19/02/2018

➡ Introduction on parallelism

Parallelism is everywhere.

Def(Flow of control): basicelly a program. Sequence of actions that come one after the other.

➡ History of computers

- Bulbs
- Transistors
- VLSI: very large scale integration
- ULSI: ultra large scale integration

Nowadays transistors and bulbs are printed on silicium.

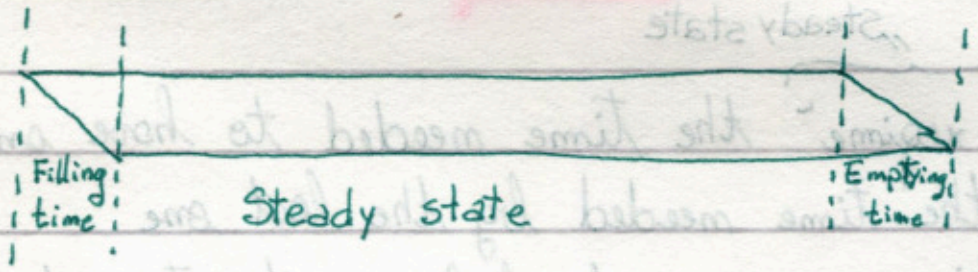
Processors changed from pipeline to superscalar.

At a certain point it wasn't possible anymore to improve performances.

Two ways of going faster:

- speculation;
- trying to guess the truth value of a formula: in particular it's done by making a probabilistic model.

- ✱ COMPLETION TIME
- ✱ SERVICE TIME: time needed to process the second piece starting from the moment I take the first piece



$$T_C = \underbrace{m}_{\substack{\uparrow \\ \text{number of items to be produced}}} \cdot T_S \qquad n_w \rightarrow \text{number of workers}$$

In the scenario of translating a book we are interested in the completion time (ONLY 1 BOOK)

$$T_C(n_w) = \frac{m}{n_w} \cdot T_{\substack{\text{translate} \\ \text{1 page}}}$$

Let us assume we have K books:

1) Teacher splits book → distributes pages → waits → collects pages

After all $T_C(n_w, K) = m \cdot T_C(n_w)$

2) Each worker translates a book.

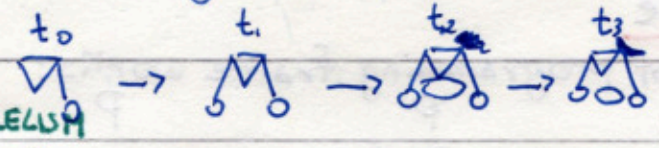
This second approach may elicit the slowdown due to a "slow translator" just by scheduling the tasks in a clever way.

$$T_c(n_w, K) = \frac{K}{n_w} \cdot m \cdot T_{\text{to translate a page}}$$

The point is that the completion time on average is the same, but hopefully more ~~them~~ ~~one~~ tasks are ready "before".

③ Building a bike (assembling parts):

Example of STREAM PARALLELISM



we want to decrease the time

Let's say 4 people may work at different parts of different bikes.

In this situation the bike should be moved and this takes time, but after all the only important time to consider is the time the slowest needs to do his part.

Aim: how many pieces may I put in the market at the end of the day.

Overhead: movement of the bike.

Balance: if all the stage need the same time.

From now on we will consider completion time (T_C) and the service time (T_S)

T_C: taking less time to complete

T_S: doing more things in the same amount of time

(*) Parallel machines do not have the same hardware structure { FPGA, GPU, shared Memory Multicores, clusters for distributed systems

21/02/2018

Three P rule

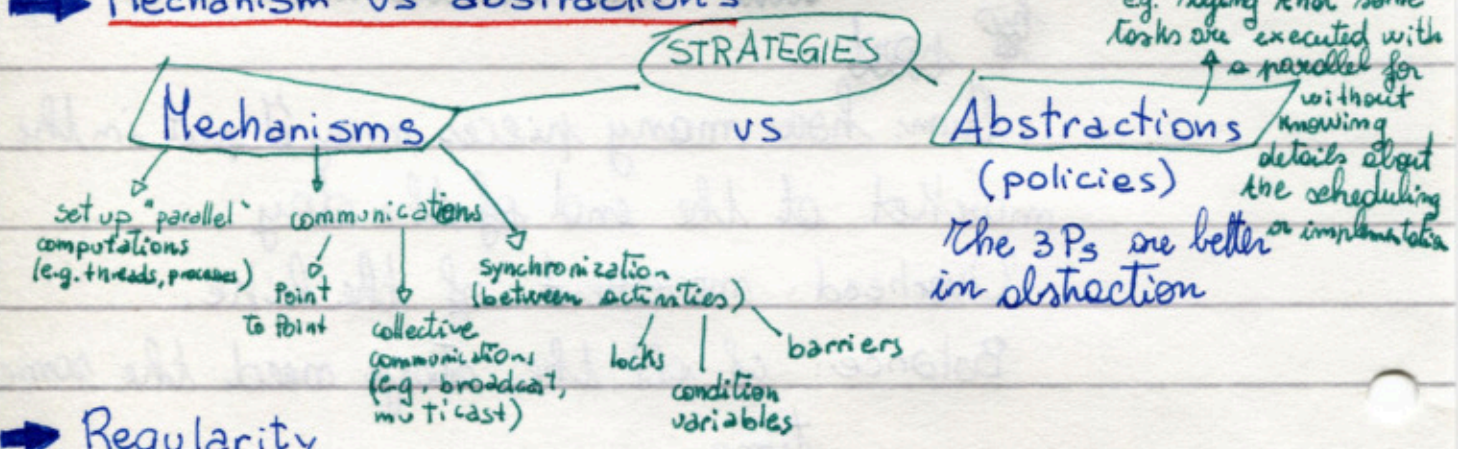
Properties of programming frame works:

P performance
↓
in terms of time & speed
Recently we are interested in energy too

P portability (*)
↓
the main structure of a computer has been the same since the beginning.
I may need to change the code in order to make it work on a cluster of workers.

P productivity (programmability)
↓
the system should provide facility to the programmer

Mechanism vs abstractions



Regularity

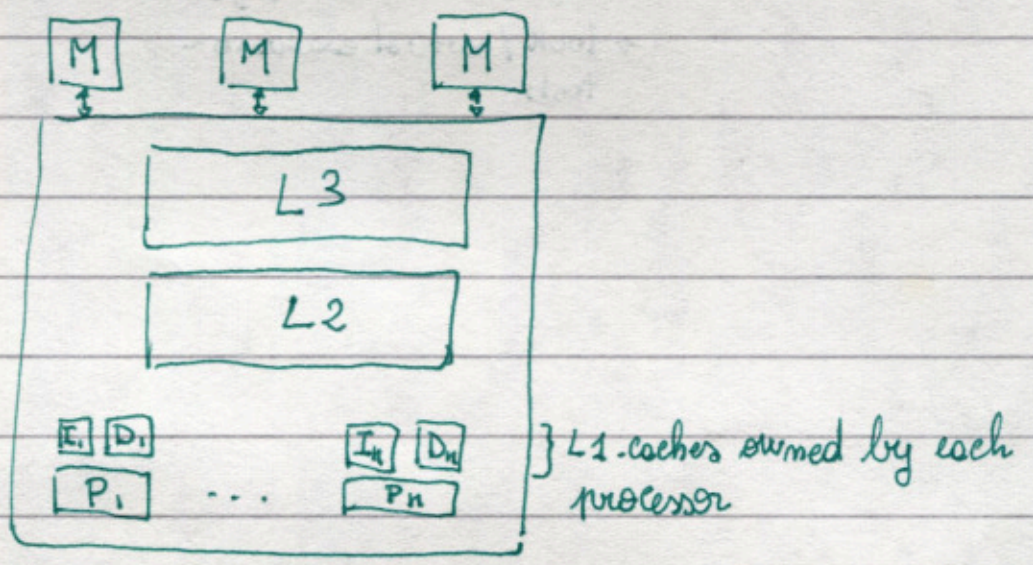
Irregular vs Regular

computing the same instructions on some data

25th February '19

- GrPPI (General Purpose Parallel Pattern Interface)
It is an interface based on OPenMP, TBB, FastFlow, C++ threads.

➡ Multicore architecture

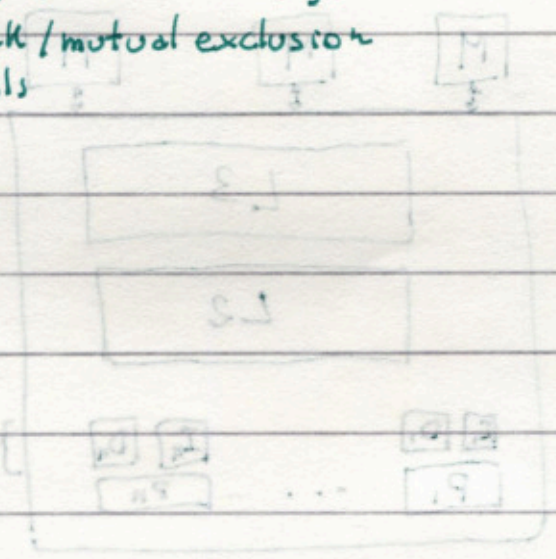


- In order to EVALUATE the performances of a parallel program we need to consider both the abstractions and the MECHANISMS, since they impact the time, depending on the implementation.

needed to be accounted for abstractions development

Mechanism ——overheads——> Abstraction

- → os mechanisms
- → memory usage
- → I/O access
- → communications
- → language features usage
- → lock / mutual exclusion tools



Map: applies the same function to all the elements of a collection

Map fusion rule: $Map(g) \circ Map(f) \equiv Map(g \circ f)$

➡ Composability (nesting)

Property that guarantees that if I know how to work on single bricks I know how to merge them together.

26/02/2014

New time table

Monday: 9-11 C

Tuesday: ~~16~~-18 C1

Thursday: 14-16 C2

➡ Fundamental properties

Def(Safety): the code need to be tested

Def(Mantainability): < modifications following updates, also I need to be aware of how I use non funtional features of a certain library, in order to be update safe.

