UPPSALA UNIVERSITY

INTRODUCTION TO PARALLEL PROGRAMMING

Assignment 1

Authors:
Erik CERVIN-EDIN
Ahmed BIHI

September 22, 2018

Exercise 1: Concurrency and Non-determinism

The program `non-determinism` outputs the following during five runs.

```
Run number: 1

Task 1 is running.
Task 1 is terminating.
Task 2 is running.
Task 2 is terminating.
Task 3 is running.
Task 3 is terminating.
Task 4 is running.
Task 4 is terminating.
Task 5 is running.
Task 5 is terminating.
Task 6 is running.
Task 6 is terminating.
Task 7 is running.
Task 7 is terminating.
Task 8 is running.
Task 8 is terminating.

Run number: 2

Task 1 is running.
Task 1 is terminating.
Task 2 is running.
Task 2 is terminating.
Task 3 is running.
Task 3 is terminating.
Task 4 is running.
Task 4 is terminating.
Task 5 is running.
Task 5 is terminating.
Task 6 is running.
Task 6 is terminating.
Task 7 is running.
Task 7 is terminating.
Task 8 is running.
Task 8 is terminating.

Run number: 3
```

```
Task 1 is running.
Task 1 is terminating.
Task 2 is running.
Task 2 is terminating.
Task 3 is running.
Task 3 is terminating.
Task 4 is running.
Task 4 is terminating.
Task 5 is running.
Task 5 is terminating.
Task 6 is running.
Task 6 is terminating.
Task 7 is running.
Task 7 is terminating.
Task 8 is running.
Task 8 is terminating.

Run number: 4

Task 1 is running.
Task 1 is terminating.
Task 2 is running.
Task 2 is terminating.
Task 3 is running.
Task 3 is terminating.
Task 4 is running.
Task 4 is terminating.
Task 5 is running.
Task 5 is terminating.
Task 6 is running.
Task 6 is terminating.
Task 7 is running.
Task 7 is terminating.
Task 8 is running.
Task 8 is terminating.

Run number: 5

Task 1 is running.
Task 1 is terminating.
Task 2 is running.
```

```
Task 2 is terminating.
Task 3 is running.
Task 3 is terminating.
Task 4 is running.
Task 4 is terminating.
Task 5 is running.
Task 5 is terminating.
Task 6 is running.
Task 6 is terminating.
Task 7 is running.
Task 7 is terminating.
Task 8 is running.
Task 8 is terminating.
```

We can observe that each thread prints to `std:cout` one at a time. Furthermore, each thread will first print that it is running and afterwards it will print that it is terminating. We can observe that the threads run `loop` in sequential order.

Since the call to `std::cout` occurs within a critical section, only one threads at a time may print to `std::cout`. The two different writes to `std::cout` occur within the same process and will thus always run sequentially.

The threads are initialized in sequential order, which is the likely reason that they carry out their writes to `std::cout` in sequential order, even though this is not guaranteed. For example, thread 2 might print that it is running before thread 1 writes that it is terminating if thread 2 enters the critical section before thread 1.

Exercise 2: Shared-Memory concurrency

Almost every time we run *shared-variable.cpp* we observe a different final printed value for the shared variable *x*. The shared variable *x* starts of as being 0 but is always, in our trial runs, a positive integer bigger than 1000. This happens because the increment thread is instantiated before the decrement thread and can therefore run several iterations before the decrement thread is instantiated. We tried to instantiate the decrement thread before the increment thread, then the shared variable *x* was always a negative integer less than -1000 in our trial runs. After both the threads are instantiated the value of x stabilizes such that the same value is printed several times, as long as none of the threads gets interrupted.

Exercise 3: Race Conditions vs. Data Races

The program *non-determinism.cpp* contains a race condition, even if the threads are initialized in sequential order, each thread will try lock `std:cout` at the same time after the first thread has locked `std:cout`.
There are no data races in *non-determinism.cpp* because none of the threads perform a write instruction to the same memory location.
The program *shared-variable.cpp* contains a race condition since the scheduling of the threads at run-time heavily influence the final printed value x when all threads are terminated. Since there are no inherent ordering of the increment/decrement threads a single interrupt could skew the value of x to either increase or decrease drastically because one of the threads were interrupted.
There are no data races in *shared-variable.cpp*, even though two different threads are performing a write instruction to the same memory location each write instruction is performed inside a critical section.

