Assignment 1 Solutions

Marco De Liso (18-120-261)

March 15, 2023

Contents

1 Task 1: Prime Counter

Write a Java program that takes as arguments two integers, T and N, and forks T threads that will together search for and print all primes between 1 and N. The program should evenly split the range [1..N] into T sub-ranges assigned to each thread. Execute the program with N=10'000'000,100'000'000 and T=1,2,4,8,16. Report execution times.

- **Answer:**
- Output:
  - All Primes found between 1 and 10'000'000 with 1 Thread:
    - ... List of all primes
    - Total time elapsed: 813 milliseconds
    - Total number of primes found: 664580
  - All Primes found between 1 and 10'000'000 with 2 Thread:
    - ... List of all primes
    - Total time elapsed: 528 milliseconds
    - Total number of primes found: 664579
  - All Primes found between 1 and 10'000'000 with 4 Thread:
    - ... List of all primes
    - Total time elapsed: 300 milliseconds
    - Total number of primes found: 664580
  - All Primes found between 1 and 10'000'000 with 8 Thread:
    - ... List of all primes
    - Total time elapsed: 232 milliseconds
    - Total number of primes found: 664579
  - All Primes found between 1 and 10'000'000 with 16 Thread:
    - ... List of all primes
    - Total time elapsed: 245 milliseconds
    - Total number of primes found: 664580
  - All Primes found between 1 and 100'000'000 with 1 Thread:

    - ... List of all primes
    - Total time elapsed: 21331 milliseconds
    - Total number of primes found: 5761455
  - All Primes found between 1 and 100'000'000 with 2 Thread:
    - ... List of all primes
    - Total time elapsed: 13550 milliseconds
    - Total number of primes found: 5761455
  - All Primes found between 1 and 100'000'000 with 4 Thread:
    - ... List of all primes
    - Total time elapsed: 7333 milliseconds
    - Total number of primes found: 5761455
  - All Primes found between 1 and 100'000'000 with 8 Thread:
    - ... List of all primes
    - Total time elapsed: 4633 milliseconds
    - Total number of primes found: 5761455
  - All Primes found between 1 and 100'000'000 with 16 Thread:
    - ... List of all primes
    - Total time elapsed: 3881 milliseconds
    - Total number of primes found: 5761455
  - Note: I added a Lock function to limit the access into the array of primes, otherwise with the concurrency multiprogramming of the threads they would have get a lot of conflicts, leading to only part of the primes to be printed. This was the only way I found to get the precise amount of primes found in total to make sure that it was working correctly. Probably the higher the amount of threads, the longer will be the waiting times to access the array, so by removing it we would get better performance for sure. By checking the time without this bottleneck: Remove line 105 and write instead $System.out.print(i + "")$

2 Task 2: Shared Counter

Modify the program of 1.1 so that it uses a shared counter to assign the next number to test to the threads. The shared counter should be protected from concurrent accesses by a monitor (e.g., using synchronized method). As before, execute the program and print the results.

- **Answer:**
- Output:
  - All Primes found between 1 and 10'000'000 with 1 Thread:
    - ... List of all primes
    - Total time elapsed: 868 milliseconds
    - Total number of primes found: 664579
  - All Primes found between 1 and 10'000'000 with 2 Thread:
    - ... List of all primes
    - Total time elapsed: 578 milliseconds
    - Total number of primes found: 664579
  - All Primes found between 1 and 10'000'000 with 4 Thread:
    - ... List of all primes
    - Total time elapsed: 722 milliseconds
    - Total number of primes found: 664579
  - All Primes found between 1 and 10'000'000 with 8 Thread:

    - ... List of all primes
    - Total time elapsed: 713 milliseconds
    - Total number of primes found: 664579
  - All Primes found between 1 and 10'000'000 with 16 Thread:
    - ... List of all primes
    - Total time elapsed: 711 milliseconds
    - Total number of primes found: 664579
  - All Primes found between 1 and 100'000'000 with 1 Thread:
    - ... List of all primes
    - Total time elapsed: 21783 milliseconds
    - Total number of primes found: 5761455
  - All Primes found between 1 and 100'000'000 with 2 Thread:
    - ... List of all primes
    - Total time elapsed: 11515 milliseconds
    - Total number of primes found: 5761455
  - All Primes found between 1 and 100'000'000 with 4 Thread:
    - ... List of all primes
    - Total time elapsed: 6989 milliseconds
    - Total number of primes found: 5761455
  - All Primes found between 1 and 100'000'000 with 8 Thread:
    - ... List of all primes
    - Total time elapsed: 8551 milliseconds
    - Total number of primes found: 5761455
  - All Primes found between 1 and 100'000'000 with 16 Thread:
    - ... List of all primes
    - Total time elapsed: 7922 milliseconds
    - Total number of primes found: 5761455
  - Note: I added a Lock function to limit the access into the array of primes, otherwise with the concurrency multiprogramming of the threads they would have get a lot of conflicts, leading to only part of the primes to be printed. This was the only way I found to get the precise amount of primes found in total to make sure that it was working correctly. From the runs it seems like that the Shared Counter is worse then expected, however, this is not the case, since in this implementation we have two synchronized blocks which cause some delay. By removing line 104 and putting instead $System.out.print(i + "")$ the scripts shows its real performance. It is interesting to observe how the concurrency (and its busy waiting in synchronized blocks) can at some point (by increasing threads) make the script even slower than with less threads.

3 Task 3: Producer-Consumer Problem

Consider the classical producer-consumer problem: a group of P producer threads and a group of C consumer threads share a bounded circular buffer of size N. If the buffer is not full, producers are allowed to add elements; if the buffer is not empty, consumers can consume elements. Write a program that can correctly coordinate the producers and consumers and their depositing and retrieving activities. For simplicity we assume that we have the same number T of producers and consumer threads. T and N are program arguments.

- **Answer:**
- The implementation runs smoothly, each producer thread produces an element called with id of the thread itself and stores it in the list, it does this if the list is not full, whereas consumer threads consume an element from the list if the list is not empty, otherwise waits. The scripts outputs whenever an element is consumed or produced. The circular buffer has synchronized access through locks and is initialized with the size of args input N given from the user.

4 Task 4: Amdahl's Law

You have a choice between buying one uniprocessor that executes five zillion instructions per second, or a ten-processor multiprocessor where each processor executes one zillion instructions per second. Using Amdahl's Law, explain how you would decide which to buy for a particular application.

- **Answer:**
- Amdahl's Law states that the speedup of a program using multiple processors in parallel computing is limited by the fraction of the program that cannot be parallelized. Therefore, we need to consider the parallelizability of the application before deciding which to buy.

  If the application can be parallelized well, meaning that a significant portion of the program can be executed in parallel by multiple processors (at least 90-95%), then the multiprocessor would be a better choice. In this case, we can expect a significant speedup by dividing the work among the ten processors.

  However, if the application has a large portion that cannot be parallelized, then the uniprocessor would be a better choice. In this case, the multiprocessor would not provide much speedup as most of the work cannot be parallelized. Therefore, it would be more cost-effective to buy the uniprocessor, which has a higher clock speed and can execute the instructions faster.

  In summary, we need to consider the parallelizability of the application before deciding which processor to buy. If the application can be parallelized well, then the multiprocessor would be a better choice, but if not, then the uniprocessor would be a better choice.