Def(Determinism): if I give as input "a" the output has to be the same during time.
On the other hand non determinism is used to $[\underbrace{G_1}_{\text{guard}} \to \underbrace{C_1}_{\text{body}}, \ldots, G_n \to C_n]$
manage events from different devices, w/o using priority between events.

→ Parallel vs distributed

Parallel	Distributed
Single system, multiple operations at the same time	N number of interconnected systems able to perform multiple operations

Rsync → software that transmits only the differences to the machines

Problems: - different clock speed between machines

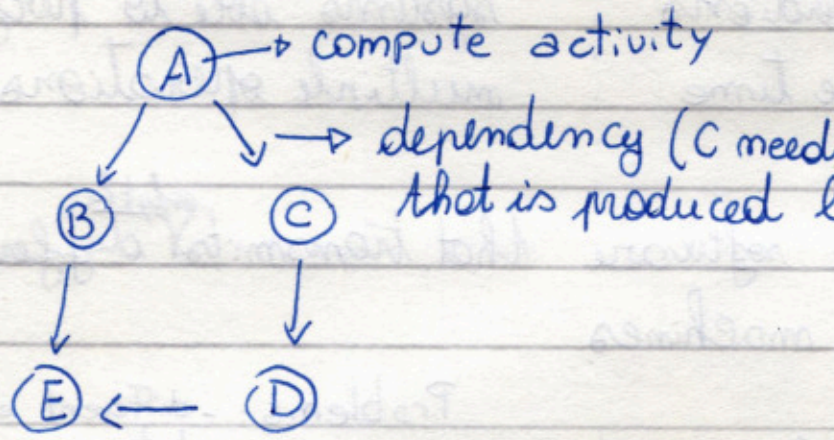
Fast and reliable COMMUNICATIONS	Slow and unreliable

Def(Concurrent activity): two different control flows that may (but not necessarily) take place at the same time.

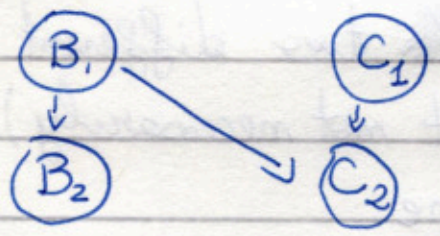
The different control flows may have the one need of the other's output. In order to read that data P_1 may have to wait for some time for that value to be calculated by P_2. Some more time (DATA DEPENDENCY) is needed for transmission.

Def(CONTROL DEPENDENCY): $A_1 ; A_2$

➡ Data flow chart



If the dependencies are different this may be written in order to isolate dependency parts

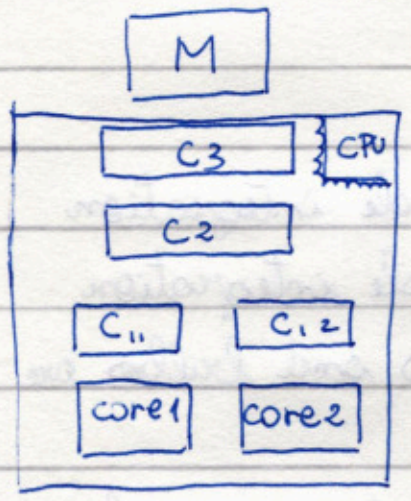


Data flow charts needs to DETECT all DEPENDENCIES in order to let us decide which parts of the program may be splitted into parallel and which may not.

Nowadays +7% in performance costs +15% in space on chips.

It became more popular to double the number of cores.

Modern processors are made like this.



There may be many copies of that CPU, that access the same memory.

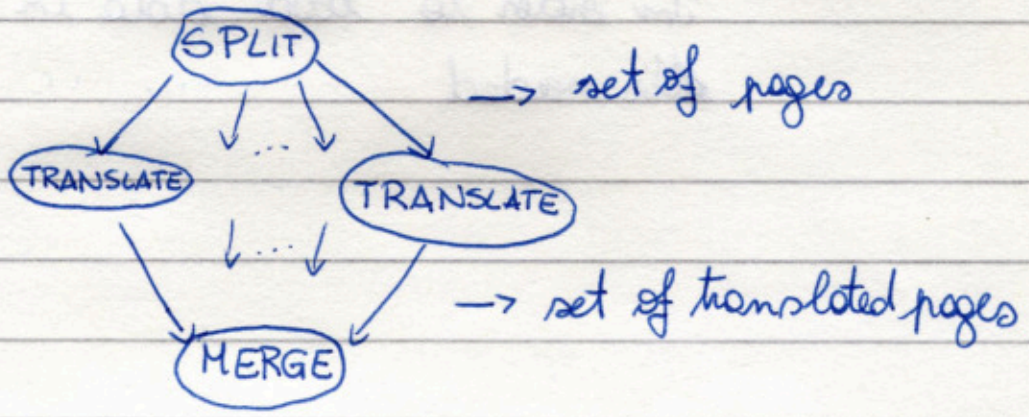
A tablet or a mobile only have one CPU. On a server there are 16 cores and 4 sockets (the state of the art).

➡ Comparison between devices

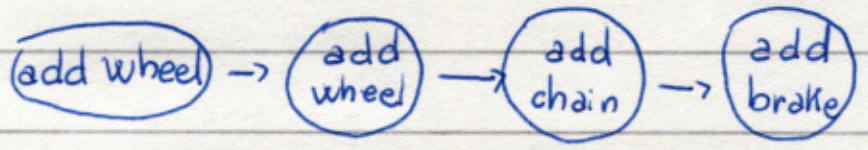
* Power PC: 2 sockets, 10 cores, 8 way-threads each } 160 threads

* Xeon PHI < KNC / KNL : 60-64 cores, single socket

Example: • Translation of a book

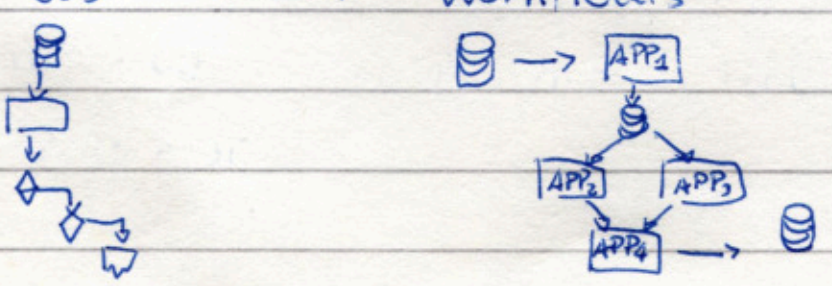


• Building of a bike



Def (SPAN): maximum distance from starting to end
- translation (split - translate (only a bunch) - merge)
- build (wheel - wheel - chair - brakes)

➡ Grids (computational grids)



Example : $x = (a-b) * (c+d)$

In order to load data in the registers the data is needed

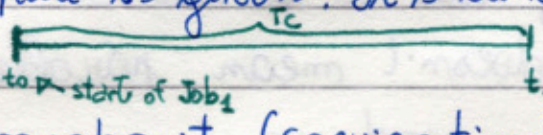
24/02/2015

Performance measures (time mostly)

Latency (completion time)

increased in data parallelism

Accounts to the total amount of Time needed to complete an operation from the time the inputs is given. It's denoted T_C.



we would like $T_C(1) > T_C(n_w)$ where n_w is the parallelism degree

Throughput (service time)

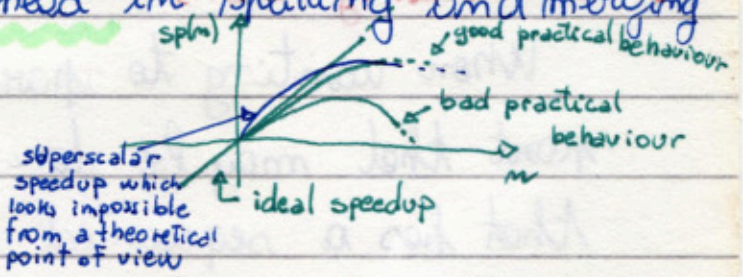
increased in stream parallelism

How many outputs can be delivered in a given amount of time. It's denoted T_S.

eg. tasks per second

Speedup

Ratio (expressed in function of the degree of parallelism) between the best sequencial time and the time needed in the parallel execution $sp(n) = \frac{T_{seq}}{T_{par}(n)} \leq n$. It's $= n$ ONLY if there's no overhead in splitting and mowing the task.



Scalability

Ratio between time spent executing the task once and the bigger task executed by many

executors.

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

Problem of scalability: a very bad ~~sequential~~ parallel algorithm may lead to an almost linear scalability although it doesn't mean parallel execution is done in a good way.

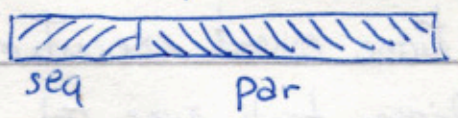
➡ Efficiency

What I would have got using n parallel degree and what I actually had in my implementation of parallelism.

$$\text{Efficiency}(n) = \frac{T_{id}(n)}{T_{par}(n)} = \frac{sr(n)}{n}$$

➡ Parallelizing tasks

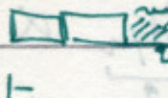
When wanting to parallelize a task there is a part that may be done in parallel and another that has a sequential nature



26th of February '19

➡ Case sp(n)>n

Why is it possible to have a speedup which is greater than the parallelism degree?

Theoretically, given a parallel implementation  if it gets sequentialized is  so the first sequential implementation was not the BEST.

In practice, the sequential program may have a too large working set but the data set we work on is splitted in the parallel version does fit the cache and this motivates the increase in speedup.

Scalability "takes into account" the overhead of creating threads. Moreover, the time needed to manage concurrency impacts T_{par} despite T_{seq}.