Exercise 4: Multicore Architectures

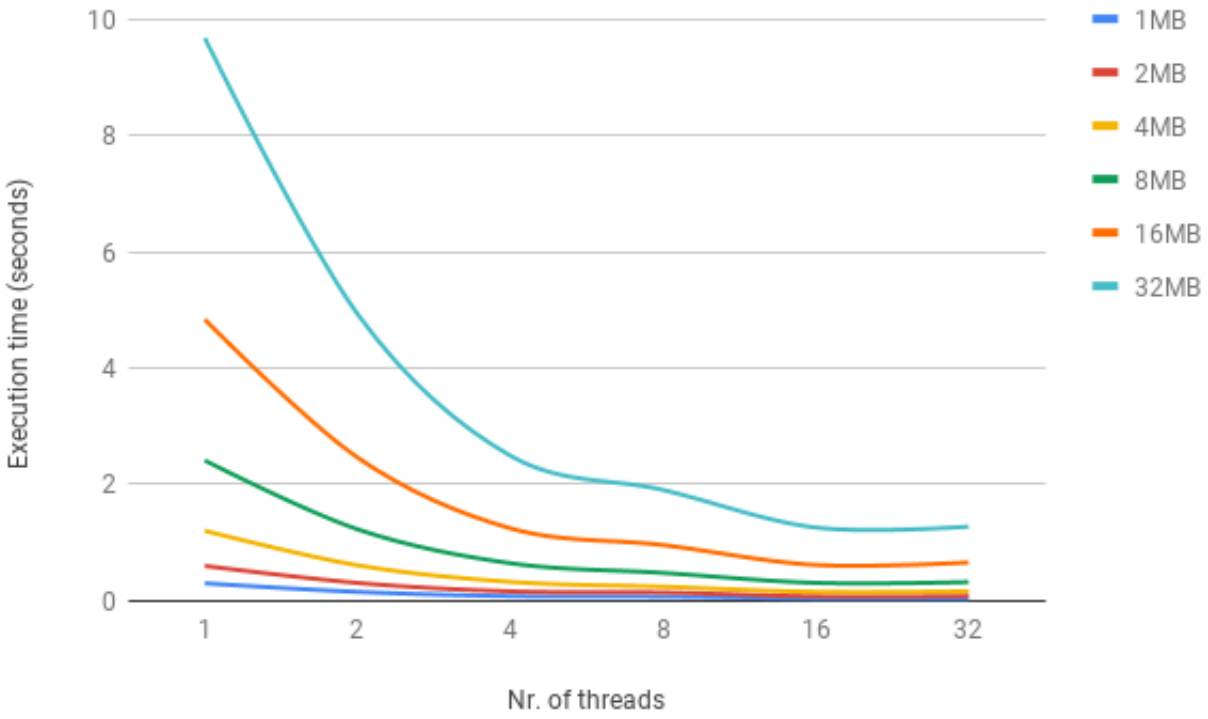
The dual socket Xeon E5520 system on siegbahn.it.uu.se has:

- Logical CPU cores: 16
- Physical processor: 2
- CPU cores: 8
- Threads per core: 2

Core0 Threads: 0,8		Core1 Threads: 1,9		Core2 Threads: 2,10		Core3 Threads: 3,11		Core4 Threads: 4,12		Core5 Threads: 5,13		Core6 Threads: 6,14		Core7 Threads: 7,15	
L1d 32KB	L1i 32KB	L1d 32KB	L1i 32KB	L1d 32KB	L1i 32KB	L1d 32KB	L1i 32KB	L1d 32KB	L1i 32KB	L1d 32KB	L1i 32KB	L1d 32KB	L1i 32KB	L1d 32KB	L1i 32KB
L2 256KB		L2 256KB		L2 256KB		L2 256KB		L2 256KB		L2 256KB		L2 256KB		L2 256KB	
L3 8192 KB								L3 8192 KB							

Table 1: Dual socket Xeon E5520

Exercise 5: Performance Measurements



The tests were run on the `siegbahn.it.uu.se` environment which was detailed in exercise 4.

The first thing we noted was that for one thread, the execution time scales linearly as the problem size increases. A 32MB array takes approximately 32 times as long to execute than a 1MB array.

This relationship is somewhat skewed as we increase the amount of threads. For example when we use 8 threads, the 32MB array only takes 25.5 times longer than the 1MB array. However, as we increase the amount of threads computing the array beyond 8 threads it starts to linearize again. At 32 threads the 32MB array takes 29.6 times the time to execute as with 1 thread.

We also observe that increasing the number of threads decreases execution time. As we go from 1 to 4 threads, the execution time decreases roughly by a factor between 3.62-3.87 for all sizes of the array. This effect levels of and at 8 threads the execution time is only between 4.08-5.07 times faster.

Increasing the thread count to 16 decreases the execution time by roughly 7.03-7.81 times whilst increasing the threads to 32 doesn't impact performance a lot. In a few cases it even performs slower.

