Uppsala University

Introduction To Parallel Programming
Assignment 1

Adolfo Hernández Signoret
Carlos Martínez

10 sept 2025

Exercise 0: Hello, World!

```
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$ g++ -std=c++11 -Wall -pth
read hello-world.cpp -o hello-world
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$ ./hello-world
Hello World!
```

Exercise 1: Concurrency and Non-Determinism

The program non-deterministic.cpp works as a simple example of how multiple threads work on a single program. The program spawns 8 threads that simply output on the terminal when they start and when they are done.

Execution A:

```
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$ g++ -std=c++11 -Wa
ll -pthread non-determinism.cpp -o non-determinism
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$ ls
assignment1.pdf  Makefile         non-determinism.cpp  shared-variable.cpp
dining.cpp       non-determinism  performance.cpp
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$ ./non-determinism
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 5 is running.
Task 5 is terminating.
Task 4 is running.
Task 4 is terminating.
Task 7 is running.
Task 7 is terminating.
Task 6 is running.
Task 6 is terminating.
Task 8 is running.
Task 8 is terminating.
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$
```

In here, each task runs and terminates mostly in order from task 1 to task 3. Task 5 started before task 4, and later task 7 was executed before task 6.

Execution B:

```
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$ ./non-determinism
Task 1 is running.
Task 1 is terminating.
Task 5 is running.
Task 5 is terminating.
Task 3 is running.
Task 3 is terminating.
Task 2 is running.
Task 2 is terminating.
Task 6 is running.
Task 6 is terminating.
Task 4 is running.
Task 4 is terminating.
Task 7 is running.
Task 7 is terminating.
Task 8 is running.
Task 8 is terminating.
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$
```

In this second execution, we can see that task 5 got done in second place; also, the order has nothing to do with the first execution, meaning the output of this program is nondeterministic.

Execution D:

```
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$ ./non-determinism
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.
sign0ret@sign0ret-msi:~/dev/student/s7/parallel-programming/assignment-1$
```

In this fourth execution, the tasks seem to be executed in order from 1 to 8, but since it has been the only time this has happened in 4 executions, it means there is no way to ensure this event happens. What is consistent is the start and end of all 8 tasks in all different executions.

