Nova School of
Science and Technology

Concurrency and Parallelism

Master in Computer Science and Engineering

2020/2021 - 2$^{\text{nd}}$ Semester

Lecture and Lab Notes

Author(s):

- Rúben Barreiro - *r.barreiro@campus.fct.unl.pt*

Last updated: March 23, 2021

Lecture 1A - Course Administrivia (March 15, 2021)

1A.1. Basic Information of Lecturers

The *professor* responsible for the *lectures* and *labs* is:

- Prof. *João* LOURENÇO - *joao.lourenco@fct.unl.pt*:
  - *Office Location*:
    - *Department of Informatics*;
    - Building II - Room P2/9, Ext. 10740;

1A.2. Discussion Forums

There will be available some *discussion forums* such as the following ones:

- *Piazza*:
  - *piazza.com/fct.unl.pt/spring2021/cp11158/home*;

1A.3. Main Bibliography

The *main bibliography* is the following:

- **Structured Parallel Programming: Patterns for Efficient Computation**:
  - *Michael* MCCOOL, *Arch* ROBINSON and *James* REINDERS;
  - *Morgan Kaufmann*, 2012;
  - *ISBN*: 978-0-12-415993-8;
  - *Click here to download*;

- **Patterns for Parallel Programming**:
  - *Tim* MATTSON, *Beverly* SANDERS and *Berna* MASSINGILL;
  - *Addison-Wesley*, 2014;
  - *ISBN*: 0-321-22811-1;
  - *Click here to download*;

- **Concurrent Programming: Algorithms, Principles, and Foundations**:
  - *Michael* Raynal;
  - *Springer-Verlag Berlin Heidelberg*, 2013;
  - *ISBN*: 978-3-642-32026-2;
  - *Click here to download*;

1A.4. Additional Bibliography

The *additional bibliography* is the following:

- **Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors**:
  - *Venkat* Suhramaniam;
  - *Pragmatic Bookshelf*, 2011;
  - *ISBN*: 978-1-934356-76-0;
  - *Click here to download*;
- **The Art of Multiprocessor Programming**:
  - *Maurice* Herlihy, *Nir* Shavit, *Victor* Luchangco and *Michael* Spear;
  - *Morgan Kauffman*, 2021;
  - *ISBN*: 978-0-12-415950-1;
  - *Click here to download*;
- **Shared-Memory Synchronization**:
  - *Michael* Scott;
  - *Morgan & Claypool*, 2013;
  - *ISBN*: 978-1-608-45956-8;
  - *Click here to download*;

- **Principles of Concurrent and Distributed Programming**:
  - *Michael* Ben-Ari;
  - *Pearson*, 2006;
  - *ISBN*: 978-0-321-31283-9;
  - *Click here to download*;

1A.5. Other Bibliography

Other *recommended bibliography* is the following:

- **Pro Git**:
  - *Scott* Chacon and *Scott* Chacon;
  - *Apress*, 2014;
  - *ISBN*: 978-1-4842-0076-6;
  - *Click here to download*;

1A.6. Syllabus

The structure of the *course* is described by the following enumerated topics:

1. **Parallel Architectures**:
   - *Flynn's Taxonomy*;
   - *Performance Theory (including Amdahl's and Gustafson's Laws)*;
2. **Parallel Programming**:
   - *The spectrum of high-demanding computational problems*;
   - *Regular and irregular problems*;
   - *Strategies for problem decomposition and their mapping to programming patterns*;
   - *The transactional and Map-Reduce models*;

3. **Concurrency Control and Synchronization**:
    - *Competition and Collaboration*;
    - *Atomicity*;
    - *Linearization*;
    - *Monitors*;
    - *Locks*
    - *Semaphores*;
    - *Barriers*;
    - *Producer-Consumer*;
    - *Multi-Reader Single-Writer Locks*;
    - *Futures*;
    - *Concurrency in Practice in Java and C*;
4. **Safety and Liveness**:
    - *Safety vs. Liveness*;
    - *Progress*;
    - *Deadlock*;
    - *Deadlock Prevention, Avoidance, Detection, and Recovery*;
    - *Livelock*;
    - *Livelock Avoidance*;
    - *Priority Inversion*;
    - *Priority Inheritance*;
    - *Lock-Free Algorithms*;
5. **The Transactional Model**:
    - *Composite Operations*;
    - *Transactions (Serializability);*
    - Optimistic Concurrency Control (OCC);
    - *Transactional Memory*;

