Lecture 1 - Why Parallel Programming

1) Because parallelism is everywhere.
While in the past it has been exclusive to high-performance environments, today parallelism is in every user-available devices, from PCs to smartphones.
This requires studying and developing parallel computing theory, practice and methodologies rather than focusing on sequential counterparts.

2) Because heterogeneous architecture are becoming important.
Most processing elements have some kind of co-processor, e.g. GPUs which feature a large number of cores only usable for data parallel computations.
Parallel programming frameworks should be able to support these architectures in a seamless fashion.
Other examples are FPGAs (Field Programmable Gate Arrays) or even other architectures like clusters/networks of workstations (COW and NOW respectively).

3) Because code reuse is paramount.
Often we speak about problems which have already been treated, and sequential code already exists that solves those problems, or part of those.
This code may have been through years of debugging and fine-tuning, and rewriting it in a parallel fashion is simply unfeasible, nor should be done (don't reinvent the wheel).
Parallel software frameworks should support the possibility to reuse sequential code, orchestrating its execution in a parallel fashion.

Parallelism: execution of different parts of a program on different **computing devices** at the same time.
We can imagine different flows of control (sequences of instruction) that all together are a program and are executed on different computing devices.
Note that more flows on a single computing device is concurrency, not parallelism.
Concurrency: things that may happen in parallel with respect to the ordering between elements, given more flows of control any schedule of these flows can be executed with interleaving.

Mind that with **computing devices** we actually mean cores.
A single core cpu is only capable of concurrency but a single cpu with 2 or more cores could run things in parallel, even if it is a single physical device.

Lecture 2

There is a dichotomy that we want to highlight first:

- **Tool** (for parallel programming): tool for writing parallel code

    - **Threads**: smallest sequence of programmed instructions that can be managed independently by a scheduler.
      Basically, a very low level tool to implement parallel programs.
      Being a basic tool, most of the things must be handled manually, for example we must be careful about shared variables.
    - **OpenMP**: standard to insert annotations in C/C++ programs to tell the system that something has to happen in parallel.
    - **Other ways**: Pipelines (where each stage can be one in parallel with others), ...
- **Libraries**: already contains parallel code
    - **OpenCV**: library to process videos/images. Some operations (e.g. blur) can be done in parallel, since the image can be split in sections, operate on those sections and re-compose the result.

Another dichotomy is in:

- **System programmers**: create the tools for the application programmers.
  The OpenCV devs are an example of them. The focus of the course is being system programmers.
- **Application programmers**: experts of a given application domain (AI, video processing, compression...), and usually want to have libraries that provides abstractions close to their way of thinking. They don't feel the need to understand what's going on inside nor how to exploit the resources of the machine they are using.

Parallel Computing

Parallel computing is the practice of identifying and exposing parallelism in our software, while understanding the cost, benefits and limitations of the tools of the chosen implementation

Example: Book Translation

Say that we want to translate a book from English to Italian.
Let's assume that the book has m pages and we spend a given amount of time $t_{page} = 1 \text{ hour}$ to translate a single page.
The overall task will require something close to m hours to complete.
We can say that the **sequential** translation of a book will take $T_{seq} = m * t_{page}$

What if a team of helpers is available? We can imagine a much more efficient procedure exploiting n helpers:

- we divide the book in n parts
- we assign each part to a helper
- helpers translate their own part
- we rebuild the translated book

How much time is needed to translate the whole book now?
Ideally each helper will get m/n pages to translate and, assuming that everyone translate at the same speed, each helper will complete its part in $m/n * t_{page}$
The point here is that all helpers may start working independently of the other helpers as soon as they get their part.
In the ideal situation they all get their part at the same time and therefore the translation of the whole book will take $m/n * t_{page}$ amount of time.

But we need to take into account some additional activities that concur to the organization of the helpers:

- count the helpers and divide the book in as many parts
- give a part to every single helper
- after the translation we need to put together the book

The amount of time needed by the three previous tasks is clearly proportional to the number of helpers and the size of our book does not impact much.
In general we may assume that the time needed to create a part is t_{split} and the time needed to put back in the right place the translated part is t_{merge} and those are smaller than the time needed to translate a page.

However we need to take into account that our parallel procedure to translate a book takes a time $T_{par} = n * t_{split} + (m/n * t_{page}) + n * t_{merge}$

We assumed that $t_{split},\ t_{merge} << t_{page}$, therefore the gain in time is due to the fact that $T_{seq} = m * t_{page}$ is divided by n and will not be impaired by the fact that we spent an additional time to coordinate the work of the helpers.
However the time spent to coordinate the helpers is pure **overhead** with respect to sequential execution.

We can now define the **speedup**, which is the **ratio between the sequential time and the parallel time**:

$$speedup(n) = \frac{T_{seq}}{T_{par}} = \frac{m * t_{page}}{n * (t_{split} + t_{merge}) + m/n * t_{page}}$$

But since we assumed that $t_{split},\ t_{merge} << t_{page}$ and that $n << m$ than the term $n * (t_{split} + t_{merge})$ become negligible and we may compute speedup as

$$speedup(n) \approx \frac{m * t_{page}}{m/n * t_{page}} = n$$

Another important measure is **scalability**, that tells us how suitable a particular program is to be used with more computing devices:

$$scalability = \frac{\text{parallel time with 1 computing device}}{\text{parallel time}}$$

Summary:

- T_{seq} vs T_{par}
- **overhead**: in order to set up parallel execution of a task a given amount of overhead is introduced that adds execution time to the time needed to execute the task sequentially;
- **coordination**: needed to organize parallel execution of tasks, which is not needed when a single worker executes the task;
- **speedup**: ratio between the sequential time and the parallel time;
- **scalability**: how suitable a particular program is to be used with more computing devices;
- **complications**:
  - logical limitations (limitations depending on the specific instance of the problem);
  - if overhead to parallelize is too high, there is no benefit in parallelizing;
  - after a certain amount of workers, the time will not decrease anymore (as the total overhead becomes too high);

Lab: Work with Remote Machines

https://classroom.google.com/c/NTgzNzY4MzU1NjI5/m/NTk3MzUxNjMwMTc0/details

Lecture 3

Base Performance Indicators

Indicators measuring **absolute time** spent in the execution of a given (part of) parallel application:

- **Latency (L)**: the time between the moment an activity receives the input and the moment the same activity delivers the output.
- **Completion time (T_C)**: the overall latency of an application computed on a given input data set: the time spent from application start to application completion;

Indicators measuring the throughput of the application (i.e. rate achieved in the delivering of the results):

- **Throughput**: number of tasks computed per unit of time (when computing a sequence of task) (expressed in task/unit-of-time)
- ****Service time (T_S)**: the inverse of the throughput: the time needed to output the results of a task belonging to a task sequence after having delivered the results of the previous task** (expressed in units of time)

Derived Performance Indicators

- **Speedup**: the ratio between the best known sequential execution time (on the target architecture at hand) and the parallel execution time.
  Speedup is a function of n, which is the parallelism degree of the parallel execution.

$$speedup(n) = \frac{T_{seq}}{T_{par}(n)}$$

- **Scalability**: the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n.
  Note that scalability tends to grow faster than speedup.

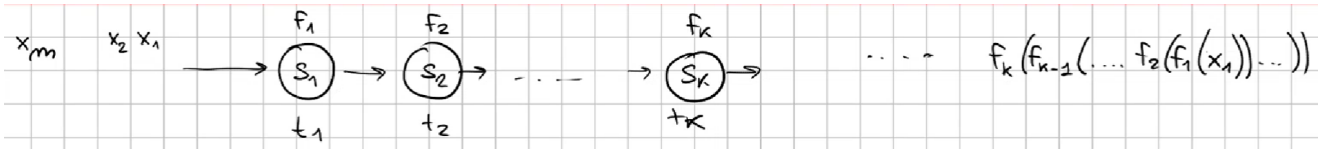
$$scalability(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

- **Efficiency**: ratio between the ideal execution time (i.e. T_{seq}/n) and actual parallel execution time. It is useful to identify the optimal number of processors to use for a given task.

$$efficiency(n) = \frac{\text{ideal parallel time (n)}}{T_{par}(n)} = \frac{\frac{T_{seq}}{n}}{T_{par}(n)} = \frac{T_{seq}}{n \cdot T_{par}(n)} = \frac{1}{n} speedup(n)$$

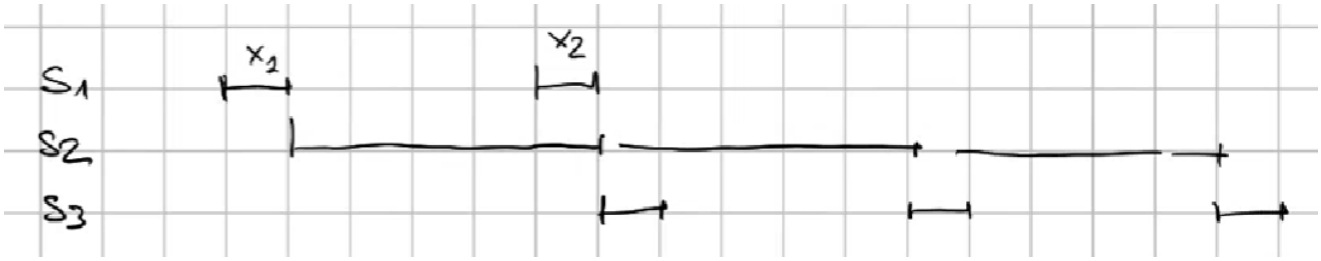
  It is important to note that different metrics may be more appropriate for different types of parallel systems and different types of tasks.
  For example, latency may be more important for real-time systems where responsiveness is critical, while efficiency may be more important for high-performance computing systems where maximizing throughput is the primary goal.

Parallelising Stages in a Pipeline



In this example, we have m inputs to feed to k stages (aka functions to be computed) of a stateless pipeline (so it does not keep memory across the stages).
In this case, the latency of the whole pipeline is: $L = \sum t_i$, as the latency is the defined as the time between the input reception and the output production.
The completion time is given by the overall latency times.

In this example, we can improve the completion time: stages in a pipeline can be computed in parallel.
Example with 3 stages:



The completion time will then depend both by the number of stages and the completion time of the slowest of all the stages:

$$T_C = \sum t_i + max\{t_i\}(m-1)$$

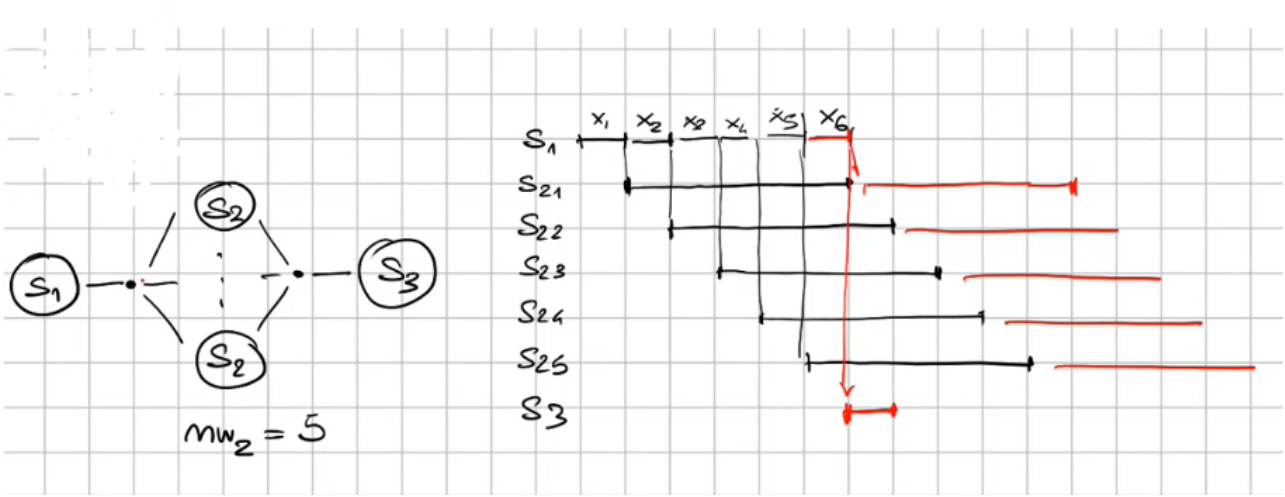
While the service time will be:

$$T_S = max\{t_i\}$$

This is an improvement because we pay the latency of each stage once, as they work in parallel the latency is amortized.
Adding the time needed by the longest state $max\{t_i\}$ executing on the last input $m-1$ is the time that hides the other stages.

If a Stage is Particularly Slow?

In the example with 3 stages, the problem is the second stage.
We can increment the performance by creating multiple instances for the second stage.
For example, assuming that this second stage is 5 times slower than the other 2, we could do something like:



We have created 5 instances of the second stage of the pipeline.
Let's better explain why we have chosen 5 by introducing an additional measure.

The **inter-arrival time** (T_A) is the time between each input arrival into the system.
In this example, $T_A = t_1 = 1$, an input arrives every 1 unit of time

Consider that each stage has a queue that contains the elements waiting to enter in the stage.

We can then compute the **utilization factor** of a queue:

$$\rho = \frac{\text{Service Time}}{\text{Inter-Arrival Time}} = \frac{T_S}{T_A}$$

In this case, the utilization factor of the queue of the 2nd stage is exactly 5.