Besides the instantiation, the program is fully parallel. Looking at the case where the array is 1MB, we can see that the speedup is basically linear, which according to Amdahl's law would

mean that the parallel portion of execution is basically 100%. The likely explanation for the observation we made comes down to how data is cached.

Each thread will prefetch the part of the array it is working on into L1, L2 and L3 cache and in this architecture each physical core shares L1 and L2 cache between two threads. One might expect that increasing the amount of threads would increase performance, as each thread would be working on smaller amounts of data and be able to fit more of the problem in a faster cache.

However, in this case the data each thread will be working on is always larger or equal to the size of the L1 cache. This means there will always be capacity misses, even in the case where the data is equal to the L1 cache (when there are 32 threads) as two threads together will work on 64KB of data and share L1 cache.

Furthermore, when 32 threads are used, all threads will be sharing cache with another thread which is reading and writing to a different part of memory. This will cause the two threads to compete for cache and they will start throwing out each others data which could explain why increasing the amount of threads beyond 16 threads doesn't speed up execution.

Exercise 6: Dining Philosophers

a) The output shows that the algorithm runs without a problem, where each philosophers thinks, picks up forks, eats, and drop forks, until we reach a deadlock state where each philosophers has one fork in their hand and is waiting for another fork to be available. But since their is not anything limiting how long a philosopher can hold a fork we will be stuck in this deadlock state indefinitely. Therefore, we need a solution to not end up in that state in the first place.

b) Our solution forces each philosopher to try to pick up the left and if that suceeds try and pick up the right fork. If the acquisition of the lock for the right fork fails then the philosopher will drop the left fork and go back to thinking. We use the *std::mutex::try_lock* method which returns true and locks the variable if we were able to pick up a fork or false if the fork is occupied. If both *left->try_lock()* and then *right->try_lock()* method calls return true then the philosopher can proceed to eat. Otherwise neither fork was picked up, in which case we continue the loop, or only the left fork was picked up in which case we drop it and continue the loop. Our solution does not reach a deadlock state and does not contain any data race according to *fsanitize*.

Source code for our changes to the philosopher function on the next page!

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex out;

void philosopher(int n, std::mutex *left, std::mutex *right)
{
  // Is the left/right fork picked up
  bool l, r = false;
  while (true)
    {
      // The philosopher starts by thinking
      out.lock();
      std::cout << "Philosopher " << n << " is thinking." << std::endl;
      out.unlock();

      std::this_thread::sleep_for(std::chrono::milliseconds((rand()%6) * 150));

      // Philosopher attempts to pick up both forks
      // If philosopher can't pick up the left fork, it will not pick up the right fork
      // Note: More than one philosopher can have two forks if philosophers >= 4
      if((l = left->try_lock()) && (r = right->try_lock()))
      {
        out.lock();
        std::cout << "Philosopher " << n << " picked up both her forks." << std::endl;
        out.unlock();

        out.lock();
        std::cout << "Philosopher " << n << " is eating." << std::endl;
        out.unlock();

        out.lock();
        std::cout << "Philosopher " << n << " is putting down her right fork." << std::
            endl;
        out.unlock();
        right->unlock();

        out.lock();
        std::cout << "Philosopher " << n << " is putting down her left fork." << std::
            endl;
        out.unlock();
        left->unlock();
        l = r = false;
      }
      // If philosopher didn't pick up both forks, it's only possible that it picked up the left fork and not the right
      else if (l)
      {
        out.lock();
        std::cout << "Philosopher " << n << " only picked up her left fork." << std::
            endl;
```

```
        out.unlock();

        // Philosopher failed to pick up right fork, put left fork down
        out.lock();
        std::cout << "Philosopher " << n << " is putting down her left fork." << std::
            endl;
        out.unlock();
        left->unlock();
        l = false;
      }
    }
}

void usage(char *program)
{
  std::cout << "Usage: " << program << " N  (where 2<=N<=10)" << std::endl;
  exit(1);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      usage(argv[0]);
    }

  // philosophers = argv[1]
  int philosophers;
  try
    {
      philosophers = std::stoi(argv[1]);
    }
  catch (std::exception)
    {
      usage(argv[0]);
    }
  if (philosophers < 2 || philosophers > 10)
    {
      usage(argv[0]);
    }

  // forks
  std::mutex *forks = new std::mutex[philosophers];

  // philosophers
  std::thread *ph = new std::thread[philosophers];
  for (int i=0; i<philosophers; ++i)
    {
      int left = i;
      int right = (i == 0 ? philosophers : i) - 1;
      ph[i] = std::thread(philosopher, i, &forks[left], &forks[right]);
```

```
    }

  ph[0].join();
  delete[] forks;
  delete[] ph;

  return 0;
}
```