6. **Concurrency without Shared Data**:
    - *Active Objects*;
    - *Message Passing*;
    - *Actors*;

1A.7. Evaluation

- 60% - **2 *Tests* (*Individual* and *Online*)** [average $\geq$ 8.5 points];
- 40% - **1 *Project* (Groups of 3 Students)** [grade $\geq$ 8.5 points];
- 3% - **Participation in *Classes' Life Cycle***:
    - *Lectures*;
    - *Labs*;
    - *Piazza*;

Lecture 1B - Parallel Programming Models and Architectures (March 15, 2021)

1B.1. Introduction

All *Classical Computers* are now **parallel**, specially, the modern ones, which support **parallelism** in *hardware*, through, at least, one *parallel feature*:

- *Vector Instructions*;
- *Multi-threaded Cores*;
- ***Multicore Processors***;
- *Multiple Processors*;
- *Graphics Engines*;
- *Parallel Co-Processors*;

This statement does not apply only to *supercomputers*, but even the *smallest* computers, such as *phones*, support many of these features, where, it is necessary to use explicit ***parallel programming*** to get the most of them.

The *automatic approaches* to *parallelizing serial code* ***cannot*** deal with the shifts in the *algorithm's structures* required for effective *parallelization*.

A *programmer*, in a *modern computing environment*, should not just take advantage of *processors* with *multi-cores*, but, must be able to write ***scalable*** *applications*, that can take advantage of **any** amount of ***parallel hardware***.

The feature of ***scaling*** requires attention to many factors:

- *Minimization of Data Movement*;
- *Serial Bottlenecks* (including *Locking*);
- Other forms of *Overhead*;

Some ***Parallel Patterns*** can help with this factors, but ultimately, is a responsibility of the *software developer* to produce a ***good*** *algorithm design*.

In the recent years, it were also made some progress in other *paradigm* of *computing*, known as, ***Quantum Computing***, which is also, *inherently* ***parallel***, taking advantage of the **quantum parallelism** offered on the *atomic* and*sub-atomic* levels of ***elementary particles***, such as, **photons**.

1B.2. Think Parallel

The ***Parallelism*** is an *intuitive* and *common human experience*, where, *programmers* naturally accept the concept of *parallel work* via a *group of workers*, often with *specializations*, in order to achieve a *major goal*.

Some *important* and *relevant* notions are:

- ***Serialization***:
  - Act of putting some set of operations into a specific order;
- ***Serial Semantics***:
  - *Semantics* used even though the *hardware* was naturally *parallel*;
- ***Serial Illusion***:
  - *Mental model* of a *computer* executing operations, *sequentially*;
  - Has the problem of programmers came to depend on it too much;

The ***Serialization*** has its own *benefits*, such as, if someone reads a piece of *serial code* from top to bottom, will also be able to understand the *temporal order* of operations from the structure of the *source code*.

It helps that modern *programming languages* have evolved to use some *structured control flow* to emphasize this aspect of *serial semantics*.

Unless it was intentionally injected *randomness*, the *serial programs* are ***deterministic***, doing always the same operations in the same order, giving the *same answer*, every time you run them with the *same inputs*.

The ***Determinism*** is useful for *debugging*, *verification*, and *testing*, but the *deterministic behavior* is not a natural characteristic of *parallel programs*.

Generally speaking, the *timing* of *task execution* in *parallel programs*, in particular the *relative timing*, is often ***Non-Deterministic***.

Given that *parallelism* is necessary for *performance*, it would be useful to find an effective approach to *parallel programming* that retains as many of the benefits of *serialization* as possible, yet is also similar to existing practice.

Some of the known ***Parallel Patterns*** provide *structure* but they can also avoid the existing **Non-Determinism**, with a few easily visible exceptions where it is *unavoidable* or *necessary* for a better *performance*.

When eliminating *unnecessary serialization*, leading to *poor performance*, the current *programming tools* may have many ***serial traps*** built into them.

The ***serial traps*** are constructs that make, often *unnecessary*, their *serial assumptions* and they can also exist in the *design* of *algorithms* and in the abstractions used, to estimate the respective *complexity* and *performance*.

Often, there are two main steps to *think* and *program* in *parallel*:

1. Learning to recognize ***serial traps***;
2. Programming in terms of ***Parallel Patterns*** that capture their best practices and using *efficient* implementations of these *patterns*;

The most difficult part of learning to *program* in *parallel* is *recognizing* and *avoiding* ***serial traps*** (i.e., *assumptions* of *serial ordering*).

These assumptions are so *common* that, their existence is unnoticed and most of the *programming languages unnecessarily* overconstrain the *execution order*, making the *parallel execution* of their *programs* to be difficult.

Parallelizing serial programs written using *iteration constructs*, requires to be *recognized* the *semantics* used and *convert* them to the appropriate *parallel structure* and even better would be to *design programs* using them.

1B.3. Vocabulary and Notation

The two fundamental components of algorithms are ***tasks*** and ***data***, where a ***task*** operates on ***data***, either *modifying* it in place or *creating* new ***data***.

In a *parallel computation*, the *multiple tasks* need to be *managed* and *coordinated*, where, in particular, ***dependencies*** between ***tasks*** need to be *respected*, regarding their particular *ordering* of *execution*.

The ***dependencies*** are often but not always associated with the transfer of ***data***, between ***tasks*** and can be categorised as the following two kinds:

- ***Data Dependency***:
  - ***Tasks*** that ***cannot*** be executed, before some of its required ***data*** be ready, which can be generated by other ***tasks***;
- ***Control Dependency***:
  - *Events* or *side effects*, such as, the ones happening due to some *I/O operations*, which need to be *ordered*, in time;

For the *task management*, there is the ***fork-join*** *pattern*, where can exist:

- **Fork Points**:
  - New *serial control flows* are created, by *splitting* an existing *serial control flow*, previously executed;
- **Join Points**:
  - Two *separate* executed *serial control flows* which are *synchronized*, by *merging* them together in one *serial control flow*;

In a *single serial control flow*, the ***tasks*** are *ordered*, according to the *serial semantics*, and due to the implicit *serial control flow* before and after these *points*, the ***control dependencies*** are also needed between ***fork*** and ***join points*** and the ***tasks*** that *precede* and *follow* them, respectively.

1B.4. Simple Approaches to Parallelization

Some simple approaches for *parallelization* are:

- ***Distributed Memory***:
  - *Parallel system* based on *processors* ***linked*** with a *fast network*;
  - The *processors* ***communicate*** with each other, via *messages*;
- ***Owner Computes***:
  - *Distribute* the ***data*** *elements* to *processors*;
  - Each *processor updates* its own ***data*** *elements*;
- ***Single Program, Multiple Data*** (***SPMD***):
  - All *machines* run the ***same*** *program* on *independent* ***data***;
  - The *dominant* form of *parallel computing*;

For ***Shared Memory***, the *latency* of *communication* between the *processors* is *low*, while, for the ***Distributed Systems***, occurs the opposite, with *high latencies* on the *communications* between the *processors*, where this *drawback* can be solved, by keeping some ***data*** *elements* of other *processors locally*, with what is called ***ghost cells***, *decreasing* a *latency* of its *communications*.

1B.5. Flynn's Characterization

The *parallel processors* can be divided into categories based on whether they have *multiple* ***flows of control***, *multiple* ***streams of data***, or on both:

- ***Single Instruction, Single Data*** (***SISD***):
  - *Standard non-parallel processor*, often referred as *scalar processor*;
  - The *individual performance* of *scalar processing* is important, since if it is slow, it can end up *dominating* the *global performance*;
- ***Single Instruction, Multiple Data*** (***SIMD***):
  - *Single task executing* simultaneously on *multiple elements* of *data*;
  - The number of *data elements* in a *SIMD*'s *operation* can vary from a small number, such as the 4 to 16 elements in short *vector instructions*, to thousands, as in *streaming vector processors*;
  - The *SIMD processors* are also known as *array processors*, since consist on an *array* of *functional units* with a *shared controller*;
- ***Multiple Instruction, Single Data*** (***MISD***):
  - Not particularly useful and is not used;
- ***Multiple Instruction, Multiple Data*** (***MIMD***):
  - Separated *streams* of ***instructions*** and ***tasks***, each one with its own *flow of control*, operating on separated ***data*** *elements*;
  - Defines *multiple cores* in a *single processor*, *multiple processors* in a *single computer*, and *multiple computers* in a *single cluster*;
  - Represents a ***heterogeneous computer***, as *multiple processors* using *different architectures*, present in the *same computer system*;
  - An example would be a *host processor* and a *co-processor* with *different* sets of ***instructions*** and ***tasks***;

1B.7. Software Taxonomies

Some commonly used *Software Taxonomies* are the following ones:

- ***Data Parallel*** (***SIMD***):
  - *Parallelism* that is a result of *identical operations* being applied *concurrently* on *different data elements* (e.g., *matrix algorithms*);
  - Difficult to apply it to some *complex problems*;
- ***Single Program, Multiple Data*** (***SPMD***):
  - A *single application* is executing across *multiple processes/threads* on a *MIMD architecture*;
  - Most processes execute the same code but do not work in *lock-step*;
  - *Dominant* form of *parallel programming*;

1B.8. Shared Memory (SM)

- ***Attributes***:
  - Has *global memory space*;
  - A *processor* use its own *cache* for a part of *global memory space*;
  - The *consistency* of the *cache* is maintained by *hardware*;
- ***Advantages***:
  - *User-friendly programming* techniques (*OpenMP* and *OpenACC*);
  - Has *low latency* for *data sharing* between ***tasks***;
- ***Disadvantages***:
  - Has *global memory space-to-CPU* path may be a *bottleneck*;
  - Has *Non-Uniform Memory Access* (*NUMA*);
  - The *programmer* is responsible for the details of *synchronization*;

1B.9. Distributed Memory (DM)

- ***Attributes***:
  - The *memory* is *shared* amongst *processors* via *message passing*;
- ***Advantages***:
  - The *memory scales* based on the number of *processors*;
  - The access to a *processor*'s own *memory* is *fast*;
  - Is *cost effective*;
- ***Disadvantages***:
  - Is *error prone*;
  - The *programmer* is responsible for the details of *communication*;
  - The *complex data structures* may be difficult to *distribute*;

1B.10. Software/Hardware Models

The *Software* and *Hardware* models ***do not*** need to match:

- ***Distributed Memory*** *software* on ***Shared Memory*** *hardware*:
  - ***Message Passing Interface*** (***MPI***), which is designed for ***Distributed Memory*** (***DM***) *hardware* but is also available on ***Shared Memory*** (***SM***) *systems*;
- ***Shared Memory*** *software* on ***Distributed Memory*** *hardware*:
  - ***Remote Memory Access*** (***RMA***) included within ***MPI-3***;
  - ***Partitioned Global Address Space*** (***PGAS***) *languages*:
    - ***Unified Parallel C*** (extension to **ISO C 99**);
    - ***Coarray Fortran*** (**Fortran 2008**);

1B.11. Difficulties of Parallelization

There are some common difficulties on ***Parallelization***, such as:

- The ***serialization*** causes ***bottlenecks***;
- The ***workload*** is not *distributed*;
- The ***debugging*** is ***hard***;
- The ***serial*** approach **does not *parallelize***;

1B.12. Performance

Often, one of the most common ***serial traps*** is the habit of discussing the *performance* of the *algorithms*, focusing only on the *minimization* of the *total amount* of *computational work* to be done, in *parallel programming.*

From this common habit, comes two main problems, such as:

- The *computation* itself, may not be the ***bottleneck***, since, the *access* to *memory* or its *communication* may constrain the *performance*;
- The *potential* for *scaling performance* on a *parallel computer* is always constrained by the *algorithm's span*, which is the *execution time* needed for the longest *chain of tasks* that, must be performed *sequentially*;

Lab 1 - π Approximation by Monte Carlo Method
(March 18, 2021)