Energy performances

The dynamic power is proportional to the square of the voltage and to the frequency

$$\underbrace{P_{dyn}}_{\text{power needed to change a 1 to 0 and viceversa}} \overset{\text{proportional}}{\alpha} V^2 f \approx f^3$$

The static power relates to the functionalities of the operating system (very low amount of computation or idle state).

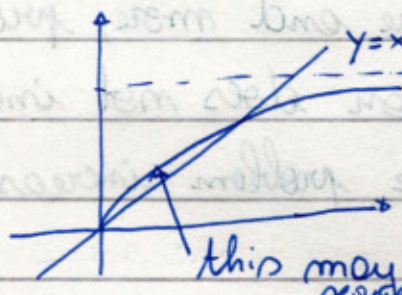
Def (serial fraction): f is the percentage of a task that is sequential. So we can say that $T_{seq} = f \cdot T_{seq} + (1-f) \cdot T_{seq}$

$$sp(n) = \frac{T_{seq}}{f \cdot T_{seq} + \frac{(1-f) \cdot T_{seq}}{n}} \Rightarrow \boxed{\lim_{n \to +\infty} sp(n) = \frac{1}{f}}$$

Amdahl law

ideal parallel time

Sometimes speedup may have this strange behaviour



this may be due to situation like the following:
- the ~~parallel~~ sequential problem may be so big so it leads to trashing
- the parallel problem may indeed fit better the cache

Obs: Amdahl law is important but there are some cases where the overhead to split the problem into n pieces is the same than to split it into m pieces, no matter on the values of n and m.

Example: Splitting an image into to pieces has the same cost if the image is small or big.

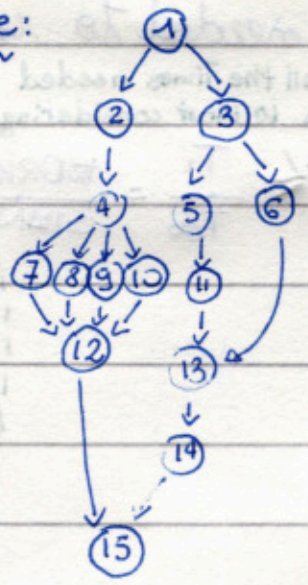
If the serial fraction becomes the smaller percentage the more I'm increasing the size of the problem.

Def(Gustafsonn): If the grain of the data becomes finer (the amount of data increases) the serial part is amortized more and more, provided that the serial fraction does not increase when the size of the problem increases.

WORK-SPAN MODEL

28/02/2018

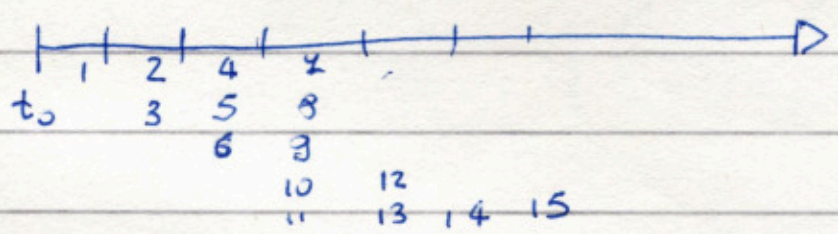
Example:



If each steps i needs t time the sequential program takes $15t$ to complete.

In order to find the execution time of the parallel algorithm the finable (ready to be executed) nodes should be found.

Def (critical path): the longest (in terms of execution time) sequential paths in the data flow that represents a lower bound to the parallel completion time.



In the previous example the critical path is 1-3-5-11-13-14-15.

The time needed to execute the critical path is called SPAN or T_∞

WORK or T_1 is the time needed to complete the sequential execution. [sum of all the times needed in the parallel execution without considering the fact that they are executed in parallel]

In these new terms $SPEED\text{-}UP \leq \frac{T_1}{T_\infty} = \frac{WORK}{SPAN}$

WORK: T_{seq} = SPAN: T_{par}

➡ Brent's lemma

$$T_{par}(n) \leq T_\infty + \frac{T_1 - T_\infty}{n}$$

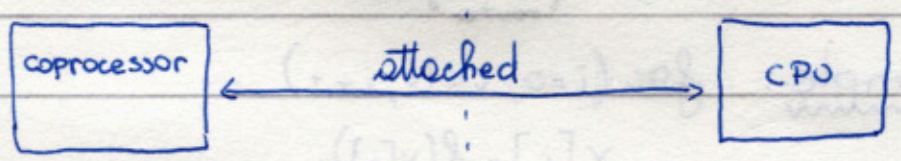
Def (race conditions): situation where more than one control flows that share memory access to the same variable (there's a least a write operation).

Def (power wall): the need of power goes with the square of the frequency.

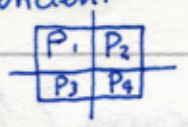
Def (problem of frequenty): if it's too high it's hard to keep it cool

GPU vs CPU

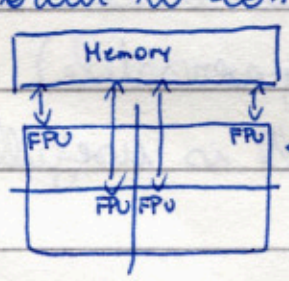
graphics processing unit



Takes care of the drawing part.
It splits the video into portions which are completely indipendent



At a certain point FPU where introduced in GPU, in order to compute easy operations.



← compute easy operations in parallel

Nowadays NVIDIA GPU have 100 FPU inside

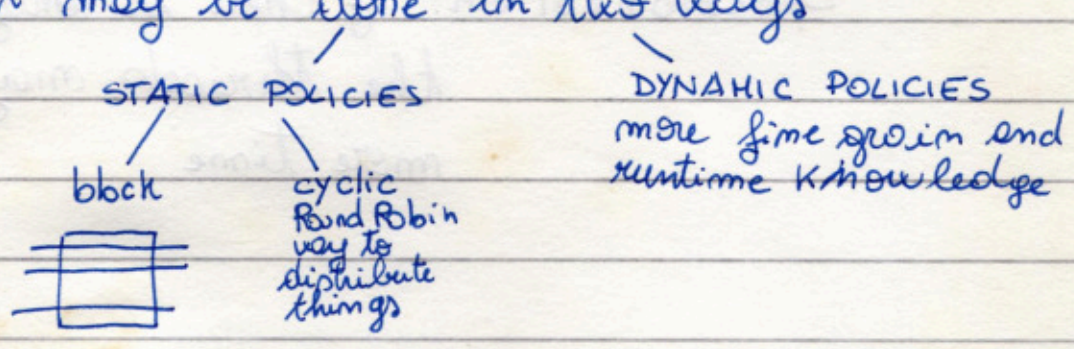
FPGA (field programmable gate arrays) is useful to compute stuff like MATRIX-VECTOR product.

6/03/2018

Def (locality): It may be in time or in space and it concerns the fact that an access may lead to an access to a near place or access to the same point in a short period of time.

Def(cache oblivious algorithms): algorithm to be efficient not only in calculations but also in cache management.

Def(load balancing): strategy used to understand if the computation is equally spread across the resources.
It may be done in two ways



Obs: the situation needs to be studied deeply in order to choose the best policy on that kind of data.

Dynamic policies

more overhead ↑

ON DEMAND — JOB STEALING

Example: |1|2|3|4|5|6|7|8|

Block:		CYCLIC:	
W_1	1 2	W_1	1 5
W_2	3 4	W_2	2 6
W_3	5 6	W_3	3 7
W_4	7 8	W_4	4 8

It's important to know the dependencies.

➡ Overhead

- Memory allocations may lead to ~~alloc~~ race for the heap
- Thread/process pool setup
- Synchronization: if there is only one lock the threads may wait for more time

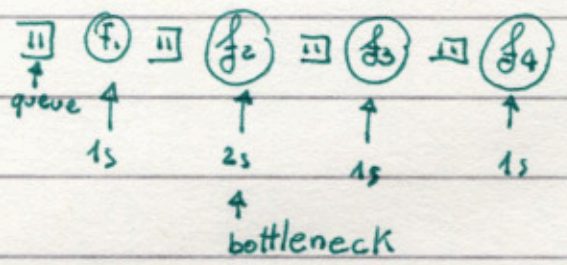
28th of February

Efficiency (def.): The ratio between the parallel time divided by the parallelism degree and the parallel time. $\varepsilon(n) = \frac{\frac{T_{seq}}{n}}{T_{par}} = \frac{T_{sec}}{n \cdot T_{par}} = \frac{sp(n)}{n}$

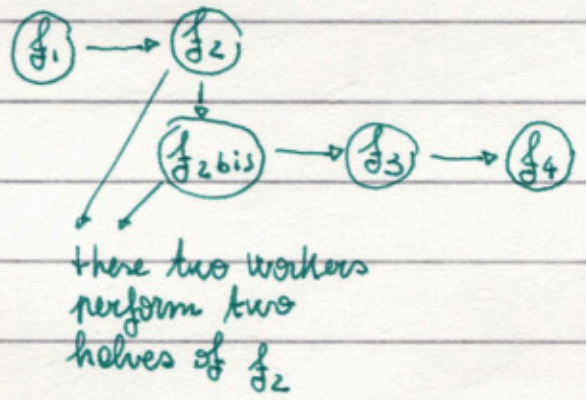
Bottleneck (def.): Those tasks f_i such that $\rho_i = \frac{T_{si}}{T_{Ai}} > 1$, hence the queue of the pipeline is filled sooner or later.

