Parallel and Distributed Systems

Federico Matteoni

A.A. 2021/22

Index

0.1 Introduction

Prof.: Marco Danelutto

Program Techniques for both parallel (single system, many core) and distributed (clusters of systems) systems. Principles of parallel programming, structured parallel programming, parallel programming lab with standard and advanced (general purpose) **parallel programming frameworks**.

Technical Introduction Each machine has more cores, perhaps multithreaded cores, but also GPUs (maybe with AVX support, which support operations floating point operations, **flops**, in a single instruction).
Between 1950 and 2000 the VLSI technology arised, integrated circuits which nowadays are in the order of 7nm (moving towards 2nm): printed circuits!
In origin, everything happened in a single clock cycle: fetch, decode, execute, write results in registers, with perhaps some memory accesses. The we had more complex control where in a single clock cycle we do just one of the phases (fetch *or* decode *or*...), like a **pipeline**. More components are used the higher the frequency but the more power we need to dissipate, and we're coming to a point were the power we need to dissipate is too much and risks to melt the circuit, so we're reaching a **physical limit** in chip miniaturization. But temperature and computing power do not go in tandem: computing power is proportional to the chip dimensions, while temperature is proportional to the area. So it's better to put more processors (**cores**) and let them work together rather than make a bigger single processor. An approach is to have few powerful cores and more less powerful cores (for example, in the Xeon Phi processors). Now, the processors follow this architecture, with the performance of a single core decreasing a bit with every generation but it's leveled by adding more cores.

Up to the 2000, during the single core era, code written years before will run faster on newer machines. Now, code could run slower due to not exploiting more cores and the decreasing in performance of the single core.
With accelerators the situation is even more different: for example GPUs, accelerator for graphics libraries, with their own memory and specialized in certain kinds of operations. This can require the transfer of data between the accelerator's memory and the main memory, so the architecture of the accelerator is impactful on the overall performance.

Capitolo 1

General Paradigms of Parallel Programming

Why Study Parallel Programming

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.

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

Because **code reuse is an issue**. Often we speak about problems which have already been treated, and sequential code already exists that solves those problem, 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 unfeasabile, 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 singe computing device is **concurrency**, not parallelism.

Concurrency It's a similar concept: **things that *may* happen in parallel with respect to the ordering between elements**. So given more flows of control, any schedule of these flows can be executed with interleaving.

Computing Devices Generic term that defines anything capable of computation:

Threads, implying shared memory

Processes, implying separated memory

GPU Cores

Hardware Layouts on a FPGA (Field Programmable Gate Array)

...

Basic Concepts

Sequential Task A **"program" with its own input data** that can be executed by a single computing entity.

Overhead **Actions required** to organize the computation but **that are not included in the program**. For example: time spent in organizing the result or dispatching the work. Basically, **time spent orchestrating the parallel computation and not present in the sequential computation**.

Speedup This is the **fundamental result** we're looking for. It's the **ratio between the sequential time and the parallel time**. Assuming the best sequential time, it is the following

$$\text{SpeedUp} = \frac{\text{Sequential time}}{\text{Parallel time}}$$

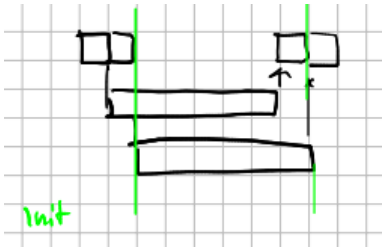
Scalability Slightly different measure that tells **how suitable a particular program is to be used with more devices**.

$$\text{Scalability} = \frac{\text{Parallel time with 1 computing device}}{\text{Parallel time}}$$

Remember the fundamental difference between scalability and speedup, although those terms can be used interchangeably in literature. Speedup is measured with the best theoretical sequential time, while scalability is measured with the time achieved by the particular program at hand.

Stream of tasks We may not want to consider just one computation but it may be useful considering a number of computations and wanting to optimize a set of tasks.

Example: Book Translation With $m = 600$ pages, for example. Let's assume I can translate a page in $t_p = 0.5$h. The sequential task is: take the book and spend time until I can deliver the translated book. The time is circa $m \cdot t_p = 300$h.
In parallel, ideally every page can be translated independently so I can split the book in two pieces of $\frac{m}{2}$ pages each (introducing some overhead), giving each half to a person. Both can translate at the same time, so ideally the time required is $\frac{m}{2} \cdot t_p$ for each, producing the translated halves. At this point I get the halves and produce the translated version (introducing some other overhead). Ideally the time required is more or less $\frac{m}{2} \cdot t_p$, with "more or less" given by the time spent in splitting the book and reuniting the two halves. So the exact time is $T = T_{split} + \frac{m}{2} \cdot t_p + T_{merge}$.
What if the two person have different t_ps? For example $t_1 > t_2$. When a translator finishes, it spends some time synchronizing its work with me. With nw "workers" (translators, in this instance) $T = nw \cdot T_{split} + nw \cdot T_{merge} + \frac{m}{nw} T_{work}$ with $nw \cdot T_{split}$ time spent delivering work to each worker and $nw \cdot T_{merge}$ time in merging each result.



Init is the time where every worker has work to do, and finish is the time where the last worker finished working. So the exact formula is

$$T = nw \cdot T_{split} + \frac{m}{nw} \cdot T_{work} + T_{merge}$$

So $\frac{m}{nw} T_{work}$ is the time that needs to happen, found in the sequential computation too, whereas the other two factors are **overhead**.

$$\text{SpeedUp} = \frac{\text{Best sequential time}}{\text{Parallel time}}$$

but the parallel time depends on the nw so

$$\text{SpeedUp}(nw) = \frac{\text{Best sequential time}}{\text{Parallel time}(nw)} \simeq \frac{\not{m} \cdot \not{t_p}}{\frac{\not{m}}{nw} \cdot \not{t_p}} = nw$$

This not taking into account the overhead. It's a realistic assumption because usually the time splitting the work is very small. But we have to take into account that, in case it's not negligible.

$$\text{SpeedUp}(nw) = \frac{m \cdot t_p}{\frac{m}{nw} \cdot t_p + \underline{nw \cdot T_{split} + T_{merge}}}$$

Example: Conference Bag $T_{bag} = t_{bag} + t_{pen} + t_{paper} + t_{proc}$ and with m bags we have $T = m \cdot T_{bag}$
We could build a pipeline, a building chain, with 4 people and each person does one task:

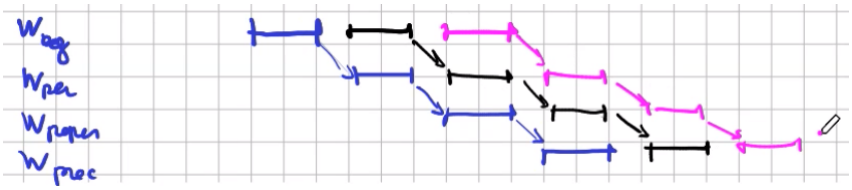
One takes the bag and gives to the next

One puts the pen into the bag and passes it

One puts the paper into the bag and passes it

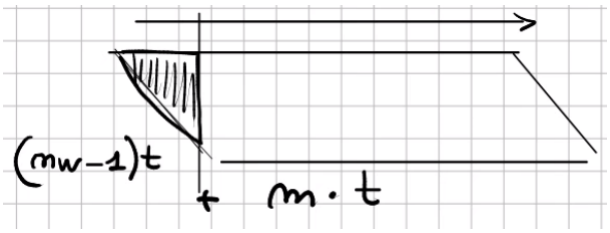
One puts the proceedings into the bag

So $w_b, w_{pen}, w_{paper}, w_{proc}$ workers. When the first worker has passed the bag, it could begin taking the next bag. Same for the others.



So in sequential we have $m \cdot (t_{bag} + t_{pen} + t_{paper} + t_{proc})$, and in parallel per 1 bag we have $t_{bag} + t_{comm} + t_{pen} + t_{comm} + t_{paper} + t_{comm} + t_{proc} + t_{comm}$ with t_{comm} spent passing the bag from one to the other, so total of $m \cdot T_{seq} + m \cdot t_{comm}$. But that's not correct, because we work in parallel: ideally we have a parallelogram of $m \cdot (t_{proc} + t_{comm}$ base, and we require $t_{bag} + t_{pen} + t_{paper} + 3 \cdot t_{comm}$ time to get up to speed and "fill the pipeline". But this required time is negligible, and in the end the overall time is given by the base of the parallelogram.

Pipeline With m tasks and nw stages, with the completion of the stage i required in stage $i + 1$. So the output is $f_{nw}(f_{nw-1}(\dots f_1(x_i)\dots))$. With t time required for each stage.



We spend $(nw - 1)t$ to get the last stage working and $m \cdot t$ time spent by the last stage to complete all the tasks.

$$T_{par}(nw) = (nw - 1) \cdot t + m \cdot t$$

$$\text{SpeedUp}(nw) = \frac{(nw \cdot t) \cdot m}{(nw - 1) \cdot t + m \cdot t}$$

So the higher the m is, the lower is the impact of the time required to get up to speed. So $m >> nw \Rightarrow T_{par}(nw) \simeq m \cdot t$

Throughput Number of **tasks completed per unit of time**.

1.1 Measures

On one side we can have **more speed with more resources** (computing devices).
On the other side we can employ **more complex applications with more resources**. For example more precise computations, so extra resources not for improving the time but to improve the quality of the computations.
A recent perspective is to **aim at computing results with less energy** thanks to parallelism.

We've seen the $\text{SpeedUp}(n) = \frac{T_{seq}}{T_{par}(n)}$, where the plot has to lie below the bisection of the cartesian graph.

1.1.1 Base Measurements

Latency L Measure of the **wall-clock time between the start and end of the single task**.
The latency related to a number of tasks is called **completion time** or T_c.

With a sequential system, we have $T_c = L \cdot m$ for $x_m, \dots, x_1$ inputs.

With a parallel system, we have $T_c \simeq m \cdot T_s$ because we reason at regime.

We are **interested in minimizing latency when we want to complete a computation as soon as possible**.

Service Time T_s It's related to the possibility of executing more tasks. It's the measure of the **time between the delivery of two consecutive results**, for example between $f(x_i)$ and $f(x_{i+1})$
Even if x_i and x_{i+1} arrive at the same time, f would still be computing $f(x_i)$ so it'll start computing $f(x_{i+1})$ when it has finished.
In a sequential system, $T_s = L$.
Also called **throughput**. We are interested in **minimizing service time when we want to have results output as frequently as possible** without caring about the latency of the single task.

Example A 3 stage pipeline, with each node being sequential and with latency L_i for node i.
At t_0 the first stage N_1 gets the first tasks and computes it in L_1, then N_2 computes in L_2 and N_3 computes in L_3 so a total of $t_0 + L_1 + L_2 + L_3$.
When the pipeline is filled, T_s is dominated by the longest L_i, so $T_s = \max\{L_1, L_2, L_3\}$ and $T_c = \sum L_i + (m-1)T_s$
If m is large with respect to n = number of stages, the "base of the parallelogram" would be very long, so $m >> n \Leftrightarrow T_c = m \cdot T_s$

1.1.2 Derived Measurements

SpeedUp

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

Could be latencies, service times... depending on what we want to measure the speedup of, and measures how good our parallelization is.

Scalability Measures how efficient the parallel implementation is in achieving a better performance on greater parallelism degrees.

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

Efficiency Measures the tradeoff between what you gain with the speedup and the cost of the speedup.

$$\text{Efficiency}(n) = \frac{\text{Ideal parallel time}(n)}{T_{par}(n)} = \frac{\frac{T_{seq}}{n}}{T_{par}(n)} = \frac{T_{seq}}{n \cdot T_{par}(n)} = \frac{\text{SpeedUp}(n)}{n}$$

Throughput

$$\text{Throughput} = \frac{1}{T_s}$$

Amdahl Law Taken the total time of a computation, T_{seq}, it can be divided into something that can and something that cannot be computed in parallel (for example, dividing the book is a sequential activity). So we can say that T_{seq} = serial fraction + parallel fraction and the **serial fraction cannot be parallelized**. $f \in [0,1] \mid f \cdot T_{seq}$ is the serial fraction.

$$T_{seq} = f \cdot T_{seq} + (1-f) \cdot T_{seq}$$

The parallel fraction can be splitted between the workers, but we would have to compute the serial fraction too. By splitting more and more and more, we have that

$$\lim_{n \to \infty} T_{par}(n) = f \cdot T_{seq}$$

$$\text{SpeedUp}(n) = \frac{T_{seq}}{f \cdot T_{seq}} = \frac{1}{f}$$

So we have a very low upper bound on the achievable speedup. This is referred to as **strong scaling**: strong meaning using more resources to get the computation faster.

Gustaffson Law

$$\text{SpeedUp}(n) = n - S \cdot (n - 1)$$

With S being the serial fraction. This comes from the fact that we're considering a different perspective: Gustaffson assumes that the computation increases with the parallelism, something that's called **weak scaling**: keeping the size of the computation the same for all the concurrent agents that we use. Given nw workers I want $nw \cdot N$ results.

Cores In modern computers, we have a main memory (slow), a disk (even slower) and the memory is connected to at least 3 levels of cache. At the bottom we have some cores (4, 8...), each one has its own level 1 cache (usually split in data and instruction cache).
With an activity with a working set that fills the cache, in case of strong scaling splitting the computation across cores we process less data per core because the size of the problem is the same.
With weak scaling, we assume that the data increases so by using more cores we process the same data on all cores but the data grows so we could have extra overhead because of the working set size.
We will have patterns of parallel computation that differentiate in how we process the data.

Application as Graphs The applications can be seen as graphs of sequential nodes with dependencies. The **work span** model defines two components:

Work: the **amount of time required on a single computing device**, of course respecting the dependencies

Span: the **length of the longest route** from source to end, in terms of computation time

The maximum speedup is $\frac{\text{work}}{\text{span}}$, because in every case I need to go from the first to the goal node. I take the longest one because at least the longest path must be computed, and all the rest can be done in parallel and I assume to have enough resources to compute the rest in the time of the span.
We **cannot use this model for reasoning about multiple tasks**, because it does not take into account the parallelism achieved in exploiting different tasks in the same program but **only takes into account what happens for a single task**. But it can be used, e.g., to reason about a single node of a pipeline.

1.2 Patterns

Computations with common and particular shapes and semantics that can be understood and implemented depending on the situations, not linked to languages and technicalities. Patterns are a useful concepts, allow programmers to reuse experience of other programmers an not reinventing the wheel.
Parallel patterns can be of two kinds: **data parallel**, **stream parallel**. For each pattern there are more than one name used in literature.

1.2.1 Data Parallel Patterns

Parallelism comes from data: we **split the data in pieces**, **compute a set of results** that can be **combined into a single final result**. The book translation example is a situation where we can apply data parallel patterns. What matters is the latency L, we want to be able to provide the result faster once I have the full collection available. The general pattern is:

Decomposition

Partial results

Recomposition

Different choices in each phase yield different data patterns.

Map Pattern Also called applytoall:

Decomposition: $\forall$ item of the collection, which is of course the minimum decomposition of a collection

Partial results: f(item), the result of a function f applied to each item

Recomposition: $\forall$ f(item), yielding an isomorphic collection to the original one.

Reduce Pattern Also called fold:

Decompose again $\forall$ item of the collection

Partial results: apply a binary function $\oplus(x, y)$ (e.g. the $+$)

Recombination: $\oplus(\oplus(a, b), \oplus(c, d))$

Stencil Pattern

Decompose into partially overlapping partitions of the data, e.g. of a matrix or an image

Partial results: the application of a function f(item)

$\forall$ f(item), again an isomorphic collection

Different kind of problems: overlapping positions will yield the new value, so we have to account for that.