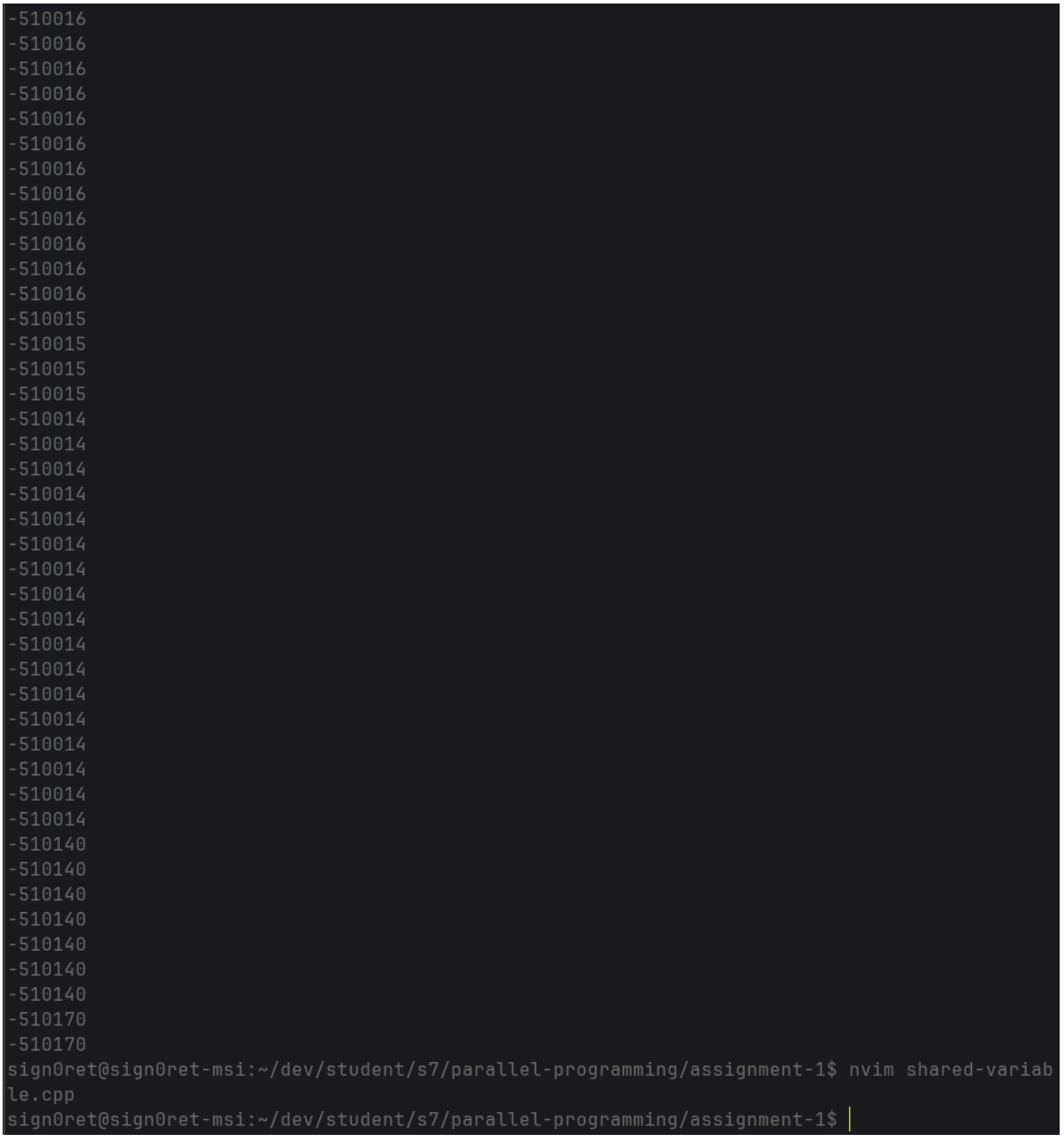
Final Observations:

From these observations, it is seen how this program cannot expect to have a deterministic output if the focus is on the order of the tasks' execution. When running this program several times, the pattern of the output becomes clear. Due to its multi-threaded nature, it's basically impossible to predict the order in which the threads will finish executing. I believe this can happen for several reasons, but I believe that is possibly because, like some threads are assigned to cores that are busier and others to faster ones, for example.

When running this program, of course, most times the first threads to finish were the first ones, like 1,2,3, but sometimes the 6th or 4th processes would finish second, which is strange. The only possible output that couldn't be reproduced in the executions made, and is curious, is if there could be a scenario in which the first process to start wouldn't be the first to finish.

Exercise 2: Shared-Memory Concurrency

Execution A:



From observing this final chunk of the output, many things can be noticed. First, as the output goes down, we notice the number printed remains the same for a while, then increases a little,

and finally decreases by a lot.. This allow us to infer that the second thread in charge of decrementing the variable by 1 overwrote the variable 510,170 times more than the first thread which is in charge of incrementing the variable by 1, or at least this is what the third thread shows, it might be that the difference was bigger, but as the many repeated prints and the huge jumps display, the third thread executed also way less times than the second thread.

Final Observations:

This program has way more variability than the previous one for the simple fact that it runs longer. The premise is simple: one counter that is shared by three threads, one adding to it every cycle, one subtracting every cycle, and the last one just prints the value of the shared variable for monitoring. After running the program several times, it can be observed that the variable remains somewhat within the bounds of -30,000 and 90,000. This program is nondeterministic again because it starts threads on infinite while loops and just stops them after a second or so; therefore, we believe that the output will be different almost every time, mainly due to the scheduling of the OS and, in general, how much each different thread manages to run.