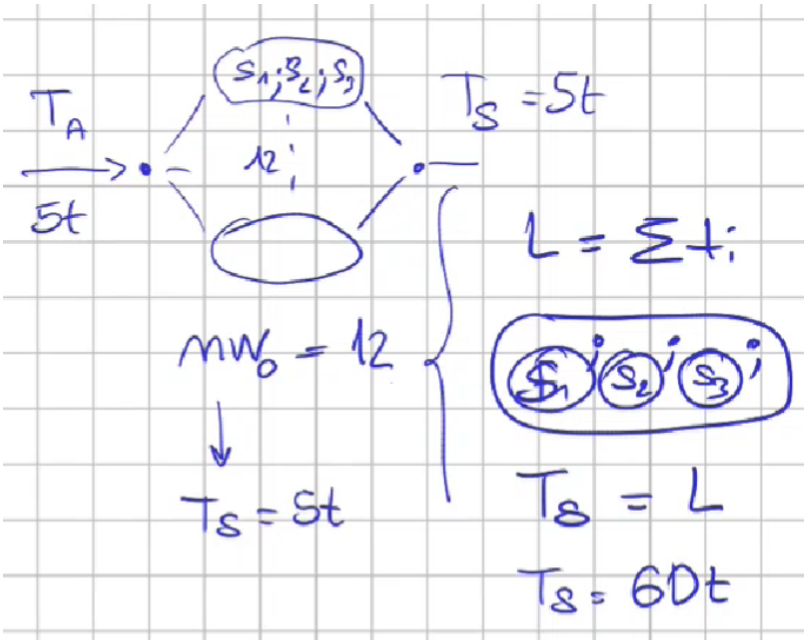
$$\rho_{stage_2} = \frac{T_S}{T_A} = \frac{5}{1} = 5$$

The idea can be further extended, by parallelizing stages differently depending on how much time they require.

Remember, in this example we didn't take into consideration that maybe a particular stage could be sped up by parallelizing it internally.

What else can we do?

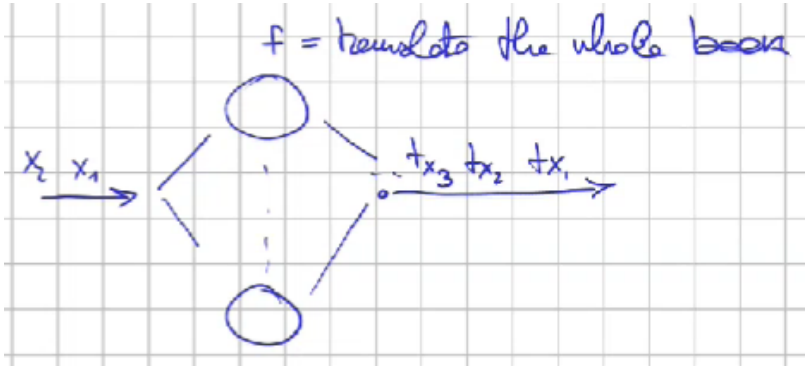
Another idea is to **parallelize the whole pipeline**, having a worker doing all the stages of it.



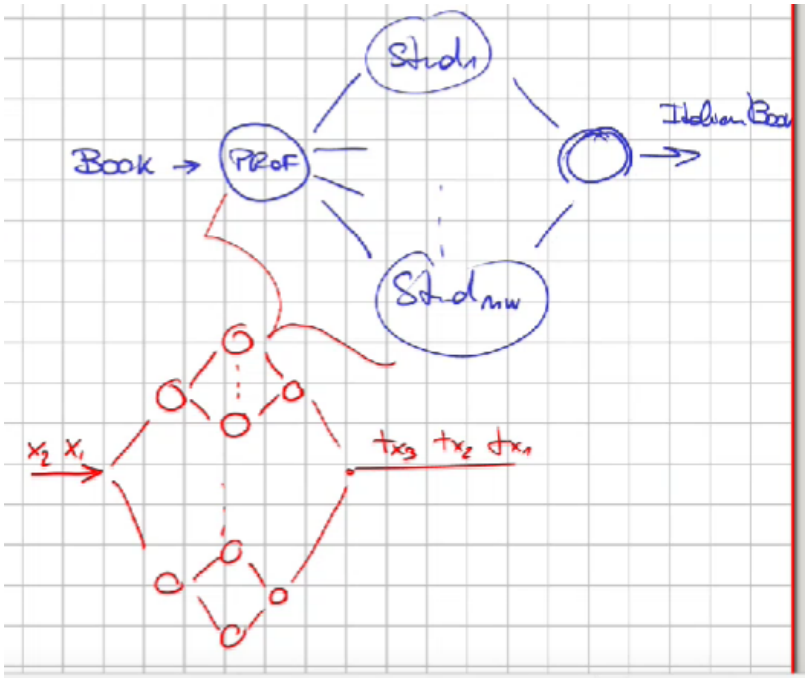
Combining Stream and Data Parallelism

Remember the example discussed in the previous lesson: given a book, we can translate it by splitting up the pages across multiple workers.

Clients sends, with an interval of T_A a request to translate a book x_i.
What we can do is to let a single book be translated by a single worker, and translating multiple books at the same time.



We can also mix this idea with the idea of the example in the previous lesson.



These two approaches focus on different parallelism:

- **Stream parallelism**: parallelism resulting from the parallel execution of different tasks available at different times on an input stream. It **increases throughput**;
  - each worker translates a book and they start translating a book just after it arrived
- **Data parallelism**: parallelism resulting from the division of an initial task in subtasks: each subtask is computed in parallel and the results are combined to obtain the result of the original task. It **decreases latency**.
  - each book is divided in parts and "subworkers" translates their part and then recombine them into the book

Depending on the particular request, we can fine-tune the degrees of the two types of parallelism.

Amdahl's Law

Amdahl's Law is a principle in parallel theory that is used to make predictions about the theoretical speedup when multiple processors are used in parallel computing.

The Amdahl's law states that any application having f of **non parallelizable** work cannot achieve a speedup larger than $\frac{1}{f}$.
Obviously the parrallelizable work is $1 - f$.
It highlights that no matter how fast we make the parallel part of the code, we will **always be limited by the serial portion**.

Proof

Assume that we have an application where:

- f of the total work is inherently sequential: it can not be parallelized;
- T_S units of time are needed by the whole application to be computed in a sequential fashion;

Say that we have n processing elements.
We use those processing elements to parallelize the $(1 - f)$ part of the program that can actually be parallelized.

Hence the best hypothetical solution is that the $(1 - f)T_S$ execution time can scaled down to:

$$\frac{(1 - f)T_S}{n}$$

Therefore the speedup we can achieve with our application is:

$$speedup(n) = \frac{T_{Seq}}{T_{Par}} = \frac{T_S}{f\,T_S\ +\ (1 - f)\frac{T_S}{n}} = \frac{T_S}{T_S(f + \frac{1-f}{n})} = \frac{1}{f + \frac{(1-f)}{n}}$$

Assuming to have infinite processing resources n available, the term $\frac{(1-f)}{n}$ clearly goes to 0, and the speedup becomes:

$$\lim_{n \to \infty} sp(n) = \frac{1}{f}$$

Therefore, even with infinite processing resources, the speedup is limited by $1/f$, □

Gustafsson's Law

We use the same notation as before:

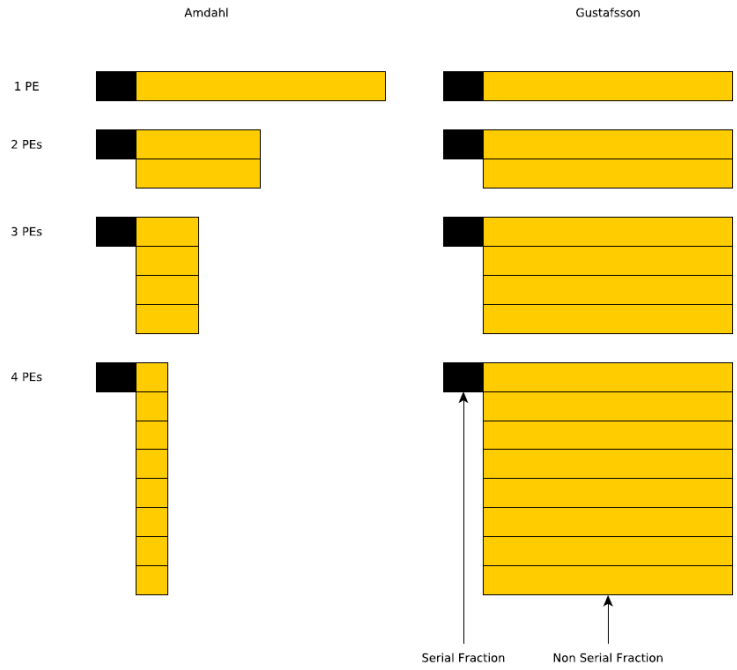
- n is the number of workers
- f is the amount of non parallelizable work
- $1 - f$ is the parallelizable work

Gustaffsson found that the speedup of a program gained by using parallel computing is

$$sp(n) = f + n \cdot (1 - f) = f + n - nf = n - f(n - 1)$$

Gustafson's law addresses the shortcomings of Amdahl's law, which is based on the assumption of a **fixed problem size**, that is of an execution workload that does not change with respect to the improvement of the resources.
Instead Gustafson propose that **the size of the problem grows proportionally to the number of processors:** programmers tend to increase the size of problems to fully exploit the computing power that becomes available as the resources improve.

The result is that a **larger problem** can be solved in the **same time** by using **more processors**.
This provides additional opportunities to exploit parallelism.



PEs stands for **P**rocessing **E**lements.
In the picture, we can see how Amdahl thinks in term of fixed size problem that can be sped up if we increase the number of workers, while Gustaffson thinks in term of how much more data we can compute in the same time by adding more workers.

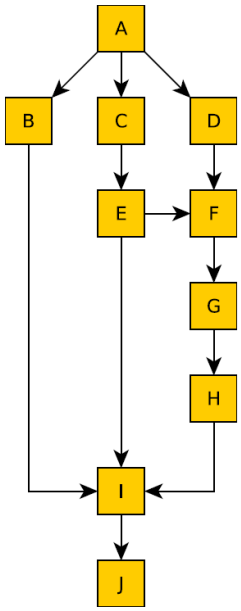
It shows more generally that the serial fraction of the program does not theoretically limit parallel speed enhancement if the problem or workload scales in its parallel components.

Said easy: Amdahl's Law limits the speedup because the size of the data that is used by the parallel part of the program remains constant.
The picture shows that with more and more computing unit we will have that execution time will be dominated by the serial part.
Instead with Gustaffson we see how we can exploit parallelism to increase the size of the data (compute more stuff).

Work-Span Model

- **Component**: a part of our application that cannot be parallelized;

- **Dependency**: given 2 components A and B, we say that $A \rightarrow B$ if the output of A is needed by B to compute its output;
- **Application Graph (C, D)**: a graph consisting of **C**omponents (nodes) and **D**ependencies (edges);



- work: the amount of work (e.g. time) required to execute the whole application
- span: total amount of work (e.g. time) to be spent executing the longest chain of components.
  Note that the “longest” chain has to be intended with respect to the target measure taken into account, in this case is time;

To execute the application in parallel we need at least the time needed for the span.
This is because the longest chain must be computed and while it is computed we assume that we can do the rest in a time $\leq$ the time needed for the span.
This is clear as the nodes are to be executed sequentially and the chain is a series of sequential task where the $i+1-th$ task need the output of the task i to be computed.

In the work-span model, the time spent computing the span chain is called T_∞, because even if you have infinite resources you cannot go better than the span chain.
In symbols:

$$T_{Par} = T_\infty$$

On the other side, **the time spent to compute the whole application with just a single processing element is clearly the time spend to compute the work**, denoted as T_1, hence

$$T_{Seq} = T_1$$

The work-span model observes that the maximum speedup we may achieve is nothing but:

$$speedup(\infty) = \frac{T_1}{T_\infty}$$

Lecture 4 - 8: C++ 101 and Parallel Patterns

These lessons focused on how to use **C++ native threads** to implement simple parallel programs.

The following are the **topics** you need to know very well, but you can simply find them online to waste less time and then look at professor's codes:

- *Thread create/join*;
- *Mutexes*;
- *Atomics*;
- *Barriers*;
- *Condition variables*;
- *Asych*;
- *Transform* (from `<algorithm>`, with execution modes);
- *Stream*;
- Jupyter notebook for C++;

Beside technicalities, professor also touched theoretical topics, which you can find in the following sections.

Vectorization

Vectorization is a way to exploit data-level parallelism by performing the same operation on multiple data elements simultaneously.

It is an optimization technique where **scalar code** is transformed into **vectorized code** that can take advantage of **SIMD** (Single Instruction, Multiple Data) instructions.
SIMD instructions allow for **parallel execution** of the same operation on multiple data elements, which can significantly improve performance.
For example, in a matrix-vector product, the instructions involving independent elements of the vector can be executed in parallel using SIMD instructions.

Many CPUs have "**vector**" or "**SIMD**" instruction sets which apply the same operation simultaneously to two, four, or more pieces of data.

"Vectorization" is the process of rewriting a loop so that instead of processing a single element of an array N times the CPU processes (say) 4 elements of the array simultaneously N/4 times.

There are two ways to vectorize a program:

- **Manual vectorization**: explicitly writing vector instructions in the code using intrinsic or other language-specific constructs;

- **Auto vectorization**: performed by the compiler itself, which automatically identifies **loops** in the code that can be **parallelized** and **transformed** into **vector operations**. It then arranges the instructions to be executed in a vectorized manner, operating on multiple data items at once.