"Google" mapreduce

Again decompose $\forall$ items

Compute f(item) + $\oplus$(item$_i$,item$_j$)
With $f \mapsto \langle K, V \rangle$ key-value pair and the $\oplus$ function that sums the values of the corresponding keys.

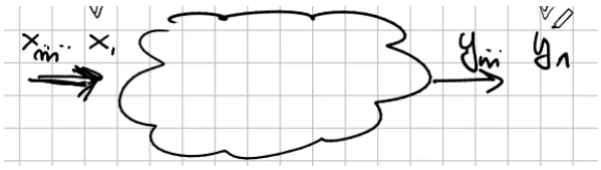
A single item

What I apply to each item is an f that maps to $\langle$key, value$\rangle$ and $\oplus$ applies the sum to each value.
For example in a document, $f(\text{word}) = \langle w, 1 \rangle$ and $\oplus(\langle w_k, v_1 \rangle, \langle w_k, v_2 \rangle) = \langle w_k, \oplus(v_1, v_2) \rangle$
"The lesson given by the professor", f will output $\langle$the, 1$\rangle$, $\langle$lesson, 1$\rangle$... and $\oplus$ will for example output $\langle$the, 2$\rangle$.
We can apply the map function over all the data distributed in various databases, for example.
This is `map(f)` and `reduce(`$\oplus$`)`, but we want something like `map(reduce(`$\oplus$`))`. Combining elementary patterns to achieve more complex results. Like two nested `for`s.
So I want building blocks, something that guarantees correctness of implementation that can be used to build upon.
For example `map(function<A(B)>, collection<B>)`
So build a bag of tools that we can combine to undertake common situations with good efficiency, speedup, scalability...

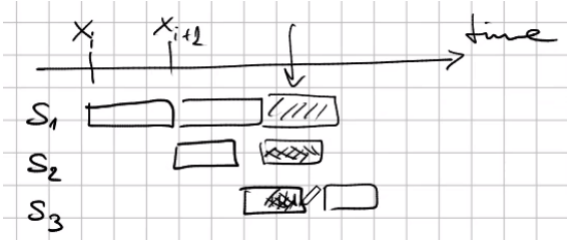
1.2.2 Stream Parallel Patterns

Stream of data, flowing in time. In data parallel we process a data collection, while in stream parallel **we don't have data appearing all at the same time**. So **stream as a collection with items appearing at different times**.
We want to take the single items and try to process in parallel, parallel execution of f over different items of the stream.
What matters is the service time T_S, to be able to sustain the flow of data and process the next available item as soon as it arrives.

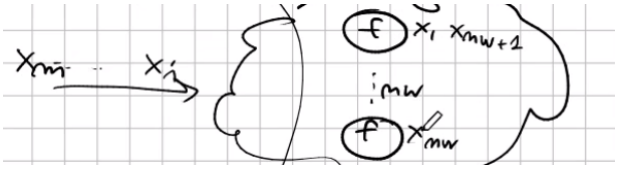
Pipeline



Inside the pipeline we have $f_1 \to f_2 \to \ldots \to f_k$ with each f_i corresponding to a phase, with f_i taking input from f_{i-1}.
So $x_i \mapsto f_1(x_i) \mapsto f_2(f_1(x_i)) \mapsto \ldots$, and the parallelism is in the computation of different phases of different items (much like what we've seen with the CPU fetch-decode-execute pipeline).



Farm We have a number nw of instances of the same function f each processing one single item.



We have no interference between computations of x_i, x_j with $i \neq j$, no need for synchronization.

The completion time T_C is $\simeq m \cdot T_S$ (number of tasks times service time). In the sequential stages, $T_S = L$. This means that given a stream of item to be computed in a parallel fashion, and I want to increase T_C or T_S, the I would need to decrease the latency of the stages in my pipeline or the workers in my farm.
If we have pipe(seq(f), seq(g)) then by decreasing the latency of f we will already improve the performances of the overall computation. Keep in mind that **not every kind of computation can be parallelized**.

Two Tier Model Let's assume a grammar of patterns.

Pat = Seq(f) | DPP | SPP

DPP = Map(Pat) | Reduce(Pat)

SPP = Farm(Pat) | Pipe(Pat, Pat)

This defines parallel computations and we aim at assuring that this can be done, a way of implementing this. So

Map(Pipe(Seq(f), Seq(g)))

can be a data parallel computation where on the single item we compute a 2-stages pipeline of f and g. But each element of the map is given to a single pipeline, or each pipeline receives a single item, so the stream parallelism is useless.

Farm(Map(Seq(f)))

Here we have a stream of items that will be processed by a Farm, each item splitted by Map and processed. This can deliver the result faster and access the next item.
So Data parallel with Stream parallel is not very good, Stream parallel with Data parallel is better: **two tier model**. So we have an initial part of the pattern which is Stream Parallel, the second part is Data Parallel and eventually the last stages (the leafs of the tree) which are sequential.

Parallel Design Patterns Also called **algorithm skeletons**: programming abstraction that model some pattern. The programmer has a framework, libraries, languages and that includes algorithm abstractions.

1.2.3 Composing

These are building blocks, so we can **compose** them. Let's see how that works and what are the expected performances.

Pipeline

We have a number k of stages for m tasks: Pipeline$(s_1, \ldots, s_k)$ meaning that this is a composition yielding $s_k(\ldots s_1() \ldots)$

$$\text{Input stream} \longrightarrow s_1 \rightarrow \ldots \rightarrow s_k \longrightarrow \text{Output stream}$$

With each $\rightarrow$ being a stream $s_i \rightarrow s_{i+1}$ and each s_i taking input from s_{i-1}.
The latency of the pipeline is the sum of the latencies of the stages

$$L(\text{Pipeline}(s_1, \ldots, s_k)) = \sum_{i=1}^{k} L(s_i)$$

We do not consider the time required to pass input to the next stage, t_{comm}, which would be based on size, nature of the computation...
The steady state is when all the stage are "filled": the longest of the stages will dominate the service time T_S

$$T_S(\text{Pipeline}(s_1, \ldots, s_k)) = \max_{i=1}^{k}\{T_S(s_i)\} = \max_{i=1}^{k}\{L(s_i)\}$$

The completion time is T_C

$$T_C(\text{Pipeline}(s_1, \ldots, s_k)) = \left(\sum_{i=1}^{k} L(s_i)\right) + (m-1)\max_{i=1}^{k}\{L_i\}$$

and when $m >> k$ we can approximate it with

$$T_C = mT_S$$

because the number of tasks required, the "base of the parallelogram", will dominate the number of workers, the "height of the parallelogram".
What if I want to achieve a given performance? For example, given a Pipeline(S_1, S_2, S_3) where S_1 and S_3 both take 1s and S_2 takes 2s, and we want a $T_A = T_S = 1$s. We can't do much, by necessity $T_S = \max\{1\text{s}, 5\text{s}, 2\text{s}\} = 5$s, and in general I don't know what's inside each stage of the pipeline so I can't perform optimization inside those. In this case, we can take the **bottleneck**, the slowest stage (S_2), and shorten the T_S by adding a farm in its place: Pipeline(S_1, Farm(S_2,), S_3). This computes the very same result:

> With the first form, Pipeline(S_1, S_2, S_3), we compute $S_3(S_2(S_1(x_i)))$
>
> With the second form, Pipeline(S_1, Farm(S_2,), S_3), we compute $S_1(x_i)$ which delivers the result to the scheduler of Farm(S_2,), where each worker computes S_2 and the scheduler selects which worker will compute it yielding $S_2(S_1(x_i))$. The farm then delivers the result to S_3, which will output the final result $S_3(S_2(S_1(x_i)))$

This works well if every timing remains the same. But if a worker in the S_2's farm takes longer we may encounter a situation where S_3 receives a x_{i+1} before receiving x_i. This may be irrelevant or be unacceptable depending on the kind of application. With 5 workers in the farm, we end up with $T_S = 1$s.
Another approach would be to consider the original pipeline as a single task computing $P(x) = S_3(S_2(S_1(x)))$ and building a farm with workers that each compute P. $P(x)$ has a latency $L = 1 + 5 + 1 = 7$s, so with a farm composed by 7 P we have a $T_S = 1$s.
Which is better depends:

> Pipeline(S_1, S_2, S_3) has $T_S = 5$s and $nw = 3$
>
> Pipeline(S_1, Farm($S_2, 5$), S_3) has $T_S = 1$s and $nw = 7$ (+2 for E and C)
>
> Farm($P, 7$) has $T_S = 1$s and $nw = 7$ (+2 for E and C)

Noting that each farm adds 2 workers. So by refactoring with the farm we need more workers, which can be a decision point in implementing this kind of refactoring.

Boundary Conditions We have to take into account the **interarrival time** T_A, time spent to get another item from the input stream, and the **interdeparture time** T_D, the time spent to get another item into the output stream. Let's suppose that $L(s_i) = i$ seconds, so ideally $T_S = k$ seconds: we process 1 item each k seconds. If $T_A > L(s_i)$ we have to wait the second item when I finish processing the first, same thing for the next after the second: the interrarival time looks like an interstage between s_{i-1} and s_i.
T_D behaves at the same way: we have to wait that T_D finishes before giving it out output, behaving like an interstage. So the previous behavior, analyzed before, happens $\Leftrightarrow T_A < T_S$ and $T_D < T_S$ **something that we have always to take into account**.

Farm

Sometimes we denote as Farm(s, nw), otherwise we omit the number of workers and simply write Farm(s).
We assume to know L_w and T_w of the workers. We have some scheduler (**emitter** E) that distributes the items from the input stream to the workers, and a gatherer (**collector** C) that gets the results from the workers and delivers them to the output stream. Those can simply be data structures: queues, for example.

$$L(\text{Farm}(s, nw)) = t_E + L_w + t_C$$

This can appear as a pipeline of three stages, where the Emitter produces to the second stage (the workers) which produce to the third stage (the Collector)

$$T_S(\text{Farm}(s, nw)) = \max\left\{t_E, \frac{T_w}{nw}, t_C\right\}$$

We assume to have m tasks

$$T_C(\text{Farm}(s, nw)) = m \cdot T_S(\text{Farm}(s, nw))$$

If we take into account the boundary conditions

$$T_S = \max\{T_s(\text{Farm}()), T_A, T_D\}$$

What if we want to achieve a given performance? Compute a nw suitable to achieve a wanted performance by inverting the very same formulas.
With a target $T_S = T_A = 1\text{s}$

$$T_S = 1\text{s} = \max\left\{t_E, t_C, \frac{10\text{s}}{nw}\right\}$$

But t_E, t_C are negligible so

$$\frac{10\text{s}}{nw} = 1\text{s} \Rightarrow nw = 10$$

Map

The map data parallel pattern is composed by three main phases:

Split: divide the collection into sub collections, a **set of subcollections**

Map: compute the **set of subresults**

Merge: produce the final **collection results** (usually in the same shape as the input collection)

With m dimension of the collection and t_f to compute the function of the map

$$L(\text{Map}) = \frac{m \cdot t_f}{nw} + t_{split} + t_{merge}$$

t_{split} and t_{merge} are non-negligible in distributed architectures, but are negligible in shared-memory architecture.

$$T_S(\text{Map}) = L(\text{Map})$$

No concept of T_C because the map is applied to a single collection.
If I have multiple collections we can consider a 3-stage pipeline composed by a splitter node as s_1, the map as s_2 and merger node as s_3, giving

$$T_S = \max\{t_{split}, t_{map}, t_{merge}\}$$

$$t_{map} = \frac{m}{nw} t_f$$

The structure of a map resembles that of a farm, thus to increment the performances we can increment the number of the workers that implement split/merge operations. This requires synchronization between the workers of the split/merge operations in order to avoid the repetition of elements.

Reduce

From a vector we want to output the sum: a scalar from a collection. We split in t_{split}, then each of the nw workers applies the function $\oplus$ to its subcollection in $\frac{m}{nw} t_\oplus$ and finally we merge in $nw \cdot t_\oplus$ because we have to compute $\oplus$ over all the nw subresults.

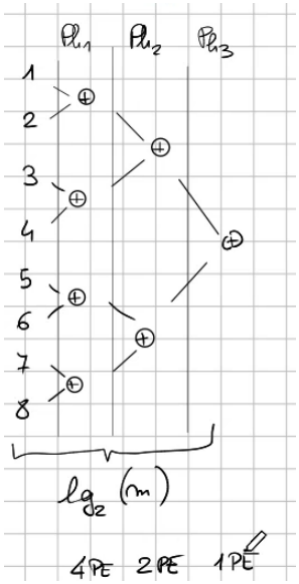
$$L = t_{split} + \left(\frac{m}{nw} - 1\right) t_\oplus + (nw - 1) t_\oplus$$

$$T_S = L$$

With multiple collections, same argument as before

$$T_S = \max\{\ldots\}$$

But the computation can be organized in a logarithmic tree, too. We would have $\log_2(m)$ phases each with half the activities of the previous phase.



But we have efficiency 1 only in the first phase, then half, then one fourth...

$$L = t_{split} + \lceil \log_2 m \rceil t_{\oplus}$$

provided that $nw \geq \frac{m}{2}$. If we use threads, the split phase is just telling the threads what they have to do, then we merge with a simple loop.

Stencil

For example computing the average of three neighbor items in a vector, with **necessary boundary conditions**.
Split phase that produces for example two halves of the vector: when computing the right extreme of the first half, I need to add the first item of the second vector, same with the left extreme of the second half: we have some shared positions. No problem when reading, the **problem arises when we write the shared position**.

> A first solution would be to **use 2 buffers**: we read from a buffer and write to another buffer, and we swap the buffer before restarting such that we use the new values to read and the old values to write.
>
> Another solution is to **use a small buffer to host modified values in the neighborhood**. The result is kept in a buffer before being written to the array. The buffers are written when we move to such a position that the edited previous position are no longer read: the buffer would need to contain at most half the radius of the stencil operation.

$$L(\text{Stencil}) = t_{split} + \frac{m}{nw} t_{stencil} + t_{merge}$$

Where $t_{stencil}$ includes buffer management with the second solution and t_{merge} includes swapping the buffers in the first solution.

Composition Given a Pipeline(s_1, s_2, s_3) with s_i sequential, s_2 may be data parallel (map), m stream items and each being a vector of k items.

$$L = L_1 + L_2 + L_3$$
$$T_S = \max\{L_1, L_2, L_3\}$$
$$T_C \simeq m \cdot T_S$$

s_2 is sequential but can be turned into a map, takes L_2 so I can imagine $t_f \simeq \frac{L_2}{k}$

$$T_S(\text{Pipeline}) = \max\left\{L_1, L_3, \frac{L_2}{nw_{\text{map}}}\right\}$$

Note how the latency decreases. This of course *if* s_2 can be turned into a map: I **need to be able to check the code**.
Another approach, as we saw before, is this: given that s_2 is the slowest stage, we can use a Farm(s_2, nw_{farm}). However in this case latency stays the same.

Single Owner Computes rule Is a more general pattern: each part of the data structure has a **single owner** who is **in charge of performing the computing** of the new values in that particular part.

1.3 Technicalities

Setup of Concurrent Activities In a parallel application a set of concurrent activities are performed by a set of parallel execution entities (threads, processes, coprocessors, machines...). Setting up these concurrent activities requires some work, varying given the tools, purpose and hardware involved. Starting a thread and waiting for its termination is in the rage of tens-hundreds μs for processors, going up to milliseconds for GPUs.

Threadpools To minimize the cost, we can use **threadpools**: a number of threads able to execute generic tasks. Each thread is created once, used and reused, only to be destroyed at the termination of the parallel computation. These threads are often called **workers**, and they obtain tasks from some **task repository**. The delivery of these tasks is the goal of **load balancing**.