(T_{Ai}: arrival time of service)

Example: Let us assume we are in this situation

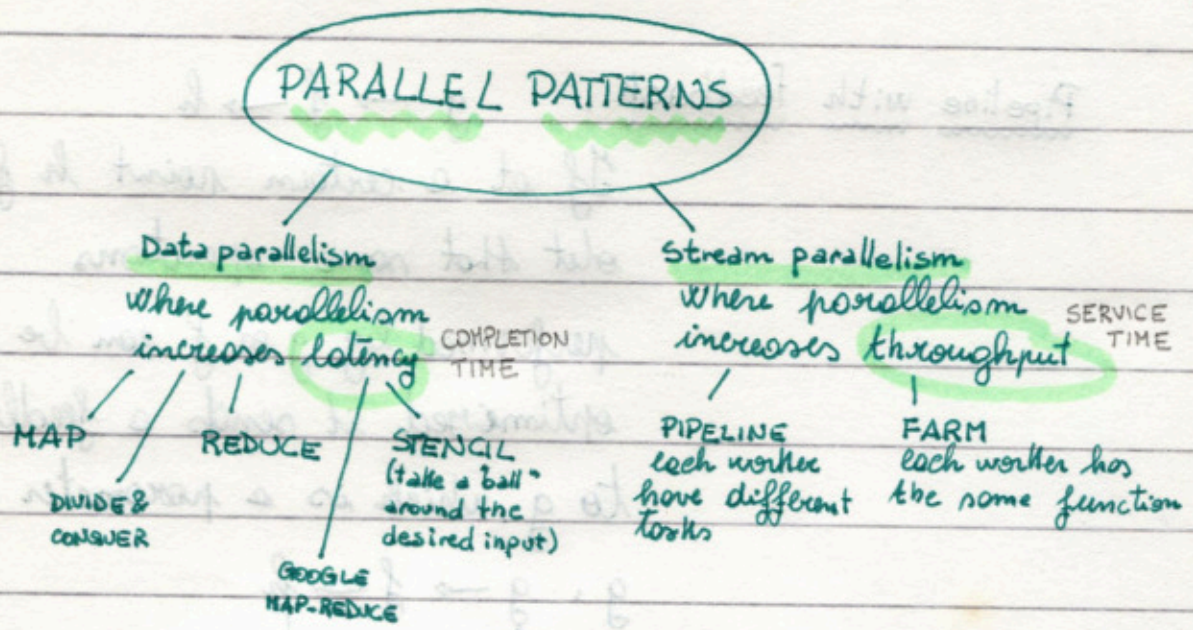


The bottleneck may be eliminated adding a worker



28th February '19

PARALLEL PATTERNS



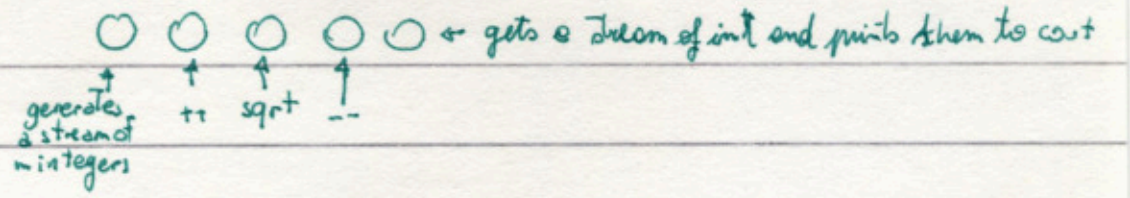
4th of March 2019

Pipeline with feedback: $g \longrightarrow f \longrightarrow h$

If at a certain point h finds out that some operations performed by g or f can be optimized it sends a feedback γ to g, which is a parameter of g: $g \longrightarrow f \longrightarrow h$, with γ feeding back from h to g.

5th of March 2019

Assignment 1. Pipeline mode of 5 stages.



These functions should use some delays (10 msec)

Goal: show that the computation takes about
10 msec * n & measure the overhead
(all time not relative to sequential computation)

Thread are numbered round robin among contexts

We must use 1 thread per stage. Possibly stick threads to cores.

USE END_OF_STREAM.
in order to check if a queue is empty

8 marzo 2018

ESERCITAZIONE

➡ Owner writes rules

Many threads may do things that it looks like they access the same part of memory. Actually they not necessarily do.

How to find out the time needed for computation?

- $t_1, t_2, \ldots, t_K \rightarrow \frac{\sum t_i}{K}$

$\rightarrow \frac{\sum^* t_i}{K}$ $\sum^*$ without outliers

$\rightarrow$ minimum

Which is the status of the machine?

- top → top activities
- uptime → Time since the last reboot + users logged + load average [-(last minute), -, -]
- w → show what people are doing
- at → to start a process at a certain time
- crontab → it can start processes many times it needs a crontab file

In order to be able to read the outputs of the program it may be written on a file or opening a script that records the output of the terminal.

- awk → text processor which gives me the chance of reading selectively the terminal output obtained through script.

Always pay attention on what is printed, because it's necessary for us to be abll to find out many things in the future.

PATTERNS
(2° part of the course)

➡ MAP pattern

It takes a collection of a given type and applies a function to all the elements of the collection

Def (Data parallel): characteristic of a set of data that allows parallel calculation

Data parallel tells us that we want to reduce LATENCY.

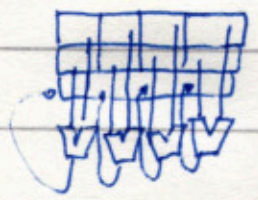
Phases :- split + assign subcomputations to the available processes

- parallel computation
- merge of results

➡ Auto vectorization

Vector registers

whose parts are inputs of many ALUs.



Example: the optimisation (regarding load-store and other operations) is made by the compiler.

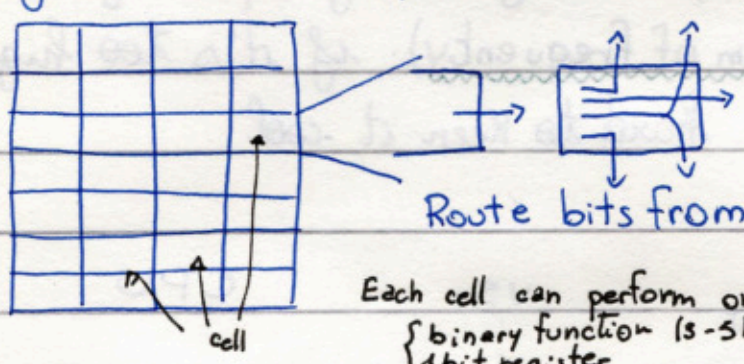
Conditions to vectorize a loop:

- countable number of iterations (NO WHILE);
- NO break points (otherwise the compiler doesn't know how to foresee behaviours);
- NO conditionals
- NO function calls, because if I call a function there is a transfer of control
  exceptions, listed in intel compiler

- fopt-info ← option to use on gcc in order to vectorize

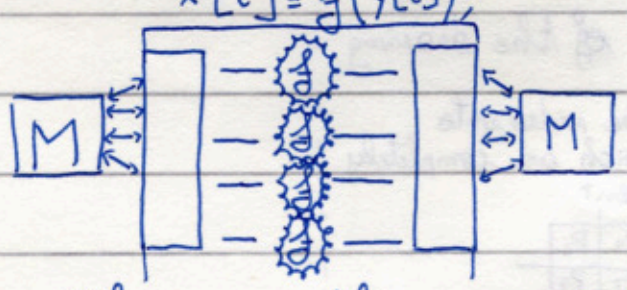
it is different from GPU because it may perform operations that are not "chosen" by the producer

FPGA is a MATRIX that keeps a bit and compute a bit function: 3-5 input bits → 1 output bit

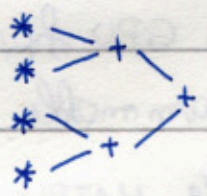


Each cell can perform one out ~~of~~ 3 operations:
- binary function (3-5 bits to 1 bit result)
- 1 bit register
- router

Example: `for (i=0; i<N; i++) x[i] = f(y[i]);`



Why using this processor? These elements need a small clock cycle and the chip may be used to calculate g instead of f (it's very versatile).
The routing skill is usefull in this situation:



7th of March 2019

Cache coherence (def.): The memory that is accessed by any thread has the same value.
There are always some instants of time in which the data stored in two different caches differs, except for the event in which there are more complicated mechanisms for example some sorts of locks that prevent threads from accessing some information until the edit performed by thread t has been propagated.

Computation grain (def): The ratio between the time needed for computation and the time needed for communication: $\text{Grain} = \frac{T_{comp.}}{T_{comm.}}$

It goes without saying that the coarser the grain the better the parallelization performance.
The issue here is that when the grain is coarse the amount of data and computation is big. hence requires a clever management of memory accesses.

False sharing (def.): it is a performance-degrading memory usage pattern that takes place when a thread periodically accesses data (in its cache) that is never altered by another thred. This leads to unnecessary overhead to manage coherence.

Single owner computes rules (def.): The situation in which two threads access the same portion of memory, ~~es~~ cached in c_1 caches of both threads (which are different).

In order to overcome this issue, we could design things in such a way the memory accessed by each thread is the beginning of a cache line.

Memory pressure (def.): The work that the operating system does in order to manage memory among many users.

Memory (or computation) intensity: (def.) the amount of load/store operations performed before some register operations.

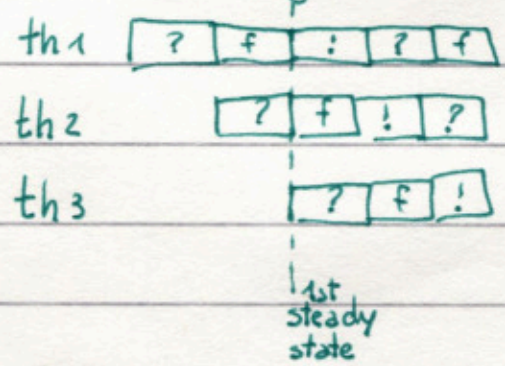
D. very high computation intensity should be managed carefully when writing parallel code

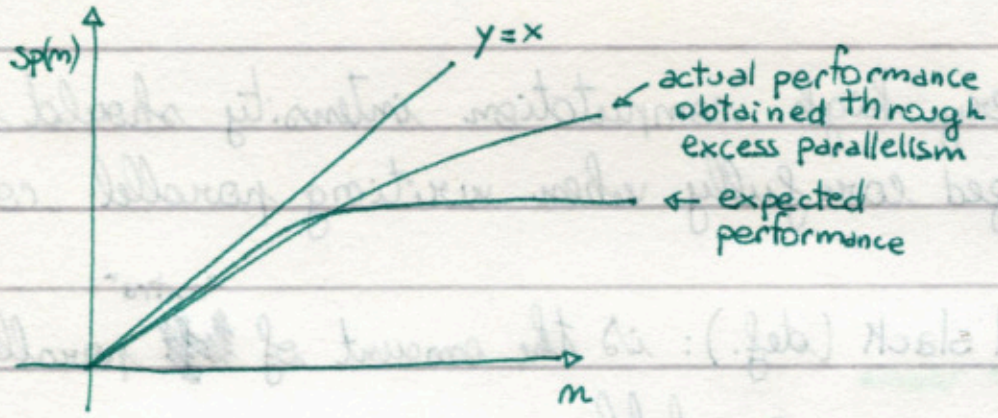
Parallel slack (def.): is the amount of "extra" parallelism available.