To be vectorized the loops must satisfy the following requirements:

1. **number of iteration:** the number of iteration has to be known
   - `for` loops can be vectorized
   - `while(c<k)` may not be vectorized
2. **can't call external code in the loop body:** no functions or libraries invocation
3. **no conditional code:** conditional code would require to vectorize the two branches, complicating things
4. **no overlapping pointers:** If the pointers overlap, the compiler cannot guarantee that the computations will be executed in the correct order, which can lead to incorrect results.

High-level Parallel Patterns

Here we see parallel patterns for

- **stream parallelism**
- **data parallelism**.

Stream Parallelism

Pipeline Pattern

Given an **input stream** of m input tasks:

$$x_m,\ \ldots,\ x_1$$

And p **functions** to compute one after each other (stages):

$$s_1,\ \ldots,\ s_p$$

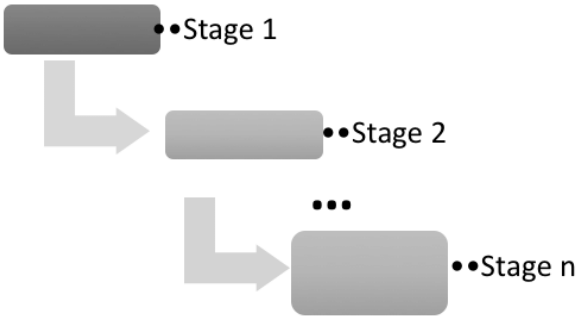
The Pipeline Pattern compute the **output stream** as follows:

$$s_p(\ldots s_1(x_m)\ldots),\ \ldots,\ s_p(\ldots s_1(x_1)\ldots)$$

The parallel semantics of the pipeline pattern ensures all the **stages** will execute in **parallel**.

The total time required to compute a single task (pipeline latency) is close to the sum of the times required to compute the different stages.

It is also guaranteed that the time needed to output a new result is close to the time spent to compute a task by the longest stage in the pipeline.



Farm

Given a **stream** of m input tasks:

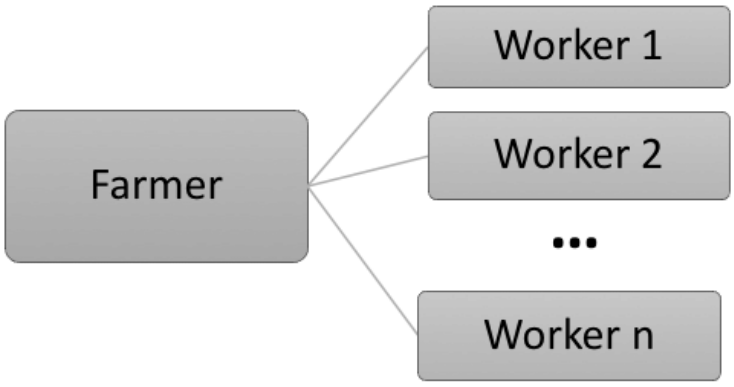
$$x_m, \ \ldots, \ x_1$$

And a **function** f to compute the output **for each input**...

The Farm computes the output stream as follows:

$$f(x_m), \ \ldots, \ f(x_1)$$

The parallel semantics of the farm pattern ensures that the single input is processed in a time close to the time needed to compute f sequentially.
The time between the delivery of two different task results, instead, is close to the sequential time divided the parallelism degree of the farm (i.e. number of workers).



Data Parallelism

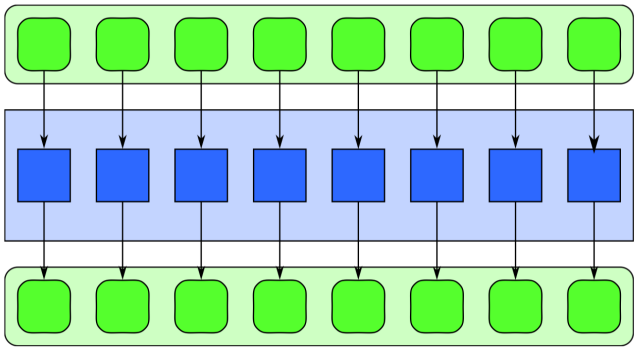
Map

Given a **function** f and a **vector** x, the map pattern computes a **vector** y whose elements are such that $\forall i \ y_i = f(x_i)$.

The parallel semantics of the map pattern ensures that **each element** of the resulting vector is computed in **parallel**.

Note that the same kind of reasoning relative to the farm parallelism degree also apply to the map parallelism degree.
The only difference is in the kind of input data passed to the parallel agents: in the farm

case, these come from the input stream; in the map case, these come from the single input data.

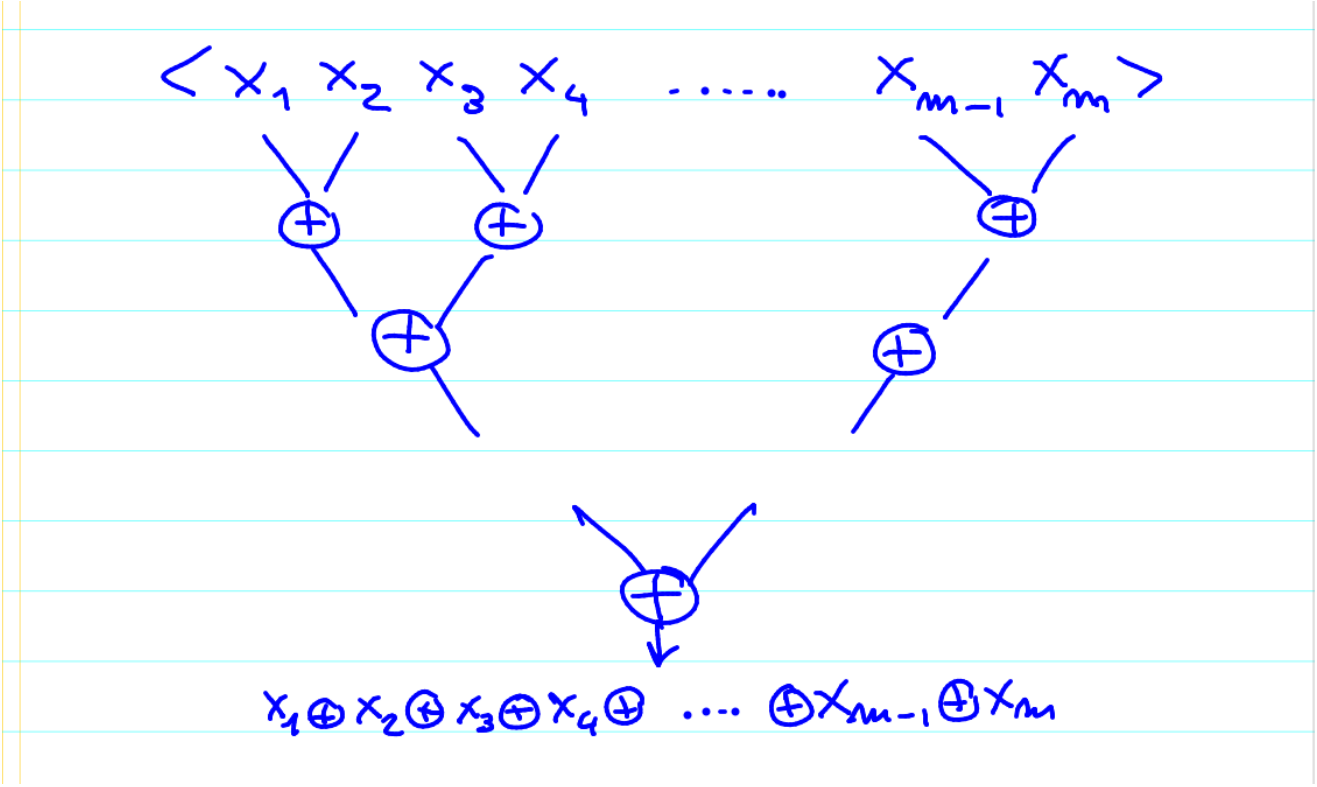


Reduce

Given a **function** $\oplus$ and a **vector** x, the reduce pattern computes a **single value** $a = x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Note that in the most general form, the function $\oplus$ passed to the reduce skeleton is assumed to be both **associative** and **commutative**.

The parallel semantics of the reduce pattern ensures that the **reduction** is performed in **parallel**, using n agents organized in a **tree**.
Each agent computes the reduction $\oplus$ over the results communicated by the son agents.
The root agent delivers the reduced result.
Leaf agents possibly compute locally a reduce over the assigned partition of the vector.



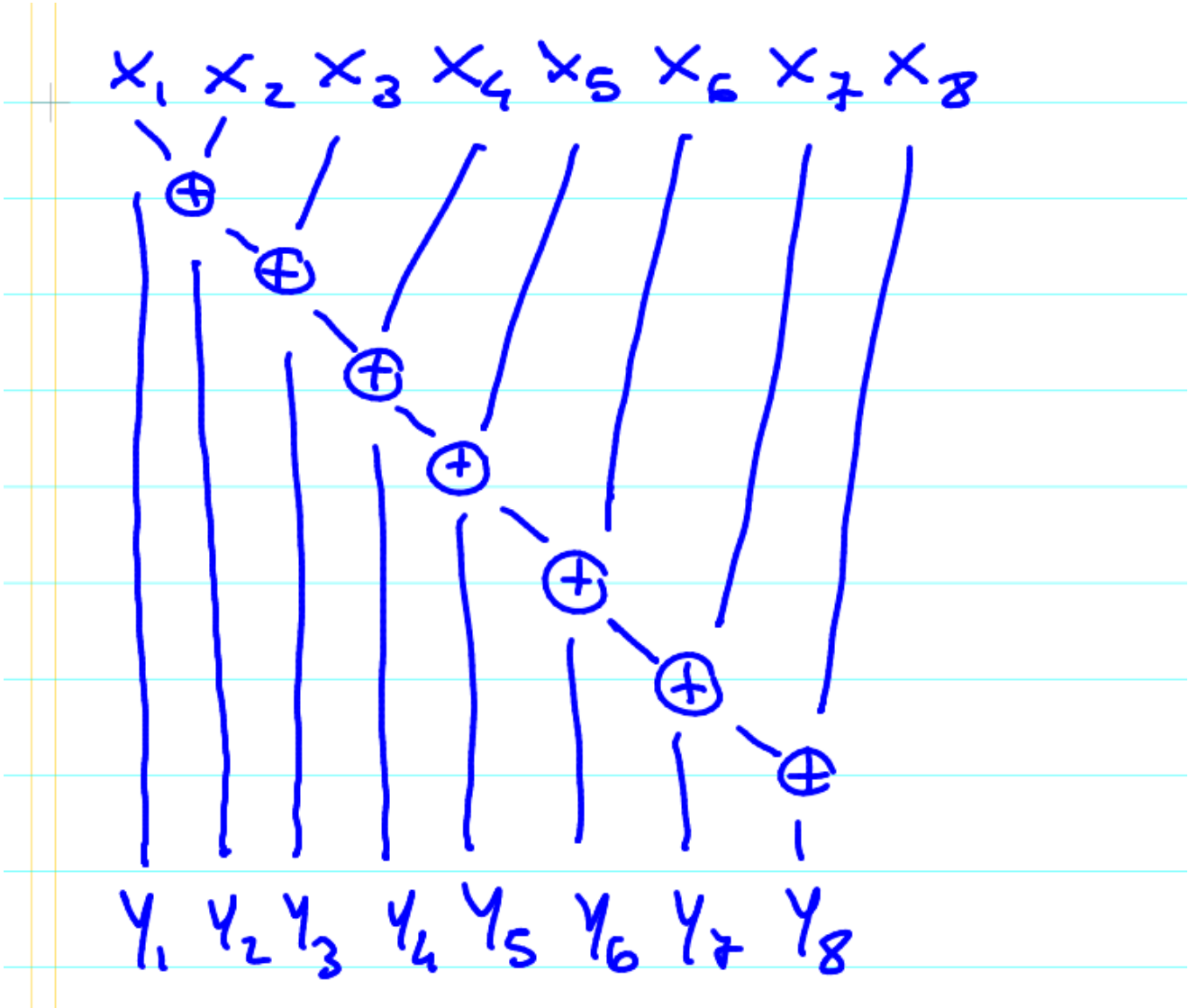
Prefix

Given a **function** $\oplus$ and a **vector** x, the prefix pattern computes a **vector** y whose elements are:

$$x_1,\ x_1 \oplus x_2,\ x_1 \oplus x_2 \oplus x_3,\ \ldots,\ x_1 \oplus \cdots \oplus x_n$$

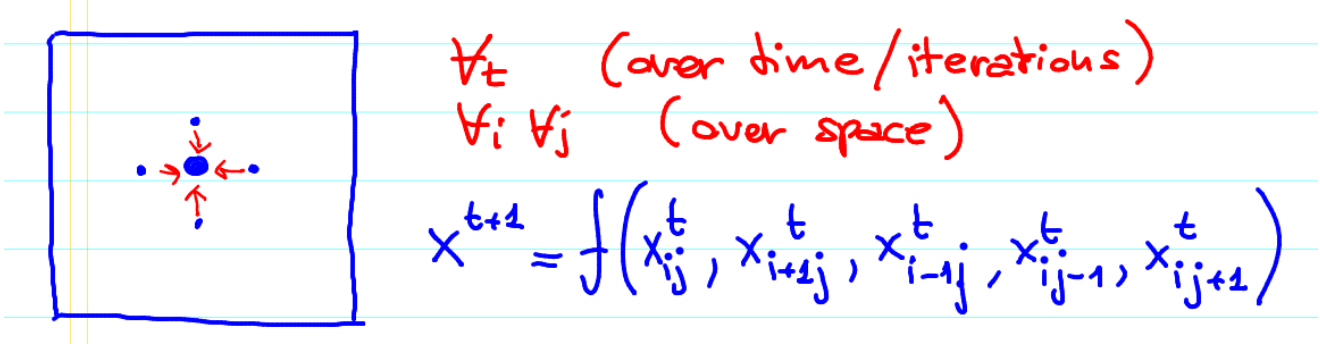
I.e. the "*partial sums*" of the vector.

