the gamble. The result of which it executes the if block inside the for loop of the main method. The main method first updates the true amount by increasing it by 10 and then spawns a thread to perform the updation in the game object's currAmount method. Let us say, the thread is currently executing the increaseAmount function and gets to execute the first statement by atomically reading the value of shared variable currAmount and storing in it's local variable temp. Similarly, let us say there is a context switch to the main thread which executes the next iteration where the gambler loses the game. The main thread then spawns a new thread to decrement the amount. The new thread(let's call it decrementer) performs the atomic read to read the currentAmount to its thread local variable temp (100) and executes the update statement to its thread local variable and updating the value to 90 (decrease by 10). Let's say the incrementer gets the chance now who executes the update to temp and then performs the update to the shared variable total atomically to 110. Now, the decrementer thread gets the chance and performs an atomic write to the shared variable currAmount and updates it to 90 (the value it had in its local temp variable). This is not the correct value as the currAmount should have been 100 after the end of the 2^{nd} iteration (100 starts with + 10 won – 10 lost). In this manner, the threads can cause memory inconsistency even after the read and modify operations to the shared variable happen in a synchronized block.

A2) The given program prints the sum of an array of 1000000 numbers. The class SerialCompute computes the sum of the array by utilizing a single thread (the main thread) and the corresponding execution time

is recorded in the variable serialTime. Now, we use the class ParallelCompute to compute the sum of the array of elements using multithreading. First, we break the 1000000 array elements into chunks of the number of threads we want to spawn. Let's say we spawn 2 threads. Then, thread 0 executes the iterations 0 to 499999 and thread 1 execute the iterations 500000 to 999999. We use an outer for loop (in the main method) to account for the calculation using 2,4,8,16 threads respectively. The inner loop creates the starting and ending index of the respective threads and spawns the threads passing the start and end index accordingly. The time taken to execute the respective functions (executed on a 72 core CPU) is stored in the time array and then for each of the time to execute the parallel code, the corresponding speedup is computed (serialTime/parallelTime with I threads where I = {2,4,8,16}). The corresponding table is shown below: -

For n = 1000000

Thread(s)	Serial Execution Time(ns)	Parallel Execution Time(ns)	Speedup
1	$1.07 * 10^7$	$2.013*10^7$	0.53
2	$1.07 * 10^7$	$1.339*10^7$	0.80
4	$1.07 * 10^7$	$0.366*10^7$	2.94
8	$1.07 * 10^7$	$0.372*10^7$	2.90
16	$1.07 * 10^7$	$0.379*10^7$	2.84
32	$1.07 * 10^7$	$0.545*10^7$	1.97
64	$1.07 * 10^7$	$0.644*10^7$	1.67

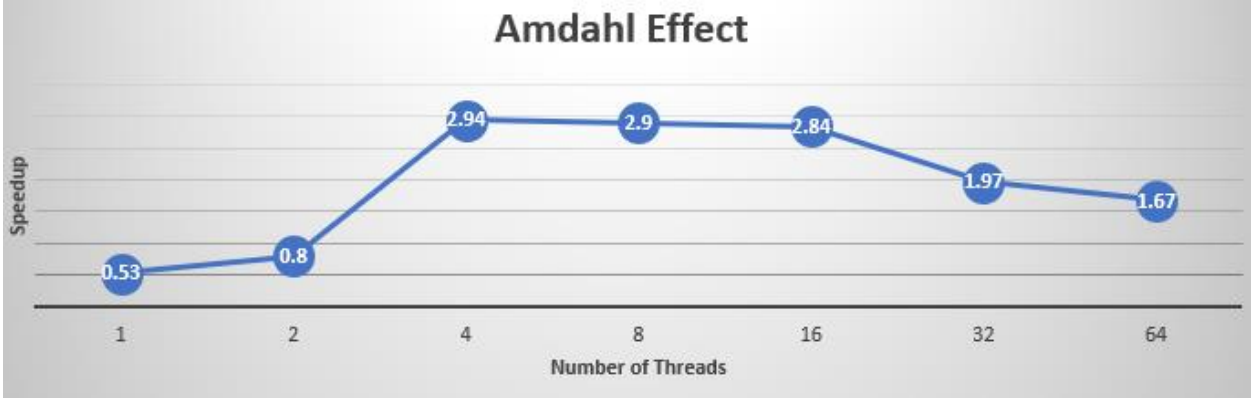


Figure 1Effect of Speedup with Increase in Number of Threads - Amdahl Effect

As we increase the number of threads (which get mapped to different parallel compute units), the speedup increases upto a certain value and then starts to decrease as according to Amdahl's Law, the speedup is dependent on the amount of parallelism exploited in the program and not on the number of parallel compute units.

3) In the given program greets a thread (by printing Hello there, followed by the name of the thread). The objective of this code is to show that the threads share the heap. In this case, we are trying to show that the threads share the wisher object w. Since the wisher object is only one time instantiated in the main, the object is allocated in the heap and so threads share the heap space. To prove this, in the thread's method run, we try to print the address of the wisher object and find it to be same for both the threads which signify both threads point to the same object in the heap. The output of the program is given below: -

```
Thread-0 is using Wisher object, details: WA.Wisher@1912b685
Hello there! Aditya
Thread-1 is using Wisher object, details: WA.Wisher@1912b685
Hello there! Krishna
```

As we see, the Wisher object is allocated in location 1912b685 which is accessible by both threads, which means they share the object on the heap.