1.3.1 Threads

To create a thread using the `std::thread` library we use `auto tid = new std::thread(function, ...);` which starts a new flow control that runs the passed function. We are used to writing instructions sequentially, now we `fork` another flow of computation: we get two flows that are executed together **in the same address space**, so **the new thread inherits all the memory of the original thread**.
Correctly handling the threads is fundamental, e.g. by `join`ing them at the end of a computation or `detach`ing them if needed: **detach separates the thread from the process** allowing the **execution to continue independently** (`detach`ed threads cannot be `join`ed).
Another useful method is `yield()`, which signals to the scheduler that this thread can release the processor and be rescheduled later. This is useful in non-timesharing contexts or when a thread describes a very long computation.

1.3.2 Synchronization

Two fundamental approached: **controlling data sharing** (`mutex`es) and **waiting on events** (condition variables).

Mutexes `std::mutex` objects are used to implement **mutual exclusion** with `lock` before and `unlock` after a certain block of code that needs to be executed by just one thread at a time. Thus `mutex` objects need to be shared.
Often the critical section that needs to be protected is very small and lasts for a small block of data. `std::lock_guard` mechanism can be declared having a `std::mutex` as parameter. It tries to acquire the mutex's lock when declared and will automatically release it when its scope ends (by its object destroyer).
Last mechanism is the `std::try_lock` function: it returns immediately with `true` if the lock has been acquired, `false` otherwise.

Condition variables `std::condition_variable` objects provide the method `wait` that takes a `mutex` and optionally a predicate as parameters, and the methods `notify_one()` and `notify_all()`.
We call the `wait` on some shared data, waiting for some kind of event. Once we call the `wait`, the thread is asleep and releases the lock on the mutex until some other thread calls one of the `notify` methods (and the relative `unlock`!).
With `notify_one` only one of the waiting threads is put in the ready list, whereas with `notify_all` every waiting thread will be put in the ready list. When a thread is awaken by a notify method, it will try to reacquire the lock. Once a thread is awake and has reacquired the lock, **it needs to re-check the condition it was waiting** because there's no guarantee that, due to scheduling, the condition has not been invalidate between the `notify` event and the thread actually being put into execution. So usually the `wait` are put inside a `while(condition)` loop.

1.3.3 Async

`std::async` are a bit different. When instantiating an `std::thread` it starts and forks the execution of the program. For `std::async` we simply tell the system that it *can* execute that portion concurrently, and it will be executed in a thread if and only if a thread is available. An async returns a `future` that can be used to know the status and return value of the computation.
A variation is in the `deferred` tasks, where the computation is only started when the result is requested. This enables the implementation of lazy evaluation properties, while an `async` starts its computation immediately.
Often these are implemented via threadpools, but these methods better expose the semantics highlighting the places where the results are needed. Overheads are in the range of tens of μs, similar to threads with condition variables.

1.3.4 Packaged Tasks

`std::packaged_task` decouples the moments of having a function to call from the moment of calling that function. With `get_future` we can get the future relative to the packaged task and use it. To compute the packaged task we call it passing the arguments, e.g. the packaged task of the function `t1 func(t2);` is `auto pt = new packaged_task(func);` has future `auto future<t1> ft = pt.get_future();` and can be called with `pt(arg);` with `arg` of type `t2`. The result can be obtained with `t1 res = ft.get();`.

1.3.5 Promises

`std::promise` can be used in two ways: using its future like before or call `set_value` on it. They can be viewed as **placeholders**.

1.3.6 Load Balancing

Load balancing is the problem of **giving the same amount of work to all the components involved**. Even if a single thread takes longer than all the other, we wait that it finishes so we have a lot of empty time in the other threads. The efficiency lowers a lot, and poor speedup too.
One of the reasons could be that the computation per sé is unbalanced, for example if we have a number of concurrent activities that is much larger than the number of processing elements, even worse if the time spent computing the single activity varies.

Static Techniques

Related to the usage of different splitting policies. Static load balancing devise a scheduling policy of concurrent activities to processing elements **before** the start of the computation.

Chunk policy The m concurrent activities are split into n chunks: the chunk i includes concurrent activities from $i \cdot \Delta$ to $(i+1) \cdot \Delta$ with $\Delta = \frac{m}{n}$

Cyclic policy The m concurrent activities are assigned to the n processing elements in a round robin fashion: i is assigned to processing element P_j such that $j = i \mathrm{mod} n$

Both are easy to implement. The chunk policy performs better when we have uniformly distributed execution times, while cyclic distributions performs better when "hot spots" are present. In both cases, there are distributions of execution times that completely invalid the load balancing.

Mix A cyclic distribution of blocks. Split into blocks and assign the first to a thread, the second to the next... following the cyclic policy.

Dynamic Techniques

We can do much more, adapting to the situation giving more things to do to the thread that so far have done less. Static load balancing devise a scheduling policy of concurrent activities to processing elements **while** the computation is running.

Autoscheduling Threads are not assigned a block/item or a distribution, but each threads *asks* for something to be computed. Some code like

```
while (more work to do) {
        ask work
        compute
        deliver result
}
```

The threads that gets longer tasks stops asking for more tasks for a while, and more tasks will be executed by the other threads. When tasks are almost finished, it may happen that some thread gets the "last" long task still taking longer than all the other threads. But it's not as impactful as stated before, as it's the worst case.
In general, prefetch $k \simeq 2$ or 3 tasks

Job Stealing Bunch of tasks, cyclic static assignment. With nw threads and m tasks, each thread gets $\frac{m}{nw}$ tasks. With job stealing, the thread that has finished its tasks and perceives that there's more to compute, steals a task from another thread. Problems: "size" of the steal, synchronize accesses, who to target... the solution is a random policy: threads that finishes their own assigned task steal a random number $\in [0, nw]$ and steal that number of tasks.

1.3.7 Overhead

Communications Dispatching tasks and results require some kind of communication between threads and processes. Inter-process communication require some copy of the message, given the separation of the address spaces. Inter-thread communication is simpler, usually involving pointers being moved around through some data structured used as a "mailbox", which requires synchronization and, thus, introduces overhead. In some cases, e.g. FastFlow, lock-free mechanisms are used. FastFlow uses a lock-free queue to communicate pointers only, reducing overhead to some 10 ns. Other actions can reduce the overhead:

> Light-overhead communication mechanisms should be preferred when the grain of the concurrent activities is small. With larger grain concurrent activities, the impact of heavier communication mechanisms will be better tolerated since there would be fewer activities, and thus the communication overhead would have smaller impact.
>
> Sending one big message usually require less time than sending 3 smaller distinct messages. This is especially true in distributed environments, like networks, where the setup of the communication requires a notable amount of time. This also may require less space, given that we would use a single pointer versus using three distinct pointers. Of course, we need to take into account the overhead required of producing the single big message versus producing the three smaller messages.
>
> Alternative algorithms, involving different amounts of computation-to-communication ratios, should be considered. In some cases, a slower algorithm that performs less communication may be preferred.

Standard computer of course have a processor and a memory, connected by some kind of bus. This bus represents a bottleneck (the **Von Neumann Bottleneck**). We can improve by having two memories: an instruction memory and a data memory.

By exploding the Von Neumann model a bit, we have a **main memory**, a number of **cache** levels faster and smaller going from the main memory to the processor, and the **processor**. Usually the last level of cache, closest to the processor, takes 1 or 2 τ (clock cycles) to perform the operations, while the main memory takes around 80 to 100 τ. This follows the **locality principle** which is divided into **spatial locality** (if a given address i is accessed, then it's likely that addresses $i + 1$ and so on will be accessed soon) and **temporal locality** (an accessed address is likely to be accessed again soon), enabling the concept of **working set** (the set of addresses that are likely of being accessed in a small amount of time).

In a multi-processor system, e.g. an 8-core processor, we have some levels of cache that are shared between cores (ideally the higher levels, closer to the main memory) and some that are independent per-core. In this case, **a shared variable should keep the same value even when we have independent caches**. This is achieved by an hardware component that implements the **cache coherence protocol**. This requires some time, of course. This is achieved by a bus (**snoopy bus**) that is placed between the last level of shared cache and the first level of independent caches. This bus detects operations on independent caches and propagates accesses to shared variables. Other systems implement a **directory of shared data** and polls it whenever shared data is accessed. The key takeaway here is that **there is something that we need to do to keep the copies consistent**.
This **cache coherency protocol** is applied to cache lines, which are the minimal units of operations on caches. So whenever the line containing the shared variable is changed, all the line is moved in all the other places. So it at least has to access the first shared level of cache, if not going up until the main memory. This has to be made for the whole working set of the thread.

Example If we use the stencil pattern to implement matrix multiplication, we may have a **cache line that is "split" between** two portions of the data, thus **two threads**. Each time a value is written inside that line, the cache coherence protocol runs. In a stencil computation this makes sense.
In, e.g., a map operation we may have a vector where $x_i^t = f(x_i^{t-1})$ performed for several ts. If this cache line is shared with another core, this may trigger the cache coherence protocol for all the other items that may be never used, resulting in the **false sharing problem**. We must try to avoid as much as possible this situation, because **this adds significant overhead** to the computation.
We use **padding techniques** that adds pad values to the original vector such that the culprit cache line is filled with dummy values to avoid sharing it with other cores.

Another overhead problem occur when **a thread is moved from a processor to another**. The receiving processor in general does not possess any element of the incoming thread working set in its cache levels.
This switch is decided and performed by the operating system, but this decision may not be optimal for some programs, especially if we want to harness the full potential of all the cores on a machine. We can **force it to not perform this operation by pinning the threads** to some particular core, by specifying its identifying number.

Another source of overhead is **communication**, e.g. between stages in a pipeline: each stage spends some time to receive and some time to send, other than the time spent computing. With threads this may be ignored, because those read and write in shared memory, but this may not be negligible in distributed systems across a network. Even on fast networks with $1\mu s$ latency, this would still be 3 orders of magnitude greater than the clock cycle time. This could result in receive and send times greater than the computation time. Sending data in the end is simply tell the network adapter the address of the data and the order. At this point, instead of waiting, the node could simply begin performing the next computation and be interrupted when the data transfer is completed (and handle errors if there are). Another solution is to have three threads and implement a kind of pipeline where the first thread receives into a buffer, the second thread computes in another buffer and the third thread sends from another other buffer. At the next stage, the first thread computes from its buffer, the second sends its result and the third receives inside its buffer. At regime, we spend $\max\{t_{rec}, t_{comp}, t_{send}\} \cdot n$ time. This allows to hide communication times if and only if t_{comp} is larger than the other two times (receive and send).

We mention this technique to be useful in clusters of network-connected workstations, but a similar situation may be present in CPU-GPU communication. GPUs are accelerators, thus are connected to the processor by some kind of bus (the PCIe bus), and have their own memory. Let's say we have a map(f) with a time-consuming f, on a huge v. We can **move v to the GPU's memory**, the **GPU kernel computes** map(f) over v and the result v' is then brought back to the CPU memory. The key point here is that we have to **move data to and from the GPU memory**. GPUs provide 1 or 2 **DMA Engines** that are in charge of this GPU-CPU data transfer. This DMA engines work like the NIC mentioned before: and address is written on the bus and it will handle read/write operations without impacting performances.
Given a vector v and a map to be computed, we can consider chunks of the vector such that the computation time over the chunks is $t_f \simeq t_{comm}$ the communication time. This technique is very common and can be achieved in two ways: moving data from CPU to GPU explicitly or use **streams** which are abstractions, streams of operations. Streams can be of three types: move CPU→GPU, move GPU→CPU or compute. We setup three streams.

Overhead related to memory allocations New objects are created when instantiating inputs and results, for example. This requires allocating memory in the heap, with corresponding `malloc`s and `free`s. So we may need to be smarter, with solutions that work on thread-level memories, small heaps where to allocate objects and releasing when no more needed.
As for threads, the operation `new` which creates a new thread, and `join` which waits for the completion of a thread, of course both take some time. There is a **tradeoff** between the time spent to setup the parallel activity and the time earned from it. If the time spent computing by a thread is smaller than the time required by creating and joining it then it is pointless to perform that computation in another thread.

`jmalloc` **library** Used in BSD systems, FireFox and Facebook among others.
It manages **chunks of memory** called **arenas**, distributed in a round robin way per thread with each thread using one arena. A metaarena is used a common place.
Arenas A1, A2,..., Ak are assigned in round robin to th1, th2.... When some data comes from an arena, and we free that data we free the original arena not the local one. `jmalloc` has its own API, but other than that it uses the classical API: `malloc` and `free`, same as stdlib.
Taking a normal program `a.out`, with `./a.out p1 ...pk` we will go with `malloc` and `free` of the stdlib. If we prefix with `LD_PRELOAD=libjmalloc.so ./a.out p1 ...pk` then `malloc` and `free` will be loaded from the `jmalloc` library.

Reducing Overhead

Let's discuss few general techniques that can be used to design better parallel application reducing different kinds of overheads.

Data Locality This principle states that **computations should take place where data is located**, basically **avoiding unnecessary data movement**. Even with the same parallelism degree, two different computations may

produce a different overhead, e.g. due to point-to-point thread synchronization, which may result in vastly different speeds. In general, data locality should be enforced as much as possible:

Tasks should be scheduled to thread pools avoiding to schedule a task whose input has been just computed by thread i to some thread $j \neq i$.

In a divide&conquer algorithm where the divide phase always generates two tasks, one of those two tasks should be kept by the thread performing the divide phase. That thread would be blocked anyway waiting for the results, therefore it can (and should) be used. Given that it already has both subtasks, using one exploits locality.

When executing a matrix multiplication, thread already accessing row $A[i][]$ and column $B[][j]$ to compute $C[i][j]$ should be preferably given tasks that use again row $A[i][]$ to preserve locality.

Overlapping Computation and Communication In general, a node (e.g. a pipeline stage) performs a process-send-receive cycle which results in the following time needed to process a single item: $T_{rec} + T_{proc} + T_{send}$. If we suppose to have hardware support to receive and transmit data, then the send/receive are dominated by latency or bandwith (**B**) and may be approximated, resulting in $T_{comm}(n) = T_{setup} + \frac{n}{\mathbf{B}}$

Triple Buffering: we setup three threads working on three distinct buffers: B_0, B_1, B_2, with thread 0 managing data in B_0 and the other two threads initially idle.
When thread 0 receives the whole input task, it moves to receive the next input task into buffer B_1, while thread 1 can start processing data in B_0. After thread 0 has received data in B_1 *and* thread 1 will have completed processing the data in B_1, then thread 0 will start receiving in B_2, thread 1 will start computing in B_1 and thread 2 will start sending the result that is now contained in B_0. From this moment the three threads will work at regime.
Each time slot has duration of $\max\{T_{rec}, T_{proc}, T_{send}\}$ and each thread that completed processing/sending/receiving in buffer B_i will repeat the operation on buffer $B_{(i+1)\%3}$
Remember that this works only for **stream of tasks** (cannot work when processing a single item) and only with hardware support for parallelism between communication and computation.

Double Buffering: we can use buffering to partially hide communication in stream processing also when we only can run one communication at the same time. This adopts the same approach of triple buffering, and can be implemented in two ways:

One buffer for receiving and one for computing/sending.
Two threads will switch buffers to receive input tasks and to compute/send results, with latency given by $\max\{T_{rec}, T_{proc} + T_{send}\}$

One buffer for receiving/computing and one for sending.
Two threads will switch buffers to receive/process the input and to send results, with latency given by $\max\{T_{rec} + T_{proc}, T_{send}\}$

1.3.8 Implementation of Patterns

We spoke about stream (pipeline, farm) and data (map, reduce, stencil) parallel patterns. Let's discuss some general techniques to implement them. The basic structure of a pipeline is: an inputstream, outputstream and a number of stages implementing functions f_i. So the parameters are $\{f_i, is, os\}$, and our tools are threads/asyncs etc. from C++. There are two main approaches to the implementation: **template based** implementation and **macro data flow** implementation (MDF).

Template Based We assume that for each pattern (pipelines, farms, maps, stencil...) we have a template, which is composed by three parts:

a **target architecture**, e.g.: shared-memory multi-core, cluster/network of workstations, cloud...

concurrent activity graph, a graph representing concurrent activities with their communication channels

performance modeling

An example is a template of a pipeline in a shared-memory multi-core environment: template(pipeline, SMmulticore). We have two choices:

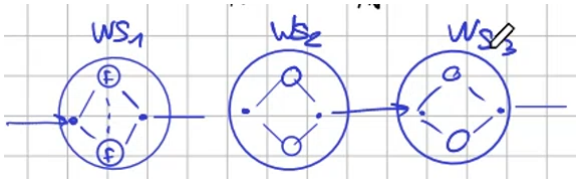
Take a number of **concurrent threads** linked by communication channels like queues. Given m stages, we have $m-1$ queues and the performance model is speedup$(m) \simeq m$ when each stage has circa the same t_S.

Use a **queue** to pile up tasks which are $\langle f, x_i \rangle$ pairs: f represents the function that needs to be computed on x_i. We have a number of threads, each takes a task from the queue, computes $f(x_i)$ and puts into the queue $\langle g, f(x_i) \rangle$ with g representing the next stage of the pipeline that needs to be computed.
There are two main issues: the queue represents a bottleneck and we need to handle the ordering of the tasks, but both can be tackled in various ways. For example, splitting the queue in a set of queues or adding the information i to the pair put in the queue to handle the ordering.
Performances of this model are very different: we don't have the limitation of having all tasks with same t_S. Provided to have a sufficient t_A (interarrival time) and given nw **workers**, not stages, we have speedup$(nw) \simeq nw$. With nw being the number of threads used, the workers, we have $nw + 2$ stages (the input and output stages count!)

We can consider another alternative

Take a number of parallel activities, with a round-robin schedule delivering data and with a sorter gathering and sorting the results. The scheduler tags each task with the appropriate number, so that the sorter can correctly sort the results.
We have nw concurrent activities, and thus nw queues plus the queue of the sorter. With a small enough interarrival time t_A, we can achieve a speedup$(nw) = nw$

So on the same architecture we can implement a pipeline in at least three different ways. What if we change architecture, e.g. from SMmulticore to COW (cluster of workstation)? A COW is a cluster of multicores, basically, so we can use any template seen before. Each workstation is a stage and the communication channels are something like TCP/IP sockets, achieving once again a speedup circa the number of stages.
Another template is to have each workstation provide a number of workers, with a scheduler queue and a sorter queue as output like the last template seen earlier. These workstation work in a pipeline, with input/output communication channels, and each implementing an array of worker threads.



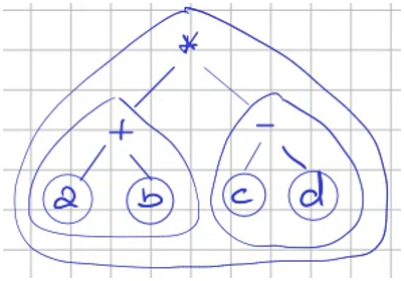
In this example, $t_S \simeq \frac{\max\{t_1,t_2,t_3\}}{nw}$ and the main source of overhead is in the scheduling/gathering part in each of the workstations. But each workstation orders the results that will be generally ignored, what's important is to maintain the arrival order in the final output. So we can take away the sort at the end of each workstation, ending up with something like $t_1 \simeq t_2 = t_{\text{scheduling}} + \frac{t_f}{nw}$ and $t_3 = t_{\text{scheduling}} + \frac{t_f}{nw} + t_{\text{sorting}}$, but given that $t_S = \max\{t_1, t_2, t_3\}$ we still have overhead. We can decouple the sorting stage outside the last workstation and add a sorting stage, with the only task of sorting what it gets.

The key idea is that **template based implementation provide a kind of library** with a number of items made of ⟨pattern, graph, hardware, performance model⟩ from which we can select the best template suited for our specific situation.

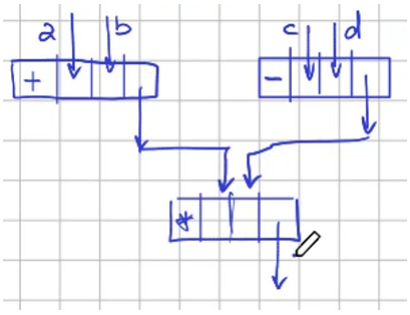
Macro Data Flow With **data flow** we denote models where the **order of the computations are dictated by the flow of data** rather than the program counter.
We are used to flows where the order of the instructions is dictated by the program counter. In data flow, we execute instructions in the order dictated by how data flows, meaning that operations are executed only when the required data is available. An example: if we want to compute $(a + b) \cdot (c - d)$ with **imperative style** programming the evaluation would go something like:

```
1 temp1 = a+b;
2 temp2 = c-d;
3 res = temp1*temp2;
```

With the program counter pointing to `1` at t_1, to `2` at t_2 and so on. With **data flow** we have a radically different approach we have a tree of operations:

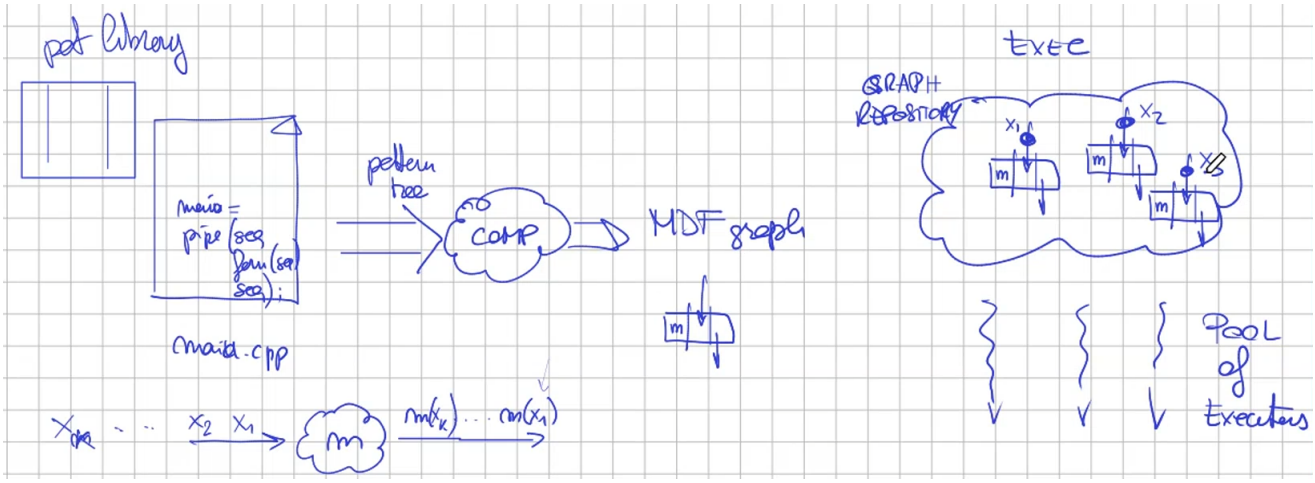


To compute $*$ I need the $+$ and $-$ trees, and each tree requires computing some other values... In a sense, the syntax dictates the order of operations. So we interpret this as having, for each operation, a node specifying the function, input arcs and output arcs. E.g. $\langle +, a, b, out \rangle$

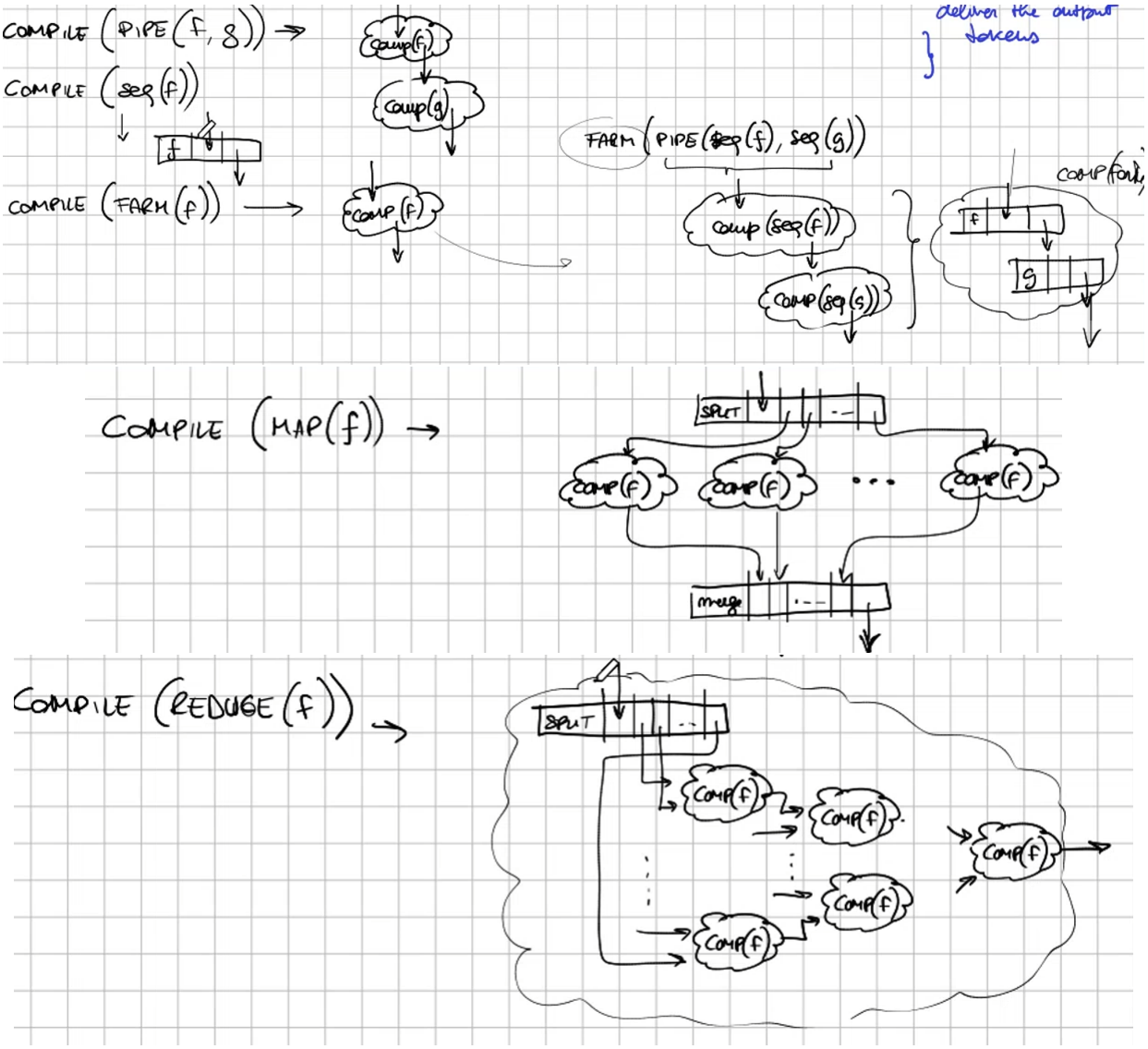


Each arc possess a token that signals its availability. If we start with a and c available, neither $+$ nor $-$ instructions are executed. Once b is available, the $+$ action becomes **fireable**. The macro data flow graph **gathers all fireable operations and executes them**. The **execution empties the input tokens of an action and fills its output token**(s), thus becoming a non-fireable action. Given a set of fireable actions, we can **execute those in every order we want**.

In ***Macro* Data Flow**, instead of considering action nodes composed by input tokens, output tokens and simple functions, we instead consider the possibility of having entire portions of code, methods, as the computable part of the node. Given a program, which uses some patterns e.g. pipe(seq, farm(seq), seq), its pattern tree is compile into a **MDF graph**. We have a **graph repository** and a **pool of executors**. Any time I have some input data, I **create a copy of the MDF graph and place it in the repository with the instantiated input tokens**. Each executor performs a loop where they get the fireable instruction, computes the action and delivers the output tokens where they need to go.



The key is the compilation of the parallel patterns.



MDF can issue some problems. First, the concurrency over the repository of course.
Another problem is the result tokens delivery: each graph can need input tokens coming from different computations, and such we need to handle it.
Another thing is maintaining the list of fireable instructions. For efficiency reasons, we maintain a **counter** inside the node (along with the function, the input tokens and the output destinations). This counter tells how many input tokens are needed to compute the function, and each time a token becomes available the counter is decreased by 1. When the counter is 0, then the instruction is fireable so the last thread who delivers the final input token is in charge of putting the receiving node in the fireable list.
Another is the out stream must be efficiently maintained, possibly reordering the results when needed.
Finally, we need to take into account some kind of graph ID to distinguish of course the various copies of the same graph.
So a node would in the end by composed by:

Graph ID, identifying the copy of the graph

Instruction ID

The **function** to be computed

Counter, the number of needed input tokens

A list of **input tokens**, each being a pointer with a presence bit

A list of **output tokens**, each containing the graph ID, instruction ID and position to where the token needs to be delivered or `NULL` if the result should be directed to the output stream.

Remember that the executors are threads in a pool. Each thread asks for a fireable action from the repository, which is unique, and computes it. Ideally, all threads begin at the same time so each thread asks the repository for a fireable action. The repository delivers to the first, then delivers to the second one (which was waiting in the meantime), then to the third (which as well was waiting for the first and the second thread to be served) and so on. This means that each thread waits a bit before beginning the computation. With double or triple buffering techniques we can mitigate this. Another way would be to deliver 2 tasks (fireable nodes) to each thread the first time. This way, once a thread delivers a result it can ask for another task while computing the task it already has.

Another issue would be present when a thread delivers a result to the repository and then gets a task which requires among its input the result that it just computed. This is a locality situation, so we can exploit some cache principles to speed up this, for example assigning a thread some task which uses tokens that it has computed (**affinity scheduling**). This optimizes computing times but it's not easy to implement, so we use caching and we take it into account when communicating between repository and executioners.

1.4 Optimizing

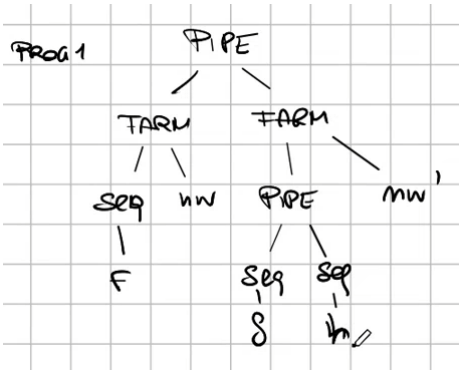
To write our programs we can use the following grammar:

$$\text{Pat} ::= \text{Seq}() \mid \text{Pipe}(\text{Pat}_1, \text{Pat}_2) \mid \text{Comp}(\text{Pat}_1, \text{Pat}_2) \mid \text{Farm}(\text{Pat}, nw) \mid \text{Map}(\text{Pat}) \mid \text{Reduce}(\text{Pat})$$

And a program is a pattern applied to some input

$$\text{Prog} = \text{Pat}{:}x$$

We describe trees



Refactoring or rewriting means looking at the semantic of the pattern and recognize shapes with the same **functional semantics** (*what* is being computed) but different **non-functional semantics** (*how* it is being computed, meaning different parallelism degrees or ordering of computation).

1.4.1 Refactoring Rules

$\text{Farm}(x, n) \equiv \text{Farm}(x, m)$ with $m \neq n$ (**pardegree change**)

$\text{Pipe}(x, y) \equiv \text{Comp}(x, y)$ (**pipe intro/elim**)

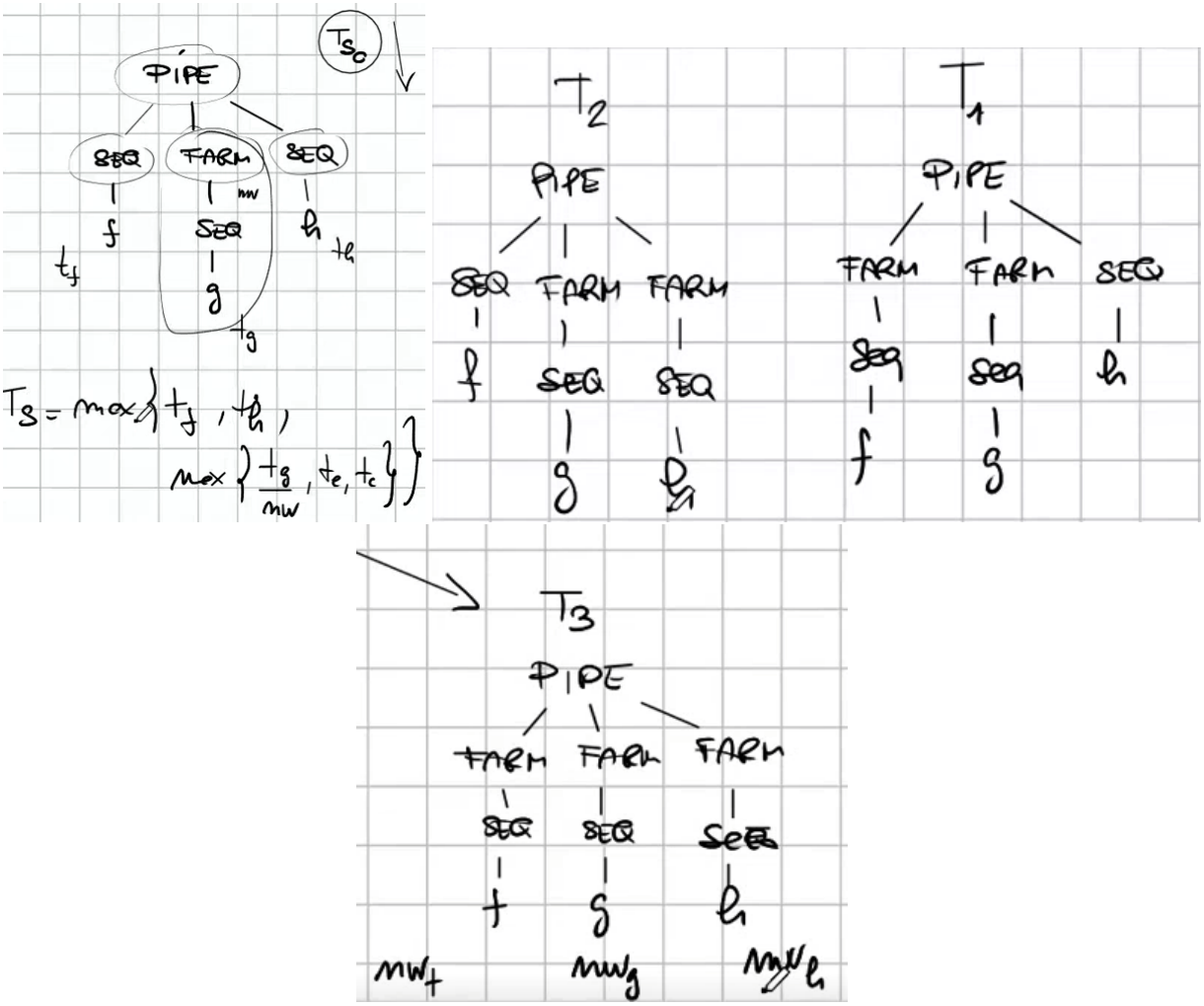
$\text{Pipe}(\text{Farm}(x,), \text{Farm}(y,)) \equiv \text{Farm}(\text{Pipe}(x, y),)$

$x \equiv \text{Farm}(x)$ (**farm intro/elim**)

$\text{Map}(\text{Comp}(x, y)) \equiv \text{Comp}(\text{Map}(x, y))$ (**map fusion**)

...

The questions revolve around choosing the best one. This is a **space exploration problem**: starting with the original tree t_0, using the refactoring rules we produce a number of derived trees and for each we measure a goal function (e.g. service time, so T_{S0} for t_0). We move toward the best one, and repeat. So we do not explore the whole tree of derived graphs, but just the sub level of the current best graph. This greedy path should bring to a better solution, a better graph thus a better pattern for the current problem. This is **not possible**. Let's see why with a counterexample, e.g. $\text{Pipe}(\text{Seq}(f), \text{Farm}(\text{Seq}(g), \text{Seq}(h))$ with T_{S0} and show a way to reach a better T_S with intermediate steps that have a worse T_S.



Each of these steps lowers the T_S: e.g. we imagine a bottleneck on one stage and introduce a farm there, then a bottleneck on the remaining stage and introduce another farm ending in t_3. But let's see another possible path: starting from t_0, in t_4 we remove the farm in the second stage, ending with Pipe(Seq, Seq, Seq). This would increase T_S. Then in t_5 we remove the pipeline, ending with Comp(Seq, Seq, Seq), once again increasing T_S. In t_5 we introduce a farm making Farm(Comp(Seq, Seq, Seq)) and here we can decrease T_S as much as we want by increasing the parallelism degree, up to $T_S = \max\{T_{\text{scheduler}}, \frac{T_w}{nw}, T_{\text{collector}}\}$ with $T_w = t_f + t_g + t_h$ in our case, so with a nw such that the maximum becomes either the time required by the scheduler or the collector.

So by performing a greedy search we easily fall into local minima, and thus we have to generate as much configuration as possible. So taken a certain tree t_0, using the rules we get all trees with depth $< k$ and then we apply map(T_S) followed by a reduce(min) (or reduce(max) depending on what we want in place of T_S).

1.4.2 Normal Form

Speaking about stream parallel computations. Earlier we saw how the best pattern for the example was Farm(Comp(Seq, Seq, Seq)). **This pattern is optimal with respect to the** T_S taking in account the resources (same or smaller T_S using the same or fewer resources).
Given a tree composed only by stream parallel patterns:

1. Get the **frontier**, which is the **set of the sequential leafs**, **from left to right**.
2. Compose the frontier **Comp(frontier)**
3. Farm it out **Farm(**Comp(frontier)**)**

This **always optimizes** the stream parallel computation in **stateless** situation. The principle is to transform a set of small computations spread around and put them into a big chunk of code with an input and on output, so **increase the grain of the computation**. The **grain** is **the ration between time spent computing and time spent organizing the computation**.

1.4.3 Performance Model Usage

Performance models are used in two ways:

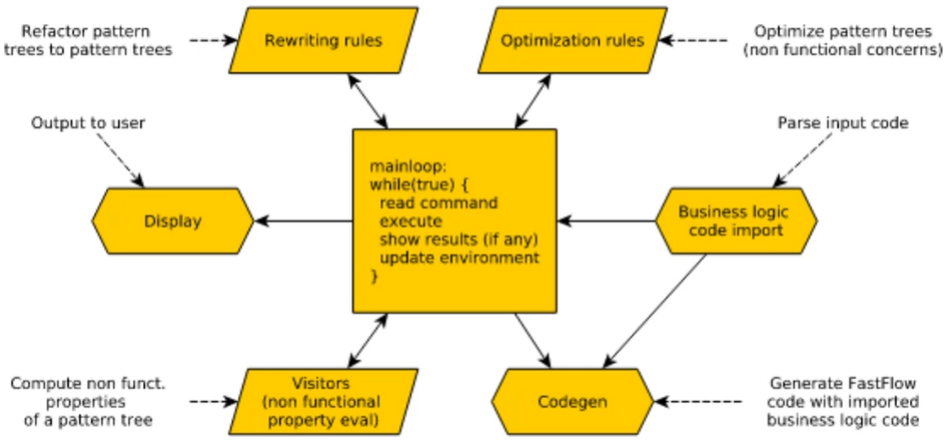
Evaluate a certain value, e.g. T_S, by visiting the tree from leafs to root and computing the required value.

Optimize by figuring out the parameters, e.g. nw

1.4.4 Optimization

In RPLSH we can do `optimize main with max resources`, with resources meaning number of available processing threads.

RPLShell



Rewriting Rules are the set of factoring rules.
For example, we have the facts that allow me to rewrite a given tree t as Farm(t) or Pipe(t_1, t_2) as Comp(t_1, t_2).

Optimiziation Rules the set of rules that for example transform a Farm(Farm(w)) into Farm(w) (because two farms linked are bad for performance), known as farmfarmopt, or the farmopt that is used when we have Farm(w) with a certain T_S of the w and a certain T_S target of the farm in order to set the nw of the farm to $\frac{T_S}{T_{S,w}}$.

Visitors the elements that visit the tree upwards.

Autonomic Management of Non Functional Features In long running computations.
For example the classical variations in network load between night and day. Autonomic because autonomous + automatic.
Born from industrial processes management, changing behaviors of the systems. Different architectures of the controller:

MAPE Loop (Monitor, Analyze, Plan, Execute loop)
Execute activities via actuators, and the system has a set of actuator objects that can be invoked to change the behavior of the system (for instance changing nw). The system has sensors, too, that provide data to the monitor part.

We have to devise a strategy to be used. E.g.: if T_A drops, most likely I could just increase nw and viceversa if T_A increases I can drop nw. Being more clever: let's suppose that T_A goes from $T_{A_{\max}}$ to $T_{A_{\min}}$ in a sin fashion, when I'm in the minimum I could take time in taking a decision to increase nw while T_A is already increasing. Same for the maximum, I could spend time making the decision and always arriving late. So our strategy can be update: I can observe the changes and avoid situations of alternating opposite changes (in the example, the adding and taking away workers constantly).

To implement these strategies, I need **sensors** and **actuators**.
We need to observe the interarrival time T_A. Also need a data structure to observe the changes, by doesn't require sensors. These sensors would be in the emitter thread.
We need to increase or decrease nw, so the actuators must be able to achieve that. To add a worker, e.g. a thread,

we need to know the queue for that worker, the collector... so the emitter sounds like a good place.

This is implemented in the **control program** of our manager. Very crude `if (condition) do actuator;`.

ECA Rules Set of **Event, Condition, Action** rules implemented in a rule system (e.g. `jboss`).

Event: **triggering action**, e.g. T_A change.
Not the event per se to be taken into account.

Condition: **predicates** on monitored values, internal state and events, that state whenever this particular rule has to be fired or not.

Action: **set of actions** on the system or the state that must be **executed after some triggering event when the condition holds true**.

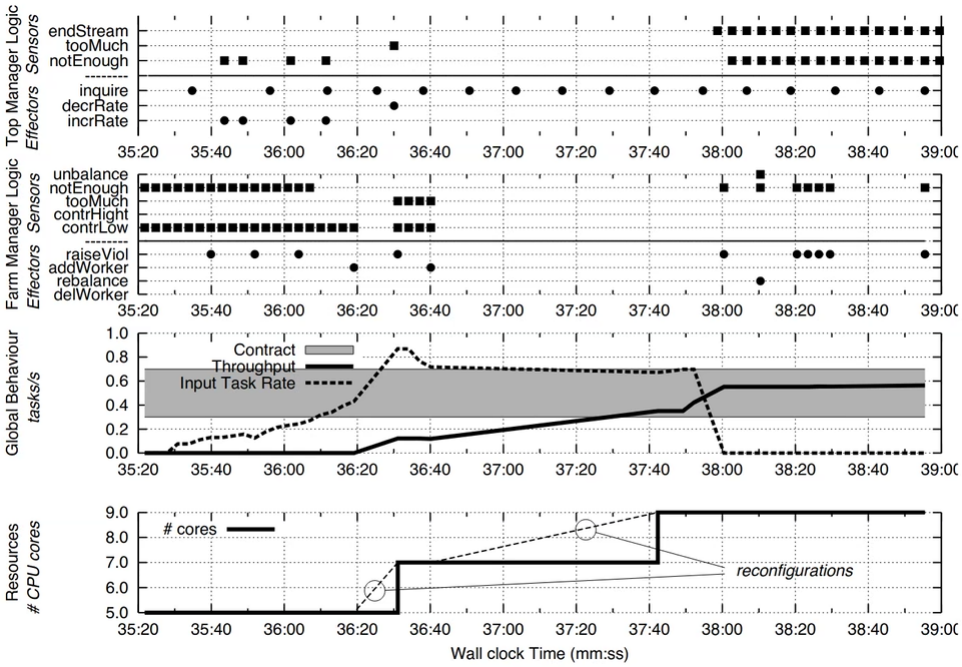
Example

	Event	Condition	Action	Priority
R_1	Change T_A	Less than the previous one	Adding a worker	LOW
R_2	Change T_A	More than the previous one	Removing a worker	LOW
R_3	Change T_A	Less than the previous one AND nw just decreased	`nop`, update T_A	HIGH
R_4	Change T_A	More than the previous one AND nw just increased	`nop`, update T_A	HIGH

Contracts

1. Propagate the "contract" top-down and try to implement it "best-effort"
2. Manage to keep contracts satisfied

We're taking into accounts performance, but often power consumption is more important. These are both non-functional features. If we take into account both, we could implement a manager for each. But the managers may take decision that contrast with each other. A solution is to build a single manager that takes into account both features, into a single MAPE loop. But this requires that both knowledge are in a single place, bad for the separation of concerns. Another solution is to have two managers, one for each feature, each with its MAPE loop, and the two can communicate: the plan phase inform each other in the hope to have authorization for a change that the execute part needs, if we don't get the authorization we go for a `nop`.

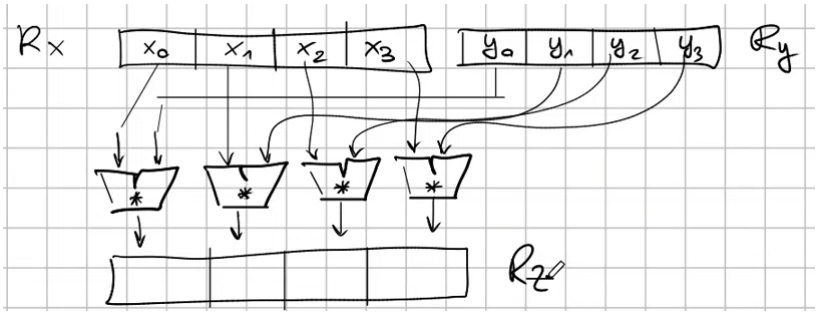


1.5 Vectorization

Very old technique, since the first computers people thought about vectorizable code with vector units attached outside the processor that could be used as I/O. The main idea is to have some **vectors of items and produce an output vector**. E.g. $\forall i \in [0, n-1]\ z_i = f(x_i, y_i)$. In most languages you have the `for` construct and we're used to think about it sequentially. But if each loop iteration is independent from the others (meaning that $z_i = f(x_i, y_i)$ is independent from $z_j = f(x_j, y_j)$ with $i \neq j$) then this each iteration is easily computable in parallel.
A classical example is matrix multiplication:

$$\forall\, i\ \forall\, j\ \forall\, k\ \ c_{ij} = a_{ik} \cdot b_{kj}$$

Using a sufficient number of ALUs we can parallelize the computation



We have ways to load/store vectors to fill contiguous portions of memory in one shot. This requires specifically designed hardware, vector processors. Registers that can be seen as vectors of tot bits numbers: 256 bits vector registers seen as 4 64 bits numbers, or 8 32 bits numbers and so on. With special commands, for example `VADD.I32 R1, R2, R3` meaning R1 = R2 + R3 with I32 meaning 32 bit numbers with I index.

Conditions In `g++` we can explicitly ask with `-O3 -ftree-vectorize` and with `-fopt-info-vec-[missed/all]` ask what cannot be vectorize.
The conditions are:

> Need to know the number of iterations.
> `for` is ok, `while(c<k)` sometimes cannot be vectorized
>
> Cannot call external code in the loop body, functions or libraries.

No conditional code.
It'd require compiling two paths, so complicates things.

No overlapping pointers.

`#pragma` Indications to the compiler
`#pragma GCC ivdep` tells the compiler whatever it thinks of the loop, to consider it as independent iterations.
`#pragma GCC unroll n` tells the compiler that the following loop should be unrolled `n` times. Useful for short loops.

So vectorization very useful for mathematical operations, such as differential equations. Requires to take a bit of care in the code to be able to vectorize. If we don't convince the compiler, we can use the `pragma`.
If we vectorize Farm(f, nw), the speedup with the vectorized f with respect to the non-vectorized f is smaller. Because the vectorized code in general is faster, so non-vectorized Speedup(nw) $\neq$ Vectorized Speedup(nw).
But vectorization can radically change the T_S of the stages of a Farm, for instance, and introduce waiting times for workers. You don't change the sequential fraction, just the non-sequential fraction.

Libraries Libraries include operations used often, e.g. `Blas` (basic linear algebra system) or `Lapack` or `NKL`.
For instance `blas` provides `gemm` for very optimized matrix multiplications.
Use libraries because they are very optimized, exploiting all the possible optimizations.

1.5.1 GRPPI

C++ library that implements common parallel patterns, using standard threads or other backends (e.g. omp, tdd or fastflow). The backend is only modeled by the `execution` parameter.

1.6 OpenMP

Part of a big family of tools useful to build parallel applications.

MPi Directed to clusters and workstations, multicores

CUDA/OpenCL For GPU

OpenMP Targets shared memory multicores (and GPU/FPGA). Very dated, since 1997 the version 1.0: no GPU, even multicores were not very popular.
At the beginning it offered very simple APIs that provided launching threads in parallel and something related to the parallel `for`. Each version improved on that, introducing: tasks, target accelerators, depend clauses on tasks (to tell that a task must be executed only after the completion of another task), memory management, task reduction (among values of different tasks), task affinity...
When we speak about shared memory multicores, we speak of some situation where the motherboard has more than one socket, with each processor being multicore.

Concepts

Directives, which OpenMP is based on. Directives, in C/C++ environment, are kind of pragmas which can be understood as portions of compiler

OpenMP is **compiler-based**

As pros:

Few lines of code are required to run parallel code, opposed to writing all the logic by oneself

Keep sequential code, means we can perform functional debugging and then adding parallel `pragma`s

As main con we need to rewrite the compiler, not entirely (such as the parser), but everything related to the usage of the pragmas must be generated at compile time.
E.g. the map seen in the GRPPI library is written at higher level but in the library, doesn't touch the compiler.

We have something used to **set up parallel activities** and something different used to **organize the parallel computation** using the parallel activities available.

`#pragma omp parallel` This must be placed before a statement, it tells the compiler that the following block must be executed in *nw* copies (threads) if and only if *nw* is the number of resources/cores of the current machine.
Most directives have clauses used to specify parameters. E.g. `num_threads(nw)` tells to use the variable *nw* instead of the predefined one (the number of cores).

```
#include <iostream>
#include <omp.h>

using namespace std;
int main(int argc, char** argv) {
  int nw = 1;
  if (argc != 1)
    nw = atoi(argv[1]);
  #pragma omp parallel num_threads(nw) // clause
  {
    auto id = omp_get_thread_num();
    cout << "Ciao from id " << id << endl;
  }
  return(0);
}
```

`#pragma omp single` Specifies that the following statement/block is to be executed by just a single thread.

```
#include <iostream>
#include <omp.h>

using namespace std;
int main(int argc, char** argv) {
  int nw = 1;
  if (argc != 1) {
    nw = atoi(argv[1]);
    #pragma omp parallel num_threads(nw)
    {
      #pragma single // this will be executed by a single thread, just one enters the following line, use a block for more lines
      cout << "Id " << omp_get_thread_num() << " of " << omp_get_num_threads() endl;
    }
  } else {
    #pragma omp parallel
    {
      auto id = omp_get_thread_num();
      cout << "Id " << id << " of " << omp_get_num_threads() endl;
    }
  }
  return(0);
}
```

`#pragma omp critical` To execute the next code in mutual exclusion.

To use OpenMP Compile with flag `-openmp`.
The `a.out` compiled, when executed, uses the linked library which provides several things including a lock/mutex machanism. So instead of the `#pragma omp critical` we can use `#pragma omp parallel` and then just the lock/unlock:

```
omp_lock_t lock;
omp_init_lock(&lock);
#pragma omp parallel
{
  // code
  omp_set_lock(&lock);
  // mutex code
  omp_unset_lock(&lock);
  // code
}
```

With the code/unlock declared with respect to the OpenMP library.

Different code for different threads We can do something like

```
#pragma omp parallel num_threads(2)
{
  if (omp_get_thread_num() == 0) {
    // code for thread 1
  } else {
    // code for thread 2
  }
}
```

This is **SPMD** code: single program multiple data.

Sections directive

`#pragma omp sections`: identifies a command which is filled by other `section` directives, each executed possibly in parallel

`#pragma omp section`

Example

```
#pragma omp sections
{
  #pragma omp section
  {
    //code 1
  }
  #pragma omp section
  {
    //code 2
  }
}
```

All the `section` may happen in parallel, how much is defined and can be inherited by other statements. For example I can have a `#pragma omp parallel` before the `#pragma omp sections`, giving the threads explained before (the number of cores as default, or as specified by `num_threads(nw)`.

Variables In most directives you can append clauses that specifies the behavior of variables.
For example in parallel pragmas you can append the clauses

`private(x)` (non initialized local copy local to each thread)

`firstprivate(x, y)` (get a local copy initialized from previous value)

`shared(y, z)` (inherit value from the global environment)

`lastprivate(x)` (this one legal only in sections, not in parallel, local copy then last `section` copied to global environment, take care last one means in syntactic-fashion, the last syntactic section updating the variables copies to global environment)

`#pragma omp for` Most famous and used.
The default behavior is that the iterations, e.g. from $i = 0$ to $i = n-1$, are made into nw chunks given each to a thread. Obviously we need independent iterations. There are others scheduling policies, specified with clauses appended to the `#pragma omp for`:

`static` the default one, with the option of specifying `chunksize(m)` specifying the number of items in each chunk, so $\frac{n-1}{m}$ chunks of m items, given to the threads in a round-robin fashion

`dynamic` where the threads ask for new chunks when they have finished, chunks are assigned on demand. Same of the static for other things

`guided` use a number of chunks smaller and smaller: large chunks in the first part of the array, then smaller chunks, then smaller and so on. Intended to be used for unbalanced computations: at the beginning we assign large chunks to each thread, but the threads may finish at different times and get smaller chunks to accomodate for load unbalancing

`auto` everything decided by the compiler

`runtime` keyword which inherits the scheduling policy from an environment variable which has to be defined before

You also have other clauses

`nowait`, removes the implicit barrier to wait all the threads, present in the parallel pragma

`reduction([+,*,and,or], x)`

`#pragma omp task` Kind of async, suggestion that if there are unusued threads use to compute the task otherwise it's sequential code.
Can define taskgroups to execute in parallel and wait with the implicit barrier at the end.
You can append `tied` or `untied`: with the first anytime you assign/reassign the execution of this task then it would be on the same thread.
The taskloop provides something similar to a map, or a pragma for, but subjecting to the rules of the tasks: could be executed in parallel given enough resources ecc.

Example Sum of a vector with a reduce(+), base case being a vector of length 1 so return the value v[0]. Otherwise the recursive case is a vector of n positions: compute the half and split in $v[0, \frac{n}{2}]$ and $v[\frac{n}{2}, n]$ recur. So we need a task for the left half, a task for the right part, checking not to be in the base case.

```
int sum (int* restricted v, int start, int end) {
  auto size = end - start;
  auto mid = size/2;
  int x, y; //shared for the subtasks, sum from left and right

  if (size == 1) return v[start]; //base case

  #pragma omp task shared(x) //to share the result with this "main" thread
  {
    x = sum(v, start, start+mid);
  }
  #pragma omp task shared(y) //same as befor
  {
    y = sum(v, end-mid, end);
  }
  #pragma omp taskwait //wait for the termination of the tasks, a "barrier"
  //now both tasks have finished, we have both x and y
  return x+y;
}

int main(int argc, char** argv) {
  #pragma omp parallel num_threads(nw) //or from the CLI "export OMP_NUM_THREADS=nw"
  //...
  #pragma omp single //single or master
  sum(v, 0, n-1);
}
```

Timing routines Also present.

Taskpool The idea is to set up nw threads to execute tasks without awaiting any kind of results from the tasks. Some king of `void` function.
We can use `auto out = bind(function, parameter);`, with `void function(float x);`,
then `out` will be a `function<float()>` so we can call `out();` because the parameter is already given.

1.7 FastFlow

Addresses the idea of structured parallel programming targeting shared-memory multicores. It's a open-source header only library (`-O3 std=c++17`). It's a complete layered system:

Parallel Applications

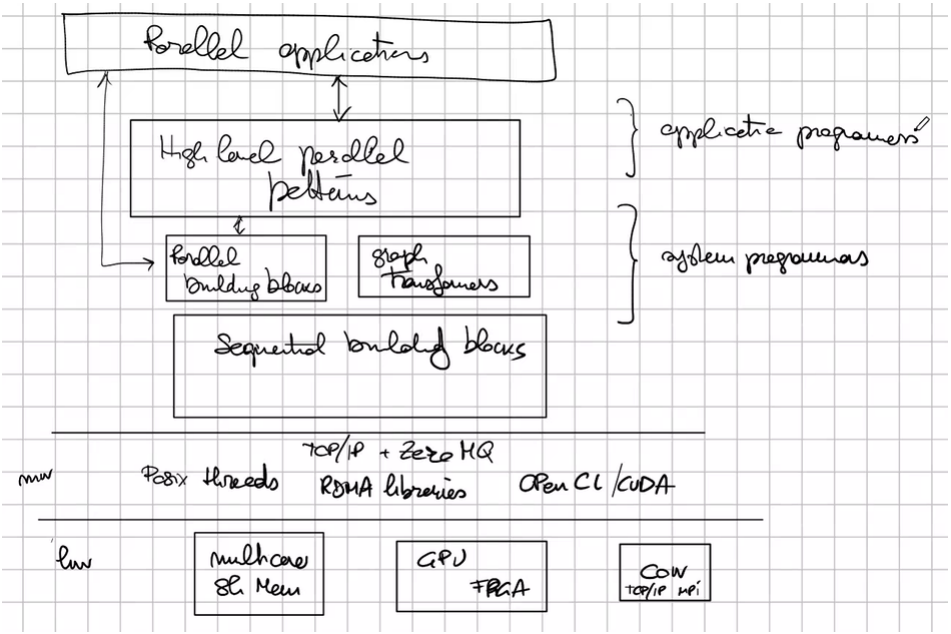
High level parallel patterns

Parallel building blocks (pipeline...), and also a small library of transformers (to change shape of structures such as pipelines...)

Core library of FastFlow: sequential building blocks

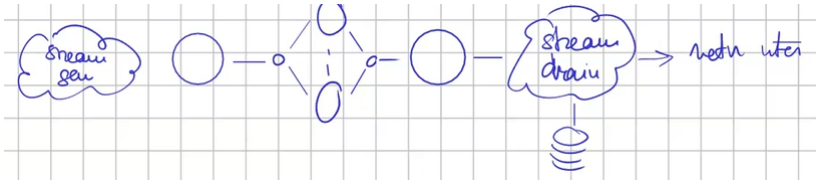
Posix threads, RDMA libraries, OpenCL/CUDA

HW layer: multicore shared-memory, but also GPU and FPGA accelerators, also COW (cluster of workstations) with TCP/IP and MPI



`ff_node` Computes some function and has two queues: a queue from where it takes items to be computed and a delivery queue.
FastFlow has stream computations and data-parallel computations.
Two nodes can be merged in a `pipe ff_node`: this merges the output queue of the first node into the input queue of the second node.
The single nodes implement the business logic, the rest is managed by the framework.
In the `svc` method, being `void*` both in parameters and in return type, you can specify whichever type you need to flow through the tasks. You can pass a task as parameter, process it and cast to `void*` its result to return it. With `ff_node_t<t1, t2>` it's introduced the type checking at template level.

Stream



The stream flows through the program and it could be infinite, in principle. We have the interarrival time to consider, so we cannot look the entire stream as a whole data structures, at most you can see a certain window.

> EOS, End Of Stream: value that signify that the stream has ended. Can be used as a condition to end loops (much like EOF for files)
>
> `ff_send_out` sends items in the stream. E.g. to end the stream, `ff_send_out(EOS)`

Shared-memory multicore, so each activity is a thread basically.

In the queues we host pointers, we don't move data. We target the fastest communication possible, non-blocking channels (without condition variables and other mechanisms): the native FastFlow queues has latency of less than tens of nanoseconds, due to cache overhead. But there are other channels, e.g.: the emitter for a farm is a 1:n communication channel so we use a father thread, slowing down the process but it's inevitable. This can be changed at compile time to use blocking communication channels (structures with mutexes, variables...).

`ff_node_t<tin, tout>` Requires the type of the input and of the output. Requires to implement a methods

> `tout* svc(tin* task);` the body, the function computed by the thread.
> Has to `return(tout*)`, or a `return(EOS)` when it wants to end (passing to `svc_end`, or a `return(GO_ON)` that tells the framework that you finished the work but doesn't put anything in the output stream. Kinda like a "`skip` and give me the next item").

Other two method are optional, provided empty but can be rewritten:

> `int svc_init();` to initialize data before the computation, executed before `svc` over the input stream, e.g. to open a file
>
> `void svc_end();` executed at the end, e.g. to close the file

`svc` stands for service.

Lifecycle of `ff_node_t`

1. `svc_init` called once when the thread is started
2. `svc` called for each item appearing on the input stream
   Returns
   - `tout*`
   - `EOF`
   - `GO_ON`
   - Other special values
3. `svc_end` called right before terminating the thread
4. `EOSnotify()` callback upon receiving `EOS`, to do something when it receives `EOS` because upon receiving that we don't call `svc` but go to `EOSnotify()` and then `svc_end`

Definition of the program I can declare several `ff_node_t`

```
ff_node_t<vector<float>> t1;
ff_node_t<vector<float>> t2;
...
```

Then we can put in a pipeline

```
ff_Pipe p(t1, t2);
```

So far no execution, only declaration

Running We call the `run_and_wait_end()` method, for example `p.run_and_wait_end();` which starts everything.

```
ff_node_t<vector<float>> t1;
ff_node_t<vector<float>> t2;
ff_Pipe p(t1, t2);

{
  utimer to("program");  // to time the execution
  p.run_and_wait_end();
}
```

Typical Patterns With capital letter are higher level, the lower letter are lower level patterns

`ff_Farm(ff_node_t worker, int nw);` build a farm with *nw* parallel degree of `workers` as nodes

`ff_Pipe(ff_node_t stage, ...);`

```
#include <ff/ff.hpp>

struct source : ff_node_t<myTask> {
  myTask* svc(myTask* t) {  // source started with null as parameter, so t can be omitted
    for(int i = 0; i < N; i++) {
      ff_send_out(new myTask(i, ...));  // whatever you need to output
    }
    return(EOS);
  }
};

struct sink : ff_node_t<myTask> {
  myTask* svc(myTask* task) {
    cout << task << endl;
    return(GO_ON);  // processed and ready for next
  }
};

struct f : ff_node_t<myTask> {
  myTask* svc(myTask* task) {
    fun(t);  // e.g. it works by side effects
    return(t);  // whatever the process I need
  }
};

int main(int argc, char* argv[]) {
  // declaration
  source node1;
  f node2;
  sink node3;
  ff_Pipe<> pipeline(node1, node2, node3);
  // implies 3 threads, one for each node (parallelism degree = 3)


  // execution
  pipeline.run_and_wait_end();
  return 0;
}
```

Compile with at least `-O3 -pthread`. This way uses non-blocking queues.

Pipeline `ff_Pipe<> pipeline(...);`

Usage statistics Use `-DTRACE_FASTFLOW` for debugging purposes, for usage statistics.

Creating tasks For example a vector of `ff_node` pointers, with `vecto<unique_ptr<ff_node>> W;`.
The `unique_ptr` is an abstraction provided by C++ to associate a usage counter to a pointer, allocated by the system.
We then add the workers to the vector with `W.push_back(make_unique<funstageF>());`.

```cpp
float fun(float x) {
  this_thread::sleep_for(2*tf);
  return x/2.0 + 1.5;
}

struct funstageF : ff_node_t<myTask> {
  myTask* svc(myTask* task) {
    task->x = fun(task->x);
    return task;
  }
}

int main(int argc, char* argv[]) {
  int m = atoi(argv[1]);
  int d = atoi(argv[2]);
  source s1(m, d);

  vector<unique_ptr<ff_node>> W;
  for (int i = 0; i < 5; i++) {
    W.push_back(make_unique<funstageF>());
  }
  cout << "Worker N. " << W.size() << endl;
  ff_Farm<myTask> f2(move(W));
  sing s4;

  ff_Pipe<> myPipe(s1, f2, s4);
  myPipe.run_and_wait_end();
  myPipe.ffStats(cout);  // compile with -DTRACE_FASTFLOW

  return 0;
}
```

Other types of nodes You can use **Multiple Output nodes** `ff_monode_t` and **Multiple Input nodes** `ff_minode_t`.
In a classical `ff_node_t` you may just `return(something)` or `ff_send_out(something)` something that goes in the output queue.
In the multiple output node you can use `ff_send_out_to(something, worker)` where you specify which worker may receive the data.
In the multiple input node you can use `ff_get_channel_id()` used to understand which channel provided the result. You also have `from_input()` that can be used to distinguish that type of channel (feedback channel or not).

RoundRobin The default policy is round robin.
If compiled with `-DFF_BOUNDED_BUFFER` and `-DDEFAULT_BUFFER_CAPACITY=k`, program wide the queues will be bounded to k elements. When the emitter finds all the queues full, it simply waits, looking for empty spots in all the queues.

Scheduling In a farm, there's the method `farm.set_scheduling_ondemand()`. It's not a true on demand, meaning that a worker without work asks the emitter for something. Instead, this forces the input buffers to be short, kind of on demand because the queues will empty faster and the emitter will fill them faster, a good approximation of the worker asking for a task without the need of a feedback channel from each worker to the emitter.

OFarm `ff_OFarm` has the same rules as the farm pattern but Ordered, meaning that the collector of this farm sorts the results.
Limitations: workers cannot change the length of the string (no `GO_ON` or multiple `ff_send_out` upon receiving a task), and workers has to be sequential.

Master Worker Pattern Master directs tasks to workers and receives results. Can be achieved with a `ff_Farm` playing with the parameters: eliminate collector, wrap-around (feedback loop) the workers. The problem is that with the wrap-around we have to be careful in handling the feedback loops.
The master also needs to discern the results of the workers (from the feedback loop) from the queue of tasks it has to send out coming from its queue. So we call the first time when there are no tasks in the queue, getting NULL as task in, and start the tasks returning `GO_ON`, then waits for the results from the feedback channel.

```
TASK* svc(TASK* tin) {
  if (tin == NULL) { //first run, start tasks
    for (int i = 0; i < m; i++) {
      TASK* t = new TASK{i, ((float) rand())/((float) INT_MAX), (rand()%100)};
      ff_send_out(t);
    }
    return GO_ON;
  } else { // getting tasks from feedback channels
    std::cout << "Feedback " << tin->taskno << " " << tin->x << std::endl;
    sum += tin -> x
    m--;
    if (m == 0) return EOS;
    else return GO_ON;
  }
}
```

`ParallelFor` Needs `#include <ff/parallel_for.hpp>` because it's a higher level pattern.

```
#include <ff/parallel_for.hpp>

ParallelFor pf; //we'll see the parameters later

pf.parallel_for(0, // initial value of the iteration value, e.g. 0 if int i = 0
        n, // limit of the iteration value, e.g. n if i < n
        1, // step incremen, e.g. 1 if i++
        0, // chunk size
        [](const int i) { }, // lambda for the ith iteration
        nw); // parallel degree
```

For example:

```
for (i = 0; i < n; i++) {
  v[i] = f(v[i]);
}
```

becomes

```
pf.parallel_for(0, n, 1, 0, [&](const int i) { v[i] = f(v[i]); }, nw);
```

Chunksize of 0 means that the vector in nw blocks of equal size and each worker gets a contiguous chunk of $\simeq$ #iterations/#workers (**static scheduling**). With a chunksize of > 0, we have **dynamic scheduling** with task granularity equal to chunksize (meaning that the vector is divided into blocks of chunksize elements, dynamically means that a idle worker gets a chunk), and with chunksize < 0 we have **static scheduling** with blocks of chunksize elements same as before.

With the `#pragma omp for` we can append many clauses, e.g. `reduction(+:sum)` (with an operator followed by a variable in the parentheses). For instance, in the following code

```
#pragma omp for reduction(+:sum)
for ( ) {
  sum += x[i];
}
```

the sum is managed in parallel exploiting the threads: each thread has its own $\texttt{sum}_i$ variable and each intermediate sum are summed together (because we specified +) when the parallel for finishes.
The `ParallelForReduce` supports the name of the variable and the lambda function to apply. An example: need to compute dot product between a and b, so $\sum_i a_i \cdot b_i$. Usually would do (sequential code)

```
int sum = 0;
for (i = 0; i < n; i++) {
  sum += a[i] * b[i];
}
```

With parallel for

```
// variable, zero value to initialize, starting from, ending to, increment, chunk size,
    lambda, final lambda, parallel degree
```

```
pfr.parallel_reduce(sum, 0, 0, n, 1, 0,
          [&](const int i) {sum += a[i]*b[i];},
          [&](int &s, const int ps) {s += ps;},
          nw);
```

`ParallelFor` is a **data parallel** pattern. At the end of the for there's a barrier that waits for all the threads involved to finish.

Divide and Conquer Pattern `ff_DC(divide, combine, base, cond, input, outpu, nw)`

`divide` splits the input

`combine` puts together partial results

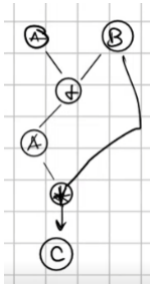
`base` function applied to short portions/single elements

`cond` is a condition that tells if you are in a short portion/single element

`input` and `output` are the input and output vectors

`nw` is the parallel degree

Macro Data Flow Pattern Used to describe custom patterns: use tags to specify nodes and dependencies. An example: compute the following graph

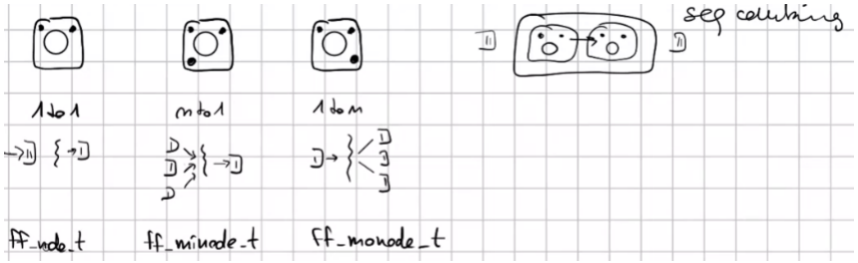


With a function that computes the sum `sum(a, b)` and one for the multiplication `mul(a, b)`. We setup some parameter info structures that tells the kind of usage of the parameters

```
// A = A+B
param_info _1 = {&A, input};
param_info _2 = {&B, input};
param_info _3 = {&A, output};
// we've described that A and B are read and A is a result
// mdf is the Macro Data Flow object initialized before somewhere
mdf.add_task(P, sum, &A, &B, size); // append the node specifying the task
```

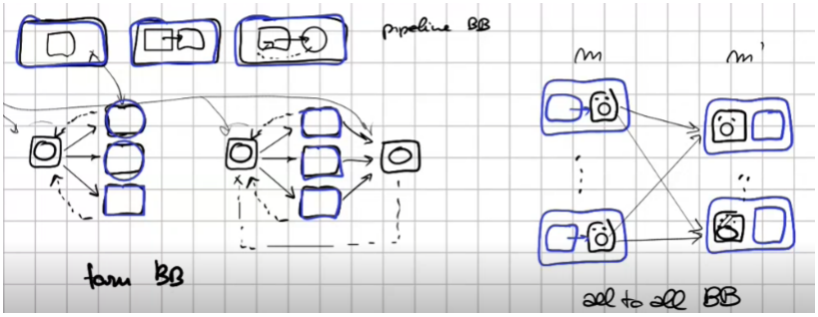
FastFlow Building Blocks Few kind of blocks: sequential, parallel...

Sequential Building Blocks Different kind based on number of inputs and number of outputs.



These can be combined sequentially.

Parallel Building Blocks Pipelines with: single block, two blocks, two blocks with a feedback channel of the second toward the first.
Farm style building blocks: emitter which connect to a set of nodes (workers), collector which takes from the workers. The emitter and collector are sequential building blocks that can be linked to parallel building blocks as workers.
All-to-all building blocks



Graph Transformers For example having a Pipeline with a Farm as intermediate stage, meaning `ff.Pipe<int> mPipe(S1, Farm2, S3);`, we can optimize it including the sequential computation of the first stage S1 in the emitter of the Farm and the sequential computation of the last stage S3 in the collector of the Farm. We can declare a `optlevel opt;` optimizer and then set `opt.remove_collector = true;`, `opt.merge_with_emitter = true;` and optimize with `optimize_static(mPipe, opt);`. All this at declaration phase.

Concurrency Throttling Useful to handle the pressure from the input, adding or removing workers accordingly. It's possible to declare a Farm with a given nw and freeze some of the workers. Meaning that the emitter only perceives the non-frozen workers, the frozen workers are "ghosts" and may be unfrozen when needed.

1.8 Parsec

Benchmarks

Blockslides Portfolio of options, to compute the prices require partial differential equations

Canneal
Simulated Annealing

Dedup

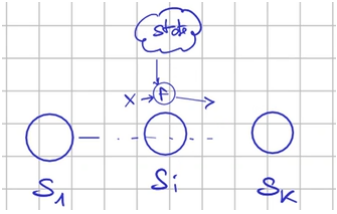
Ferret Pipeline

1. Come up with an initial solution Pipe(S1, S2, S3, S4, S5)
2. Refactoring techniques and possible explore the solution space to find the better solution
   Before coding! Insights on possible performance, possible solutions to discern the better one.
3. Implementation phase of the better solution: OMP...

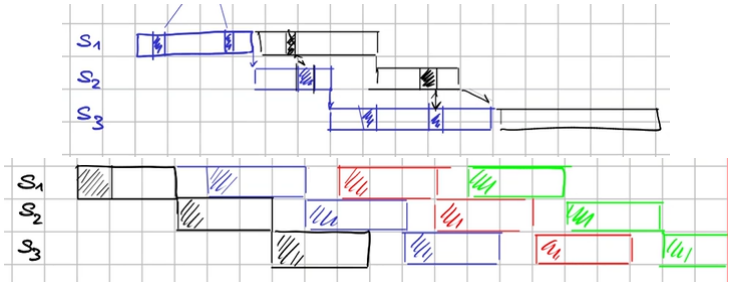
In the project, figure out from the sequential code the values of interest (e.g. time spent in the stages) and once we have a clear idea, refactoring and then implement the best solution.

1.9 Stateful Computations

So far assumed that stateless stages/workers/components. E.g. in a Pipeline, the stages are functions $f, g, h, \ldots$ where we move $f(x)$ to g, $g(f(x))$ to $h \ldots$ and we just compute function, no interaction between the stages.
In a stateful computation the computation depends on: input, state and the function. So not $f(x)$ but $f(x, state)$



We don't mind the order of accesses in the state but it has to be accessed in mutual exclusion. If the accesses to the state happen at the same time, then it would be difficult to compute the times. So we make another assumption $t_i = t_{state_i} + t_{fun_i}$ with t_{state_i} being the mutex.



Due to the state, the mutex, we have an increased T_S due to the fact that each time a job arrives to be computed, it needs to wait that the state is accessed and updated.
We would like to reuse as many concept as possible, from the ones we've seen so far with stateless parallel computations. With data parallel computations, we take a collection of type T, split into chunks and give each to some workers, building an output collection: the idea is that each worker takes something from the original collection and puts something into the output collection. Distributing data from the original collection to the workers, in real-world applications at a scale, is a very costly operation, often implemented with ad-hoc communication channels. So usually, when operating with many data parallel computation, we try to use the map-fusion refactoring rule as much as possible, ideally keeping just one distribution phase and not spending time redistributing data to workers at each stage. So with stateful computation we need to **look at the patterns that can be reasonably exploited**.

Accumulator We have a parallel pattern and a **state pattern** (accumulator). Each parallel activity updates the pattern using a function f (associative and commutative). The result is the final value of the state.
Characteristics are: type of global states and functions and whether or not the intermediate results means something.

Successive Approximation Set of worker in a master-worker configuration (one master emits tasks and receives answers from a set of workers). In this example, the goal is to find a certain value by successive approximations: we are exploring a tree of solutions and we are going towards some best value. What if two workers, w_1 and w_2, return both a solution that is better than the current one maintained by the master? So, $w_1 \Rightarrow \langle$sol'$\rangle$ and $w_2 \Rightarrow \langle$sol"$\rangle$ and we do not know the relationship between the two solutions other than each of them is better than the current sol. Only one of them is the best solution, and in the meanwhile there could be other w_i performing computations that could become useless once we know the new best solution.
What should be avoided, e.g. when sol' is better than sol", is that w_2 updates the state sol *after* w_1 has updated it. So we need an atomic step that performs read-and-update.

Example An emitter and nw workers, followed by a collector.
Without state, $t_S = \max\{t_e, t_c, \frac{t_w}{nw}\}$ because nw tasks, nw workers each t_w giving nw results.
With state, nw tasks with $t_f + nw \cdot t_s$ giving nw results. So $\frac{t_f}{nw} + t_s$. The Speedup$(nw) \simeq nw$. With m tasks

$$\frac{m(t_f + t_s)}{m(\frac{t_f}{nw} + t_s)}$$

$$\lim_{nw \to \infty} = \frac{t_f + t_s}{t_s} = \frac{t_f}{t_s} + 1$$

With k stages, m tasks and $t_f + t_s$ time in each stage per each task

$$\frac{k \cdot m(t_f + t_s)}{m(t_f + k \cdot t_s)} = \frac{t_f + t_s}{\frac{t_f}{k} + t_s}$$

With a large enough k the $\frac{t_f}{k} \to 0$ so the approximation goes like the farm.

Common State Patterns

Resource: lock, replace, unlock

Accumulator: $x_i, s = f(x_i, state)$ with f associative and commutative.
We can use the global value (only) or as local state and finally update the global

Owner Computes: the state is a vector and the ith position can only be updated by the ith concurrent activity. Attention: if using an actual vector this is a source of false sharing overhead, possibly resolved with padding techniques.