The parallel semantics of the prefix pattern ensures that **prefix** is computed in **parallel** by n agents with a schema similar to the one followed to compute the reduce.
Partial results are logically written to the proper locations in the resulting vector by intermediate nodes.



Stencil

This pattern computes the new item in a data structure as a function of the old item and of the neighboring items.



The parallel semantics of the stencil pattern ensures that every x^{t+1} is computed in **parallel**.

Divide&Conquer

This pattern recursively divides a problem into sub-problems.
Once a given "**base**" case problem has been reached it is solved directly.
Eventually, all the solved sub-problems are recursively used to build the final result computed by the pattern.

The parallel semantics of the Divide&Conquer pattern ensures that the computation of **sub-problems** deriving from the "divide" phase are computed in **parallel**.

Lecture 9

Google MapReduce

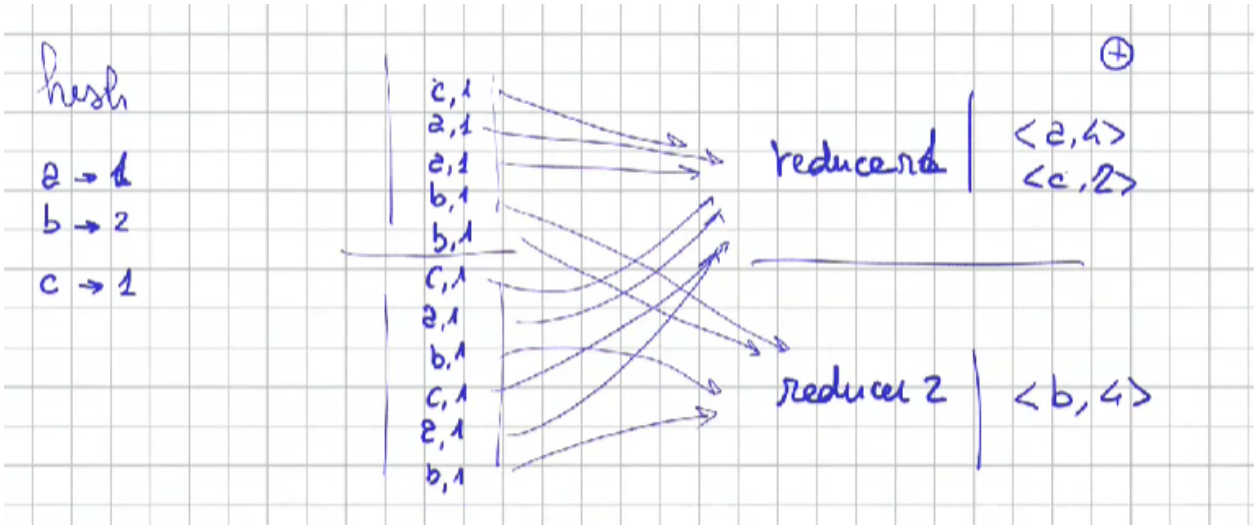
MapReduce is a programming model and an associated implementation for processing and generating large data sets.
The MapReduce model is designed to handle large-scale data processing tasks by automatically parallelizing and distributing the computation across a cluster of machines.

The MapReduce model consists of two main functions: the **map function** and the **reduce function**.
The map function takes a set of data and converts it into a set of key-value pairs.
The reduce function then takes the output of the map function and combines the values associated with each key to produce a smaller set of output values

The key contributions of the MapReduce framework are not the actual map and reduce functions (that previously existed), but rather the **scalability** and **fault-tolerance**.
As such, a single-threaded implementation of MapReduce is usually not faster than a traditional (non-MapReduce) implementation; **any gains are usually only seen with multi-threaded implementations on multi-processor hardware**.

Logical View

- `map` (f): takes a data with a type in one data domain, and returns a list of pairs in a different domain: `map(x)` $\rightarrow$ `(k,v)`;
- `shuffle`: collects all pairs with the same key and groups them together, creating one group for each key.
  Hashing techniques are typically used for the grouping, which can be performed in parallel.
  The items grouped together are then sent to the same reducer (to the same machine, using hashing to determine which), to exploit the *locality principle*.
  This is the key point of MapReduce which make the task heavily parallel;



- `reduce` (⊕): reduce is now applied to collapse all the pairs having the same key into a single pair: `reduce((k,v1), (k,v2))` → `(k, v3)`, where `v3` = `v1` ⊕ `v2`;

Consider that there are **mappers** and **reducers**, which takes as input the output of the mappers.
They are parallel agents that perform their respective tasks.
It is obvious that reducers can start as soon they get their input from at least one mapper, to optimize the computation.

Implementing characters counting

NOTE: these are very fast notes taken about how to implement character counting using MapReduce. For full details, check the lecture.

- First idea: produce `<char, 1>` pairs for every char in the input in the `map`, then send it to *reducers* machines;
- The problem with this idea is in the **communication**: it is very costly to send all these pairs;
- Second idea: after producing `<char, 1>` pairs, perform a local reduce before sending it to reducers, to reduce the cost of communication.

Legacy

MapReduce was primarily designed for **batch processing** of large datasets.
It lacks built-in support for **stream processing**, where data is processed continuously as it arrives. This limitation became increasingly important as organizations sought to process and analyse data in **real time** for immediate insights and decision-making.

Moreover, it incurs significant overhead in **reading** and writing **intermediate** data to **disk** after each `map` and `reduce` phase, which can impact performance for certain types of processing tasks.

Apache Hadoop (*discussed in the final lectures by Professor Dazzi*) mainly took over Google's MapReduce.

Map fusion

An idea to improve a composition of sequences of MapReduce comes from a theoretical result named **map fusion**.
It aims to **move** as **less data as possible** to give the final result.

The map operation applies a given function to each element of a collection, producing a new collection with the transformed elements. When multiple map operations are performed one after another, it can lead to **unnecessary intermediate collections** and redundant iterations over the collection. Map fusion aims to eliminate these overheads by **fusing** or merging consecutive map operations into a single operation.

Consider the following code:

```
val numbers = List(1, 2, 3, 4, 5)
val result = numbers.map(_ + 1).map(_ * 2)
```

Map fusion can optimize this code by merging the 2 operations into a single operation:

```
val numbers = List(1, 2, 3, 4, 5)
val result = numbers.map(x => (x + 1) * 2)
```

By fusing the map operations, the code avoids creating an intermediate collection after the first map operation.

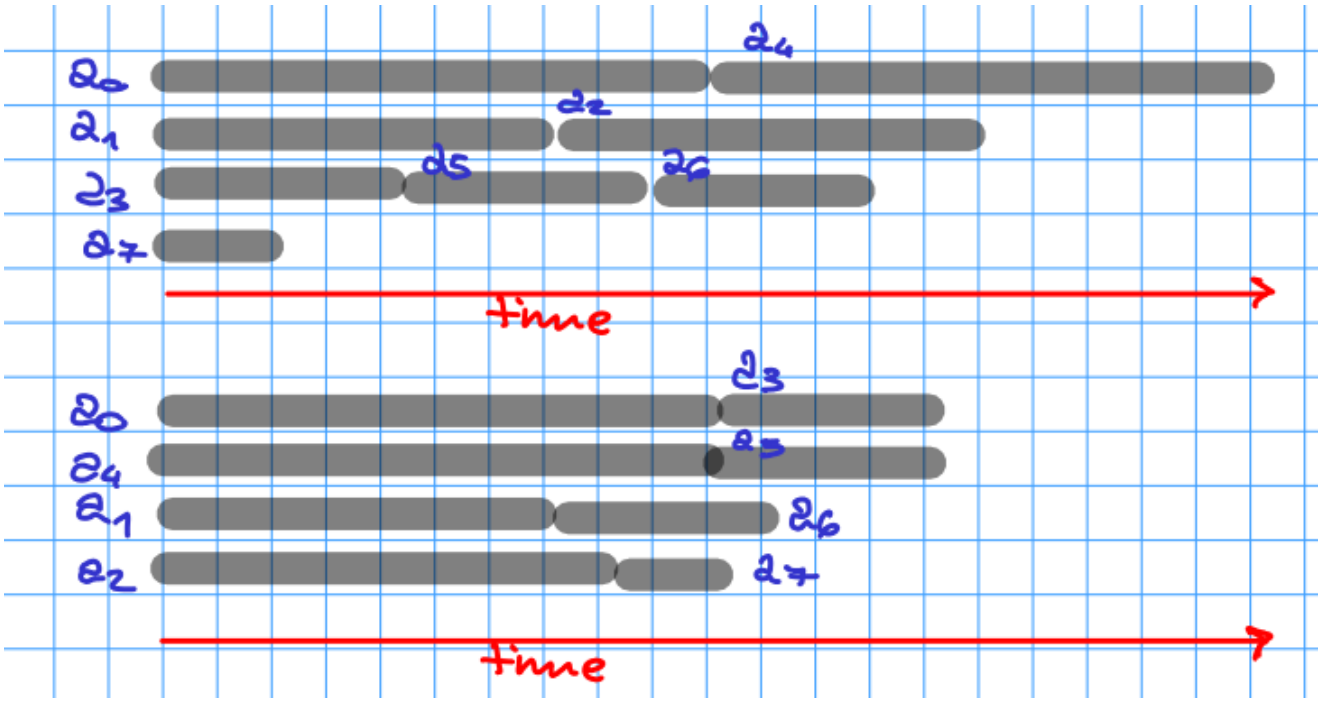
Chap 5.5 of Danelutto Notes gets in depth in this topic.

Load balancing

Load balancing is the process of distributing a set of tasks over a set of resources (computing units), with the aim of making their overall processing more efficient.

Load balancing can optimize the response time and avoid unevenly overloading some compute nodes while other compute nodes are left idle.

Example showing the alternative scheduling of activities on 4 processing elements:



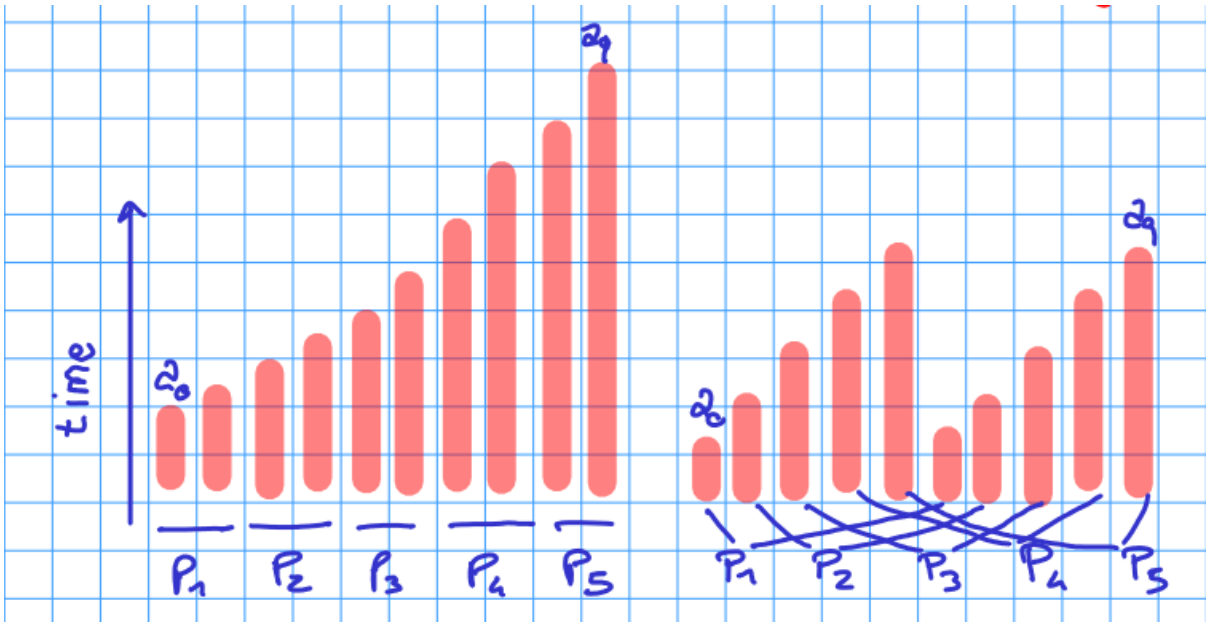
Static load balancing

In static load balancing, the scheduling **policy** is devised **before** the **computation starts** (can only rely on "compile time info").

2 commonly used policies:

- **Block distribution**: partition the activities into n chunks, and assign each chunk to a processing unit:
  - Performs better in case of **uniformly distributed** execution times;
- **Cyclic distribution**: assign activities to processing units in a **round robin fashion**, i.e. activity a_i is assigned to processing element P_j such that $j = i \mod n$:
  - Performs better when "**hot spots**" are present.
    Hot spots are portions of concurrent activities with consecutive indexes that require execution times higher (or lower) than the average execution times.
    This is intuitive: having consecutive indexes we are assured that the modulo operation distributes evenly such activities to different processing elements.

As one can easily think, bad time distributions exists that can make these policies perform badly.



Dynamic load balancing

In dynamic load balancing, the scheduling policy is devised while the computation is running (can rely on both "compile time" info and "run time" info).

Auto-scheduling

In auto-scheduling, **tasks are not assigned to the processing units, but instead the idle processing units will ask for a task to compute to a centralized or distributed concurrent activity repository.**

It is worth noting that this load balancing strategy works both for:

- Activities with quite **different execution times**;
- Processing elements with **different compute capabilities**;

Auto-Scheduling Dependency Handling:
In case there are dependencies among the concurrent activities, these dependencies have to be recorded in the repository and the entity answering the computation requests from the idle processing elements must take care to answer only concurrent activities whose input dependencies are satisfied.
In turn, once a computed concurrent activity is completed, the solved dependencies must be communicated as well, in such a way the entity assigning the concurrent activities to the processing elements could properly record the event in the dependency graph.

A little con of the auto-scheduling is that the computing units have to stay idle until they get a task assigned.
A simple way to tackle this problem is to assign two activities at a time, such that when I finish computing the first one, I can send a request for a new one while I compute the second one.
However, assigning two concurrent activities to processing elements could impair the load

balancing policy: a single processing element may receive two “long” concurrent activities while the other ones may get only “short” ones.

If it is not enough clear, Chap. 5.6.2 of Danelutto Notes goes into details with a lot of examples.

Variable Chunks

Given an initial set of concurrent activities to be computed an initial large portion of the activities is assigned to processing elements statically with either a block or a cyclic policy. The rest of the concurrent activities to be computed is divided in smaller and smaller chunks that are assigned to those processing elements that finish their previous assignments.

As an example, consider the assignment of 100 concurrent activities to 4 PEs.
We may initially assign the first 64 concurrent activities in static, block chunks.
The remaining 36 of the concurrent activities may be divided into 4 chunks of 6 concurrent activities and 4 chunks of 2 concurrent activities and 4 chunks a single concurrent activity.

In presence of differences in the times spent in the computation of each one of the concurrent activities the processing elements terminating their assignment will get more tasks to compute with a smaller probability of load imbalance due to the smaller size of the chunks.

Said easy: the at first we give to everyone the same amount, then we give less and less to the faster ones.

Job Stealing

Very popular technique: basically, PE can take tasks from other PEs if they don't have anything to do anymore.

In details, we need to define:

- **Assignment of activities to PEs**: any policy previously described can be used, but in general a static policy is preferred;
- **Method to list PEs still having something to compute**: it is not easy, and 2 possible approaches may be:
  - **centralized entity** to which PEs can report updates: easy to implement, but introduces a SPOF (single point of failure) and a bottleneck;
  - **heuristic**: the idle PE will decide the PE to steal at by random or using other heuristics. It does not guarantee that the chosen candidate will have something to get stolen, but it's trivial to implement. Stealing at random is usually the policy implemented;
- **A protocol for the stealing**: the victim must handle "stealing requests" (e.g. with a concurrent activity beside the executor one), and also any task queue must be synchronized (as multiple PEs may try to steal at the same time).

While this technique can be particularly effective, it is difficult to implement because it is necessary to ensure that **communication** does not become the **primary occupation** of the processors instead of solving the problem.

Lecture 10 - 11

Overheads

Anything added to the sequential code is overhead and must be reduced as much as possible.
In the following we show the **sources** of overheads.

Communications

There are three types of communication overhead:

- **Inter-threads**: usually it is sufficient to send pointers.
  But even using simple mailboxes is costly, as they are usually implemented with a shared data structure that needs to be thread-safe (and thus, adds overhead).
  In order **to minimize the overhead**, we can:
  - **light (overhead) communication** mechanisms if the granularity (work load) of concurrent activities is small (as for those with higher grain, this overhead will have less impact);
  - **Pack messages** as much as possible: sending one big message usually require less time than sending 3 smaller distinct messages;
  - **Change algorithm**: we may choose an algorithm that compute slower but performs less communication;
    Moreover, **lock-free** mechanisms can be implemented (e.g. *FastFlow* communications use lock-free message queues, generally boosting the speed by a 1000 times).
- **Inter-process**: usually it requires to send a **copy** of the **message**, as the address space are separated.
  However, *zero-copy* communications can be implemented to save CPU cycles and memory bandwidth (details here), *but not discussed in class*);
- **Distributed systems**:
  - **Consider a 2-stages pipeline:**
    If we compute the 2 stages in 2 **different machines**, we will need to transfer data each time. Instead, **data locality** (both **spatial** and **temporal**) should be enforced as much as possible: in this case, we could create 2 pipelines each performing the 2 stages sequentially, without impacting the parallel degree;
  - **Consider a n-stages pipeline distributed across different machines**, and to have hardware supporting parallelism in between communication and computation. Then, the time spent on each stage will be given by the sum of the times spent on **receiving**, **processing** and **sending** the data.

Given our assumption, a technique named **triple buffering** can be used to improve the performances:

- Set up **3 threads** ($T_{\text{rec}}, T_{\text{proc}}, T_{\text{snd}}$), working respectively on receiving, processing and sending, the threads are sharing 3 distinct buffers (B_0, B_1, B_2);
- There is initially a phase to reach regime:
  - As soon as the 1st task arrives, T_{rec} will receive the input on B_0;
  - Once the 1st task has been received, T_{proc} can start computing with the newly arrived input in B_0 and T_{rec} can receive the 2nd input on B_1;
  - Once the 1st task has been computed AND the 2nd input arrived:
    - T_{rec} will **receive** a 3rd input on B_2;
    - T_{proc} will **compute** the 2nd input in B_1;
    - T_{snd} will **send** the 1st output in B_0;
- At this point we have reached regime, and the 3 threads will keep going in a synchronized way, every time working on a different buffer in a cyclic way.
  The final time will then be given by: $\max(T_{\text{receive}}, T_{\text{process}}, T_{\text{send}})$.
  If T_{process} is greater than the other 2, we are talking about "**communication hiding**".
- As a final point, a technique named **double buffering** can be used if we have the possibility to run only one type of communication (send or receive) at the same time. In this case, we use 2 threads and 2 buffers only, thus leading to **partial communication hiding**:
  - B_0 for receiving and B_1 for computing/sending, OR
  - B_0 for receiving/computing and B_1 for sending;

Shared Memory Issues

- **Memory (de-)allocation**: concurrent access to the **heap** leads to overhead caused by **synchronization**.
  A classical technique to address this issue is **per-thread pre-allocation**, consisting in **allocating large memory chunks to each thread and using it instead of the global heap**.
  It is important to remark that managing this "per-thread heap" is not an easy task, and it should be delegated to proper libraries (e.g. `jemalloc`, the professor explained in class how to use it);
- **False sharing**: when memory caching is used, multiple threads may unintentionally **share** the **same cache line** while **modifying different data**. Consider the code used by these 2 threads:

```
int A;
// ... code executed by Thread 1
A = 10;

int B;
```

```
// ... code executed by Thread 2
B = 20;
```

In this scenario, if the variables A and B are located close to each other in memory and happen to reside in the same cache line, the processor's caching mechanism may treat them as a **single unit**.
When Thread 1 modifies A, the cache line is loaded into the processor's cache, including both A and B.
If Thread 2 subsequently modifies B, it will invalidate the cache line and cause Thread 1 to reload it, even though Thread 1 is not accessing B at all (aka, the cache line is now "wrong" because thread B modified it).
This **reloading process** (which needs to be propagated to all the cache levels by some **cache coherency protocol**) introduces unnecessary **overhead** and can significantly degrade performance.
This problem can be easily solved by looking at the cache size of the processor and introducing some **padding** (trivially by allocating some empty space before/after the variable we are interested in) or also by **re-arranging data**.
NOTE: the professor went into more details on how caching is usually implemented (and also the notes goes into these details), I skipped it since it should be trivial, it is mostly about different level of caching, write-through vs write-back mechanisms...

- **Thread moved from a processor to another**: it could be that the OS decide to move a thread to a different processor in order to optimize the computation, but it may not be optimal in some cases (e.g. the local cache of the new processor does not contain anything about the incoming thread).
  We can address this issue by **thread-pinning** to a particular core.

Setup of concurrent activities

The cost to start a thread and wait its termination (fork-join model) is in the range of 10 to $100\mu s$ on state of the art multicores.
Common techniques to minimize the cost needed to set up a collection of concurrent activities include usage of **thread pools** and **asynchronous tasks**.

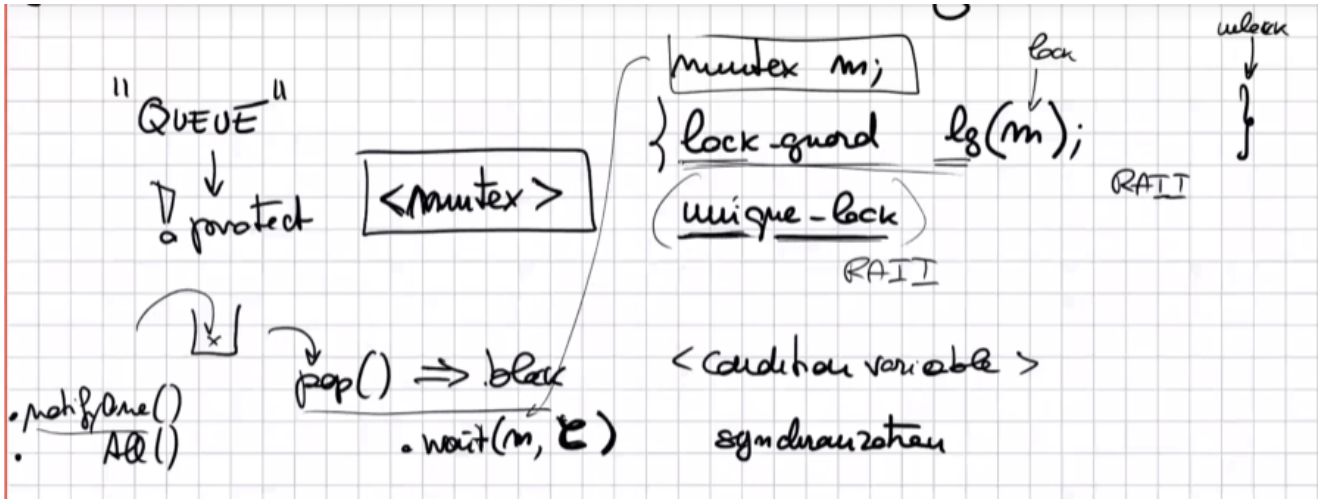
Thread pool

A **thread pool** maintains multiple threads waiting for tasks to be allocated - for concurrent execution - by the supervising program:

- Threads are **created once** and destroyed all after the parallel computation is terminated;
- Workers obtain tasks to compute from some centralized or decentralized task repository, thus implementing any of the **load balancing** techniques previously seen;

We must take care in implementing the **queue** managing the tasks: it must handle concurrency, for example using **mutexes** (to grant exclusive access) and **condition variables** (to implement the waiting).

Details:



The tasks in the queue are composed by **data** and **code**, and produce a **result**.
Thread pools may improve performances of plain sequential code where **function call** may be clearly identified as **independent** and, therefore, possibly concurrent.

Lecture 12: Implementing Frameworks

NOTE: professor only briefly stated what the 2 standards are without getting too much into details. To better understand it, I briefly summarized what can be found on the Danelutto Notes.

Algorithmic Skeletons were defined as pre-defined patterns encapsulating the structure of a parallel computation that are provided to the user as building blocks to write applications.
Each skeleton corresponds to a single parallelism exploitation pattern.
As a result a skeleton-based programming framework was defined as a framework that gives to programmers algorithmic skeletons that may be used to model the parallel structure of the application at hand.
In the skeleton based programming framework the programmer has no other way but instantiating skeletons to structure his parallel computation.

To **implement** parallel programming **frameworks**, there are 2 standards:

- implementation templates
- macro data flow

Implementation Templates

Implementation template is the model used in the firsts algorithmic skeletons framework implementations.

In a template-based implementation of a skeleton framework each skeleton is eventually implemented by instantiating a "process template".

We use the term "process" because historically the typical concurrent entity was the process.

A **process template** is a parametric network of concurrent activities, specified by means of a graph $G = (N, A)$.
Each node represents a concurrent activity and each arc denotes a communication moving data (or pure sync signals) between two nodes.
Each node has associated an informal "behavior", that is the program executed on that node.
The parameters of the network may vary, as an example the parallelism degree is often a parameter.

A process template also has some further **metadata** (e.g. target architecture, a description of the performance achievable).

The **compilation** then can then be obtained as follows:

1. Build a **skeleton tree**: a hierarchical outline of the activities and their relationships, providing an overview of the process flow without including detailed information about each task;
2. **Template assignment**: a process template is assigned to each skeleton node such that the process template is suitable and it possibly demonstrates better performances than any other template;
3. **Optimization**: the skeleton tree annotated with templates is visited again to **fill-up the parameters** of the templates;
4. **Code generation**: the framework is generated, comprehending codes in a high-level language with calls to suitable communication/synchronization libraries (e.g. OpenMP), as well as makefiles, metadata...

Macro Data Flows

The idea is to denote models where the **order of the computation** is dictated by the **data flow** rather than by PC, and where we can abstract complex functions (thus "macro").

Rather than compiling skeletons by instantiating proper process templates, the skeleton application are first compiled to **macro data flow graphs** and then instances of this graphs (one instance per input data set) are evaluated by means of a distributed macro data flow interpreter.

A *macro data flow graph* is a graph where **nodes** contain a function to compute, a list containing inputs and instructions on where to put the output, while an **arc** denotes that a data item (output token) is moved from one node to the other. For every input to the program, a copy of the graph is created, and the workers will operate on it in order to run the

program. Note that it is more complex to maintain (synchronization issues also have to be taken into account, such as workers competing for getting a fireable node). *NOTE: details on this type of graph and issues can be found on Danelutto notes.*

Template-based implementations achieve **better performances** when computing tasks with **constant execution time**, as the template based implementation has **less run-time overhead** with respect to the macro data flow based implementations.

Porting of MDF-based frameworks to new target architectures is **easier** than porting of template based implementations. In fact, MDF implementation porting only requires the development of a new MDF concurrent interpreter, whereas porting of template based implementations requires a complete rewriting of the skeleton template library, including performance models of the templates.

Template based implementation are more suited to support **static optimization techniques**, such as those rewriting skeleton trees, as a substantial compile phase is anyway needed to implement skeletons via templates. The very same optimizations in MDF-based implementations require some additional effort due to the fact the run-time has actually no more knowledge of the skeleton structure of the program it is executing: it just sees MDFI (basically nodes of the graph) from a MDFG.

Lecture 13

Refactoring

Refactoring means rewriting a program, obtaining a new one with same functional semantic but different non-functional semantic, e.g. service time, latency, completion time, lower energy consumption, lower memory, better security...

Patterns

We can organize a parallel application as a composition of **patterns**.
The patterns are given by the following **grammar**:

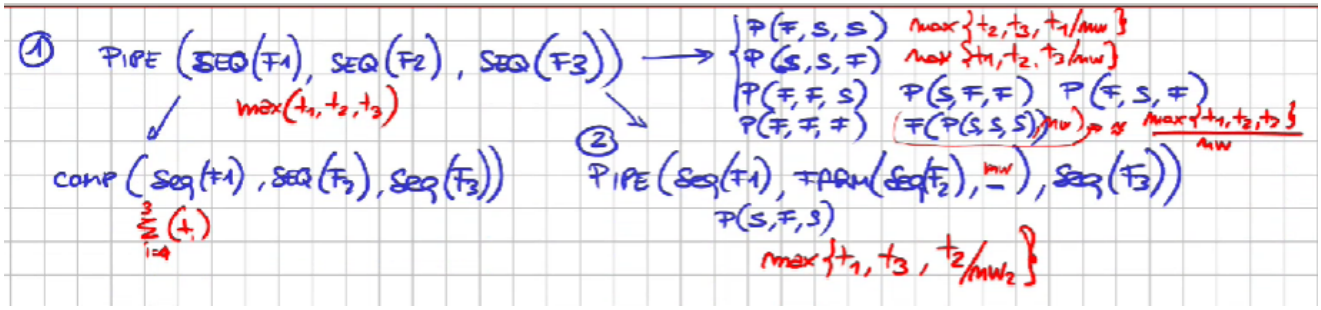
$Pat ::= Seq(f) \mid Pipe(Pat1, Pat2) \mid Comp(Pat1, Pat2) \mid Farm(Pat, nw) \mid Map(f) \mid Reduce(\oplus)$

NOTE: comp is the sequential composition of 2 patterns.

Rewriting rules

- **Parallel Degree changing**: $Farm(x, n) \equiv Farm(x, m)$, with $m \neq n$ (different parallel degree);
- **Pipeline elimination ($\rightarrow$) / introduction ($\leftarrow$)**: $Pipe(x, y) \equiv Comp(x, y)$;
- **Farm elimination ($\rightarrow$) / introduction ($\leftarrow$)**: $Farm(x, nw) \equiv x$;

Example:



NOTE: these were the rules illustrated during the lecture, but there are many others, that can be found in section 11.2 of Danelutto Notes.

Performance Model

A model can be seen as a **function mapping** a **pattern** to a set of **non-functional features**.

Rewriting rules can be exploited to improve the performance of the model, and then we can evaluate the newly rewritten program using PMs.

Exploring all the possible rewritings could take a large amount of time, and employing **heuristics** (such as performing a greedy search, where at each step we only choose to explore the path leading to better performances) could lead to falling into local minima. Thus, generally, we generate the tree **up to a certain depth** k and then we evaluate all the possibilities, getting the one maximizing/minimizing our performance model.

RPLShell Tool

Is a programming tool that allows to **define** and **apply rewriting rules** and performance rules (loop unrolling, caching, algorithmic improvements...) to manipulate and optimize programs.

NOTE: in the second part of the lecture, professor showed how to use RPLShell. I don't think it is object of the oral exam, but if you are interested you can watch the lecture yourself.

Lecture 14 - 16

OpenMP

OpenMP is an API that supports **multi-platform shared-memory multiprocessing programming**.

It is compiler-based, and it basically adds to sequential programming languages **annotations** by adding "pragmas", as `#pragma omp ...`, thus requiring few lines of code to parallelize.

The following are the **most important** pragmas, to append to `#pragma omp` :

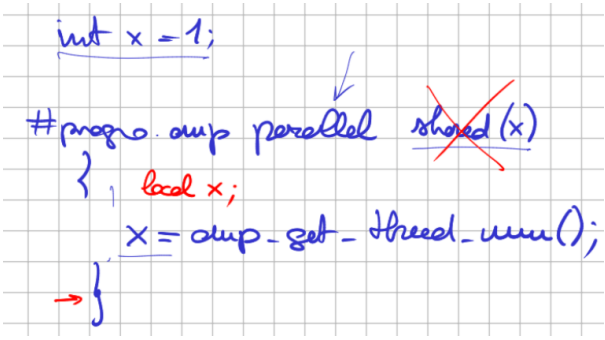
- `parallel`: execute statement in parallel with as many threads as the ones available. Optionally: `parallel num_threads(...)`;
- `critical`: entering a critical section;
- `single`: a single thread can execute the code;
- `atomic`: atomic section;
- `sections` followed by 1 or more `section`: distributes work among threads;
- **Work-sharing pragmas**:
  - `for`: independent loop iterations inside the block are distributed among nw threads. *NOTE: it has to be preceded by parallel to distribute threads among the cores of the processor.* The iterations are grouped into chunks, which are distributed depending on the specified [policy](OpenMP - Scheduling(static, dynamic, guided, runtime, auto) - Yiling's Tech Zone | 风逝无殇的瞎逼逼基地 (610yilingliu.github.io)) (to write after `for`):
    - `static[, x]`: chunks (of optional size `x`) are distributed among threads from the beginning;
    - `dynamic`: only a part of the chunks is distributed at the beginning, while the other can be requested as soon as the thread has finished;
    - `guided`: as dynamic, but size of the chunks is not fixed, and it is set to be smaller and smaller;
    - `auto`: decision delegated to compiler;
    - `runtime`: decision inherited from an environment variable to be defined;
  - `task`: the following code region can be scheduled for execution by threads. *NOTE: the parallel construct creates a set of implicit tasks.* If we want to wait for the end of the tasks, there is the primitive `taskwait`. There are pragmas that are related to it as well:
    - `task loop`: creates a task for the iterations of a following loop. It is useful because it accepts a `grainsize` parameter, that allows to combine several loop iterations into a single task;
    - `task group`: all tasks generated inside a `taskgroup` region are waited for at the end of the region;
      NOTE: Danelutto spent a lot of time on examples of tasks in [lecture 15](spm-2223-15.pdf - Google Drive).

You can also get **runtime information** through `omp_get_wtime()`.

You can get/set the number of **threads** through `omp_get_num_threads()` and `omp_set_num_threads()`. You can get the thread ID through `omp_get_thread_num()`.

You can also specify the **scope** of the **variables** in most pragmas, by appending:

- `shared(...)` (default): variable shared between thread. *Must be protected during the critical section*;

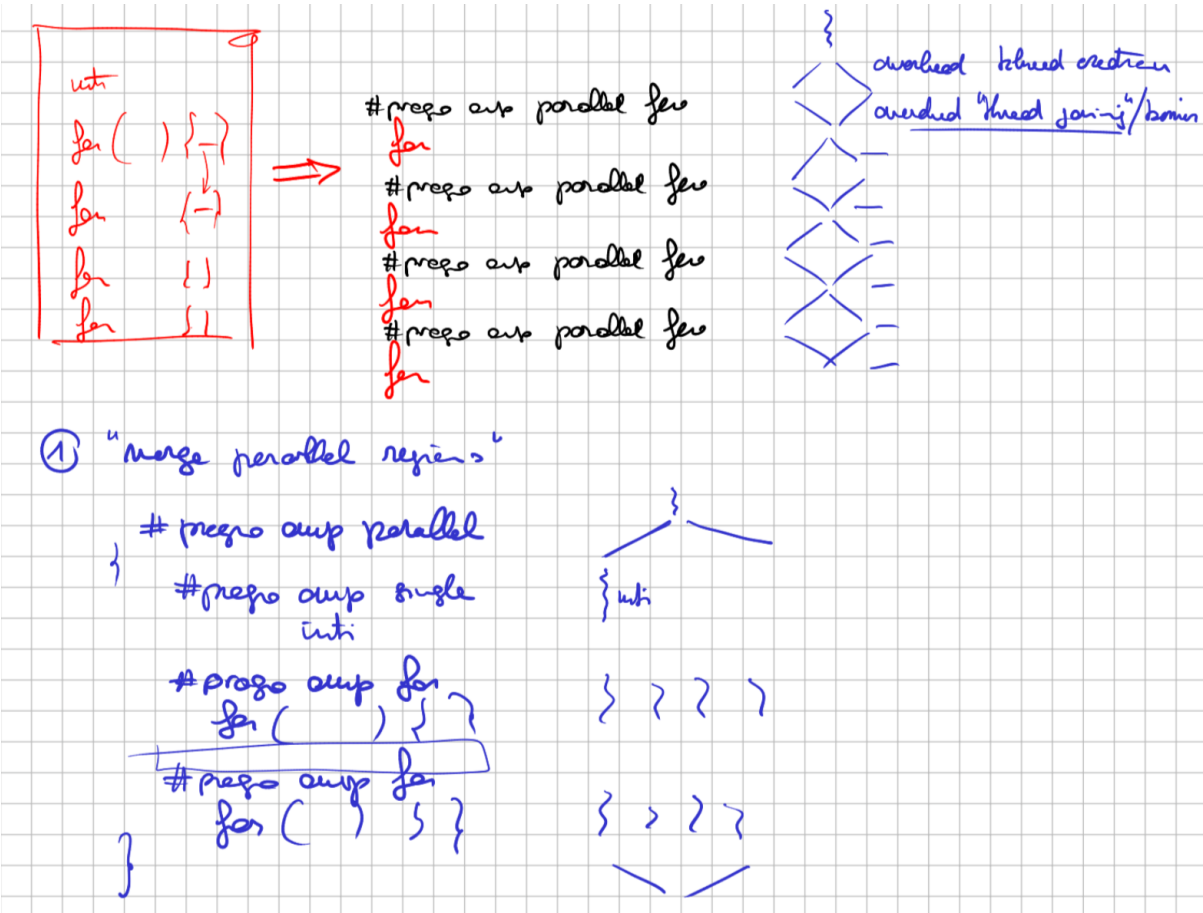


- `private(...)`: non-initialized local copy of a variable for each thread;
- `firstprivate(...)`: initialized (with the value before the pragma) local copy of a variable for each thread;
- `lastprivate(...)`: non-initialized local copy of a variable for each thread, but the value of the variable after the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section;

There are **locking primitives** as well: `omp_init_lock(&...)`, `omp_set_lock()`, `omp_unset_lock()`;

Optimization of OpenMP code

- Merge parallel regions;



- `#pragma omp for nowait`, removes the implicit barrier to wait all the threads, present in the parallel pragma;
- Make variables private as much as possible;

Lecture 17

GRPPI

C++ library that implements common **parallel patterns**.

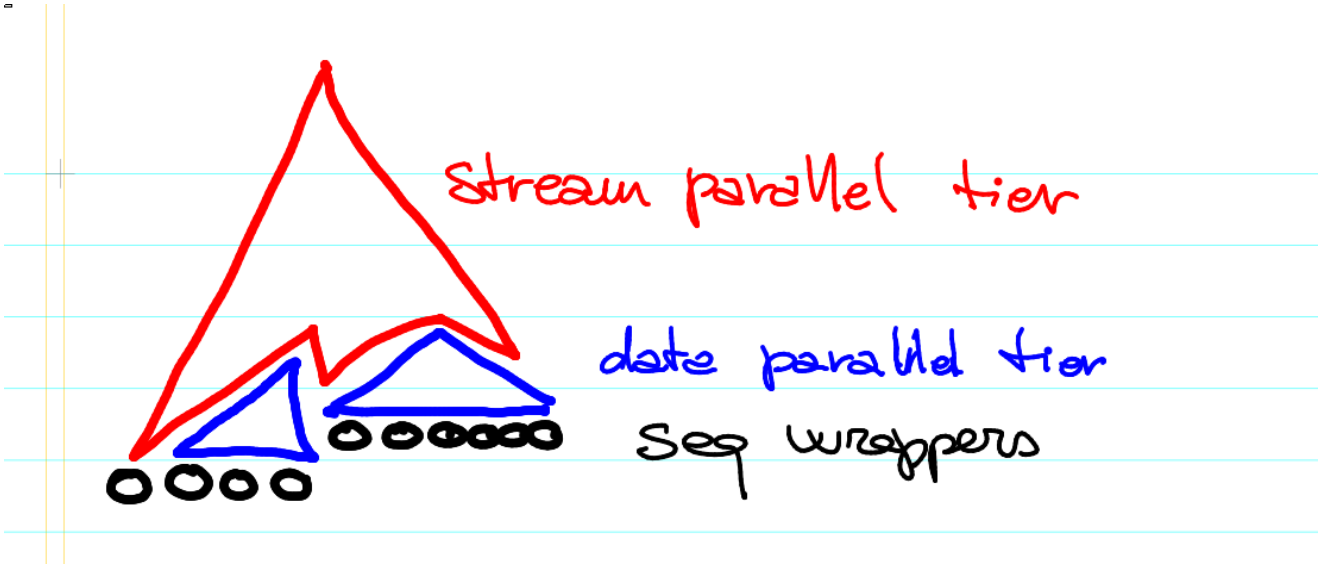
NOTE: as it does not appear to me that it is relevant for the oral exam, I skipped it. Professor dedicated few time to it as well.

Lecture 18 - 19

Two tier composition rule / Two tier model

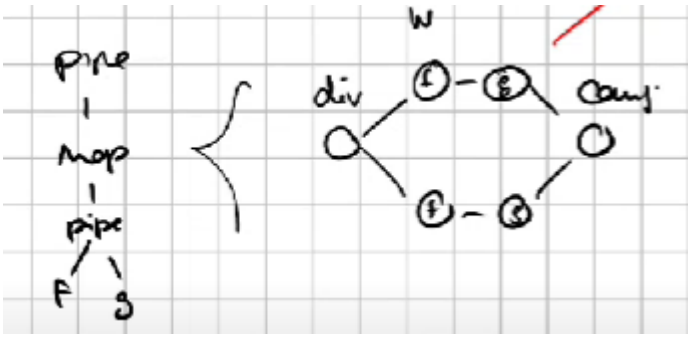
In a program, we usually need to **nest parallel patterns** inside one another.

However, it was observed that **stream parallel patterns** should always be used **on top** of **data parallel patterns** (two tier composition rule).



Example: $Map(Pipe(Seq(f), Seq(g)))$

NOTE:
There could be rare cases where there is a possibility to exploit stream parallel patterns inside a map, when the map is used into another stream parallel construct.
For example, here we could see the map as something dividing the data into two partitions, to which then the pipe is applied and finally there is a composition of the results.
Danelutto analysed further the time required by these strategy against the sequential time, but I think it is too much to know.



Portability

Portability of programs is an issue typical of parallel programming.
In the sequential programming world it has a minor impact.
The problem is mostly determined by the different features sported by different parallel and distributed architectures.

There are 2 kinds of portability:

- **Functional portability**: we want to be able to perform the **same functions** across portings, thus obtaining the same results (e.g. functional portability of an image filtering application means that we can move/recompile the application to the new machine preserving the possibility to filter images obtaining the same set of filtered images we previously get from the application running on the first architecture);
- **Performance portability**: we want to have the same non-functional features across portings (e.g. performance, security, power management, fault tolerance...);

Simple recompilation of an application on a new target architecture can give you functional portability but may be it does not guarantee performance portability.

NOTE: Danelutto Notes goes into more details and technicalities here, but professor during the lecture kept it simple.

Autonomic Management

Consider the following simple example:
$Pipe(S_1, S_2, S_3)$

If the time required by S_3 is greater than the time required by S_1 and S_2, it might be better to merge S_1 and S_2 into a unique stage.

Sometimes it could be possible to recognize this waste by ourself, but sometimes the time required by the stages could change during time, making it difficult to analyse it and decide for the best solution to implement / for the best parameters to set.

Thus, autonomic management is the best solution for these cases.
When reasoning about it, we often use a concept called **MAPE** loop:

1. **Monitor phase**: reads from the system (the parallel program) some measures through sensors;
2. **Analyse phase**: the measures are analysed to understand whether they are good/bad;
3. **Plan phase**: if the measures are bad, actions are defined to improve the situation;
4. **Execute phase**: application of the changes through actuators.

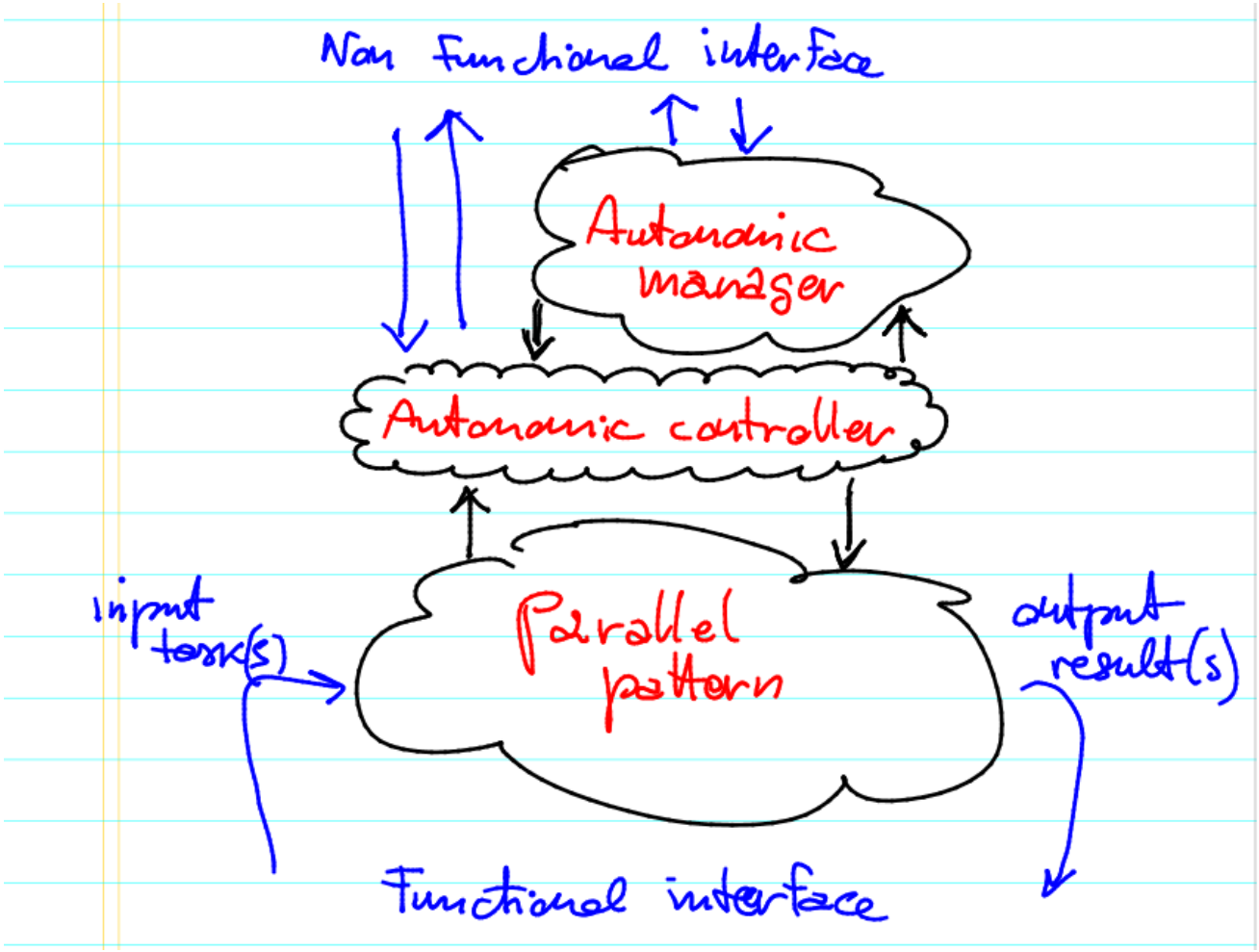
NOTE: Details about these phases can be found on chap. 11.4 of Danelutto Notes, but most of the observations seemed trivial to me.
However, there are some examples that might be worth checking.

All the phases are generally cheap apart from the execute phase, since it could involve stopping and restarting entire services.
This has to be taken into account, since we're doing autonomic management for improving performances, but implementing it could lead to worse performances sometimes.

NOTE: Danelutto went into exercises where it reasoned about what kind of measures could be taken, and what kind of actions could be implemented if some scenarios happened, but it is not part of the theory. It analysed possible scenarios for farms, maps and pipes.

Behavioural Skeletons

A behavioural skeleton is the result of the **co-design** of a **parallelism exploitation pattern** and of an **autonomic manager**, taking care of some non functional feature related to the parallelism exploitation pattern implementation.



There are 3 components:

- **Parallel pattern**: the parallel pattern implementation;
- **Autonomic controller**: interfaces needed to operate on the parallel pattern execution, both the monitoring/triggering and the actuator ones;
- **Autonomic manager**: this is the manager taking care of non functional features related to the pattern implementation. It implements the adaptation policies;

Of course, since we are generally talking about parallel patterns nested one into another, we could think of having a nested structure of Autonomic Managers as well, each one handling a different pattern and reporting to its parent.

This implements a **separation of concern** between pattern implementation and the autonomic stuff.

NOTE: Danelutto Notes went into far more technical details here, but professor just barely mentioned the concept of Behavioural skeleton, using it then to further go into examples.

Power consumption

We're taking into accounts performance, but often power consumption is more important.
If we take into account both, we could implement a manager for each.
But the managers may take decision that contrast with each other.

The power consumption is more or less proportional to the **cube of the frequency**.

One technique to affect power consumption is called **DVFS (Dynamic Voltage Frequency Scaling)**: you can change in a processor the voltage they are using to operate (actually changing the frequency) through system calls.
A core can also generally be tweaked to go into two modes:

- **sleep mode**: it is very fast to wake up, but still consuming a bit
- **deep sleep mode**: consume close to 0 power, but takes more time to awake.

Of course, reducing the frequency means increasing the total time: one must seek a **trade-off** between the two, often by taking the **maximum (useful) time** possible for the computation but spending less power to compute it.

Lecture 20 - 23

Stateful patterns

*NOTE: Chap. 6.5 of Danelutto Notes goes into details.

So far we have only seen stateless patterns (e.g. in a pipeline, we have never seen interaction between the stages).
In a stateful computation, the computation depends on: **input**, **state** and the **function**. So not $f(x)$ but $f(x, state)$.

This could lead to the concept of **shared state**: it may be useful to maintain in a state variable some counters relative to the current parallel computation.
Or, in other cases, it is worth allowing concurrent activities to (hopefully not frequently) access some centralized and shared data structure.

Shared states does not usually come with good news: it generally make more complicated to add this concept to previously seen patterns.
For example, consider pipelines, if we add a **state inside** each stage and then we decide to farm (duplicate) a particular stage, it could become very difficult to keep the duplicated stages with a consistent state.
We could think of having a **shared data structure**, but it could quickly become a **bottleneck**: when shared data structures must be accessed concurrently, the concurrent activities rarely may achieve ideal speedup.

However, implementing **shared states** is possible. There are **4 types** of shared states:

- **Read only**:
  - *What it is*: not truly a shared state, as no modification is made to the shared structure. Concurrent accesses only read concurrently;
  - *Implementation*: shared states can be easily copied and require no synchronization nor cache coherence;

    - *Impact*: no impact on parallelism;
- **Owner writes**:
    - *What it is*: one writes, multiple read;
    - *Implementation*: the owner has variables allocated in its private memory and other entities will access its memory (through remote access in case of distributed architectures or some kind of cache coherence algorithm in case of shared memory target architecture);
    - *Impact*: a possible small overhead may occur if reads occur at the same time of the writing of the owner;
- **Accumulate**:
    - *What it is*: multiple write (but only the increase operation is allowed as write operation), multiple read;
    - *Implementation*: separate concurrent activity handling write/read requests, possibly implementing a "flush" of all the pending writes before a read is performed;
    - *Impact*: most of the accesses to accumulator will be implemented asynchronously, not impacting parallelism. However, if some specific "flush/barrier" operation is needed, some activities may need to be serialized;
- **Resource**:
    - *What it is*: multiple write, multiple read, but only one entity can access variables to modify it (through a locking mechanism);
    - *Implementation*: full synchronization, as exclusive access is needed;
    - *Impact*: accesses may be completely serialized;

Fastflow

NOTE: followed the lecture, as it was mostly practice and it is not relevant for the oral exam.

FastFlow provides a base class called `ff_node`, encapsulating the logic and behaviour of individual parallel tasks or stages of computation.

Key concepts:

- **Deriving from `ff_node`**: To define a custom node, you create a new class that derives from the `ff_node` base class. This class will represent your specific parallel task or computation stage. You can override the virtual member functions provided by `ff_node` to define the behavior of your node;
- **Implementing the `svc` Function**: The most important function to override is `svc()` (short for "service"). This function contains the main computational logic of the node. It is executed by worker threads in parallel;
- **Message Passing**: Nodes in FastFlow communicate with each other by exchanging messages. Messages can be any type of data that you define. To send a message from one node to another, you use the `ff_send_out()` function within the `svc()` function of

the sending node. The receiving node can then process the received message in its own `svc()` function;

- **Handling Input and Output**: FastFlow provides input and output channels for nodes to receive and send messages. You can define input channels by deriving from the `ff_in` template class, and output channels by deriving from the `ff_out` template class. Nodes can have multiple input and output channels to handle different types of messages and communication patterns;
- **Building the Computational Graph**: Once you have defined your custom nodes, you can build the computational graph by connecting the nodes together. This is done by specifying the input and output channels of nodes and establishing the message flow between them. FastFlow provides pattern classes (such as `ff_farm`, `ff_pipeline`, etc.) that help you organize and connect nodes based on common parallel patterns;
- **Running the FastFlow Application**: To execute the parallel application, you create an instance of the pattern class that represents the overall structure of your parallel computation. You pass the necessary input channels, output channels, and nodes to the pattern class. Finally, you invoke the `run()` function on the pattern instance to start the execution of the parallel application.

During the execution of the application, FastFlow manages the distribution of tasks among worker threads and handles the message passing between nodes.
It provides automatic load balancing and efficient task scheduling to maximize parallelism and performance.

NOTE: After introducing these concepts, professor went into details on the patterns that FastFlow contains (e.g. pipeline, farm, map, for, reduce...) and a lot of advanced features available in FastFlow (like pinning processes to processors, optimizations, compilation flags...). I suggest going with: Tutorial [FastFlow] (unipi.it)

Lecture 24 - 27: Message Passing Interface

Standard interface for **message passing**.

3 main entities:

- **Communicator**: object that encompasses a *group of processes* that have the ability to communicate with one another (point-to-point communications);
- **Group**: a collection of processes;
- **Process**: has a rank (id) assigned. They can:
  - **Send/receive** a message to/from another process by providing a *rank* and a *tag* to uniquely identify the message. It can employs *collective communications* to gather multiple messages and reduce the network overhead (a message is not sent until a process acknowledge that it wants to receive it);

Practice

Every time:

- `MPI_Init(...)`: initialize the **environment** and spawns a communicator (which is assigned to a variable passed by reference), assigning a rank to every process alive;
- `MPI_Comm_size(...)`: returns the **size** of a communicator (n. of processes);
- `MPI_Comm_rank(...)`: returns the **rank** of a communicator;
- `MPI_Finalize(...)`: cleans up the MPI environment;

Datatypes: MPI has the classic signed/unsigned datatypes (short, int, long...);

Messages:

- `MPI_send`;
- `MPI_receive`;

Messages (used for collective communication):

- `MPI_Barrier(MPI_Comm comm)`: synchronize all of the processes inside a communicator;
- `MPI_Bcast(...)`: broadcast;
- `MPI_Gather(...)`: inverse of broadcast, a process receives `n` messages from other processes;
- `MPI_Scatter(...)`: distribute a message (e.g. an array) among multiple processes;
- `MPI_Reduce(...)`: similar to `gather`, but it also perform a reduce operation on the incoming elements (one of the pre-defined ones, typically max/min, arithmetic operations, bitwise operations...);
- `MPI_Allreduce(...)`: similar to `reduce`, but every process get the reduced result;

Communicators:

- `MPI_Comm_split(...)`: sometimes you may want to perform operations only on a subset of the processes. This method "splits" the communicator into sub-communicators.

Groups:

- Instead of splitting a communicator (which is a *remote* operation and requires for all the processes to synchronize and communicate about it), working with groups is a **local** operation;
- `MPI_Group_union(...)`: performs the union of 2 groups;
- `MPI_Group_intersection(...)`: performs the intersection of 2 groups;
- Other less interesting operations are available as well (check slides);

Lecture 28 - 30: Apache Hadoop

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming

models.
It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

Hadoop is a scalable, reliable, and cost-effective solution for **storing** and **managing** large **datasets**, supporting Java and Scala.

Hadoop Filesystem: HDFS

HDFS stands for Hadoop Distributed File System.
HDFS operates as a distributed file system designed to be fault-tolerant and it designed to be deployed on low-cost, commodity hardware.
HDFS provides high throughput data access to application data and is suitable for applications that have large data sets and enables streaming access to file system data in Apache Hadoop.

Generally, files are divided in fixed-size **blocks**, easy to mange and replicate.
The default size is **64MB**: it is very large compared to competitors in order to minimize overhead and allowing almost constant disk seek time, but it makes working with small files difficult.

HDFS has a **master-slave** architecture:

- **NameNode (NN)**: act as *master*, keeping metadata about the cluster (e.g. which node contains which file block). A system could have multiple NameNodes, they could be used as *checkpoint nodes* in case the primary one fails;
- **DataNode (DN)**: act as *slave*, managing data (e.g. store, replicate, change a file).

HDFS is best used with Hadoop, since it provides **data awareness** to the computation jobs (thus leveraging data locality).

In order to **READ/WRITE** a file, clients only communicate with NNs in order to find/decide the data blocks and then **directly read/write blocks** from DNs.
This approach prevents NNs from being bottlenecks.

For the reading part, when given multiple replicas of the same file, it considers 2 factors:

- **Proximity** to the reader;
- **Rack awareness**: choose replicas on the same rack or nearby racks, reducing network traffic;

For the writing part, in case of node failures NNs will figure it out and replicate the missing blocks.
By default, the replication factor of a block is 3, and the placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack.

This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of node failure; this policy does not impact data reliability and availability guarantees.

MapReduce

Atop the file system comes the **MapReduce** Engine, consisting of 2 main components:

- **JobTracker**: **manage/coordinate** the execution of MapReduce jobs (e.g. job scheduling, resource management, task assignment...);
- **TaskTracker**: **execute** individual task on the cluster nodes and does also some additional functions (e.g. status reporting, task retry, speculative execution...).

PageRank with Hadoop

PageRank is a ranking algorithm used by search engines to **rank web pages** based on their **relevance** to a search query.

Check slides for an implementation of it, both theoretically and practically. The following are just sparse notes.

- For each node, we need a list of its incoming links
- Given a node, its new pagerank is given by a reduction of the pagerank of the incoming nodes.

KNN with Hadoop

KNN is a simple machine learning algorithm that **classifies a new data point** based on the k nearest data points in a training set.

Check slides for an implementation of it, both theoretically and practically. The following are just sparse notes.

- For each element, find its k nearest point and their labels, and assign the label of the majority to the element.

Lecture 31 - 35

Apache Spark

It is a unified analytics engine for **large-scale data processing**.
It **extends Hadoop** approach: it does not only support MapReduce, but also SQL queries, data streaming, machine learning and graph algorithms.

The main component is the **driver program**: it runs the `main()` function and creates the `SparkContext`, which is the entry point to any Spark functionality. The driver program is

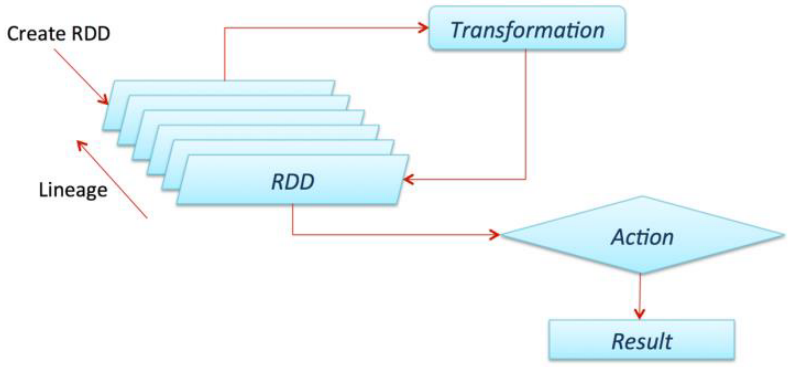
responsible for **dividing** the work into **tasks** and scheduling them across the cluster.

It supports the concept of **shared variables**, variables shared across tasks.
They can be of 2 types:

- **Broadcast**: **read-only** variables that are cached on each machine rather than providing a copy of it with tasks. Broadcast variables allow programmers to keep a read-only variable cached on each machine, which helps to reduce communication cost. They can be used, for example, to give a copy of a large input dataset in an efficient manner to every node;
- **Accumulators**: **write-only** variables that can only be increased by the tasks. They can be used, for example, for *reduce* operations, so that the driver can then retrieve the final result.

RDD

RDD (Resilient Distributed Datasets) is a fundamental data structure of Spark: it is an **immutable distributed** collection of objects (e.g. Python objects, Java objects, Scala objects...).



- An RDD is **created** starting from one or multiple sources (e.g. in memory data, on disk data...);
- The RDD is then **logically subdivided** in partitions, such that they can be worked on in parallel;
- **Transformations** (i.e. mappings) can be applied on RDD in order to generate a new RDD, thus creating a versioning of the starting one (thus offering resiliency, as it can reconstruct the original RDD in case of failure);
- **Actions** (i.e. reductions) can finally be applied on RDD, producing a result to the user;

Note that transformations are executed in a **lazy** way (transformations are only evaluated when an action is requested).

GraphX

GraphX is a **distributed graph processing framework** built on top of Spark. It supports **basic graph operations** (e.g. vertices/edges counting, reversing, subgraphing...) and also

has **builted-in algorithms** (e.g. PageRank, Connected Components, Shortest Path...).

It is implemented using the concept of **Think Like A Vertex (TLAV)** and it leverages 2 models: **Bulk Synchronous Parallel (BSP)** model and **Actor Model**.

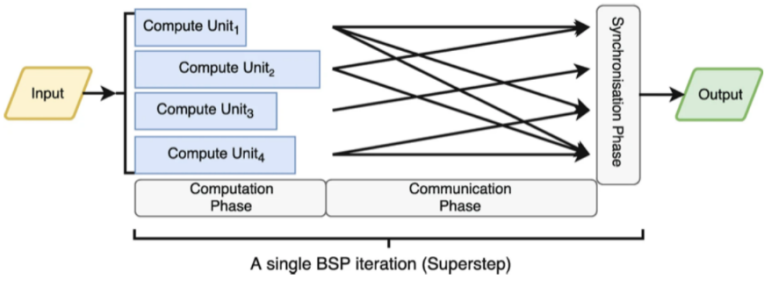
Think Like A Vertex

GraphX uses the concept of **TLAV (Think Like A Vertex)** to represent a graph and enabling efficient distributed graph processing.

The vertex is the **primary unit of computation**, maintaining a state (e.g. label, attributes...) and **sending messages** to its neighbours, which can in turn update their state based on those messages.

Bulk Synchronous Parallel (BSP) model

It is a **parallel computing model** that divides computation into **supersteps**, where each superstep consists of a **computation phase**, a **communication phase** and a **synchronization phase**.
No 2 supersteps can overlap thanks to the synchronization phase.



The TLAV concept is implemented here: the compute units are the vertices, that can communicate and exchange messages with each other and synchronize at the end of every superstep.

Actor model

Very similar to TLAV: the **actor** is the primary unit of computation, maintaining a state, creating other actors and **sending messages** to its neighbours, which can in turn update their state based on those messages. The main difference is that Actors can do the said **actions** (send messages, create actors and change their behaviour) **concurrently**: it leads to **indeterminism**, differently from BSP.

The said communication is also **one-way asynchronous**: once a message has been sent, it is the responsibility of the receiver.

Because of the indeterminism, it is important that the implemented actions are **quasi-commutative**: the order in which messages are sent between actors must not affect the final outcome of the system.

Note that due to the said features, the Actor model supports **fault-tolerance** (actors can be restarted/moved to another machine) and **scalability** (new actors can be created).

CAP Theorem

Fundamental concept in distributed systems that states that it is **impossible** to **simultaneously guarantee** all three of the following properties:

- **Consistency**: reads are always up-to-date data;
- **Availability**: always get a response (no guarantees it is the most recent write);
- **Partition Tolerance**: system continues to operate despite network partitions (communication failures) that may occur.

According to the CAP theorem, in the presence of a network partition, a distributed system can only provide either consistency and partition tolerance (CP) or availability and partition tolerance (AP). To be more precise, it is not a **binary decision**: perfect consistency cannot coexists together with perfect availability, but rather systems relax one condition in favour of the other.

To support this idea, several **types** of **consistency** exists:

- **Strong consistency**: subsequent reads return the same up-to-date value;
- **Weak consistency**: subsequent reads are not guaranteed to return the same up-to-date value;
- **Eventual consistency**: weak consistency, but lazy propagation leads to strong consistency after some time.

PACELC Theorem

Extension of the CAP theorem, standing for **Partition tolerance**, **Availability**, **Consistency**, **Else**, **Latency** and **Consistency**.

The theorem states that in the presence of a network partition, a distributed system must choose between availability and consistency, as per the CAP theorem. However, when the system is running normally in the absence of partitions, it must choose between latency and consistency.

Lecture 36: Akka

Akka is an open-source toolkit and runtime for building **highly concurrent**, distributed, and fault-tolerant applications in the Java Virtual Machine (**JVM**) ecosystem. It provides a powerful model for building scalable and resilient systems by leveraging the **Actor Model**.

Here's a high-level overview of how Akka works:

- **Actor Hierarchy**: Actors in Akka are organized in a hierarchical structure. Each actor has a parent and can have multiple children. This hierarchy allows for supervision and fault-tolerance mechanisms. If an actor encounters an error or fails, its parent can handle the error and decide how to recover or escalate the failure.
- **Supervision and Fault Tolerance**: Akka provides built-in mechanisms for handling errors and failures in a distributed system. When an actor fails, its supervisor can decide how to handle the failure, such as restarting the actor, stopping it, or applying more complex strategies for fault tolerance. This hierarchical supervision structure allows for the creation of resilient systems that can recover from failures gracefully.
- **Concurrency and Scalability**: Akka is designed to handle high levels of concurrency and scale efficiently. It achieves this through the use of lightweight actors, which can be scheduled and executed concurrently on a small number of threads. This approach minimizes the overhead of thread creation and context switching, allowing for better utilization of system resources.
- **Clustering and Distribution**: Akka provides tools for building distributed systems. It includes features like clustering, where multiple instances of an application can form a cluster and communicate with each other transparently. Akka also supports remote actors, allowing actors to be located on different machines and exchange messages over the network.

Overall, Akka simplifies the development of highly concurrent and distributed applications by providing a powerful actor-based model, fault-tolerance mechanisms, and tools for scalability and distribution.
It enables developers to build resilient and reactive systems that can handle large workloads and adapt to changing conditions.