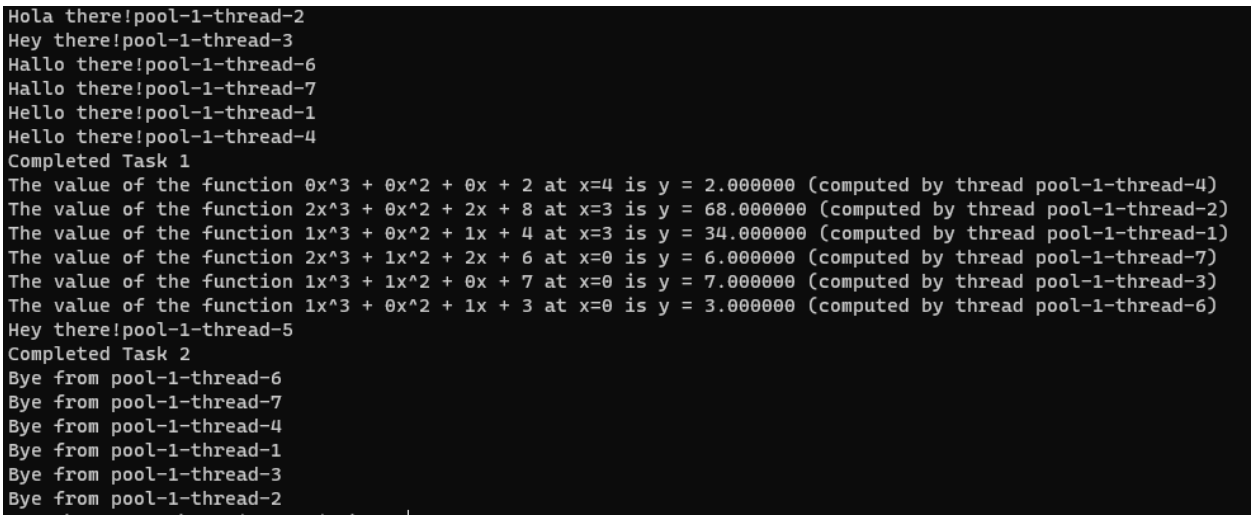
4)In the question 1a, the start to the thread (S5) happens before any of the statements are executed in the run () method which constitutes the job of a particular thread. Also, for a particular thread, the S1 HB S2 and also S3 HB S4 to maintain sequential ordering in a thread based on the program order. Also, statements S6 HB S7 since they are executed by only the main thread and they maintain sequential order. The corresponding constructor calls when executing S6 and S7 execute in sequential order as well respectively. The statement S10 HB S11 since the main thread waits for all the threads to complete its job in S10 and being in waiting state will only proceed to execute S11 once it has completed its waiting for other threads to finish its job. Thus, any statement executed by the thread will also happen before S11 by law of transitivity. If {S1, S2, S3, S4} HB S10 and S10 HB S11 then {S1, S2, S3, S4} HB S11.

In the question 1b, the statements S20 to S24 and according to the if condition, S26 or S29 are executed by only the main thread so sequential order is maintained and each statement occurs before any subsequent statement. The start method at S27 or S30 (depending on the predicate of the if condition) is executed before any of the thread's job (run ()) statements are executed. Inside the run method, either S18 or S19 gets executed only after the start method is invoked for the respective thread. If the thread executes S18, statements S13 and S14 always happen before the synchronized block S15 and S16 is executed to ensure the updates are visible to all threads. If the thread executes S19, statements S9 and S10 are executed before the synchronized block S11 and S12 is executed to ensure the updates are visible to all threads. To retrieve the game amount, the statements S5 always happen before synchronized block from S6 and the synchronized block is executed before the return statement S8.

5a) The given program illustrates how by using synchronized keywords, deadlocks can occur. The given class A has two methods a1 and bye which are synchronized methods, thereby only one thread would be allowed to enter the method at any moment. Similarly, class B has two methods b1 and bye which are also synchronized methods thereby allowing only one thread to execute the method at any moment. The main function creates an object of type Question5a and calls the runner method to start the thread and also execute the a1 method by passing b as the object in the parameter. The new thread created (let's call it t) executes the b's object b1 method by passing a object as parameter. Let us say t gets the chance first to execute the b1 method. Once t prints to the console, the thread calls the yield method to give the chance to other waiting threads. The main thread now gets the chance to execute the a1 method of object a. It prints the output to the console (by executing the first line) and calls the yield method to allow other waiting threads to proceed. The thread t then tries to obtain a lock on the object a and executes the bye method of the object (defined in class A). Since the main thread is already executing a synchronized method on the object a, the lock will not be granted until the main thread finishes executing the function (a1). So, the main thread gets the chance now to execute and wants to obtain a lock on the object b to execute the bye method (defined in class B). Similarly, since the object lock on b is already acquired by t (to execute its synchronized method), the main thread is not allowed to get the lock and goes to the waiting state. Hence, the main thread waits for a lock on object b and the thread t waits for a lock on the object a. Hence, they go in a deadlock.

5b) The given program performs three tasks. First to print a message to the console then computes the value of a randomly generated cubic equation (with random coefficients) followed by printing a message

before completing its job. A barrier is used to achieve synchronization among the threads. Let us see the below output of the program and analyze it better: -



The threads 1 to 7 are spawned from the main method by using a fixed thread pool. A cyclic barrier is used for synchronization among the threads. In the above output, threads 1,2,3,4,6 and 7 complete the first task and decrement the barrier count. As soon as 6 tasks are finished, the barrier is resetted to its default value (6) and the 6 threads continue to execute the second task. The threads 1,2,3,4,6 and 7 perform the 2nd task and also, the thread with id 5 gets the chance to execute and waits at c1. The threads 1,2,3,4,6 and 7 await at c2 and once they all reach move to execute the next task. In the meanwhile, the thread 5 keeps on waiting at c1 for 5 other threads to call their await method to proceed. The thread with id 5 waits indefinitely. This causes a deadlock.