Parallelism for Constraint Programming

Jean-Charles Régin
Université Nice-Sophia Antipolis,

jcregin@gmail.com

Plan

- Background (Parallelism, Concurrency, distributed computing …)
- Parallelism in CP
  - Distributed CSP
  - Parallel propagation
  - Portfolio method
  - Parallel Search
- Time is money: how to manage the solving time

Background

- General definition
- Concurrent programming
  - Race condition, lock, atomicity
- Distributed programming
- Data parallelism
- Pattern: MapReduce
- Some general rules

Parallelism

- Different kinds of parallelism
  - Shared memory (multi-cores)
  - Distributed
  - Cloud

- Depends on the kind of communication you can have
  - Memory
  - Ad-hoc
  - Message passing

Parallelism

- Use machines (resources) at the same time (in parallel) in order to improve the resolution of a problem
- Non parallel = sequential

- Machine, core, resource, multi sockets, lame …
  - Abstract entity = **worker**
  - Worker = core on multicore, on multisocket etc…

Parallelism

- Goal
  - gain a linear factor (#workers)
  - gain a super-linear factor

Parallelism: example

- Pb: count the number of houses on a maps
- //: split the problem into disjoint regions
  - Solve the pb for each region independantly
  - Sum up the results of each region

- We can expect a linear factor

Parallelism: example

- Pb: sort an array of numbers
- //: use a merge sort
- We split the array into k disjoint groups
  - 1) We sort each group in //
  - 2) We merge the groups: hierarchically 2 by 2

- We can expect a linear gain for Point 1)
- Point 2) in linear? More Difficult

Parallelism: example

- Search for a shortest path
- Not easy at all
- A Depth First Search?
- Very difficult to be linear
- // algorithms are quite different than sequential ones

Parallelism: example

- Search for the first white pixel in an image
- If you are lucky you can have a super linear gain!
- Extreme results: we can gain a lot or nothing!
- Increasing the number of workers is not a guarantee of improvements!

Parallelism performance

- Nqueens problem in C
  - Run in sequential
  - Run in // 4 times the same problem
  - Run in // 8
- Same experiments in Java

Benchmark: requires a lot of memory

#threads	I7 920		I7 2720 QM		FX 8150		I7 3820		I7 3930	
	ms	Perf/ thread	ms	Perf/ thread	ms	Perf/ thread	ms	Perf/ thread		
1	835		715		757		680		730	
2	1000	500	853	426	1100	550	750	374	790	395
4	1365	341	1300	324	1800	450	915	228	900	225
6	1900	316	1820	303	2600	433	1220	203	1100	184
8	2460	307	2375	296	3325	412	1530	191	1330	167
10	3240	324	3050	305	4120	412	1970	197	1630	163
12	3850	318	3600	300	5000	417	2250	187	1890	158
		ratio		**ratio**		**ratio**		**ratio**		**ratio**
		2,71		2,42		1,84		3,6		4,6

Benchmark: almost no memory

#threads	I7 920		I7 2720 QM		FX 8150		I7 3820		I7 3930	
		ratio		ratio		ratio		ratio		ratio
		4,0		3,7		5,7		4,0		5,5

Parallelism performance

- Be careful with
  - Hyper threading
  - Number of memory channels

Background

- General definition
- **Concurrent programming**
  - Race condition, lock, atomicity
- Distributed programming
- Data parallelism
- Pattern: MapReduce
- Some general rules

Concurrent programming

- Model with shared memory
- Important to understand some fundamentals concepts of parallelism
  - Lock, mutual exclusion, critical section…
- In CP, often used by a master

Concurrent programming

- The instructions between the two programs may be interleaved in any order

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A: Write back to variable V	3B: Write back to variable V

- **Race condition:** If instruction 1B is executed between 1A and 3A the program will produce incorrect data.
- Solution: use a **lock** to provide **mutual exclusion.**
- A lock allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked.
- The thread holding the lock is free to execute its **critical section** (the section of a program that requires exclusive access to some variable), and to unlock the data when it is finished.

Race condition

- A lock manage a mutual exclusion
  - Only one thread can access to the critical section

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A: Write back to variable V	3B: Write back to variable V

Thread A	Thread B
1A: Lock variable V	1B: Lock variable V
2A: Read variable V	2B: Read variable V
3A: Add 1 to variable V	3B: Add 1 to variable V
4A: Write back to variable V	4B: Write back to variable V
5A: Unlock variable V	5B: Unlock variable V

  - One thread will successfully lock variable V, while the other thread will be locked out (unable to proceed until V is unlocked again).

Lock

- Drawbacks of locks
  - Possibility of program deadlock
- Deadlock :
  - If two threads each need to lock the same two variables using non-atomic locks, it is possible that
    - one thread will lock one of them
    - the second thread will lock the second variable.
  - In such a case, neither thread can complete, and deadlock results.
- Solution lock of lock (atomic lock)

Lock

- Drawbacks of locks
  - May slow down the program
- Lock-free algorithm
  - Try to avoid locks, use atomic operation.

Atomicity

- An operation (or set of operations) is **atomic**, if it appears to the rest of the system to occur instantaneously.
- Guarantee of isolation from concurrent processes.

Atomic operations

- From Intel

$= x$	read the value of x
$x=$	write the value of x, and return it
x.fetch_and_store(y)	do $x=y$ and return the old value of x
x.fetch_and_add(y)	do $x+=y$ and return the old value of x
x.compare_and_swap(y,z)	if x equals z, then do $x=y$. In either case, return old value of x.