Exercise 3: Race Conditions vs Data Races

The non-determinism.cpp script has a race condition, meaning that the threads execute at different timings, and depending on the run time of each thread depends the position in the output. The shared-variable.cpp script has a data race, in which, depending on the execution position, 3 different threads want to read the value of a shared attribute, as well as 2 writing to modify its value constantly, and simultaneously, affecting the expected execution process.

Exercise 4: Multicore Architectures

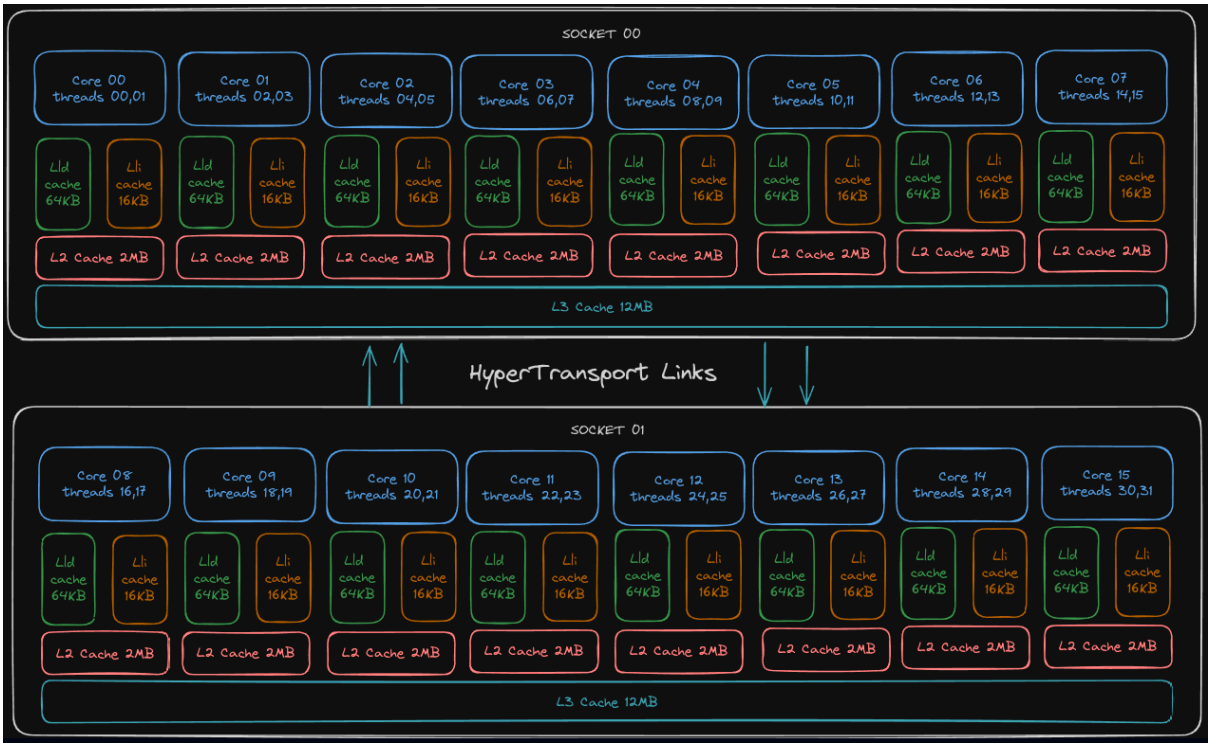
By running lscpu inside the machine of tussilago, we get the following attributes:

```
adhe6733@tussilago:~$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          48 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   32
  On-line CPU(s) list:    0-31
Vendor ID:                AuthenticAMD
  Model name:             AMD Opteron(tm) Processor 6282 SE
    CPU family:           21
    Model:                1
    Thread(s) per core:   2
    Core(s) per socket:   8
    Socket(s):            2
    Stepping:             2
    BogoMIPS:             5200,19
```

From this image, the following data is extracted:

- logical CPUs: 32
- Physical processors (sockets): 2
- Cores per socket: 8
- Hardware threads per core: 2

b. The following diagram displays a simple representation of the tussilago's AMD processor's architecture:



Differences between the Intel I5-450M and the AMD Opteron(tm) Processor 6282 SE

1. Socket level: It differentiates mainly from the Intel-based processor in that it contains multiple sockets, which need the hypertransport links to be able to communicate, allowing for vertical scaling of the compute power of the machine.
2. Core level: The AMD processor has 8 cores per socket, which is 4 times the cores than the ones inside the Intel I5-450M.
3. L1 Cache level: Each core in the AMD processor contains 64 KB of cache for data and 16 KB for instructions. This is different from the Intel architecture, which has 32 KB for instructions and another 32 KB for data. The distribution in the AMD processor gives 4 times more room for data storing, but only half for instructions, in comparison with the Intel option. This distribution of the memory in the cache benefits AMD in workloads that are data-intensive, allowing for broader, quicker access to recently used data, reducing latency. The downside would be the potential misses in the instruction cache. The reason for this unequal distribution is the focus of the machine, since it has many more threads; also, the amount of instructions would be distributed, and the concern of not having space to store them in each core becomes less of an issue. In simple terms, the cache

distribution for the L1 in the AMD processor is optimized for the types of operations that will be executed in this machine, which mostly would be distributed and ideally parallelized.

4. L2 Cache level: the AMD processor has a 2MB L2 Cache, while the Intel one has 256KB. This matches how the AMD processor aims for data-intensive routines to be executed in it, at least 8 times more per Core.
5. L3 Cache level: the AMD processor has a 12MB L3 cache memory compared to the 3MB of Intel; this is mainly based on that the AMD processor uses the 12 MB for 8 cores, not just for 2 as Intel does. If we divide the 12 MB / 8 Cores, the ratio becomes 1.5 MB per Core, the same ratio as in the Intel processor: 3 MB / 2 Cores = 1.5 MB per Core. At this level, both architectures use a similar approach.

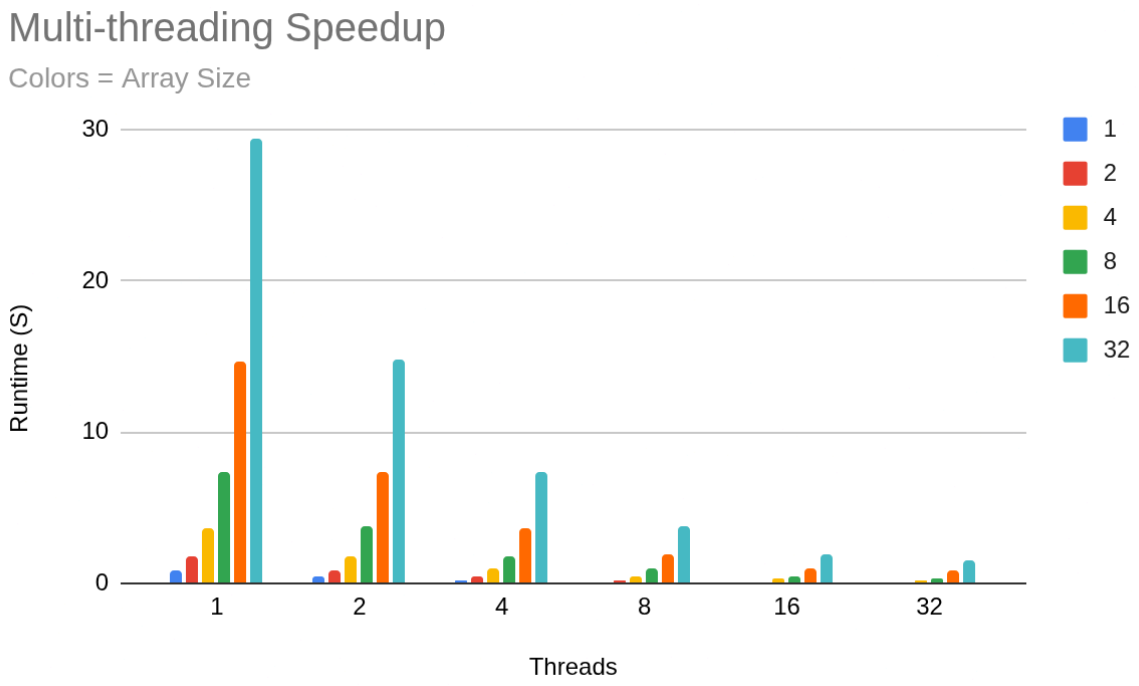
After analyzing the individual parts of both architectures, it is easy to identify the use cases for each processor. One is more of a personal computer that does simple tasks, in which the data should be almost equal to the amount of instructions; this is the recipe for a personal computer for normal tasks. The AMD architecture proposes a different usage for data-intensive applications.