EXCESS PARALLELISM AND PARALLEL SLACKNESS

Double/triple buffering (def.): it is the strategy of masking the idle time assigning to a waiting thread some other activities.

Example: Let us assume there is a shared buffer and there are 3 kinds of operations (SEND, COMPUTE, RECEIVE).
At the steady state the operations are spread among the threads





Notice that there is a very thin range of parallelism degrees that take advantage of extra parallelism without suffering for the increasing overhead.

Notice also that the usage of condition variables, or locks leads to more SHARED MEMORY between multiple threads, hence incurring in a more severe FALSE SHARING.

12th of March '19

→ OMP syntax

#pragma omp task depend(in: x) → does not start executing until x is ready

#pragma omp task depend (out: x) → provides x as an output

#pragma omp task wait() → executes the code that follows as soon as the two threads finished

open MP is not good to use in pipeline algorithms.

OpenMP is the standard tool used for implementing parallelism in operations, because it takes care of managing more machine. Our point of view is that open MP is a bit primitive. Example:

#pragma omp task shared(i1)

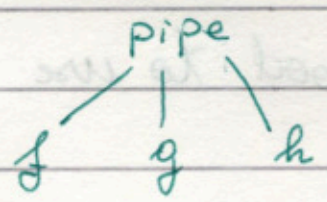
It is atos low level library

OpenMP is GREAT for independent Tasks.

➡ ~~RPLSH~~ RPLSH

This library is born from the idea of finding the BEST PARALLEL SOLUTION of a certain problem that SATISFIES REFACTORING RULES.

Example



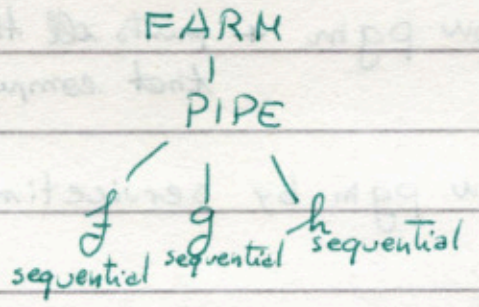
s.t. t_f = time needed to compute f(input)

$t_g = 10\, t_f$

$t_h = 9\, t_f$

If we build a FARM on g and h, using the appropriate amount of workers to reduce the waiting time. How to choose the proper amount of workers? We need to tune the algorithm or USE A LIBRARY that does such a thing by itself.

On alternative solution would be parallelise as



This process is very difficult to execute at run time, hence we would like to use some theoretical arguments.

rplsh

```
s1 = seq()
s2 = seq()
s3 = seq()
pgm = pipe (s1, s2, s3)
annotate s1 with latency 10
         s2          100
         s3           90
show pgm
show pgm by service time
show pgm by latency
```

rewrite pgm with allrules

show pgm ← prints all the possible patterns trees that computes the same result

show pgm by servicetime ← prints each pattern sorted by service Time

optimize pgm with farmopt

show pgm by servicetime, resources
↳ return the actual number of threads needed to implement such solutions

We can do even better: we can tell to rppl sh to the number of context we have available on our machine

MISSING COMMAND

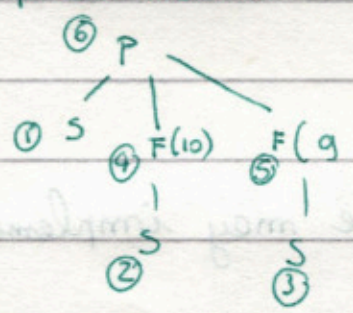
If we have a number of contexts which is << #resources of the solution we may proceede decreasing the number of resources used by some parts of the code.

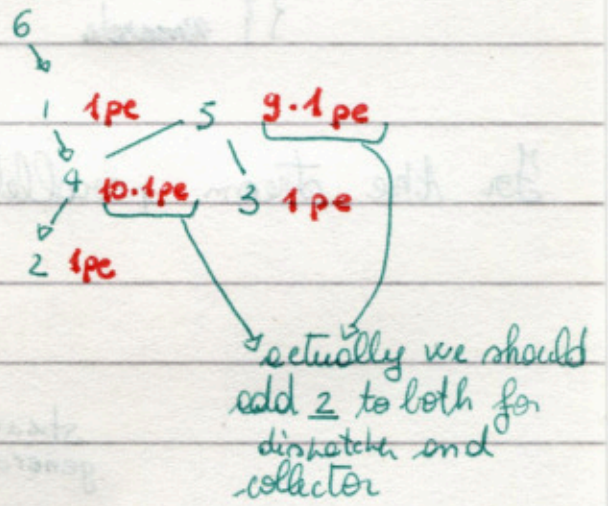
Fast flow also takes as input real c++ functions

PARALLEL TREES GENERATION

19th of March 2019

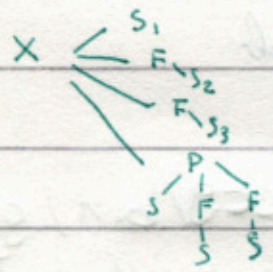
How rplsh works.





Overall we need $1 + 10 \cdot 1 + 2 + 9 \cdot 1 + 2 = 24$ pe

We are interested in GENERATING NEW PATTERNS.
How? Find a target x and substitute it with all the possible patterns.



eg. substitute F / S2 with F / F / S2

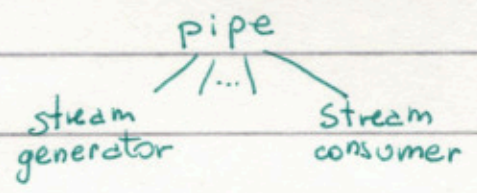
GrPPI
General Parallel Pattern Interface

Stream -> is not a type, but it is implemented as

optional<T> [loaded using #include<experimental/optional>]

In order to express "no value" we may use {}

For the stream parallelism we may implement a pipeline



In order to compile: g++ p1.cpp -I usl/local/include/grppi -o p1 -pthread

! farm can only live inside a pipeline !

Download at: arcosuc3/grppi on Github

Compile with /usr/local/gcc9/bin/g++

cat ~/.bashrc add two lines (listen to last 10 minutes of lenau)

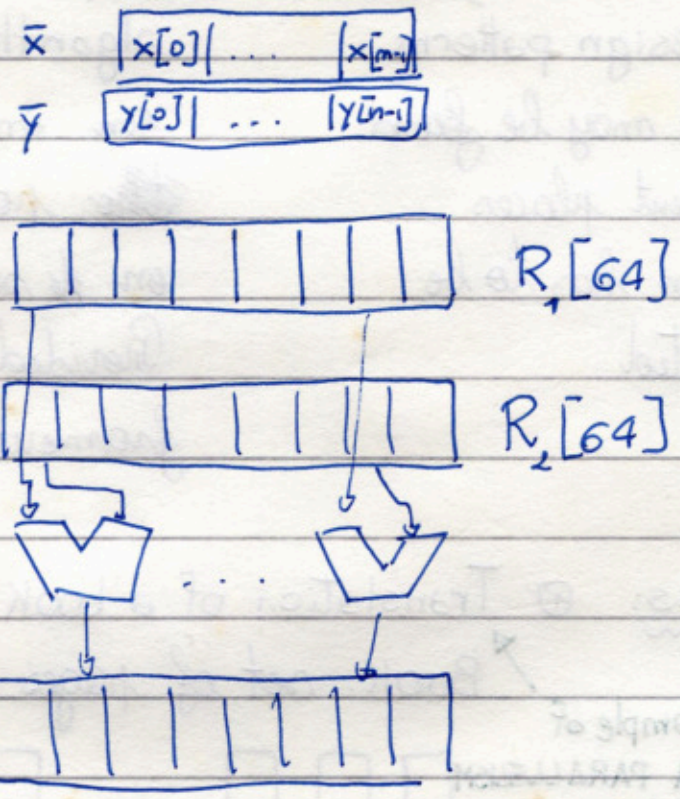
Basic tools for parallelism

23/02/2018

FPGA: Altera → bought by Intel
      Xilinx

Def (Vectorization): technique that allows to make operations all together instead of one after the other.

Example:

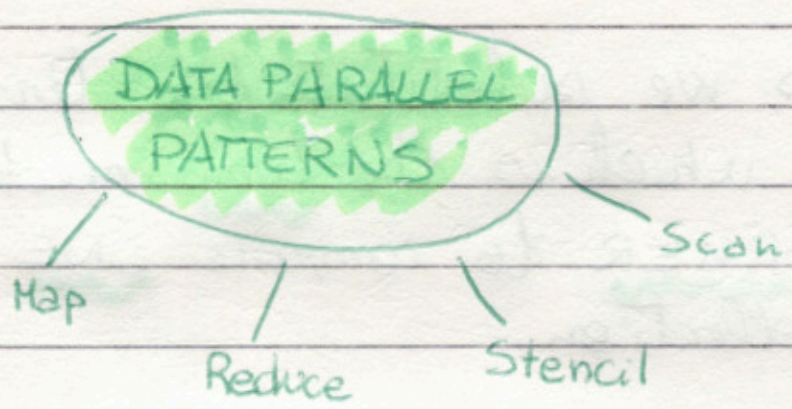


Vectorization is possible only in that cases where operations are parallelizable.