Atomic operation

- Attention,
  - x=12 is atomic
  - x=y is NOT atomic, because one read and one write!
- x.compare-and-swap (y,z)
  - if x equals z, then do x=y. In either case, return old value of x.
  - ```
    UpdateX() {
    do {
      oldx=globalx
      // Compute new value
      newx = ...expression involving oldx....
      // Store new value
      // if another thread has not changed globalx.
    }while(globalx.compare_and_swap(newx,oldx)!=oldx);
    ```
    ```

Synchronized Priority Queue

- Usually this is the only one synchronized data structure that is needed in parallel CP algorithms.

Concurrent programming

- Implementation: thread or process?

- In the past there was a difference
- Today, almost no difference on Linux
- Processes are slower on windows (but not a lot)

Background

- General definition
- Concurrent programming
  - Race condition, lock, atomicity
- **Distributed programming**
- Data parallelism
- Pattern: MapReduce
- Some general rules

Concurrent or distributed?

- Since concurrent programming is not easy, we can consider distributed programming
- We avoid shared memory
- However, some other problems appear…

Distributed/Cloud

- A **distributed system** is a model in which components located on networked computers communicate and coordinate their actions by passing messages.
- **Distributed computing** refers to the use of distributed systems to solve computational problems.
  - a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other by message passing.
- Main issues:
  - Load balancing (also for threads)
  - Communication time
  - Fault tolerance (not considered here)

Load balancing

- Very important
- **Load balancing** distributes workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units...
- Load balancing aims to optimize resource use
- The usage of resources (workers) is well balanced if the amount of work performs per each worker is globally equivalent
- Bad load balancing: **starving**. Some workers have no longer any work

Communication impact

- May be the bottleneck of the application
- Cost of message transmissions
- Number of communications

- That’s the case with SAT solvers which communicates no-goods

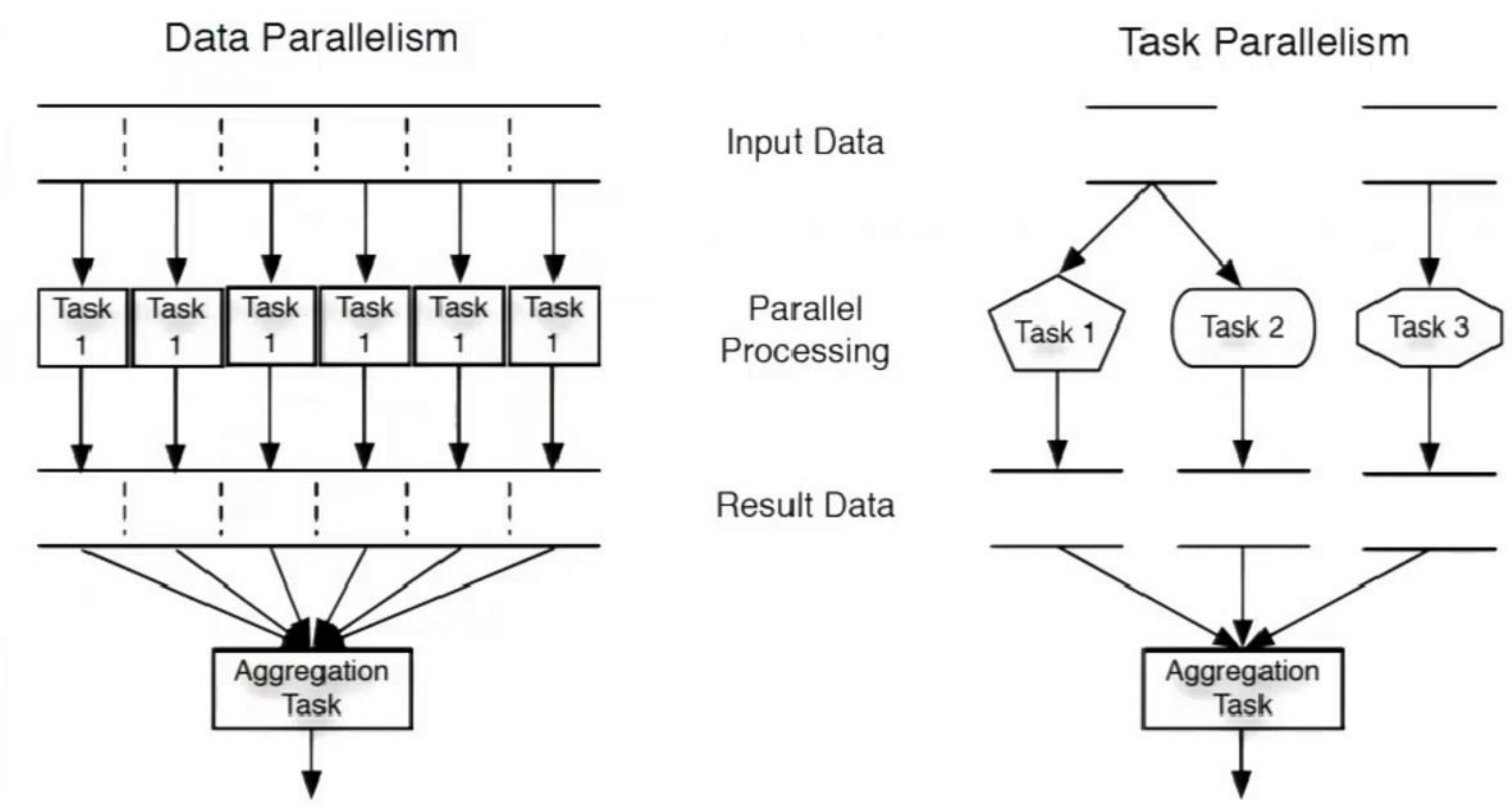
Background

- General definition
- Concurrent programming
  - Race condition, lock, atomicity
- Distributed programming
- **Data parallelism**
- Pattern: MapReduce
- Some general rules

Data parallelism

- Data parallelism focuses on distributing the data across different parallel computing nodes.
  - It contrasts to task parallelism as another form of parallelism.
- Data parallelism emphasizes the distributed (parallelized) nature of the data, as opposed to the processing (task parallelism).
- In a multiprocessor system executing a single set of instructions, data parallelism is achieved when each processor performs the same task on different pieces of distributed data
- Same task = exactly the same task. This means that the same branches are used when an "if" occurs. Otherwise, a factor 2 is lost

Data parallelism



Data parallelism

- Data parallelism via GPGPU
  - Data locality
  - Communication between CPU and GPU is slow (500 cycles)
  - Stream processing
- A *stream* is simply a set of records that require similar computation. Streams provide data parallelism.
- GPUs process elements independently so there is no way to have shared or static data.
  - For each element we can only read from the input, perform operations on it, and write to the output. Never a piece of memory that is both readable and writable.
- Ideal GPGPU applications have large data sets, high parallelism, and minimal dependency between data elements.

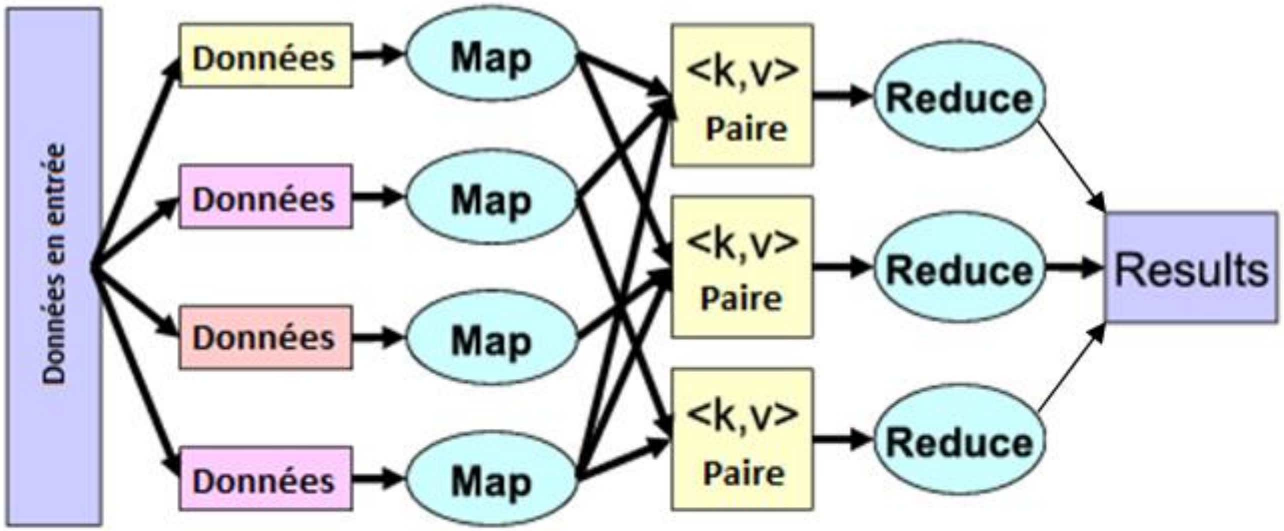
Background

- General definition
- Concurrent programming
  - Race condition, lock, atomicity
- Distributed programming
- Data parallelism
- **Pattern: MapReduce**
- Some general rules

Pattern: MapReduce

- A MapReduce program is composed of a **Map** method that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce** method that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).
- The "MapReduce System“ orchestrates the processing by distributing the data, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.
- Hadoop in Java
- The Map and Reduce functions are both defined with respect to data structured in (key, value) pairs.
- The MapReduce framework transforms a list of (key, value) pairs into a list of values.

Pattern: MapReduce



Map

- Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:
  - Map(k1,v1) → list(k2,v2)
- The Map function is applied in parallel to every pair (keyed by k1) in the input dataset.
  - This produces a list of pairs (keyed by k2) for each call.
  - The MapReduce framework collects all pairs with the same key (k2) from all lists and groups them together, creating one group for each key.

Reduce

- The Reduce function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:
  - Reduce(k2, list (v2)) $\rightarrow$ list(v3)
- Each Reduce call produces
  - either one value v3 or an empty return (one call is allowed to return more than one value).
  - The returns of all calls are collected as the desired result list.

Background

- General definition
- Concurrent programming
  - Race condition, lock, atomicity
- Distributed programming
- Data parallelism
- Pattern: MapReduce
- **Some general rules**

Some general rules

- Be careful with simulations of parallelism in sequential
  - Restart idea: kind of parallelism
  - O. Lhomme's idea:
    - A constraint is 10x slower to propagate than another one.
    - We run it only for 1/10th of the nodes.
    - That's a kind of parallelism

Some general rules

- The most important (IMHO)
- **You should try to avoid doing in parallel things that you would have not done in sequential**
- Not so easy: search for the first white pixel.
  - In parallel you may consider pixels that are not consider in sequential

Some general rules

- Determinism
  - We should be able to obtain the same results of a program if we rerun it on the same machine (same conditions)
    - The same solution must be obtained (exactly the same with the same cost)
  - We should be able to obtain the same results of a program if we rerun it by using a different system
    - The increases of the number of workers should not changed the solution
- Very difficult in general

Exercices

- Documentation:
- https://docs.python.org/2/library/multiprocessing.html
- https://docs.python.org/2/library/multiprocessing.html#shared-ctypes-objects

Exercices

- Shared data val: one thread adds one to val, another multiplies val by 2. Each operation is repeated 200 times. Can you predict the final result?
- Bank account: 3 threads want to access to a back account in order to withdraw some cash (10 units). Money can be withdrawn only if there is enough money. A fourth thread adds money to the bank account (by increment of 9 units). Describe the behavior of the system:
  - Without lock
  - With a lock on the value
  - With a lock on the test of money on the account
- Count the number of prime numbers int the range [1,1M]. To check whether a number x is prime or not, we try to divide x by the number from 2 to sqrt(x).

Plan

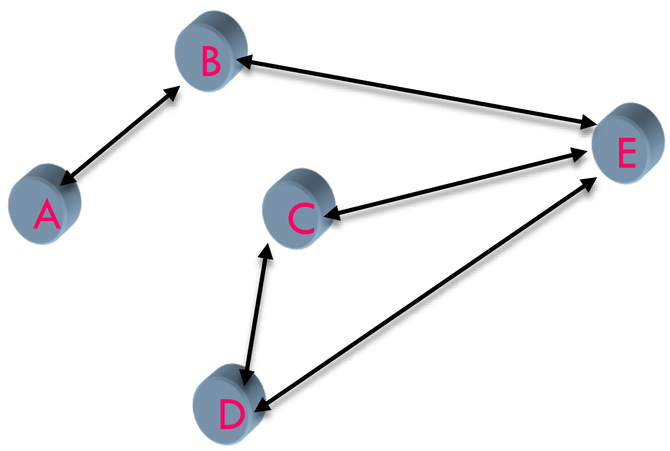
- Background (Parallelism, Concurrency, distributed computing …)
- **Parallelism in CP**
  - Distributed CSP
  - Parallel propagation
  - Portfolio method
  - Parallel Search
- Time is money: how to manage the solving time

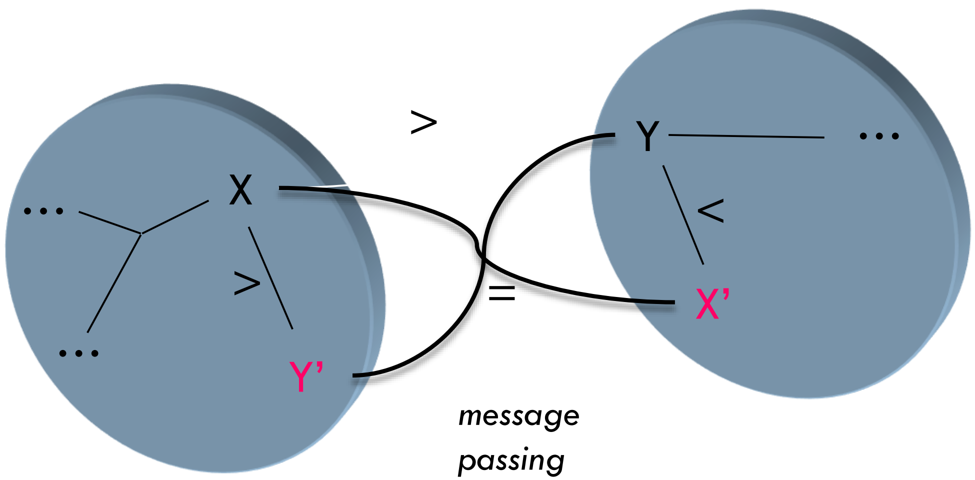
Distributed CSP

- A dedicated method for solving CSPs on a distributed network.
- Ad-hoc method
  - For filtering algorithm
  - For tree search traversal
- Difficult to integrate with an existing solver

The Distributed Constraint Satisfaction Problem

- A system of agents collaborating through peer-to-peer interactions to solve a Constraint Satisfaction Problem
- P = (X, D, C, A),





Problems of Distributed Tree based Search

- Problem 1: Idleness
  - Consequence of, **load imbalance**

Problem class		ABT	IDIBT
easy randoms		87%	92%
hard randoms		92%	96%
n-queens		91%	94%
hard quasi-groups		87%	93%
	Idleness time (average)		
	10-100 runs		

Problems of Distributed Tree based Search

- Problem 2: Randomization risk
  - Consequence of, **message interleaving**
  - R-Risk (Def) the standard-dev of a deterministic distributed tree-search algorithm applied multiple time to one instance.

	min time	max time	R-risk
ABT	297ms	5374ms	807
IDIBT	1640ms	1984ms	96
	10 queens, lex orderings		
	100 runs		

Problems of Distributed Tree based Search

- Problem 3: Selection risk
  - Consequence of, **wrong heuristic**
  - Heuristic (Def) partial order of the agents + value (local solution) ordering
  - S-Risk (Def) of a set H of heuristics is the standard-dev of the performance of each $h \in H$ applied the same number of time to one instance.

Distributed CSP

- There are some solutions to these problems
- Have look at the work of Youssef Hamadi, Christian Bessiere etc…

Knowledge aggregation

- Performance of maxSupport
  - Hard randoms, hard quasigroups
  - IDIBT, 100 instances, median
  - 10 partial orders: (Max-degree, domdeg,mindom,lex)

Heuristics	hard randoms	hard quasigroups
	message	message
minUsed	367	102000
minBT	392	104000
maxUsed	379	40000
maxBT	433	43000
maxSupport	57	1900
none	409	73000

Idleness

Problem class	ABT	IDIBT	ABT-10	IDIBT-10
easy randoms	87%	92%	56%	47%
hard randoms	92%	96%	39%	59%
n-queens	91%	94%	48%	52%
hard quasi-groups	87%	93%	28%	59%
Idleness time (average)				
10-100 runs				

Plan

- Background (Parallelism, Concurrency, distributed computing …)
- **Parallelism in CP**
  - Distributed CSP
  - **Parallel propagation**
  - Portfolio method
  - Parallel Search
- Time is money: how to manage the solving time

Parallel Propagation

- Distributed CSPs use a worker per variable
- Parallel Propagation uses a worker per constraint

- Not really studied
  - Cannot allocate one worker per constraint because of idleness (bad load balancing)
  - Difficult to use more workers than constraints
  - Strong synchronization issues

Parallel filtering algorithms

- GAC4R in // with openmp
- Propagation per variable
- Directive #pragma omp parallel for before loops
- Performance between 350 and 425 ms on my laptop (instead of 715 ms)
- Performance on a 3930K : 175 ms instead of 730 ms
- Problem: we cannot parallelize again ☹

Plan

- Background (Parallelism, Concurrency, distributed computing …)
- **Parallelism in CP**
  - Distributed CSP
  - Parallel propagation
  - **Portfolio method**
  - Parallel Search
- Time is money: how to manage the solving time

Portfolio Method

- Idea: run in parallel different methods
- Method is general:
  - Variable-value strategy,
  - Model
  - Solvers…
- Exploit the facts that there are
  - Great disparities between methods
  - No dominant method
  - Very difficult to determine a priori the best method

Portfolio Method

- Efficient for SAT solvers
- In CP: CPHydra (E. Hebrard) is such a solver
- May lead to super linear gain!

- Consider M1, M2, M3, M4
- Running in // all the methods leads to a wall clock time of t=min(time(M1),time(M2),time(M3),time(M4))
- Best possible time t/4
- Worst possible time max(time(M1),time(M2),time(M3),time(M4)) /4
- We lose a factor but we have a guarantee!

Portfolio method

- A good idea.
- Sometimes difficult to accept intellectually.
- Never forget it and always try to compete with it.

Plan

- Background (Parallelism, Concurrency, distributed computing …)
- **Parallelism in CP**
  - Distributed CSP
  - Parallel propagation
  - Portfolio method
  - **Parallel Search**
- Time is money: how to manage the solving time

Parallel search for solutions

- We have k workers (CPU, cores, …)
- How can we use the k workers in order to speed up the search for solutions ?

- Hypothesis:
  - If we split a problem into sub-pb then the sum of resolution times of subproblems is equal to the resolution time of the initial problem.
    - In CP, it seems to be right, but not in MIP
    - Be careful with some learning strategies

Static Decomposition

- We have k workers,
  - We split the problem into k subproblems:
    - (x={1,2}) (x={3,4}),(x={5,6}),…
  - We give one subproblem to each worker
- Pros
  - Very simple
  - Not intrusive
- Cons
  - Total time = the time of the longest subproblem
    - Pb with the homogeneity of decomposition

Static decomposition

- Sometimes it works well
  - Nqueens problem
- Often the results are not good and this does not scale up.

Work stealing

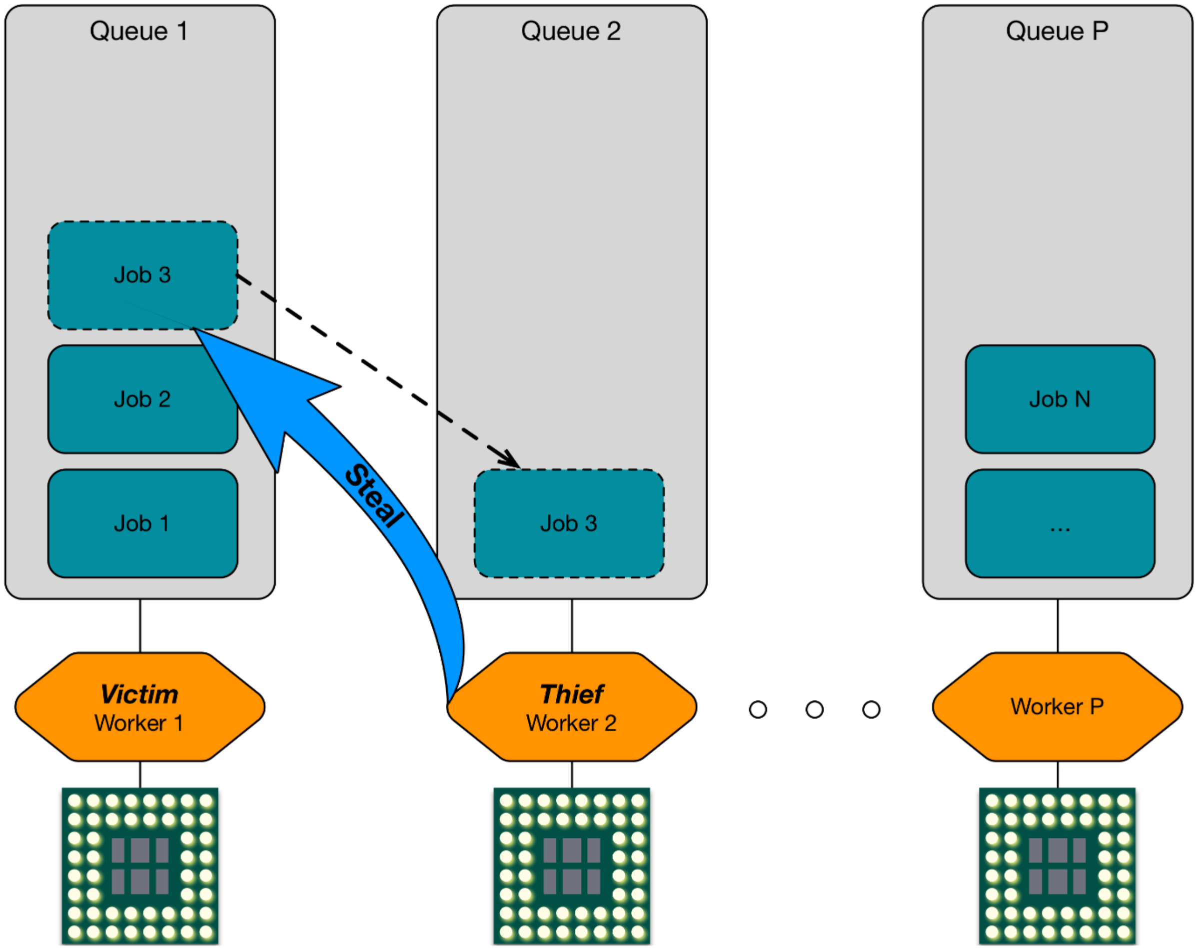
- **work stealing** is a scheduling strategy for multithreaded computer programs.
  - solves the problem of executing a *dynamically multithreaded* computation, one that can "spawn" new threads of execution, on a *statically multithreaded* computer, with a fixed number of workers.

Work stealing

- We have k workers,
  - We split the problem into k subproblems,
  - We give a subproblem for each worker
  - When a worker finishes its work, it asks another worker which works. This latter gives it a part of its remaining work.

- How to select a victim for stealing?
- How much to steal?

Work stealing



Work stealing

- **Pros**
  - Better repartition of the work (dynamic)
- **Cons**
  - Very intrusive in the solver
    (avoided by the work of B. Le Cun, Bob++)
  - Easy tasks should be avoided
  - At the end, almost all the workers ask for some work all the time.
    We need to manage that.

Work stealing

- Ajouter des slides
- Et des dessins (chercher sur le web)

Work stealing implementation

- How do we interrupt a worker to ask for some work?
- Use a timer
  - Not a good solution, but time is not a constant notion in parallelism: not reproductable (we can be interrupted or stoped)
- Use a counter of instructions
  - Better solution because we can expect a constant notion (not related to time) that is reproductable

Embarrassingly Parallel Search

- Static decomposition is simple, but it is difficult to split into equal parts

- Solution :
  - We split into more subproblems than workers
- We hope that the sum of resolution times will be well balanced.

- **The greatest importance is not to split into equal parts, but it is to equilibrate the sum of the resolution times of subproblems for each worker**

EPS

- Main idea:
  - This is not the subproblems that have to be well balanced but the overall solving time of each worker

- Assumption: solving independently 2 disjoint subparts of a problem must not be longer than solving the whole problem
  - Not true with MIP Solver
  - Possible to deal with this point

Embarrassingly Parallel Search

- 2 steps
  - Decomposition
  - Resolution
- **Decomposition**
  - We divide the problem into q subproblems to get a partition of the initial problem (static decomposition)
  - We put these subproblems into a queue
    - This process is **static**
- **Resolution**
  - When a worker needs some work, it takes a supbroblem from the queue. (dynamic choice)
    - This process is **dynamic**

Massive Decomposition

- Decomposition
  - One task requires 140s
  - We divide it into 4 tasks requiring 20,80,20,20s : not well balanced : 80s (max), 20s (min)
  - We split again into 4 parts: (5+5+5+5)+(20+10+10+40)+(2+5+10+3)+(2+2+8+8)
  - w1: 5+20+2+8=35; w2: 5+10+2+10=27
    w3: 5+10+5+3+2+8=33; w3=5+40=45 gives : 45s (max) et 27 (min)

Massive Decomposition

- We have more chance
  - to equilibrate the sum of workload for each worker
  - to break large subproblems
    and to reduce their relative importance
  - The relative importance of maximum (40 vs 80) is reduced.
    The inactivity time (max-min) also (80-20=60) vs (45-27=18)

Embarrassingly Parallel Search

- How do we decompose ?
  - We want to split into q subproblems
- Solution 1
  - We take p variables for which the cartesian product of their domains is close to q.
    (we adjust the last domain if needed)
- Results
  - Work very well with some problems
  - Work badly with others, because a lot of generated problems are trivially inconsistent

Decomposition

- Solution 1:
  - We take p variables with which the cartesian product of their domains is close to q.

- If x,y and z are implied in alldiff constraint then
  - the cartesian product is a bad idea. (a,b,a) must not be considered, the same thing for (a,a,b)
  - they are not considered with a sequential resolution

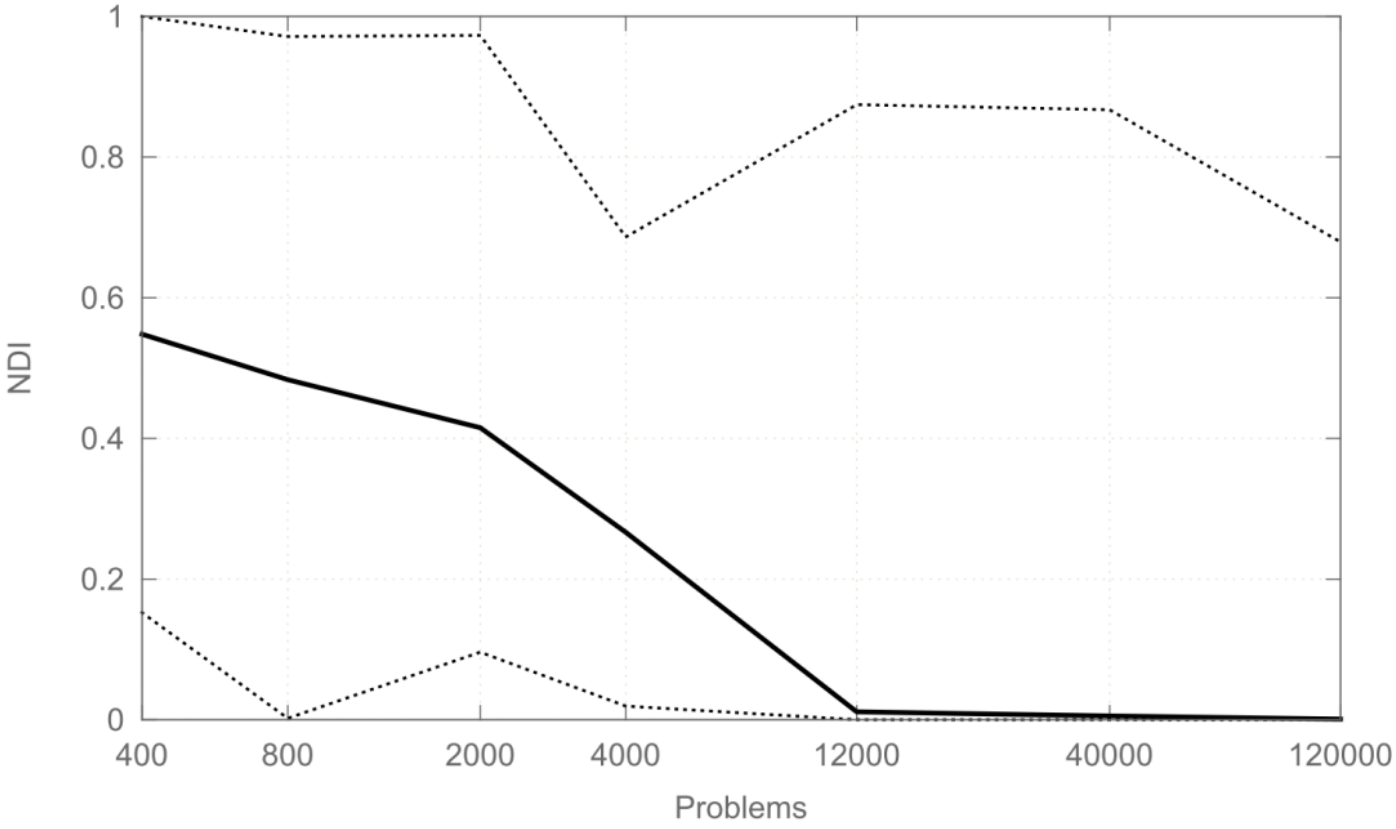
Decomposition

- Solution 1:
  - We take p variables with which the cartesian product of their domains is close to q.

- **We should avoid considering in a parallel resolution , problems that would have not been considered in a sequential resolution**

- This solution generates too many problems non consistent with the propagation (alldiff case)

Decomposition



Decomposition

- How do we decompose ?
  - We want to split into q subproblems
- Solution 2
  - We take p variables in order to have a cartesian product close to q.
  - We generate all combinations step by step with eliminating problems non consistent with the propagation
  - If we do not generate the desired number of subproblems then we restart the process with more variables

Decomposition

- To generate all combinations, we simulate a BFS with Bounded DFS
  (fixed number of choiced variables)

- We introduce a table constraint containing combinations for each level to avoid repeating the bad branches between two DFS.

Resolution for Satisfaction Problems

- We dynamically take a subproblem in the queue
- The ordering seems not really be important
  - We can use the one of the decomposition

Resolution for Optimization Problems

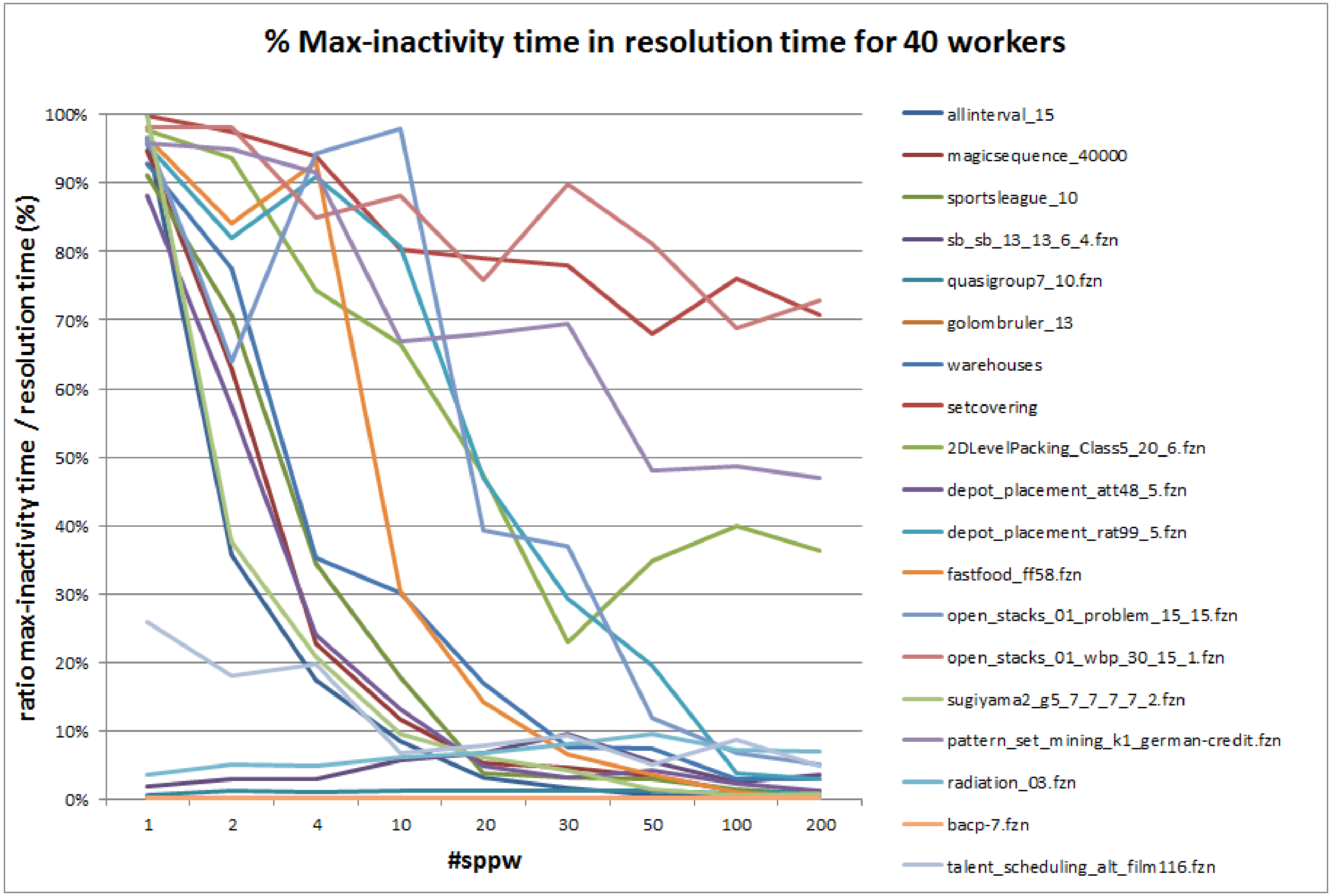
- The subproblems queue keeps the best current value of the objective
- **A current resolution is never interrupted**
  - Neither to communicate a better solution
  - Or to receive a new value of the objective
- However, when a worker finishes to solve a subproblem, the value of the objective can be used as a better bound

How many subproblems ?

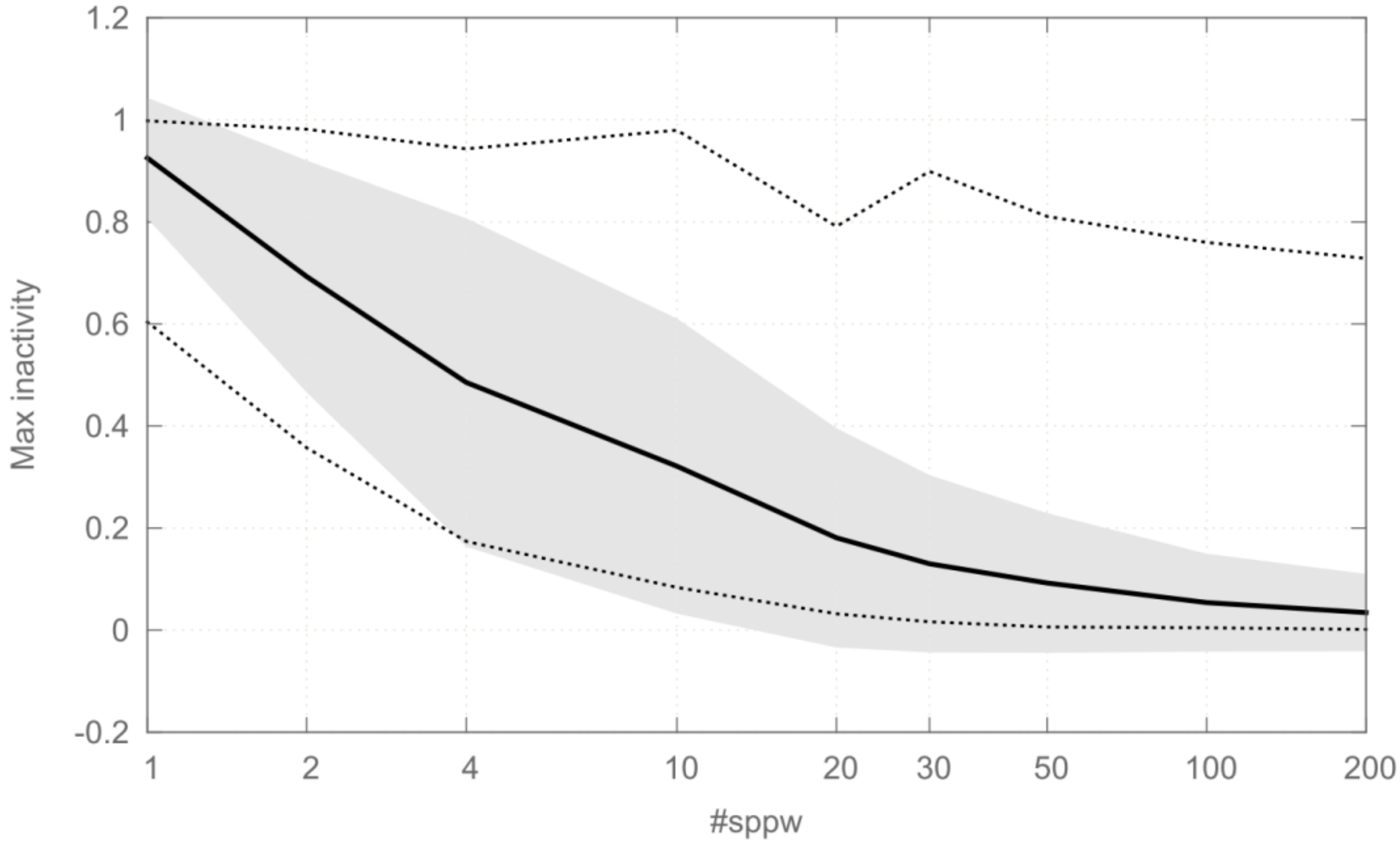
- Tricky question
- As we want to equilibrate the workload of workers,
  we propose to define a number of sub-pb per worker (#sppw)
- According to our experimentations :
  - It appears to be decorrelated the problems !
  - If the value is too small, it is difficult to equilibrate.
  - It the value is too high then the decomposition takes a lot of time

- A value between 30 and 100 subproblems per worker is good.
  The best result is obtained with 30 sub-pb per worker
- The results always include the decomposition time
  (except for precision)
- Means are geometric

Inactivity time



Inactivity time



Results on 40 cores

Instance	Seq.	Work stealing		EPS	
	t	t	s	t	s
allinterval_15	262.5	9.7	27.0	8.8	**29.9**
magicsequence_40000	328.2	592.6	0.6	37.3	**8.8**
sportsleague_10	172.4	7.6	22.5	6.8	**25.4**
sb_sb_13_13_6_4	135.7	9.2	14.7	7.8	**17.5**
quasigroup7_10	292.6	14.5	20.1	10.5	**27.8**
non_non_fast_6	602.2	271.3	2.2	56.8	**10.6**
golombruler_13	1355.2	54.9	24.7	44.3	**30.6**
warehouses	148.0	25.9	5.7	21.1	**7.0**
setcovering	94.4	16.1	5.9	11.1	**8.5**
2DLevelPacking_Class5_20_6	22.6	13.8	1.6	0.7	**30.2**
depot_placement_att48_5	125.2	19.1	6.6	10.2	**12.3**
depot_placement_rat99_5	21.6	6.4	3.4	2.6	**8.3**
fastfood_ff58	23.1	4.5	5.1	3.8	**6.0**
open_stacks_01_problem_15_15	102.8	6.1	16.9	5.8	**17.8**
open_stacks_01_wbp_30_15_1	185.7	15.4	12.1	11.2	**16.6**
sugiyama2_g5_7_7_7_7_2	286.5	22.8	12.6	10.8	**26.6**
pattern_set_mining_k1_german-credit	113.7	22.3	5.1	13.8	**8.3**
radiation_03	129.1	33.5	3.9	25.6	**5.0**
bacp-7	227.2	15.6	14.5	9.5	**23.9**
talent_scheduling_alt_film116	254.3	13.5	**18.8**	35.6	7.1
total (t) or geometric mean (s)	488.2	1174.8	7.7	334.2	**13.8**

Scaling

- Instead of 40 cores we want to deal with 1,000

- Let's go!

Results (1)

Comparison of different decompositions (512 workers)

Instance	Seq. t_0	su_{res}	Dec_{seq} t_{dec}	Dec_{seq} su	$Dec_{//1}$ t_{dec}	$Dec_{//1}$ su	$Dec_{//2}$ t_{dec}	$Dec_{//2}$ su
	s	r	s	r	s	r	s	r
market_split_s5-02.fzn	3314.4	459.5	3.6	305.5	1.3	388.7	1.0	405.9
market_split_u5-09.fzn	3266.6	455.0	3.0	321.2	1.1	394.7	0.8	411.8
market_split_s5-06.fzn	3183.9	436.0	4.4	272.0	2.2	334.8	1.0	384.0
prop_stress_0600.fzn	2729.2	213.9	54.4	40.7	21.3	80.0	7.5	193.1
nmseq_400.fzn	2505.8	429.7	33.7	63.3	14.9	120.9	4.6	240.4
prop_stress_0500.fzn	1350.6	265.2	22.7	48.6	9.3	93.7	3.3	161.6
fillomino_18.fzn	763.9	301.9	19.8	34.2	6.4	85.7	2.5	150.7
steiner-triples_09.fzn	604.9	443.8	3.3	130.8	1.8	191.5	0.5	332.0
nmseq_300.fzn	555.3	309.0	18.7	27.1	7.9	57.1	2.4	131.7
golombruler_13	1303.9	492.0	9.4	92.7	1.4	322.9	0.4	427.9
cc_base_mzn_rnd_test.11.fzn	3279.5	196.5	83.8	32.6	35.5	62.8	10.1	122.6
ghoulomb_3-7-20.fzn	2993.8	279.2	112.6	24.3	50.0	49.3	12.1	131.1
pattern_set_mining_k1_yeast.fzn	2871.3	285.5	51.3	46.8	21.0	92.4	5.6	183.2
still_life_free_8x8.fzn	2808.9	331.0	82.0	31.1	33.2	67.4	8.3	166.9
bacp-6.fzn	2763.3	473.1	13.4	143.5	5.4	245.0	1.5	378.9
depot_placement_st70_6.fzn	2665.1	345.6	29.9	70.9	12.5	131.8	3.6	235.1
open_stacks_01_wbp_20_20_1.fzn	1523.2	280.7	35.4	37.3	15.6	72.3	4.0	160.8
bacp-27.fzn	1499.7	445.3	11.0	104.5	4.4	193.8	1.2	326.5
still_life_still_life_9.fzn	1145.1	347.9	25.2	40.1	9.4	90.4	3.0	182.9
talent_scheduling_alt_film117.fzn	566.1	386.4	15.0	34.4	6.0	75.8	1.8	175.8
total(s) and geom. average(r)	**41894.5**	**347.1**	**632.6**	**66.4**	**260.8**	**124.8**	**75.0**	**223.9**

Scaling

- Results
  - Degradation of the results from hundred cores

- **Why?**
  - We still observe a nice speedup for subproblems solving
  - The decomposition becomes slow and it slows down the overall solving time

Decomposition

- For 40 cores we need to find 40*30=1,200 subproblems
- For 500 cores we need to find 500*30=15,000

- When the number of workers increases,
  - the decomposition has more work to do!
  - The ratio // vs sequential augments
- A sequential decomposition can no longer be used!

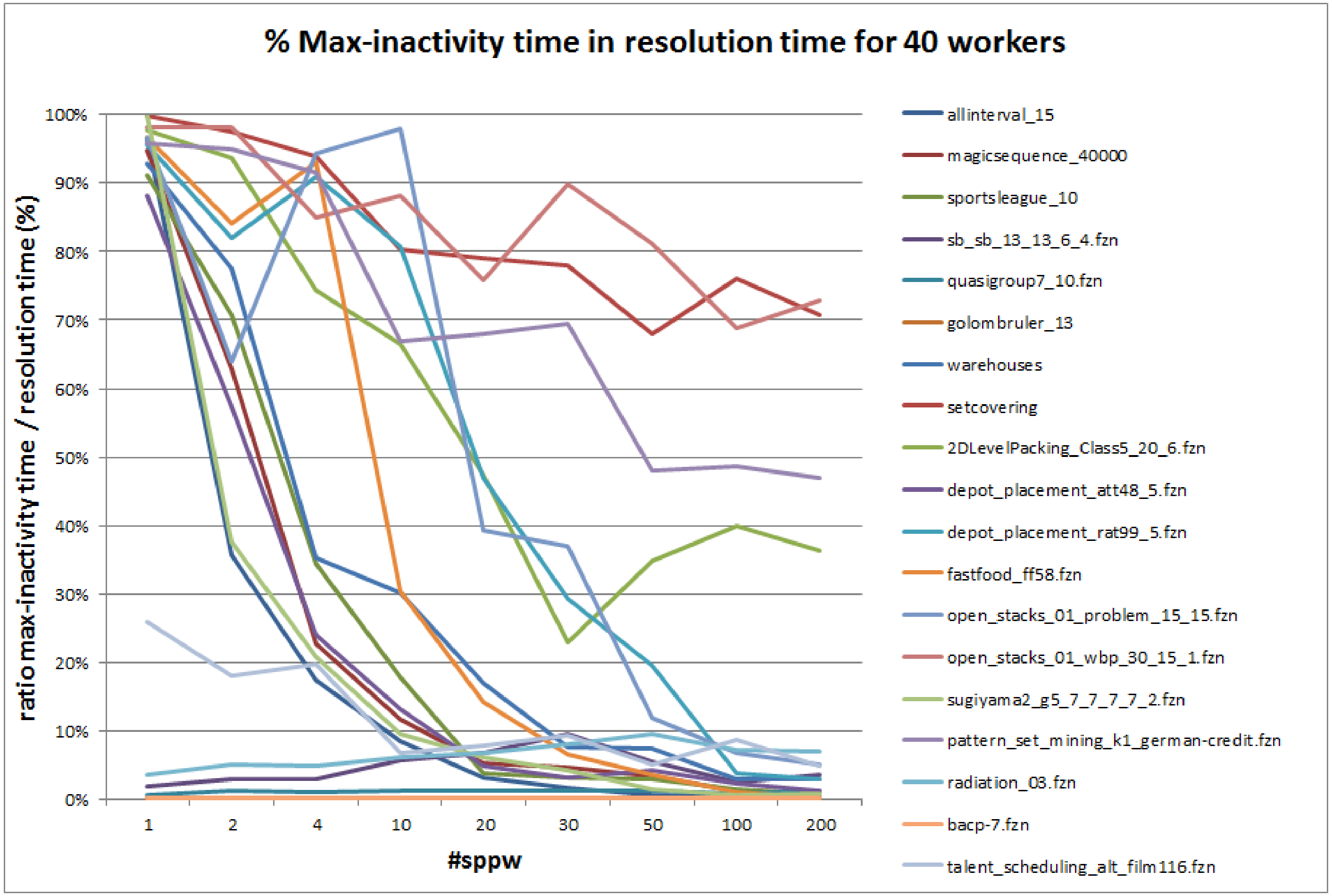
Parallel decomposition

- The decomposition is made by the workers
- Question: what are the problems that you give to workers (this is our goal!)
- A first naive parallelization:
  - We have k workers and we want to split into q subproblems
  - First, we split into k subproblems
  - Each worker split into q / w in order to reach q subproblems
- Results: Bad load balancing ☹

Parallel decomposition

- Idea: proceed step by step
  - Introduce stops (synchronization) for redistributing the work.
- Decomposition into 30 subproblems
  - Use of intermediate steps
  - Search for generating 5 sspb/worker, then 10, then 20 then 30.
- The more we synchronize, the more we have well balanced workload
- The more we synchronize, the more we slow down the process
- Consequence:
  - We synchronize but not too much
  - We prefer synchronization close to the top

Inactivity time



Intermediate phases

- After tests, we found that it is better to achieve the following phases:
  - Split the problem into k subproblems in sequential
  - Look for 1 sspb / worker (First Stop) in //
  - Look for 5 sspb / worker (Second Stop) in //
  - Look for 30 sspb / worker (Third Stop) in //

- Note:
  - No need of synchronisation for the last phase
  - It is rather robust and does not depend of the type of the problem
  - The parallel decomposition generates the same subproblems as the sequential decomposition does

Experimental Protocol

- 20 selected instances (take a long time for solving)
  - min 1500 seconds in sequential
- Data center (1152 cores)
- Reservation of 512 cores (difficult to have)
- Implementation with Gecode

Results (1)

Comparison of different decompositions (512 workers)

Instance	Seq. t_0	su_{res}	Dec_{seq} t_{dec}	Dec_{seq} su	$Dec_{//1}$ t_{dec}	$Dec_{//1}$ su	$Dec_{//2}$ t_{dec}	$Dec_{//2}$ su
	s	r	s	r	s	r	s	r
market_split_s5-02.fzn	3314.4	459.5	3.6	305.5	1.3	388.7	1.0	405.9
market_split_u5-09.fzn	3266.6	455.0	3.0	321.2	1.1	394.7	0.8	411.8
market_split_s5-06.fzn	3183.9	436.0	4.4	272.0	2.2	334.8	1.0	384.0
prop_stress_0600.fzn	2729.2	213.9	54.4	40.7	21.3	80.0	7.5	193.1
nmseq_400.fzn	2505.8	429.7	33.7	63.3	14.9	120.9	4.6	240.4
prop_stress_0500.fzn	1350.6	265.2	22.7	48.6	9.3	93.7	3.3	161.6
fillomino_18.fzn	763.9	301.9	19.8	34.2	6.4	85.7	2.5	150.7
steiner-triples_09.fzn	604.9	443.8	3.3	130.8	1.8	191.5	0.5	332.0
nmseq_300.fzn	555.3	309.0	18.7	27.1	7.9	57.1	2.4	131.7
golombruler_13	1303.9	492.0	9.4	92.7	1.4	322.9	0.4	427.9
cc_base_mzn_rnd_test.11.fzn	3279.5	196.5	83.8	32.6	35.5	62.8	10.1	122.6
ghoulomb_3-7-20.fzn	2993.8	279.2	112.6	24.3	50.0	49.3	12.1	131.1
pattern_set_mining_k1_yeast.fzn	2871.3	285.5	51.3	46.8	21.0	92.4	5.6	183.2
still_life_free_8x8.fzn	2808.9	331.0	82.0	31.1	33.2	67.4	8.3	166.9
bacp-6.fzn	2763.3	473.1	13.4	143.5	5.4	245.0	1.5	378.9
depot_placement_st70_6.fzn	2665.1	345.6	29.9	70.9	12.5	131.8	3.6	235.1
open_stacks_01_wbp_20_20_1.fzn	1523.2	280.7	35.4	37.3	15.6	72.3	4.0	160.8
bacp-27.fzn	1499.7	445.3	11.0	104.5	4.4	193.8	1.2	326.5
still_life_still_life_9.fzn	1145.1	347.9	25.2	40.1	9.4	90.4	3.0	182.9
talent_scheduling_alt_film117.fzn	566.1	386.4	15.0	34.4	6.0	75.8	1.8	175.8
total(s) and geom. average(r)	**41694.5**	**347.1**	**632.6**	**66.4**	**260.8**	**124.8**	**75.0**	**223.9**

Results (2)

2nd stop in the parallel decomposition (512 workers)

Instance	2ème étape				
	3	4	5	6	7
prop_stress_0600.fzn	10.5	8.6	7.5	9.1	13.0
cc_base_mzn_rnd_test.11.fzn	21.5	12.1	10.1	14.5	17.8
ghoulomb_3-7-20.fzn	16.4	13.3	12.1	16.5	18.1
pattern_set_mining_k1_yeast.fzn	8.5	6.9	5.6	9.2	13.4
still_life_free_8x8.fzn	11.5	8.1	8.3	12.7	14.6
total decomposition time(s)	68.4	49.0	43.6	62.0	76.9

Results (3)

Work stealing vs EPS (512 workers)

Instance	Seq.	Work stealing		EPS	
	time(s)	*time(s)*	*ratio*	*time(s)*	*ratio*
market_split_s5-02	3314.4	-	-	8.2	405.9
market_split_u5-09	3266.6	-	-	7.9	411.8
market_split_s5-06	3183.9	-	-	8.3	384.0
prop_stress_0600	2729.2	1426.4	1.9	14.1	193.1
nmseq_400	2505.8	-	-	10.4	240.4
prop_stress_0500	1350.6	670.0	2.0	8.4	161.6
fillomino_18	763.9	-	-	5.1	150.7
steiner-triples_09	604.9	79.0	7.7	1.8	332.0
nmseq_300	555.3	-	-	4.2	131.7
golombruler_13	1303.9	15.5	83.9	3.0	427.9
cc_base_mzn_rnd_test.11	3279.5	-	-	26.8	122.6
ghoulomb_3-7-20	2993.8	575.4	5.2	22.8	131.1
pattern_set_mining_k1_yeast	2871.3	299.8	9.6	15.7	183.2
still_life_free_8x8	2808.9	1672.8	1.7	16.8	166.9
bacp-6	2763.3	330.1	8.4	7.3	378.9
depot_placement_st70_6	2665.1	1902.9	1.4	11.3	235.1
open_stacks_01_wbp_20_20_1	1523.2	153.9	9.9	9.5	160.8
bacp-27	1499.7	579.6	2.6	4.6	326.5
still_life_still_life_9	1145.1	140.1	8.2	6.3	182.9
talent_scheduling_alt_film117	566.1	95.5	5.9	3.2	175.8
total (time) or geometric average (ratio)	[illegible]	[illegible]	5.4	195.7	223.9

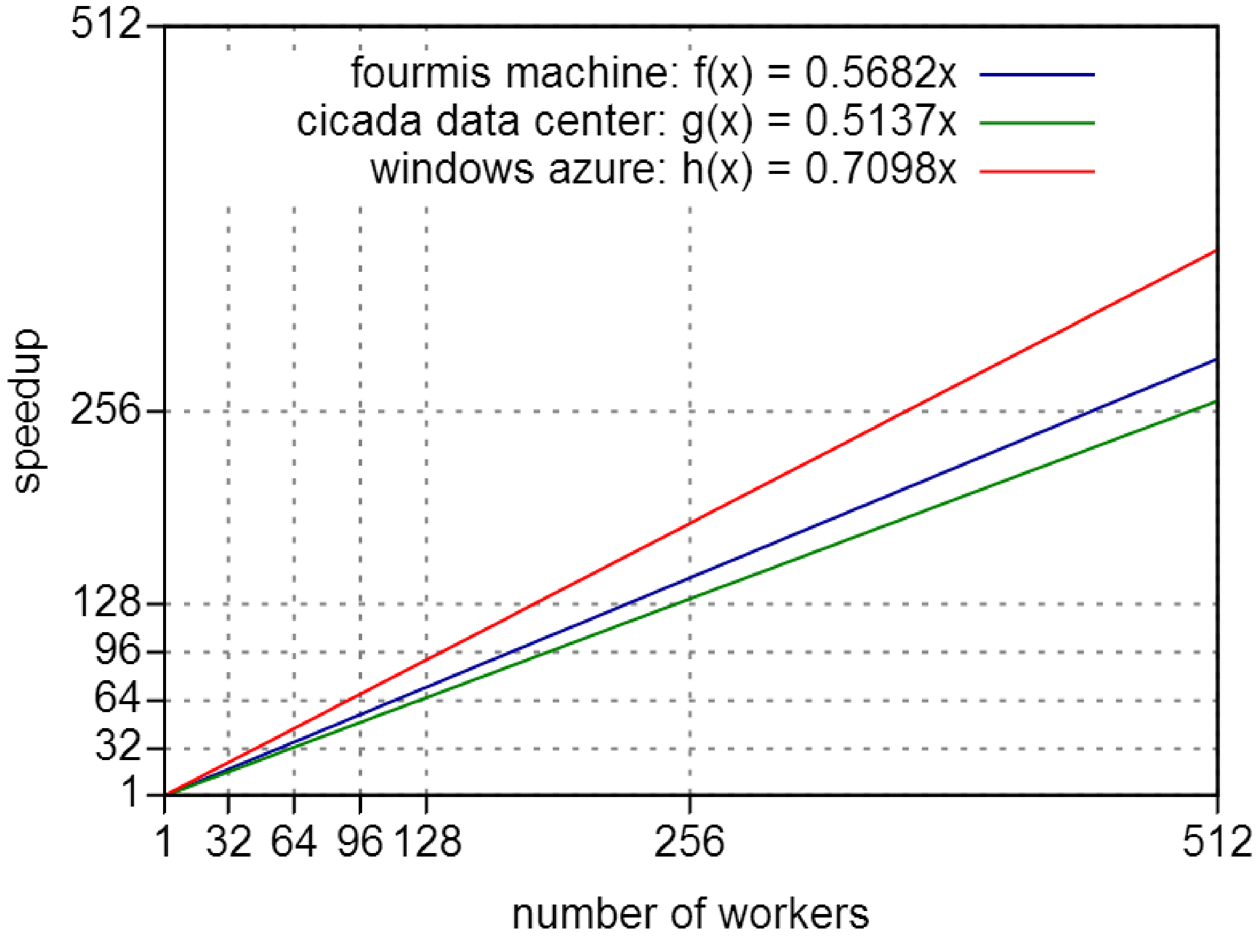
Azure cloud

		Gecode 4.0.0			
		work stealing		EPS	
		24 workers			
instance	seq. time (s)	// time (s)	speedup	// time (s)	speedup
bacp-27	4256,8	548,4	7,8	260,2	16,4
depot_placement_att48_5	298,5	21,3	14,0	17,6	16,9
depot_placement_rat99_5	52,5	10,1	5,2	8,5	6,2
depot_placement_st70_6	7929,0	1172,5	6,8	433,9	18,3
fastfood_ff58	63,1	11,3	5,6	3,2	20,0
fillomino_18	2227,1	184,6	12,1	160,2	13,9
golombruler_13	3167,3	210,4	15,1	154,0	20,6
market_split_s5-02	11367,4	658,6	17,3	467,1	24,3
market_split_s5-06	11039,8	650,7	17,0	452,7	24,4
market_split_u5-09	11421,6	609,2	18,7	468,1	24,4
open_stacks_01_problem_15_15	284,5	38,4	7,4	26,4	10,8
open_stacks_01_wbp_20_20_1	5338,7	374,1	14,3	302,7	17,6
open_stacks_01_wbp_30_15_1	521,0	72,3	7,2	30,0	17,4
pattern_set_mining_k1_german-credit	247,8	31,4	7,9	29,7	8,3
pattern_set_mining_k1_yeast	7938,7	482,7	16,4	172,1	46,1
quasigroup7_10	683,5	67,1	10,2	31,8	21,5
radiation_03	274,4	46,2	5,9	30,6	9,0
sb_sb_13_13_6_4	257,7	23,3	11,0	18,4	14,0
still_life_still_life_9	3187,4	196,8	16,2	189,0	16,9
sugiyama2_g5_7_7_7_7_2	602,6	47,1	12,8	32,2	18,7
talent_scheduling_alt_film117	1677,8	110,5	15,2	22,7	74,0
			10,7		17,9

Cloud comparison

Instance	10 workers			20 workers		
	fourmis	cicada	azure	fourmis	cicada	azure
fillomino_18	5.4	5.8	5.8	9.1	8.8	12.0
market_split_s5-02	10.0	10.1	10.2	18.6	18.3	20.3
market_split_s5-06	10.1	10.2	10.3	19.1	18.8	20.3
market_split_u5-09	10.0	10.1	10.3	18.9	18.3	20.3
quasigroup7_10	7.4	7.6	9.2	12.7	12.3	18.3
sb_sb_13_13_6_4	3.9	4.7	6.0	5.4	5.1	11.8
bacp-27	10.8	10.8	6.8	19.1	16.8	13.8
depot_placement_att48_5	6.8	6.9	7.0	12.0	16.1	14.3
depot_placement_rat99_5	2.4	2.8	2.9	4.1	8.0	5.5
depot_placement_st70_6	7.6	7.6	7.6	15.9	14.4	15.3
golombruler_13	8.5	8.5	8.7	18.1	17.0	17.5
open_stacks_01_problem_15_15	4.4	4.5	4.6	10.6	9.8	9.4
open_stacks_01_wbp_20_20_1	7.0	7.4	7.4	18.2	17.4	14.9
open_stacks_01_wbp_30_15_1	7.4	7.4	7.5	12.1	11.8	15.7
pattern_set_mining_k1_german-credit	3.6	3.6	3.6	5.9	5.8	7.1
pattern_set_mining_k1_yeast	16.2	16.7	19.4	23.4	22.4	39.4
radiation_03	3.1	3.5	3.8	5.8	5.4	8.0
still_life_still_life_9	7.0	6.7	7.1	13.6	13.4	14.3
sugiyama2_g5_7_7_7_7_2	6.9	7.1	8.4	10.5	10.8	16.2
talent_scheduling_alt_film117	45.0	42.6	32.1	66.4	65.7	66.2
geometric average (su)	7.2	7.4	7.3	13.2	13.6	14.7

Scaling factor



EPS determinism

- We want to be able to always compute the same solution
- If we consider the subproblems along with the decomposition ordering then when a solution is found in sp_k then
  - some subproblems whose index is $< k$ have already been solved
  - some subproblems are currently being solved by some workers.
    - Some of them have an index $i < k$
    - Some of them have an index $j > k$
- If we wait for the resolution of all the subproblems with $i < k$ currently being solved, then we will be able to define the FIRST subproblem having a solution. SO we will be able to repeat the run independantly of the ordering in which the subproblems are solved!

Embarrassingly Parallel Search

- In computer science, a problem that is obviously decomposable into many separate subtasks is called **embarrassingly parallel**

- Comes from the french expression “avoir l’embarras du choix”.

- Properties
  - Computation can be easily divided into several independent parts, each part can be executed by a processor.
  - No or very few communication between processus
  - Each process works regardless of others

EPS Advantages

- Simple
- No or very few communication
- Not intrusive in the solver (we just need to get a subproblem and to test the propagation)
- We can easily replay the resolution
  - We just have to save the order of solved problems and the assigned problems to the workers
- Competitive with work stealing

EPS advantages

- At anytime, gives an idea about the quantity of the problem that has been soved
- Determinism is possible
  - Replay (Easy and fast)
  - change of machine (More complex: the decomposition should respect the order)

Plan

- Background (Parallelism, Concurrency, distributed computing …)
- Parallelism in CP
  - Distributed CSP
  - Parallel propagation
  - Portfolio method
  - Parallel Search
- **Time is money: how to manage the solving time**

Question

- We have a problem P to solve
- We have unlimited resources but they cost money
- We have a limited resolution time

- Question : **how can we solve P for the minimum cost while respecting the resolution time?**

Question

- How can I use computers to speed up the solving time of a problem?
  - I want to gain a factor of 10. How can I reach that goal?
  - I have a maximum amount of time for solving a given problem, how many machines should I use?
  - Resources cost money, how can I solve my problem in less than x hours for the minimum cost?

- Unfortunately, using k computers does not mean gaining a factor of k

Time is money

- Thanks to cloud computing we can have the power that we want!
  - It just costs money…
- I have something to do which requires x units of computation time on my machine
  - Can I solve it on the cloud within a certain amount of time (x/5; x/100)? How much it will cost?

Speed up the resolution

- Any type of computer can be used
  - Local parallel machines
  - Local distributed machine
  - Computer center – supercomputer
  - Cloud infrastructure

- Any type of parallelisation technique can be used
  - Work stealing
  - Embarassingly parallel search

Azure cloud

		Gecode 4.0.0			
		work stealing		EPS	
		24 workers			
instance	seq. time (s)	// time (s)	speedup	// time (s)	speedup
bacp-27	4256,8	548,4	7,8	260,2	16,4
depot_placement_att48_5	298,5	21,3	14,0	17,6	16,9
depot_placement_rat99_5	52,5	10,1	5,2	8,5	6,2
depot_placement_st70_6	7929,0	1172,5	6,8	433,9	18,3
fastfood_ff58	63,1	11,3	5,6	3,2	20,0
fillomino_18	2227,1	184,6	12,1	160,2	13,9
golombruler_13	3167,3	210,4	15,1	154,0	20,6
market_split_s5-02	11367,4	658,6	17,3	467,1	24,3
market_split_s5-06	11039,8	650,7	17,0	452,7	24,4
market_split_u5-09	11421,6	609,2	18,7	468,1	24,4
open_stacks_01_problem_15_15	284,5	38,4	7,4	26,4	10,8
open_stacks_01_wbp_20_20_1	5338,7	374,1	14,3	302,7	17,6
open_stacks_01_wbp_30_15_1	521,0	72,3	7,2	30,0	17,4
pattern_set_mining_k1_german-credit	247,8	31,4	7,9	29,7	8,3
pattern_set_mining_k1_yeast	7938,7	482,7	16,4	172,1	46,1
quasigroup7_10	683,5	67,1	10,2	31,8	21,5
radiation_03	274,4	46,2	5,9	30,6	9,0
sb_sb_13_13_6_4	257,7	23,3	11,0	18,4	14,0
still_life_still_life_9	3187,4	196,8	16,2	189,0	16,9
sugiyama2_g5_7_7_7_7_2	602,6	47,1	12,8	32,2	18,7
talent_scheduling_alt_film117	1677,8	110,5	15,2	22,7	74,0
			10,7		17,9

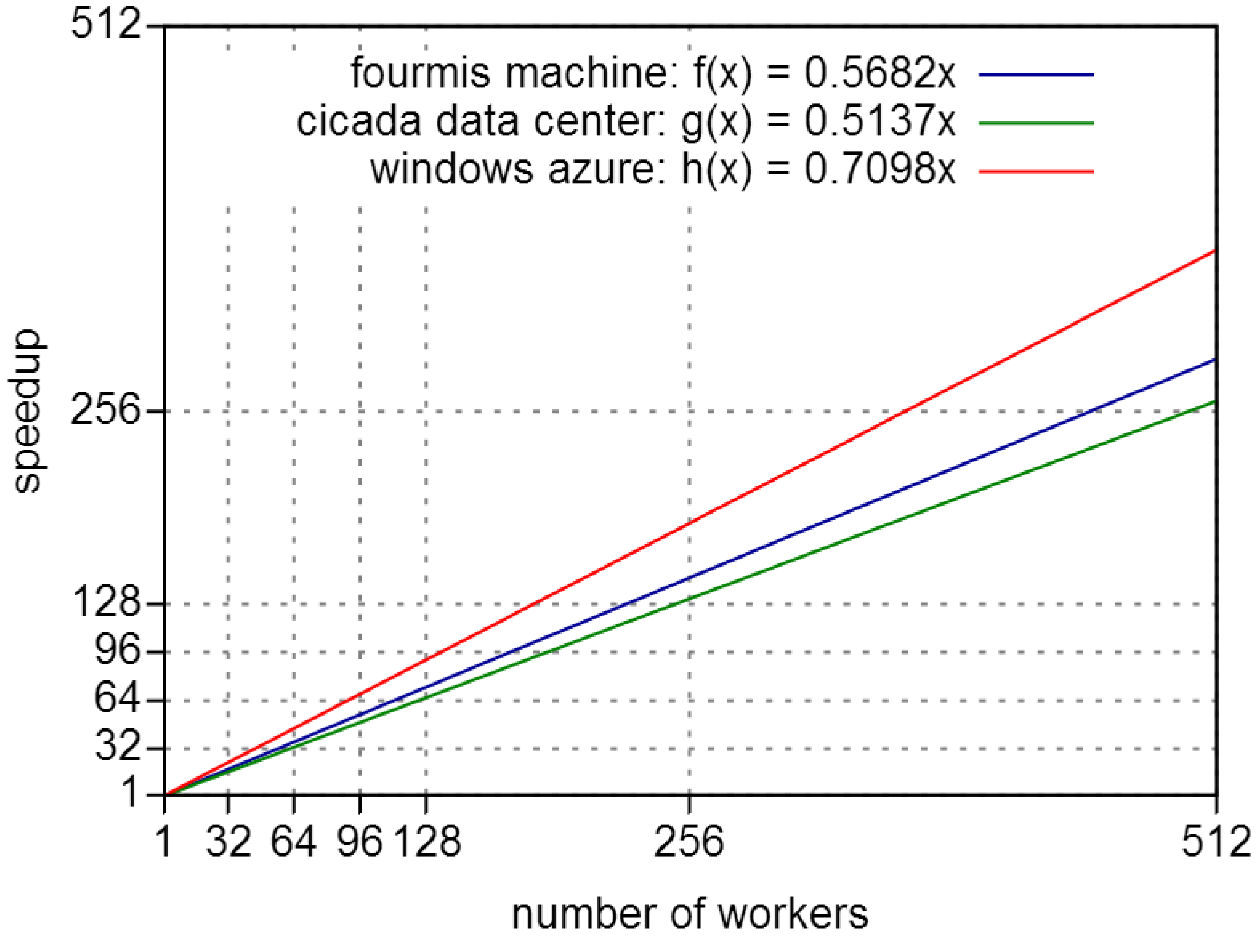
Cloud comparison

Instance	10 workers			20 workers		
	fourmis	cicada	azure	fourmis	cicada	azure
fillomino_18	5.4	5.8	5.8	9.1	8.8	12.0
market_split_s5-02	10.0	10.1	10.2	18.6	18.3	20.3
market_split_s5-06	10.1	10.2	10.3	19.1	18.8	20.3
market_split_u5-09	10.0	10.1	10.3	18.9	18.3	20.3
quasigroup7_10	7.4	7.6	9.2	12.7	12.3	18.3
sb_sb_13_13_6_4	3.9	4.7	6.0	5.4	5.1	11.8
bacp-27	10.8	10.8	6.8	19.1	16.8	13.8
depot_placement_att48_5	6.8	6.9	7.0	12.0	16.1	14.3
depot_placement_rat99_5	2.4	2.8	2.9	4.1	8.0	5.5
depot_placement_st70_6	7.6	7.6	7.6	15.9	14.4	15.3
golombruler_13	8.5	8.5	8.7	18.1	17.0	17.5
open_stacks_01_problem_15_15	4.4	4.5	4.6	10.6	9.8	9.4
open_stacks_01_wbp_20_20_1	7.0	7.4	7.4	18.2	17.4	14.9
open_stacks_01_wbp_30_15_1	7.4	7.4	7.5	12.1	11.8	15.7
pattern_set_mining_k1_german-credit	3.6	3.6	3.6	5.9	5.8	7.1
pattern_set_mining_k1_yeast	16.2	16.7	19.4	23.4	22.4	39.4
radiation_03	3.1	3.5	3.8	5.8	5.4	8.0
still_life_still_life_9	7.0	6.7	7.1	13.6	13.4	14.3
sugiyama2_g5_7_7_7_7_2	6.9	7.1	8.4	10.5	10.8	16.2
talent_scheduling_alt_film117	45.0	42.6	32.1	66.4	65.7	66.2
geometric average (su)	7.2	7.4	7.3	13.2	13.6	14.7

Resolution Speed up

- sf: scaling function
  - Ratio of the scaling obtained for a given number of cores
  - sfA for the cloud and sfM for the machine
- The number of cores needed to increase by a factor of p the power of a machine using k cores is
  - $x = sfM^{-1}(p*sfM(k))$

Scaling factor



Performance ratio

- pr(A/M) is the ratio of
  - the performance of ONE core of the machine A, and
  - the performance of ONE core of the machine M.

  - 3167/1355= 2.34
    - (3167 Azure, 1355 Cicada or // machine)

Power equivalence

- M1 machine with k1 cores
- The number of cores of the machine M2 needed to have an equivalent power than the machine M1 is
  - $k2 = sfM_2^{-1}(sfM_1(k1)pr(M2,M1))$

Power equivalence

- For 20 cores:
  - Azure $sfA(x)=0.71$
  - // Machine $sfF(x)=0.66$
  - Data center $sfC(x)=0.68$

- $k2 = sfM_2^{-1}(sfM_1(k1)pr(M2,M1))$

	Microsoft Azure cloud	fourmis (server)	cicada (data center)
$\#cores$	20	9,19	8,92

Time is money

- Fourmis (server = 40 cores) 88 Azure cores, hourly cost €6.45
- Cicada (data center = 1150 cores). 2579 Azure cores, hourly cost €177.69

Time is money

- Machine M1 solve the problem P with k1 cores in t1 unit of time
- We can solve P in t2 unit of time with the machine M2 by using k2 cores defined as

- **$k2=sfM_2^{-1}(t1/t2\ sfM_1(sfM_2^{-1}(sfM_1(k1)pr(M2,M1))))$**

- Eg.
  - 3 hours of computation on Parallel machine (40 cores);
  - compute the problem in less than 1h on the cloud

Controlling the resolution time

- 3 hours of computation on Parallel machine (40 cores); Compute the problem in less than 1h on the cloud
- We need to speed up the resolution by a factor of 3.
- On the cloud we need 2.4 times more
- We need 40*2.4=96 cores on the cloud
- We need an improvement by a factor of 3: 96*3=288
- Hourly cost is €0.07 per core that is €20

Time is money

- Fourmis (server = 40 cores) equivalent of 88 Azure cores, hourly cost €6.45
- Cicada (data center = 1150 cores) equivalent of 2579 Azure cores, hourly cost €177.69
  - Be careful this calculation assumes a perfect scaling factor

Controlling the resolution time

- 3 hours of computation on Parallel machine (40 cores); Compute the problem in less than 1h on the cloud
- We need to speed up the resolution by a factor of 3.
- On the cloud we need 2.4 times more cores (performance ratio)
- We need 40*2.4=96 cores on the cloud
- We need an improvement by a factor of 3
  - We have a scaling factor of 0.71 on the cloud
  - Result: 96*3/0.71=405
- Hourly cost is €0.07 per core that is €28.4

Conclusion

- EPS works well on a cloud infrastructure
- The speed-up is comparable with the speed-up obtained with a parallel machine
- We propose to compare machines and compute the number of cores for having an equivalent power
  - We can deduce how much it will cost for obtaining a certain power (time is money)

Conclusion

- Not sure that it is still worthwhile to buy machines dedicated to computation

- We need to test on more cores
  - We asked Microsoft for that
  - We could also buy some resources (€93/h for a comparison with 512 cores of the Nice’s data center)

- We should compare with Amazon EC2 and Google compute engine

Plan

- Background (Parallelism, Concurrency, distributed computing …)
- Parallelism in CP
  - Distributed CSP
  - Parallel propagation
  - Portfolio method
  - Parallel Search
- Time is money: how to manage the solving time

General Conclusion

- Parallelism is challenging and difficult, but it may be fun!
- Understand well the basic concepts
  - Race condition, critical section, atomicity
  - Load balancing, starving
- Try to minimize the communication
- Try to avoid synchronization in the code (barrier)
- Do not forget to look at the behavior of your method according to the number of workers

General Conclusion

- A simple method which works well in practice is certainly one of the best compliments in computer science