Exercise 5: Performance Measurements

In the chart below is recorded the runtime of the program performance.cpp

N/Threads	1	2	4	8	16	32
1	0.916793	0.459622	0.231991	0.117166	0.0614773	0.0478791
2	1.82651	0.918728	0.46281	0.233322	0.124335	0.104987
4	3.67285	1.84662	0.929369	0.499556	0.258977	0.189396
8	7.33039	3.74571	1.84034	0.930834	0.489535	0.376511
16	14.6463	7.41049	3.69808	1.97342	1.01983	0.800992
32	29.3549	14.7588	7.41306	3.74491	1.97534	1.49789

In the graph below, the speedup achieved is demonstrated by involving more cores and multi-threading on the same program with different parameters



Exercise 6: Dining Philosophers

a. The program is getting stuck after a few iterations when it should go on indefinitely. After observing the output, it was found that it was deadlocking. All the philosophers were grabbing the left one, and when they tried to grab the right one, they couldn't; they were stuck there trying to grab the right one, so the threads stopped, and the program couldn’t move forward

b. Our solution was simple: if a philosopher couldn't get the right fork after getting the left then he would drop the left one so the other philosophers could have it. When we implemented this, we saw an immediate improvement since it was not deadlocking now; however, the program tended to livelock for a couple of seconds every now and then because sometimes the philosophers just grabbed and dropped the forks too fast.

   To solve this problem, we added a sleep timer when a philosopher stops eating and another one when the philosopher drops the left fork so others could use it. With this, the program runs a bit slower due to the timers, but it never livelocks, and all philosophers can eat. The only problem we couldn’t address was the implementation of fairness in the program

```
void philosopher(int n, std::mutex *left, std::mutex *right)
{
  while (true)
    {
      out.lock();
      std::cout << "Philosopher " << n << " is thinking." << std::endl;
      out.unlock();

      left->lock();
      out.lock();
      std::cout << "Philosopher " << n << " picked up her left fork." << std::endl;
      out.unlock();

      // try locking the right fork and if not posible drop the left one and wait so others can eat
      if(right->try_lock()){
          out.lock();
          std::cout << "Philosopher " << n << " picked up her right fork." << 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 forks" << std::endl;
          out.unlock();
          right->unlock();
          left->unlock();

          // make the thread sleep after eating to allow others to eat
          this_thread::sleep_for(chrono::milliseconds(100));

      }else{
          out.lock();
          cout<<"Philosopher " << n << " Could't get right fork so he is droping left" << std::endl;
          out.unlock();
          left->unlock();
          // make the thread sleep for a bit when unable to eat and allow others to eat
          this_thread::sleep_for(chrono::milliseconds(200));
      }

    }
}
```

This file will be submitted to Studium under the name of nicedining.cpp , to compile and execute it should be used:

g++ -std=c++11 -Wall -pthread nicedining.cpp -o nicedining

./nicedining 2

./nicedining 10

References:

AMD Opteron 6282 SE Specs. (n.d.). TechPowerUp. https://www.techpowerup.com/cpu-specs/opteron-6282-se.c1298

google. (n.d.). *ThreadSanitizerCPPManual*. GitHub. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

std::mutex - cppreference.com. (n.d.). https://en.cppreference.com/w/cpp/thread/mutex.html

Wikipedia contributors. (2025, August 8). *Dining philosophers problem*. Wikipedia. https://en.wikipedia.org/wiki/Dining_philosophers_problem