Example: `for (int i=0; i<N; i++) y[i] = x[i] * x[i]` ≠ `for (int i=1; i<N; i++) y[i] = x[i] * y[i-1]`

(y[i] depends on y[i-1])

18th of March 209



MAP

collection <type> —— f: type -> typeout

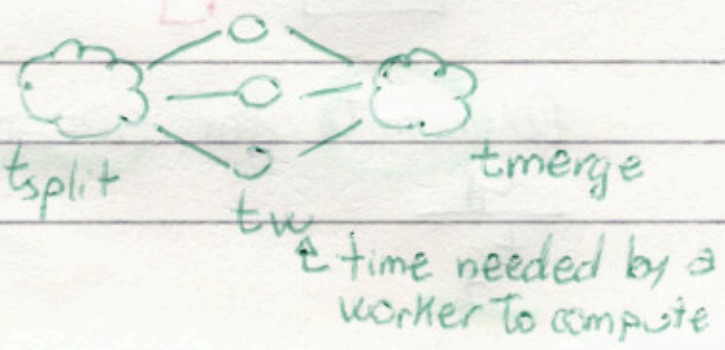
collection <type:out>

The main idea is to split (partition) a collection of data in order to divide the execution time.

We need to take into account overheads

- split : collect -> {partitions}
- merge : {results} -> res_collection

We model this situation as follows



t time needed by a worker to compute

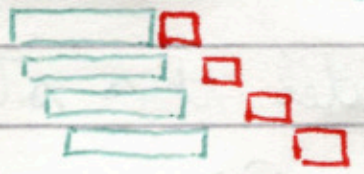
Sometimes we substitute the term t_w with t_g, which is the time needed by a WORKER to compute ONE ELEMENT of the collection.

What if the amount of work to do is not divided by the number of workers?

The REMAINDER of the division is split among the workers.

Variation: a STATE s needs to be computed as well.

Issue: if we need mutual exclusion for the state access the speedup is theoretically bounded



□ computation of g function for the state

O(limit for SPEEDUP is $\frac{t_f}{t_g}$

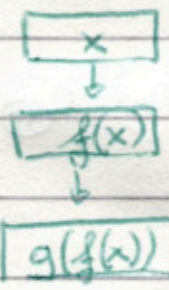
STENCIL

The input is "enriched" by a portion of its neighbourhood

Issue: cannot compute "in place"

Example → array. f computes the sum of an item and predecessor and successor and divides it by 2

MapFusion Rule (def.): comp(map(f) map(g)) = map(f∘g)



It works with pipeline instead of comp as well

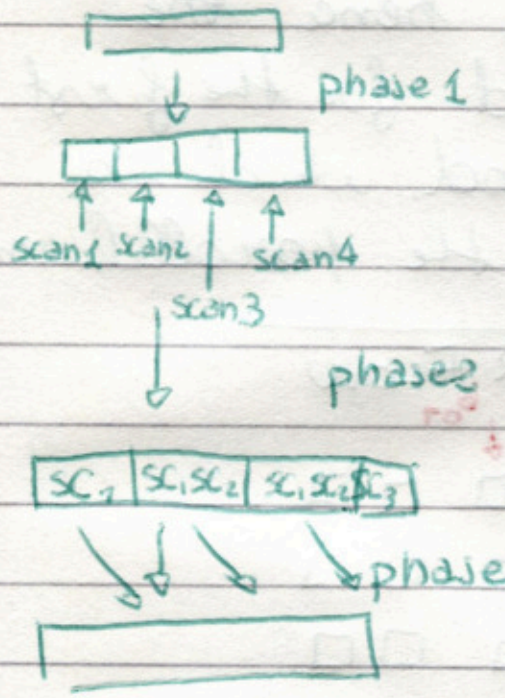
Idea: in the RIGHT part the map operation

is performed in COARSER GRAIN, hence leading to smaller overheads.

There are some libraries such as SKETO that perform optimizations based on the MAPFUSIONRULE.

REDUCE	SCAN
Applies a function to a collection and obtains one item	The function is "incremental" with the elements of the collection
ASSUMING COMMUTATIVE AND ASSOCIATIVE OPERATOR ⊕	$\oplus: x_1 x_2 x_3 x_4 \rightarrow x_1 \mid x_1 \oplus x_2 \mid x_1 \oplus x_2 \oplus x_3 \mid x_1 \oplus x_2 \oplus x_3 \oplus x_4$
Parallelism may be implemented via a "tree" of binary computations or via a queue of "do available" operations	$T_{seq} = m \cdot t_{\oplus}$ [since we stored the partial result]

Example of scan



1 2 3 4 | 5 6 7 8

1 3 6 (10) → this is the candidate to be summed to the final result

1 2 | 3 4 | 5 6 | 7 8 |

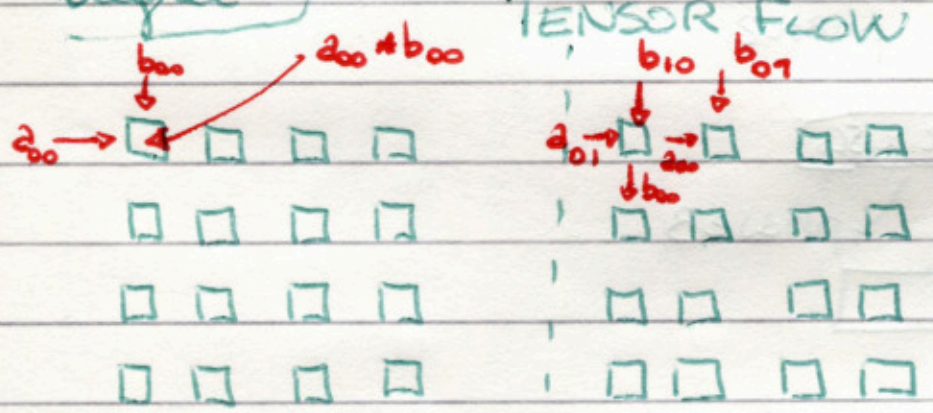
1 (3) | 3 (7) | 5 (11) | 7 15 |

1 3 | 6 (10) | 5 11 | 7 15 |

1 3 | 6 10 | 15 (21) | 7 15 |

1 3 | 6 10 | 15 21 | 28 36 |

The number of operations is twice the sequencial case, hence the speedup is bounded for the first part by this overhead, which is later amortized by the parallelism degree



19th of March 2019

Examples of projects

→ Parallel prefix

→ Image Watermarking

We does not preprocess the WM.

Requirements: check dimensions of images and dtomp

syntax CImg<unsigned char> img = ...;

img.width == ... && img.height

1st parallel solution: parallelizing on the images: any thread proceses a bunch of contiguous images in the order they are returned by folder/*

Obs: assigning new threads we decide that the last one takes all the images left (potentially more than the other threads).

21st of March 2019

In IMAGE WATERMARKING we can imagine that the time needed for computing on image is

$$t_w = t_{wPAR} + t_{wSEQ}$$

Hence the model with n_w parallelism degree is $\frac{t_{wP}}{n_w} + n_w t_{wS}$ (for the disk operations)

The parallelism on ALL THE IMAGES may be implemented via OPENMP in two ways

- Parallel for
- thread + wait thread ; we need to identify the tasks

```
#pragma omp parallel
#pragma omp task before
the call
```

```
pragma omp parallel (on top)
  pragma omp single
    for( ... ) {
      pragma omp task
        forEachImage()
```

GrPPI

```
pipeline (
          step 1 <- should create a stream of integers
          step 2 -> implemented via farm
          step 3
)
```

Once we have encapsulated step2 into a function, we want to SUBSITUTE the FOR LOOP with a PIPELINE

```
auto genstream = [&] () -> experimental::
                           optional<int>{
  static int i = 3;
  if (i < argc)
    return (i++);
```

end of stream in GrPPI

```
    else return {};
};

auto fakedrain = [&] (int i) {
        cout << "Image " << argv[i] <<
            "has been processed" << endl;
};
dynamic_execution thr = grppi::parallel_execution_native{};
pipeline (thr,
            genstream, farm(nw, forEachImage),
            fakedrain );
```

to compile: g++ -lX11 -I usr/local/
include/grppi/ -pthread -o
exe prog.c

+ we need #include<grppi.h>

obs: in the check of dimensions
in forEach Image we should
not use "continue" (we are

The first use of "for" allows the programmer to use "parallel_for", so that the optimization g++ -fopenp parallelizes the "for".

ABSTRACTIONS

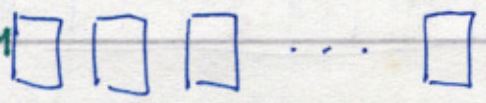
parallel design patterns
A pattern may be found in different places
A pattern has to be implemented

algorithmic scheletons
an implementation of patterns that depends on some custom values
Provided on programming frameworks

Examples: ① Translation of a book

Book: set of pages

Example of DATA PARALLELISM called MAP or PARALLEL FOR



problem: what if a sentence goes across pages? for simplicity we won't consider it.
If it takes to me ½ h to translate a page I will need $\frac{m}{2}$ h to translate the book made of m pages.

not in a for loop anymore).
We should use instead "return(0);

How to check that the real bottleneck is disk access?

Replicate in memory one image, copy it in RAM and spread the computation on many resources.

The results should be discarded in order to avoid the bottleneck in disk writes (exept for the first, otherwise it is hard to check if the result is correct).

Notice that the optimization flag O3 may force a store of the final result.