Read only: shared state is read only, we can just copy it

1.10 MPI

Standard for clusters of workstations and the high-performance (HPC) community, while OpenMP is the standard for shared-memory multicores.

Message Passing Interface Set of processes plus this MPI that allows to send/receive data or to act with more structured communications. So no pointers but buffers, that are copied. Interfaces means that we need to provide implementations.
So we have some processes, meaning **programs each with their own memory** space, and this **message passing interface** that allows the exchange of data and **enables the possibility of defining more structured computations**.

SPMD Single Program Multiple Data. Reasoning about the structure of the programs, we have a main program that is run in more than one instance and each provides a function that can be used to identify the instance. Given the identifier, we can switch between different portions of code.

Use the library With `mpicc` compiler.

`MPI_Init(&argc, &argv)` at the beginning of the main.

`MPI_Finalize()` at the end of the main.

The code included between these two calls **runs on all the instances** following the SPMD model. It relies on the concept of **communicator**: set of processes capable of communicating. Simplest communicator: `MPI_COMM_WORLD`, hosts all the processes in MPI. Can be splitted in subcommunicators: scatter is a broadcast (everyone gets data including the one that scatters).

To know who I am: `MPI_Comm_rank(communicator, &me)` getting in `me` my ID (one of the numbers between 0-15 if called with `mpirun -np 16`)

To know how many we are: `MPI_Comm_size(communicator, &me)`, returns the total number.

These calls can also be done into a subcommunicator. Use the subcommunicators for example for two set of workers.

Send/Receive Many ways to send and receive.
The message has an envelope. The message has a pointer, size of the data (send contiguous addresses) and type of data. The envelope has ⟨from, to, tag⟩. Tag can be used to discern normal messages, init messages, EOS messages...

`MPI_Send(`

`void* data`, where to take the data to be sent

`int count`, how much data to be sent

`MPI_Datatype type`, e.g. `MPI_Byte`, `MPI_Int`, ecc...

`int destination`

`int tag`, for the envelope

`MPI_COMM communicator` to be used, where the communication will take place

`)`
`data`, `count` and `type` compose the **message descriptor**, `destination` and `tag` compose the **envelope**

`MPI_Receive(`

`void* data`, `int count`, `MPI_Datatype type`

`int source` to tell if I want to receive from a particular host or from any host.

`int tag`

`MPI_COMM communicator`

`MPI_Status* status` to get the final state of the communication

`)`

They are **blocking calls**: the **sender only waits until the message is copied in the buffer**, and the **receiver only waits until the buffer is full**.

```
MPI_Init(...)
MPI_Comm_rank(..., &me)
switch(me):
  case 0:
    MPI_Send(...)
    MPI_Receive(...)
    break;
  case 1:
    MPI_Send(...)
    MPI_Receive(...)
    break;
```

If, with this code, we want that worker 0 sends something to worker 1 and receives something from it, and viceversa worker 1 sends something to worker 0 and receives something from it, this could be a problem if both blocks on the `MPI_Send`. If we need this pattern of communication we usually do something different, e.g. by inverting the order of the calls in the block for worker 1:

```
switch(me):
  case 0:
    MPI_Send(...)
    MPI_Receive(...)
    break;
  case 1:
    MPI_Receive(...)
    MPI_Send(...)
    break;
```

Another, perhaps better, solution would be to use the non-blocking versions: `MPI_IReceive` and `MPI_ISend`. Both use the same parameters but only start the communication, which would be closed by a `MPI_Waitall()` call for both sending and receiving data.
Another way is to use calls that are more general than single send/receives calls: `MPI_Sendrecv()`, with parameters from both and that enables cross-communication between processes and the exchange of data between them.

Collective Distributing data. E.g.

`MPI_Bcast`, broadcast, one sending (root) and all receiving (sender included): the data in the buffer of the sender is copied in each receiver's buffer.

`MPI_Scatter` is similar, but the the buffer of the sender is divided and each receiver gets a portion. You specify the cell counts.

`MPI_Reduce` takes the values in the buffers of the processes applying the operation (e.g. plus) and puts the result in the root's buffer.

`MPI_AllReduce` is similar but the result is duplicated in each processe's buffer.

`MPI_Gather` gets the values and concantenate them into the root's buffer, no operations. `MPI_AllGather` for putting the results in each process's buffer. Remember that it's not shared memory, so with `MPI_Gather` the result can be seen only by the root and there's no way of getting it from the other processes.

1.11 Accelerators

1.11.1 GPUs

Graphic Processing Units Aimed mainly at managing objects on the screen, mainly moving (copying) shapes on the screen. For each pixel, you can do a number of operations, typically in a data parallel fashion.

GP-GPUs General Purpose. We use the same devices engineered to operate in parallel, because the general idea is **vector processing**. They are composed of processors, each takes the instruction flow from the memory and sorts it into smaller children units. In CUDA terminology the processors are called SM (Streaming Processor) and the child units are Cores (powerful ALUs). It follows the SIMD (Single Instruction Multiple Data) model: single instruction to the processor and instructions with multiple data to the cores (vector processing).
GPU's memories have wider access paths that allow to access more addresses at the same time. Can use the most significant bit to select the memory module (each with $k\times$wordsize capacity), using a multiplexer with that bit as selector. If I want to read more than one position at the same time, I can select the other bits to get the "same" address in each module at the same time. This in theory, with **Coalesced Accesses**: accessing blocks of memory that start with the same bits (e.g. changing only the last k bits).

Memory Most GPUs distinguish between global, read only and local memory.
Read only for historical reason, kind of RAM, e.g. storing fixed image patterns used many times.
Global memory is accessible by each core but its slower.
Local Memory is bind to the cores, accessed by clock cycle. Not like normal registers but close.

External GPU are used as accelerators, meaning that they are external devices with respect to the CPU, connected with the PCIe bus. It has its own memory, so when you need to do something on the GPU you need to transfer data to it. In order to compute, for example, a $map(C, f)$ so $\forall\, i \in C\ \ i' = f(i)$, we can do it on the CPU like we've done until now or on the GPU with three steps:

1. Assuming C is in the central memory M, we need to copy C from M to the GPU's global memory
2. Execute the *map* on the GPU that'll write the resulting collection C' somewhere in the GPU's global memory
3. So we need to copy C' from the GPU's memory to M

PCIe has a number of lanes, e.g. 16.
GP-GPU usually have a DMA device that are able to do independent transfers on the PCIe from/to the general memory of the CPU where the bus is attached. The DMA devices are often two, with a specific intent: moving data to/from the GPU while the GPU is computing something else. This means we have to program them.

Integrated GPUs Two kinds:

> GPU on the same chip with a sort of dedicated PCIe express connecting directly to the CPU, otherwise same mechanisms as before
>
> Still GPU on the same chip with a dedicated interconnection with the CPU and possibly with the memory (cache), and perhaps its own memory. This means faster connection because, being a dedicated link there's less overhead due to management protocols.

Modern Non-Integrated GPUs Still GP-GPUs with the option of having dedicated interconnection between them (SLI) such that if we have n GPUs of x SM each, we see the whole assembly as a single nx SMs GPU.

Using the GPU

1. Manage data: moving data to/from GPU memory and also managing allocations in GPU memory.
2. Orchestrate parallel computation, meaning managing communication and synchronization: telling to streaming processors and to cores what to do, meaning what to compute and to which data.
   Usually vendors provide abstractions.

Abstractions nVidia terminology

Grids: groups of threads

Group of threads: a number of warps

Warps: groups of threads (usually 16) need to execute linear, non-diverging codes. Coalescent accesses, one per cycle

So a single activity is denoted by a triple ⟨grid component number, block number, thread number⟩ = $\langle GC, B, th\rangle$ (at least block and thread number). This means that what we have to tell the GPU is what the single thread performs: the unit of computation is called **kernel** and tells what to do in the $\langle x, y, z\rangle$ thread. The values are taken with abstractions provided by the GPU, and you only need to write the computation.

```
cuda Malloc*
cuda Memcpy* //data from host to device
<< >>() //kernel computation
cuda Memcpy* //device to host
cuda free*
```

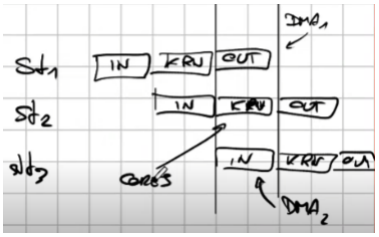
We move $O(m)$ data to the device, compute $O(m)$ and then move back the $O(m)$ resulting data. From the point of view of times: start at t_0 setting up the bus, then send $O(m)$ with a time depending on the speed of the bus, compute $O(m)$ and take back $O(m)$ results. The transfers to and from the GPU is overhead (from our point of view of parallel computation, it's in the range of setting up the data and collecting it, not useful computation). More overhead we have more parallel computation we need to amortize it. So the GPU is useful when the computation on $O(m)$ data is far longer than the time needed to transfer the same $O(m)$ data back and forth. E.g., matrix multiplication: transfer $O(n^2)$, compute $O(n^3)$ operations and transfer $O(n^2)$.
In both cases we can use the **stream**.

Streams Object we declare to send operations to the GPU, made of copies, in and out, and kernels. For example, with DMA_1 copying from the GPU and DMA_2 copying to the GPU, with the cores computing, we can have a situation like

$\text{Stream}_1 = \text{cpyin}(\text{ds}_1)$, kernel, $\text{cpyout}(\text{ds}_1)$

$\text{Stream}_2 = \text{cpyin}(\text{ds}_2)$, kernel, $\text{cpyout}(\text{ds}_2)$

$\text{Stream}_3 = \text{cpyin}(\text{ds}_3)$, kernel, $\text{cpyout}(\text{ds}_3)$



```cpp
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

using namespace std;

void vecaddGPU(float *r, float *a, float *b, int n) {
  #pragma acc kernels loop copyin(a[0:n], b[0:n]) copyout(r[0:n])
  for (int i = 0; i < n; i++) r[0] = a[i] + b[i];
}

// note that the GPU memory is managed automatically and not explicitly by the programmer
int main(int argc, char* argv[]) {
  int n; // vector length
  float * a, * b, * r, * e; // inputs, output and expected vectors
  int i, errs;

  if (argc > 1) n = atoi(argv[1]);
  else n = 100000; // default vector length
  if (n <= 0) n = 100000;

  // allocate vectors on the CPU
  a = (float*) malloc(n*sizeof(float));
  b = (float*) malloc(n*sizeof(float));
  r = (float*) malloc(n*sizeof(float));
  e = (float*) malloc(n*sizeof(float));
  // populate vectors
  for (i = 0; i < n; i++) {
    a[i] = (float) i+1;
    b[i] = (float) 1000*i;
  }

  // compute on the GPU
  vecaddGPU(r, a, b, n);

  // compute on host to compare
  clock_t start = clock();
  for (i = 0; i < n; i++) e[i] = a[i] + b[i];
  clock_t stop = clock();
  printf("Seq time is: %f\n", ((double) (stop - start)/CLOCKS_PER_SEC));

  //compare results
  errs = 0;
  for (i = 0; i < n; i++) {
    if (r[i] != e[i]) errs++;
  }

  printf("%d errors found\n", errs);
  return errs;
}
```

Kernels Even with annotated code, where we don't have to write all the surrounding logic regarding the parallel computations, you have to be aware that you're executing a kernel which is a for loop of independent iterations.
Given some C code, which we want to turn into a kernel, it's relatively easy to turn into an AST (Abstract Syntax Tree). The we can take the list of variables and see those that are read before being written, those are plausibly the parameters we're getting from somewhere else, and those that are written and then never read anymore, which plausibly are the outputs to be reflected on the main memory.
You need a lot of code to **specify everything**. In particular three kinds of commands that you need to take into account:

Figure out the configuration of the board

Define kernel

OCL: compile a string, e.g. a file read from somewhere and with a specific command produce a kernel from the string that is directed to the GPU

CUDA: special keywords that gets the code from the CUDA compiler, which is a wrapper on top of the C++ compiler

Direct command queue: copy out, copy in, kernel exec

OpenCL is both for CPU and GPU, so figuring out the config means how many cores on CPU too. On CPU, can do conditionals and divergent code in general, things you can't do on GPU.
OpenCL can be used for FPGAs too.

1.11.2 FPGAs

Devices with components that can be programmed to act as other components by simply configuring them at the beginning, somehow. Basically we have components like logical gates, small registers, small ALUs, etc. that we can place and configure (almost) as we like.
A FPGA is a matrix where a **single cell** can act like:

a small (5-7 bit input and 1 bit output) **boolean network**. It is made with a kind of EPROM which uses the input bits as an address and stores a single bit output

a **1-3 bit register**

a **router**, redirecting to a neighbor cell (North, East, South, West)

A recent addition: **columns can host** some

RAM blocks 1KB memory for intermediate values

DSP blocks which basically multiply addresses: multiply two inputs, adds the result with a register, stores the result of the addition in that register while propagating it too

These are **organized in columns** so that **there are columns composed solely of cells interleaved with RAM/DSP columns**: a cells column may implement a certain function that stores the result in a RAM column, that may be routed in a way to do come floating point operations in a DSP column, and so on.

RTL Register Transfer Level code used to program FPGA. An example is Verilog:

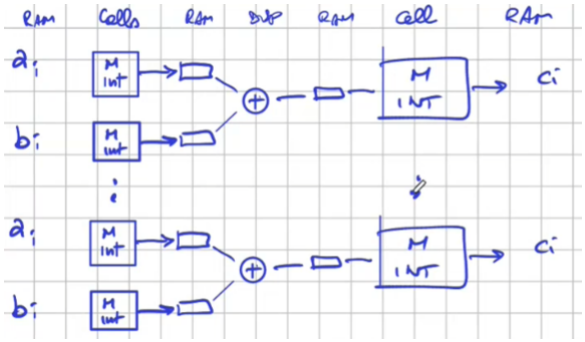
```
module Name (input [N-1:0]x, input [N-1:0]y, output [N:0]z);
  // code example
  assign z = x + y; // creates a block which computes the sum and outputs in z
end module;
```

FPGAs are very good at executing pipelined code, with better performance than CPUs. For example

$$\forall\, i \;\; a_i = b_i + c_i$$

a FPGA may operate with a memory interface able to load a_i and b_i into registers, then an adder which outputs to another register that delivers to a memory interface outputting c_i in the memory. It costs a clock cycle per register level: first level of two registers, then adder and register and a final level with the memory interface which outputs c_i. Hence it costs 3 clock cycles.
On a CPU this would've took 2 loads, the add and the final store: 4 operations each of the fetch-decode-execute pipeline.

Also, this structure (which is a RAM column for the origin of a_i, b_i, a cells column for the memory interface, a DSP column for the add, another cells column for the memory interface and a final RAM column for the output c_i) can be easily replicated in multiple copies obtaining an hardware parallel computation of something already much faster than an original CPU code.



To achieve this you need a complex workflow which can go multiple ways. The classical is:

Write some RTL code

Run the compiler, which solves many problems. In particular, FPGAs are graphs and a task is mapping between tasks which is NP-hard and may take hours to complete

You get a bit stream file that we send to the FPGA (in configuration mode): is basically the entire configuration of the chip, and takes μseconds.

This is simplified, usually the RTL code is firstly run into simulators to check everything from performance to results.

Xilinx Assumes a kernel code written in C with no pointers, no recursion... and other limitations, filled with pragmas with calls to a library of components (e.g. communication channels to interconnect different kernels to a buffer in hardware). With HLS (High Level Synthesis) tools, the "compiler", you get the bit stream.
To operate the bitstream it's a different story. You have .ocl code where normally you have the `createProgram` with the GPU specified, that outputs a kernel that you can submit for execution in a queue, instead of the GPU you specify the bitstream obtained.
Then, you can run on the FPGA or simulate it on CPU/GPU.