- farm(pipe(L, P, S))
- pipe(L, P, S)
- farm(comp(L, P, S)) ← NORMAL SOLUTION
- farm(pipe(L, farm(process), S))

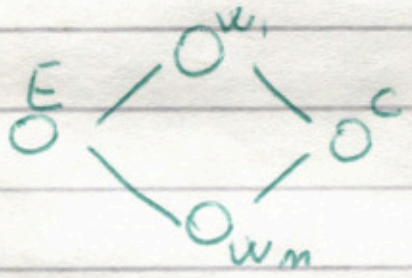
which is the once we computed all the rpl5h models the theoretical problem is that there is the DISK bottleneck

26th of March 2019

TEMPLATE BASED

In the pipeline pattern the service time is the MAXIMUM of the service times of all the tasks $\{T_S = \max\{T_{S_i}\}\}$

For example the FARM pattern



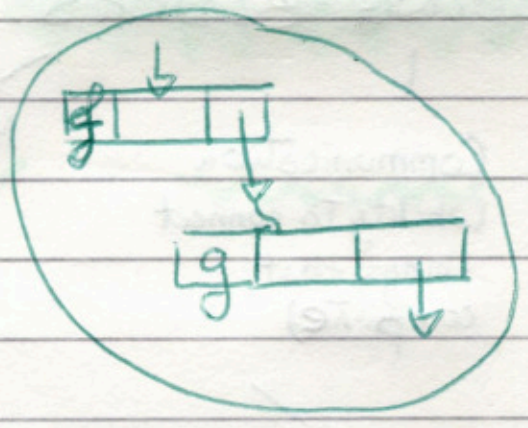
may implement EMITTER and COLLECTOR as threads or as a passive data structure.

Here, $T_S = \max\{t_e, t_c, \frac{t_w}{n_w}\}$ for the bandwidth

MACRO DATA FLOW BASED

Here we have a graph of dependencies

farm(pipe(f, g), 2)



this happens to any input x:

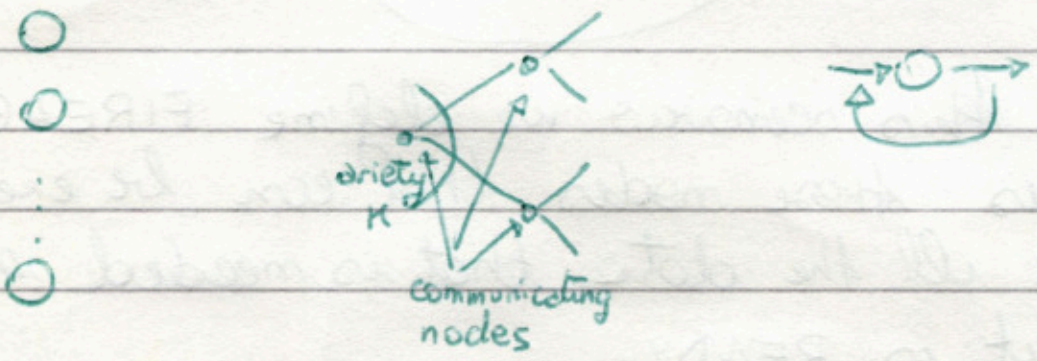
In this scenario we define FIREABLE nodes those nodes that can be executed since all the data that is needed as its input is READY.

Implementation: each thread of a thread pull pops a FIREABLE task from a 'REPOSITORY of MACRO DATA FLOW INSTRUCTIONS'.

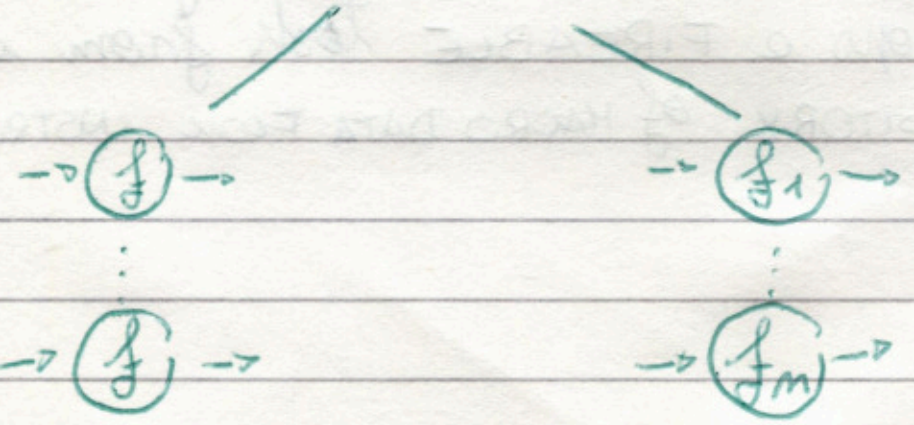
We are now interested in finding the constituting elements of both templates and MACRO DATA FLOW structures

PARALLEL BUILDING BLOCKS

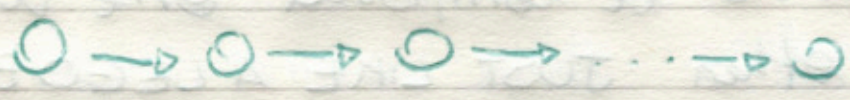
- **Computational** (e.g. number of identical replicas of a pbb)
- **Communication** (ability to connect things that compute)
- **Modifiers**



- COMPUTATIONAL BUILDING BLOCKS



A computational stage may be used in the context of a pipeline



On the other hand, also a REDUCE pattern may be considered a COMPUTATION.

- COMMUNICATION BUILDING BLOCK
  from one to many
  or
  from many to one

- FEEDBACK MODIFIER
  One predicate tells whether to go one direction or the other way round

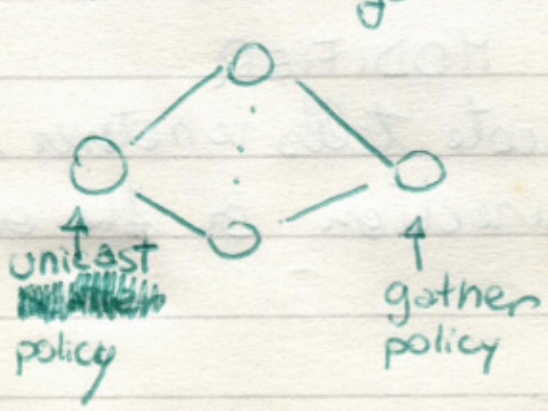
a merge operation may be needed

All these building blocks need INPUT and OUTPUT channels in order to be composed one with the other JUST LIKE A LEGO BRICK.

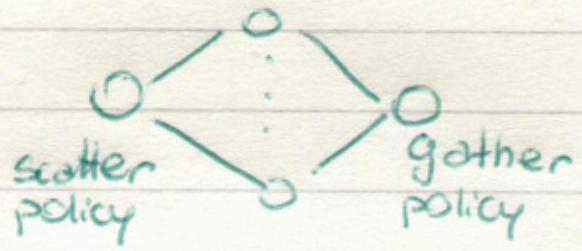
CHANNELS — COMPUTATION: N to N & 1 to 1
COMMUNICATION: 1 to N & N to 1
N to M (policies)

Notice that such input-output channels can be composed iff the varieties coincide and the size of input and output is correct.

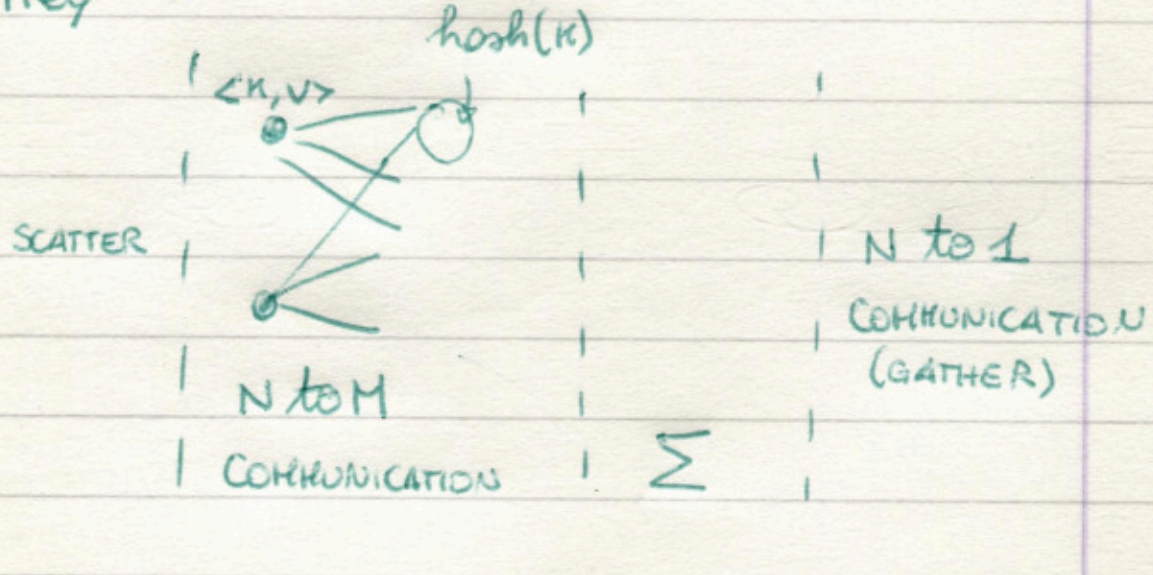
Example: a form



on the other hand, a map template is represented as



Google MAPREDUCE: sum all the values corresponding to the same key



CLASSES I MISSED

11th of March (introduction to OPENMP)

26th of March (data for boxed implementation)
[replayed at home]

Algorithmic skeleton: a higher order function which takes as an input another skeleton and portions of sequential code

We ded with STREAM PARALLEL SKELETONS.

Stream (def.) ordered collection of homogeneous data items.

Stream parallel rewriting rules

$\sigma \rightarrow \Diamond(\sigma)$
$\Diamond(\sigma) \rightarrow \sigma$
$((\sigma_1 | \sigma_2) | \sigma_3) \rightarrow (\sigma_1 | (\sigma_2 | \sigma_3))$
$(\sigma_1 | (\sigma_2 | \sigma_3)) \rightarrow ((\sigma_1 | \sigma_2) | \sigma_3)$
$(i_1 ; (i_2 ; i_3)) \rightarrow ((i_1 ; i_2) ; i_3)$
$((i_1 ; i_2) ; i_3) \rightarrow (i_1 ; (i_2 ; i_3))$
$i \rightarrow ; i$
$i ; \rightarrow i$
$i_1 | \dots | i_n \rightarrow i_1 ; \dots ; i_n$
$i_1 ; \dots ; i_n \rightarrow i_1 | \dots | i_n$

$\Diamond$ → Farm
$|$ → pipeline
$;$ → sequential

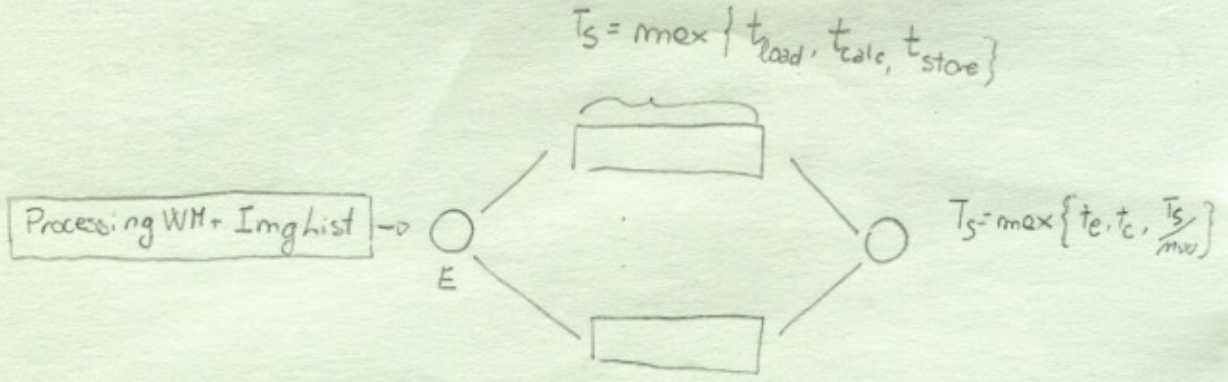
We want to obtain a NORMAL FORM for a skeleton that provides LOWER BOUNDS to the SERVICE TIME.

Service time (def.): time occurring between the delivery of two distinct, consecutive result data items.

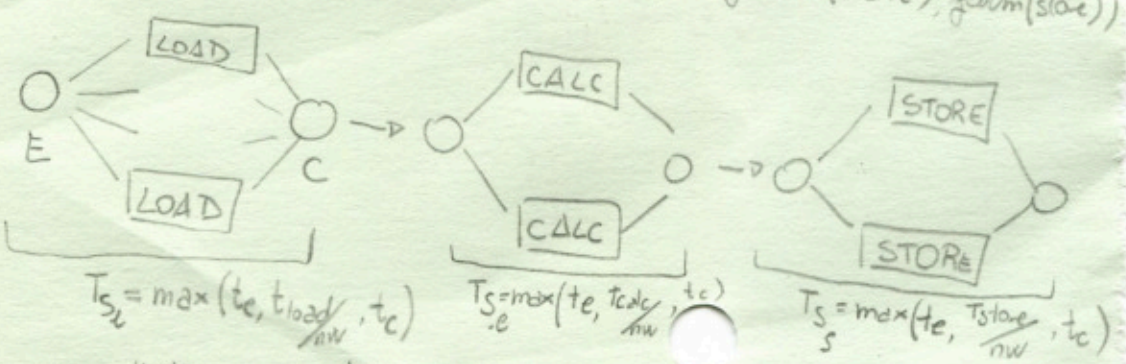
Fringe (def.): ordered list of all the sequential portions of the code in Δ. Inductively:

$$\begin{aligned}
&fringe(i) = i \\
&fringe(i_1; \ldots; i_n) = [i_1, \ldots, i_n] \\
&fringe(\sigma(\diamond)) = fringe(\sigma) \\
&fringe(\sigma_1 | \sigma_2) = append(fringe(\sigma_1), fringe(\sigma_2))
\end{aligned}$$

Normal form (def.): $\overline{\Delta} = \diamond(; (fringe(\Delta)))$



rplsh suggests pipe(farm(Load), pipe(farm(Calc), farm(store)))
14 25 19



overall $T_S = max\{T_{S_L}, T_{S_C}, T_{S_S}\}$

If we take a group of pages each (≈ n_w chunks per each). Some time is needed to divide the job and also to gather all the parts at the end.

$$T_{id} = \frac{\frac{1}{2} h \cdot m}{m_w}$$

$$T = T_{id} + \overbrace{m_w T_{give} + m_w T_{get}}^{\text{overhead}}$$

Parallelizing is a good idea if $T_{give}, T_{get} \ll T_{id}$

What happens if each of us needs different amount of time or one of us gets much text then the others whose pages are full of images? Somebody should decide the "translational weight" of each page and assign it equally.

In another scenario some pages may be left and given to the ones who finish first (those people will form a queue).

② Couting money in a room.

Everyone knows how much money he owns.

$T = \underbrace{(t_{ask} + t_{sum}) + \dots + (t_{ask} + t_{sum})}_{m_w \text{ times}}$

We could have a different approach:
we split in groups where the sum is calculated and then the sum is communicated to the leader $T = \#groups \cdot \{t_{ask} + \sqrt[m_w]{t_{sum}}\}$

The sums of group amounts may be done using a binary tree needing $\log_2(\#groups)$ sums.

The overhead in this situation is in the communication part

Problem: comparable quantity of money and laziness of counting of people.

This kind of operations don't need any particular ordering.

Difference from book case: no split needed, since the money is owned by workers in the first place + operations are commutative

19th of February '19

MAP-REDUCE

$f: \alpha \longrightarrow \beta$

<key, value>

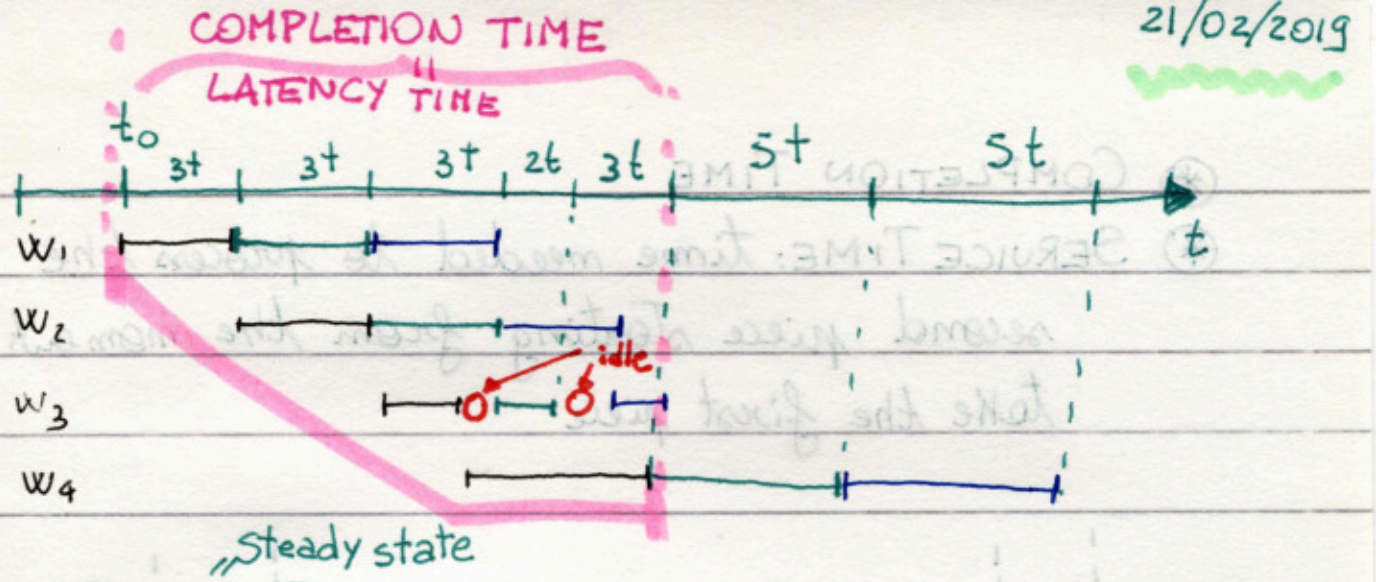
We use greek letters for types

$\oplus$: <key, value> ⟶ <key, value>

o-plus

where "value" is the result of summing all the values corresponding to the key "key".

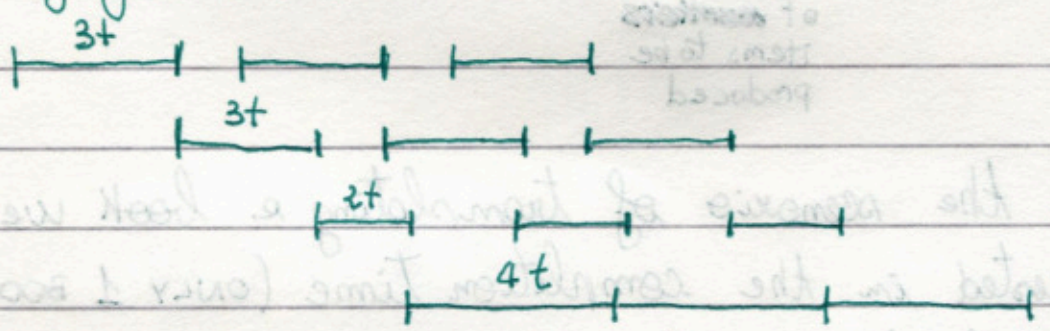
21/02/2019



"A regime" the time needed to have an output is the time needed by the last one

This scenario needs buffering to store the result of each step in the case of very fast tasks.

The solution to this issue is to rearrange the tasks forcing faster ones to wait.



$$\sum_{i=1} t_i + (m-1) \cdot \max_{i=1,\dots,4} \{t_i\}$$ \max\limits \sum\limits_ } } ^ {

We measure the time needed to process data from two different points of view: