Parallélisme: lien entre adaptation et complexité algorithmique

Parallelism: algorithmic complexity and adaptation

Jean-Louis Roch
Moais team-project – LIG

Séminaire LIG, 7 Juin 2011

Grenoble INP
GRENOBLE UNIVERSITÉS
LIG
INRIA RHÔNE-ALPES

« The Top 10 Algorithms of the 20^{th}»

[J. Dongarra, F. Sullivan editors, Computing in Science and Engineering, Feb. 2000]

- 1946: The Metropolis Algorithm for Monte Carlo.
- 1947: Simplex Method for Linear Programming.
- 1950: Krylov Subspace Iteration Method.
- 1951: The Decompositional Approach to Matrix Computations.
- 1957: The Fortran Optimizing Compiler.
- 1959: QR Algorithm for Computing Eigenvalues.
- 1962: Quicksort Algorithms for Sorting.

- **1965: Fast Fourier Transform**. ***« An algorithm the whole family can use »***
  - « (…) *the most ubiquitous algorithm in use today to analyze and manipulate digital or discrete data. The FFT takes the operation count for discrete Fourier transform from $O(N^2)$ to $O(N \log N)$.* »

- 1977: Integer Relation Detection.
- 1987: Fast Multipole Method.

between the publication of GAUSS' algorithm and the modern rediscovery of this approach by COOLEY & TUKEY.

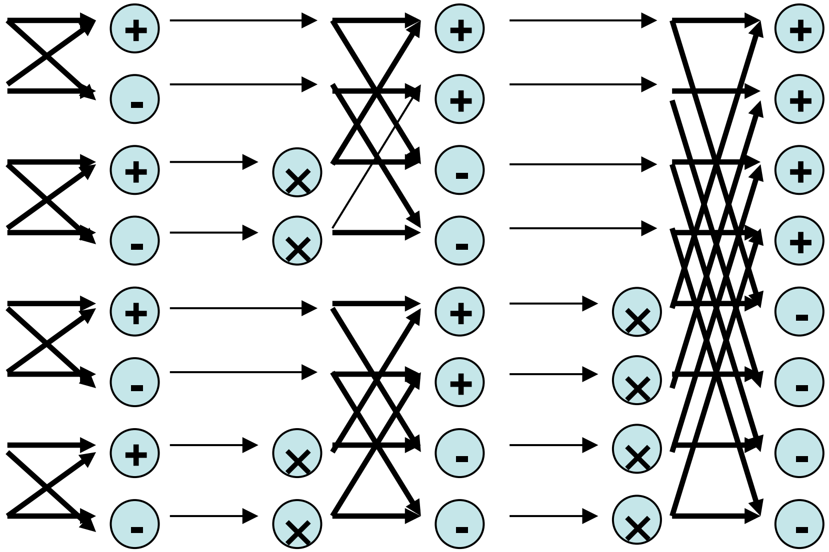
Principal Discoveries of Efficient Methods of Computing the DFT

Researcher(s)	Date	Lengths of Sequence	Number of DFT Values	Application
C. F. GAUSS [10]	1805	Any composite integer	All	Interpolation of orbits of celestial bodies
F. CARLINI [28]	1828	12	7	Harmonic analysis of barometric pressure variations
A. SMITH [25]	1846	4, 8, 16, 32	5 or 9	Correcting deviations in compasses on ships
J. D. EVERETT [23]	1860	12	5	Modeling underground temperature deviations
C. RUNGE [7]	1903	2^nK	All	Harmonic analysis of functions
K. STUMPFF [16]	1939	2^nK, 3^nK	All	Harmonic analysis of functions
DANIELSON & LANCZOS [5]	1942	2^n	All	X-ray diffraction in crystals
L. H. THOMAS [13]	1948	Any integer with relatively prime factors	All	Harmonic analysis of functions
I. J. GOOD [3]	1958	Any integer with relatively prime factors	All	Harmonic analysis of functions
COOLEY & TUKEY [1]	1965	Any composite integer	All	Harmonic analysis of functions
S. WINOGRAD [14]	1976	Any integer with relatively prime factors	All	Use of complexity theory for harmonic analysis

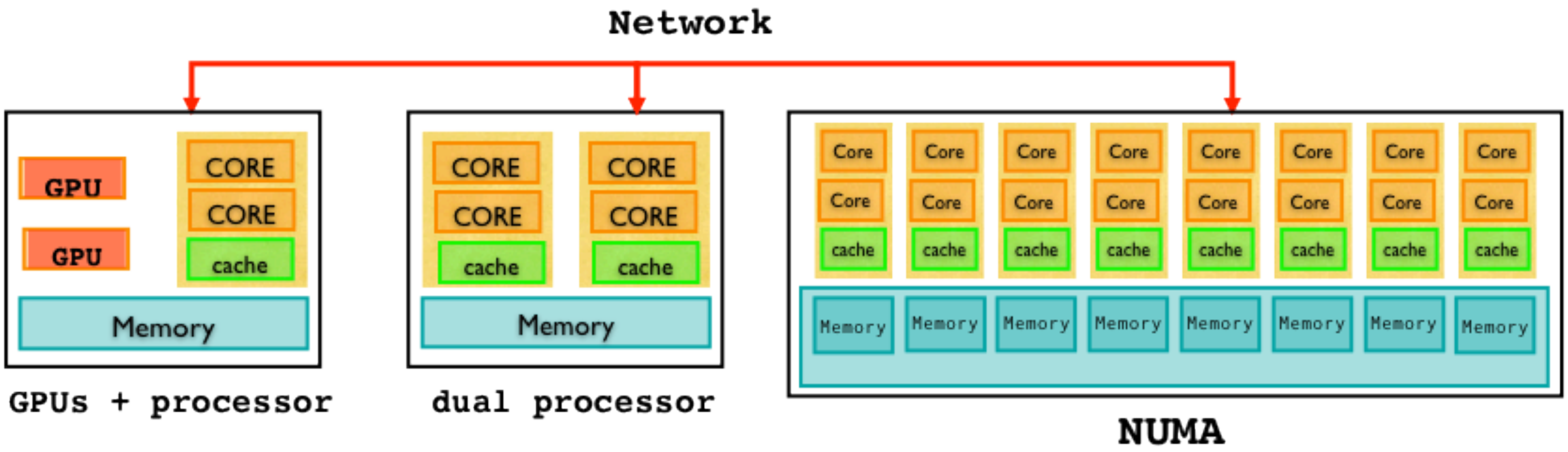
M. T. Heideman, D. H. Johnson, C. S. Burrus, *Gauss and the history of the fast Fourier transform*,

IEEE ASSP Magazine, 1 (4), 14–21 (1984)

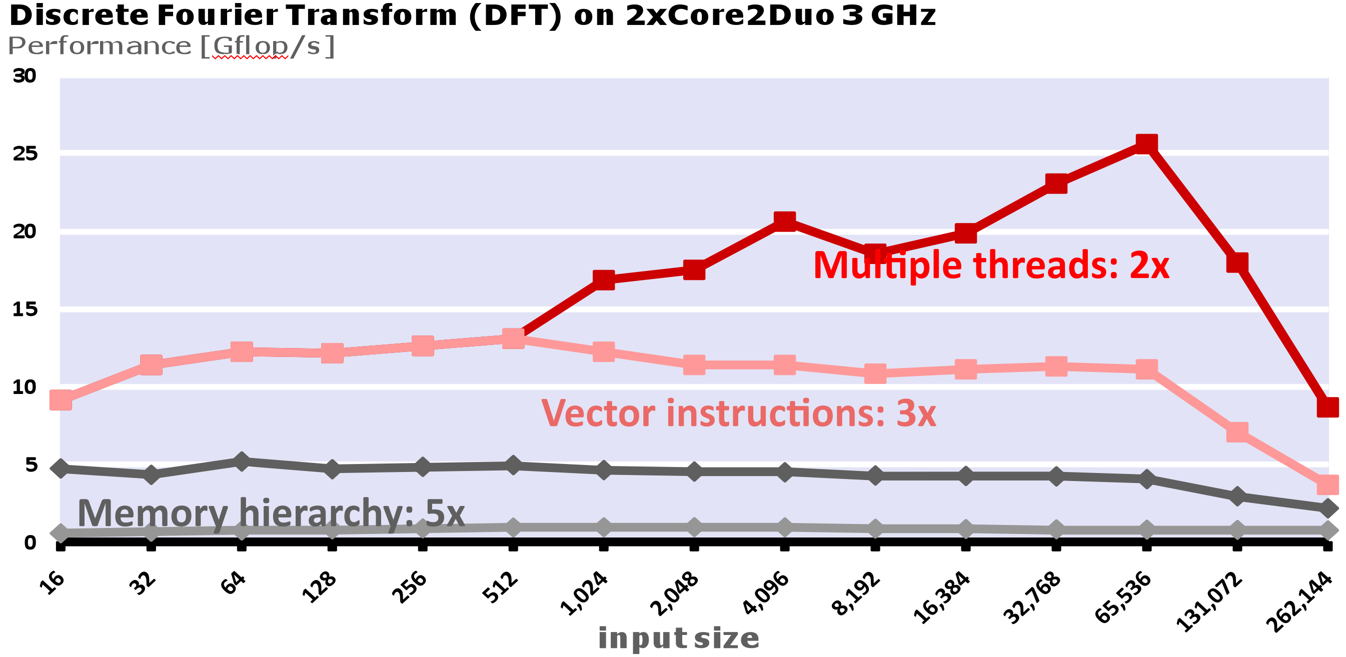
FFT radix 2 - butterfly



Context



- More and more cores per chip
- Non-uniform (GPU, FPGA, …)
- Hierarchical, non uniform memory access



[J. Johnson, SPIRAL]

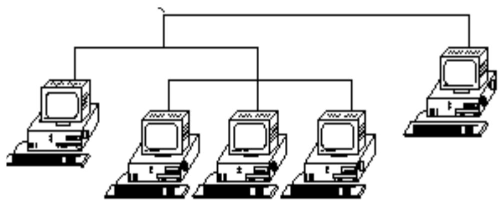
La programmation devient un cauchemar…

Comment synthétiser des programmes efficaces:

- de nouveaux modèles (abstraction support d'exécution)
- de nouveaux schémas algorithmiques pour les exploiter
- de nouveaux langages de plus haut niveau pour les compiler.

Why adaptive algorithms and how?

Resources availability
is versatile



Input data vary

$$\begin{bmatrix} 7 & 3 & 6 \\ 0 & 1 & 8 \\ 0 & 0 & 5 \end{bmatrix}$$

Measures on resources

Measures on data

Adaptation to improve (multicriteria) performances

Scheduling
- partitioning
- load-balancing
- work-stealing

Choices in the algorithm
- ***sequential / parallel(s)***
- ***approximated / exact***
- ***in memory / out of core***
- ***…***

Calibration
- tuning parameters
  - block size/ cache
  - choice of instructions, …
- priority managing

Def: "An algorithm is « hybrid » iff there is a choice at a high level between at least two algorithms, each of them could solve the same problem"

[Cung, V.D., Danjean, V., Dumas, J.G., Gautier, T., Huard, G., Raffin, B., Rapine, C., Roch, J.L., Trystram, D..TC2006]

Modeling an hybrid algorithm

- Several algorithms to solve a same problem **f** :
  - Eg : ***algo_f***$_1$, ***algo_f***$_2$(block size), … ***algo_f***$_k$:
  - each *algo_f*$_k$ being “recursive”

```
algo_fi ( n, … )     {
        ….
        f ( n - 1, … ) ;
        ….
        f ( n / 2, … ) ;
        …
}
```

Adaptation
to choose *algo_f*$_j$ for each call to f

E.g. ***“practical” hybrids****:*
- *Atlas, Goto, FFPack*
- *FFTW, SPIRAL*
- *cache-oblivious B-tree*
- *any parallel program with scheduling support: Cilk, Athapascan/Kaapi, Nesl,TBB…*

.

Example : Discrete Fourier Transform

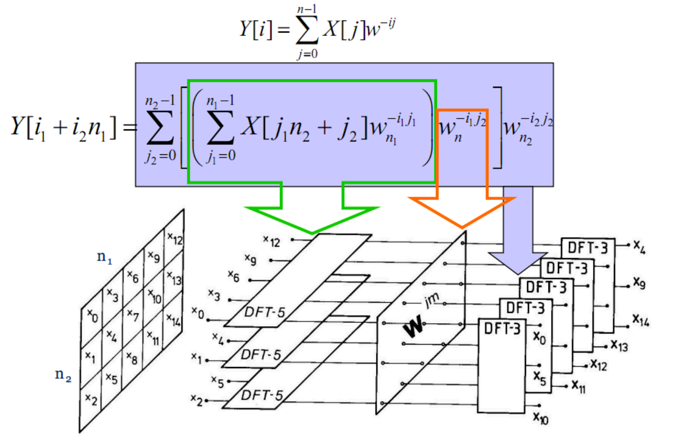
- FFT(vector of size ***n* ≤ *k.m***) reduces to :

  k* FFTs** with size ***m

  \+ **1 transpose** ***m*×*k* → *k*×*m***

  (twiddle factors)

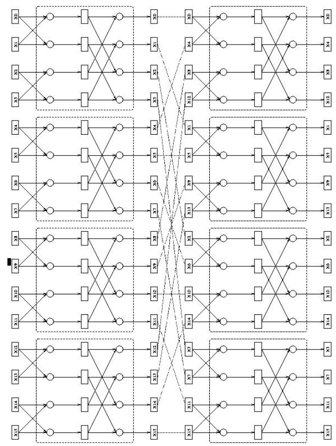
  \+ ***m* FFTs** with size ***k***



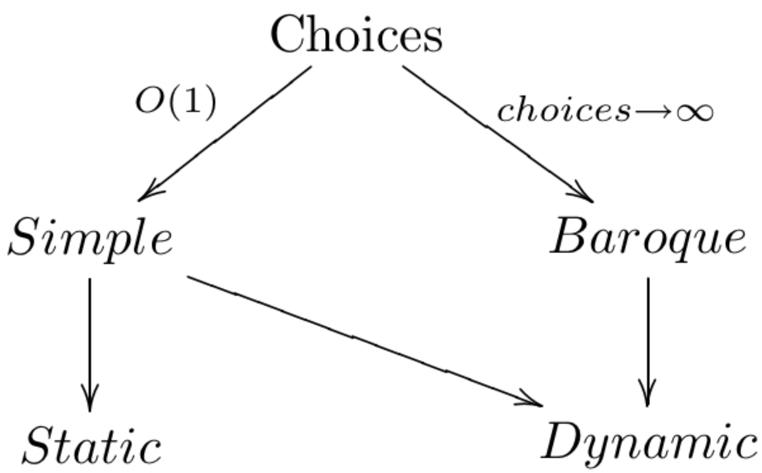
How to manage the choices ?

(? m, k, and among other DFT algorithms.

« adaptive algorithm »

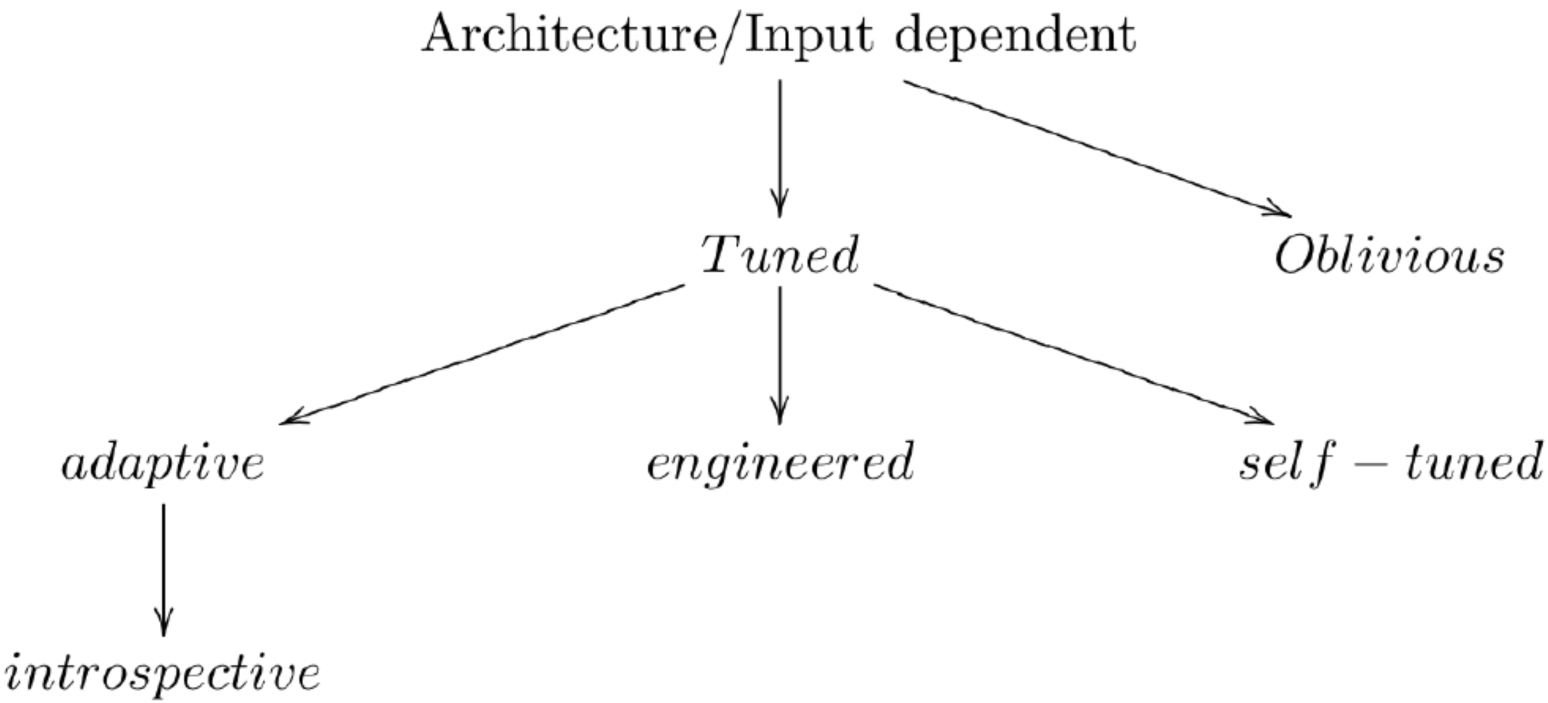


- How to manage overhead due to choices ?
- Classification 1/2 :
  - ***Simple*** *hybrid* iff O(1) choices
    [eg compiler optimizations, block size in Atlas, …]
  - ***Baroque*** *hybrid* iff an unbounded number of choices
    [eg recursive splitting factors in FFTW]
    - choices are either dynamic or pre-computed based on input properties.



- Choices may or may not be based on architecture parameters.

- Classification 2/2. :



Oblivious: « *An algorithm is said **oblivious** if no program variables dependent on hardware configuration parameters need to be tune to reach optimal performances* » [Prokop&al]

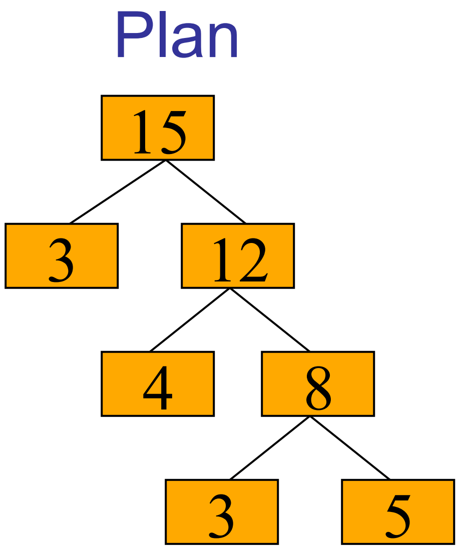
La bibliothèque FFTW [Frigo, Johnson IEEE05]

http://www.fftw.org/ [3.2.2])

- Utilise des “codelets”: code optimisé pour les FFTs de petite taille.

- Pour un *n* fixé (paramètre), combine ces codelets via par découpe récursive de la FFT (Split-radix)
  - Différentes stratégies, in-out, ...

- Utilise la programmation dynamique pour trouver une combinaison efficace



FFTW : utilisation

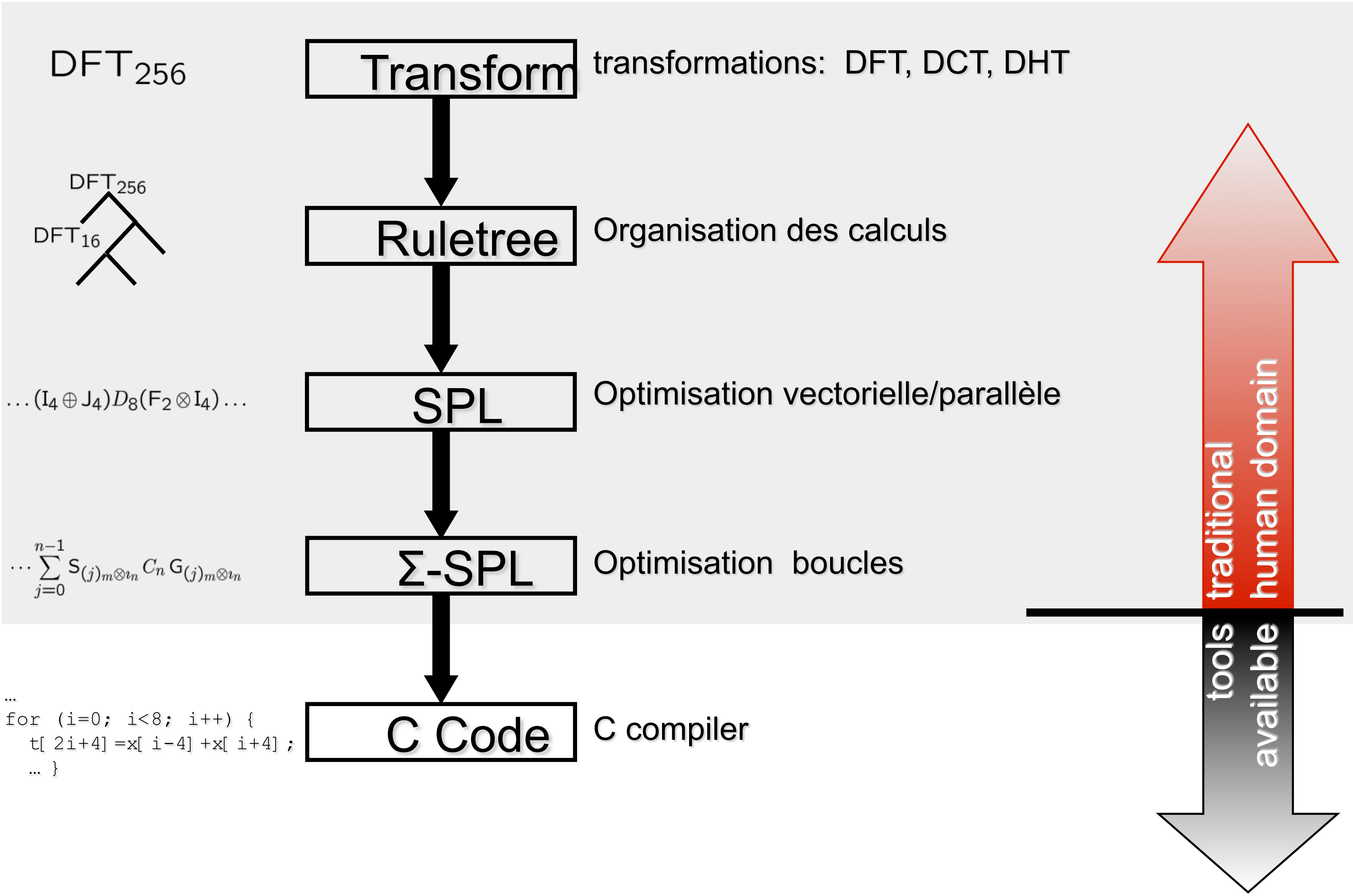
```
fftw_plan plan;
int n = 1024;
COMPLEX A[n], B[n];

/* plan the computation */
plan = fftw_create_plan(n);

/* execute the plan */
fftw(plan, A);

/* the plan can be reused */
fftw(plan, B);
```

Surcoût du calcul du plan amorti par sa réutilisation.

[Johnson&al. PLDI 05]

Synthèse par compilation: SPIRAL

www.spiral.net



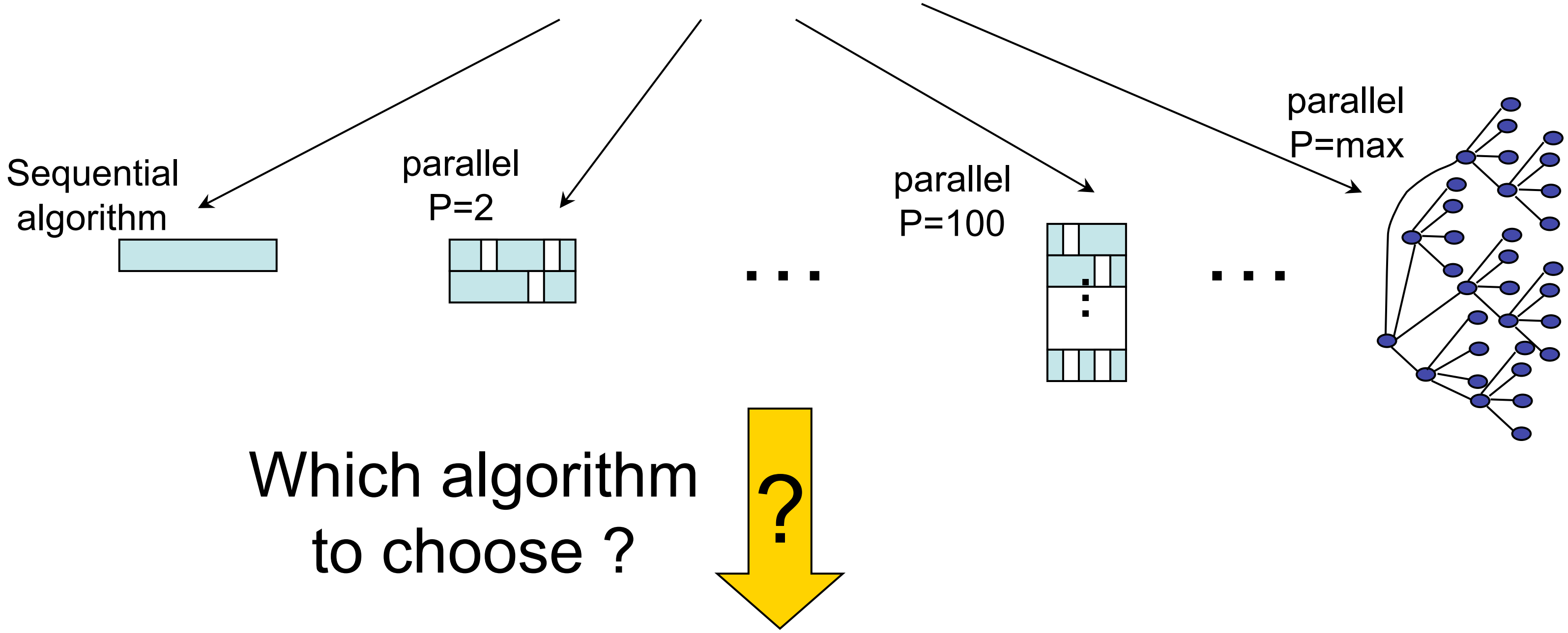
Performance : Oblivious adaptive

- Both :
  - « Poly »-algorithms
    - For a fixed input, several control flows are possible
  - And an adaptation technology

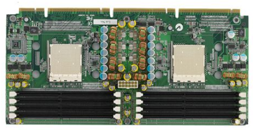
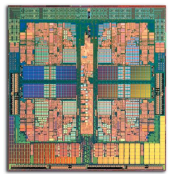
- But only the performance of the effective control flow matters
  - **Graal** : near-optimal performance « *An algorithm is said* ***oblivious*** *if no program variables dependent on hardware configuration parameters need to be tune to reach optimal performances* »

Towards oblivious algorithms

To design a single efficient algorithm
with provable performances on an arbitrary architecture

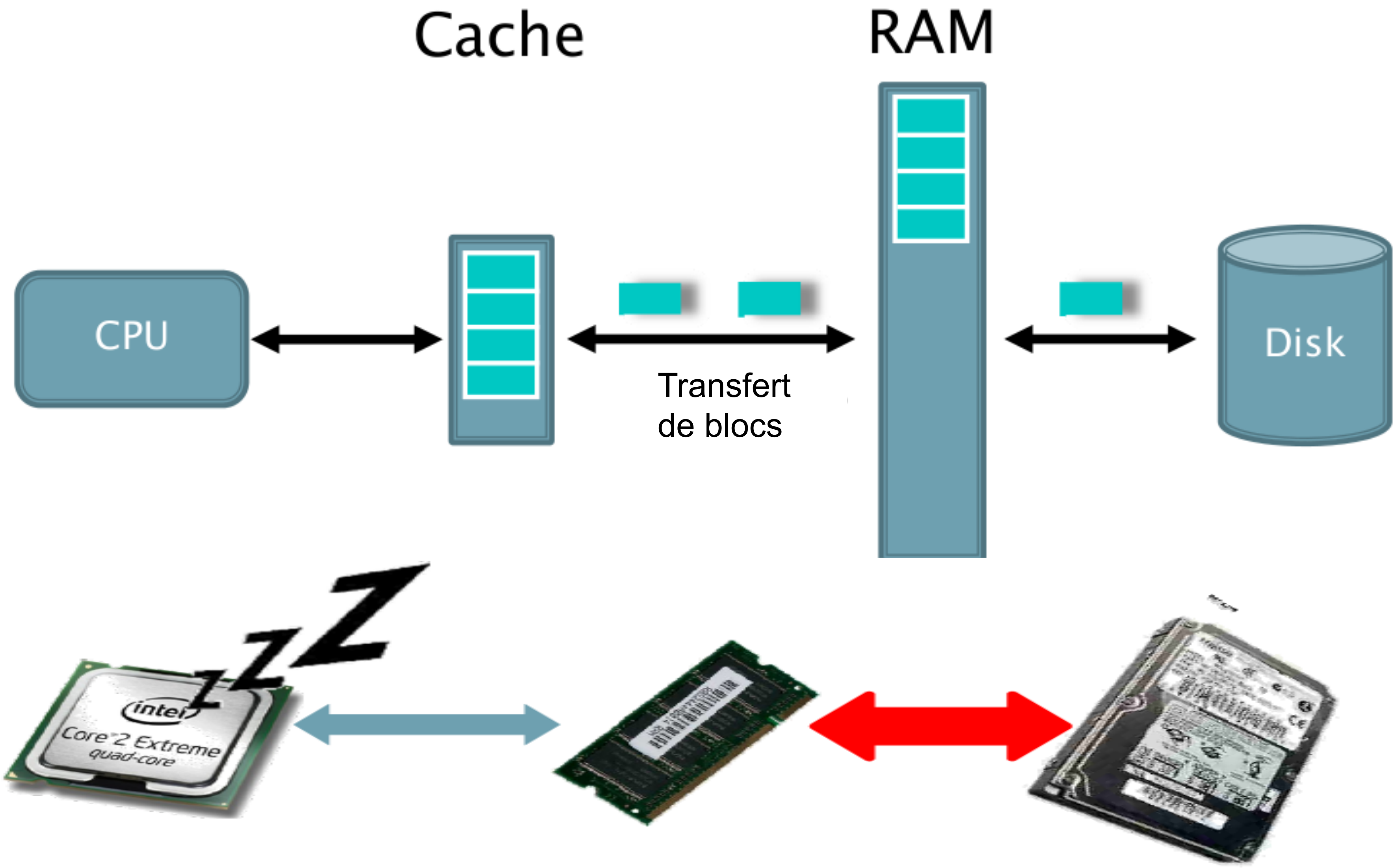





Exemple 1: Cache oblivious

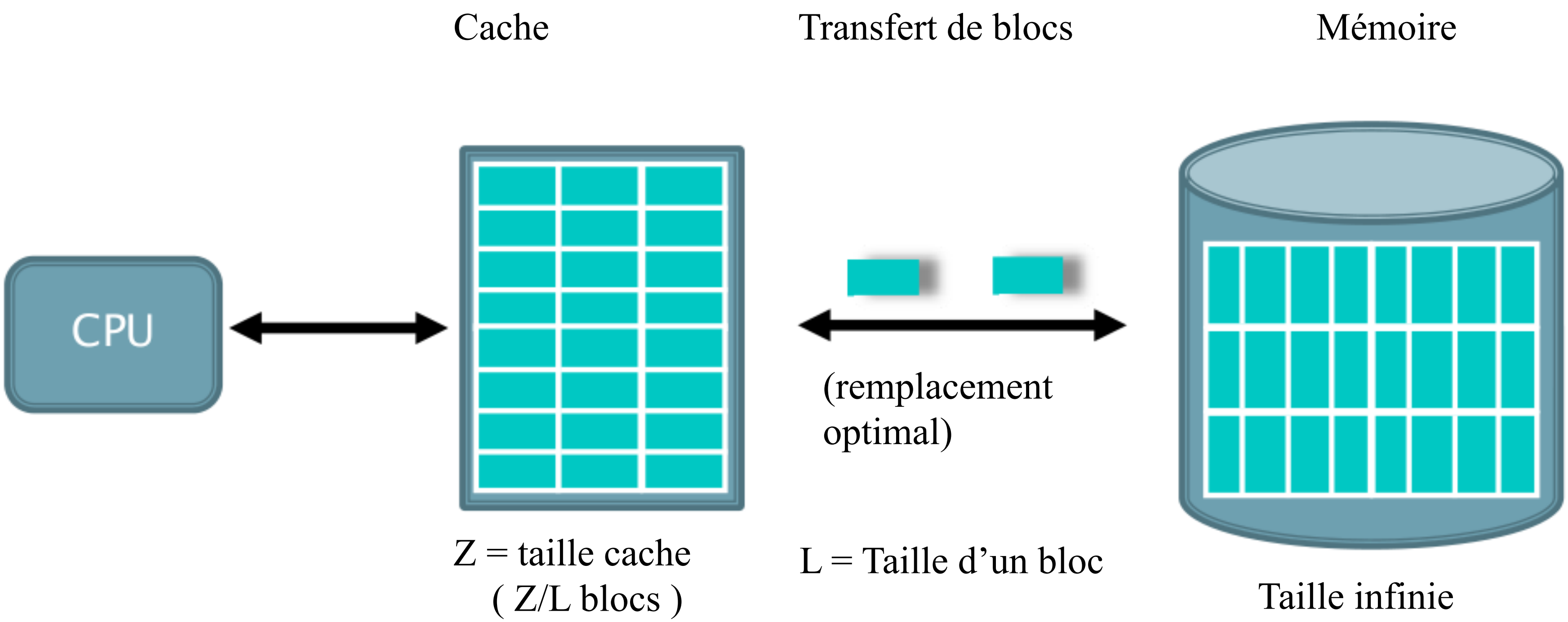
Hiérarchie mémoire et cache



[Aggarwal & Vitter 1988]
[Frigo & al 1999]

Modèle de cache

or external memory
out-of-core
disk access machine
I/O model



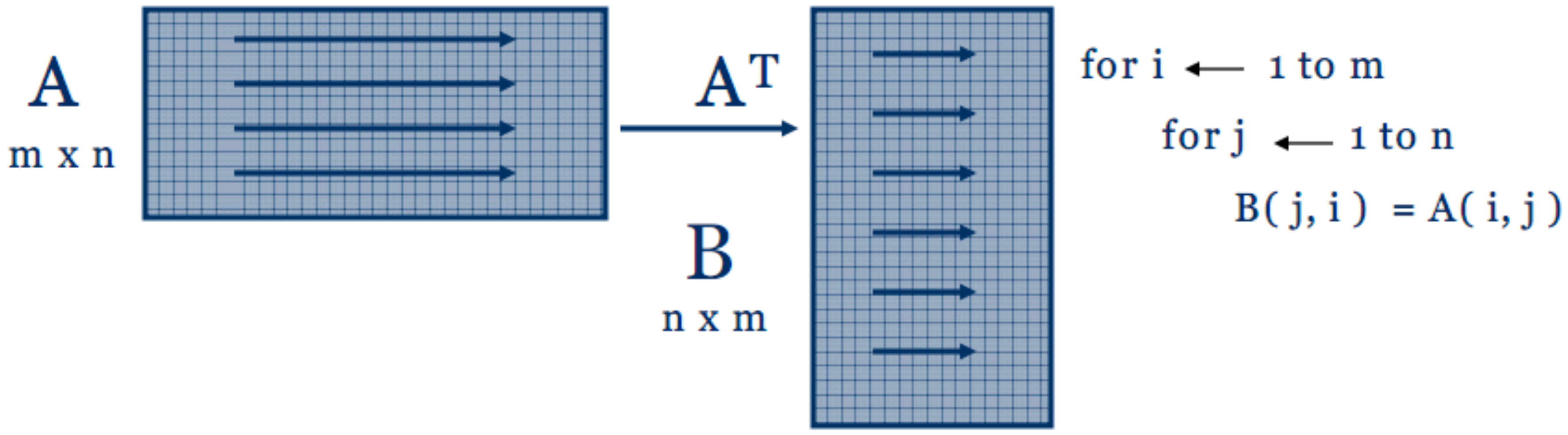
Travail (W) = #opérations

Défauts de cache: Q(n, L, Z) = #transferts de blocs

Exemple:Transposition de matrice

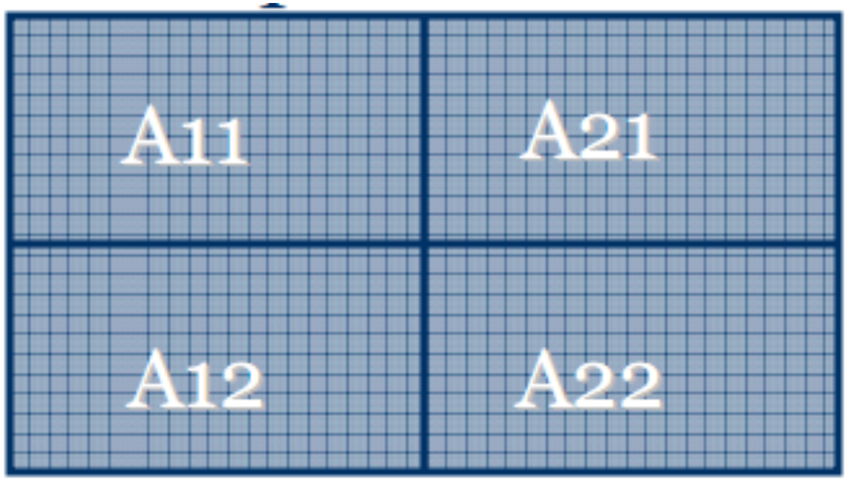
- Le disque = séquence de mots
  - Matrices stockées par lignes

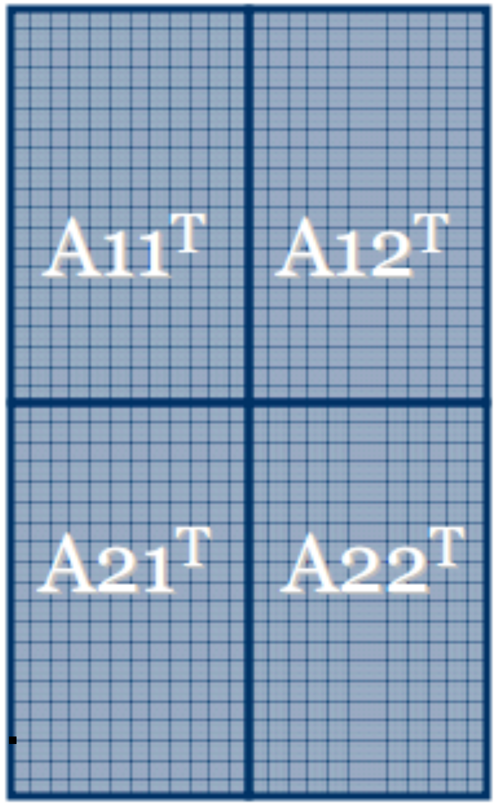
- Si n très large, l'accès de B par colonne cause un défaut de cache à chaque top



Transposition de matrice optimale

- Partitionner A selon la plus grande dimension, et récursivement transposer chaque bloc



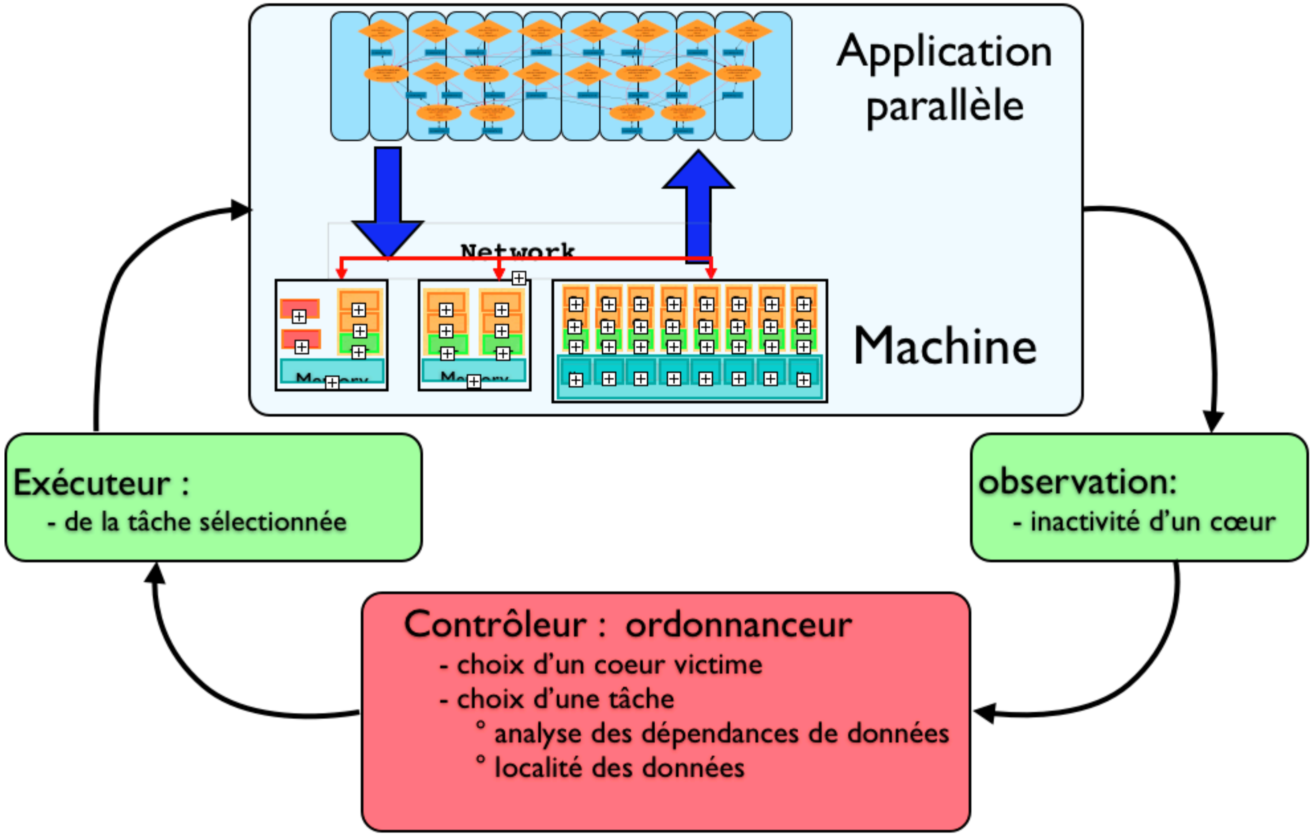


- Q(m,n) = O(1 + m.n/L) : optimal.
- « **Cache-oblivious** »: optimal sans référencer Z et L dans le programme.

Parallel processor-oblivious algorithms

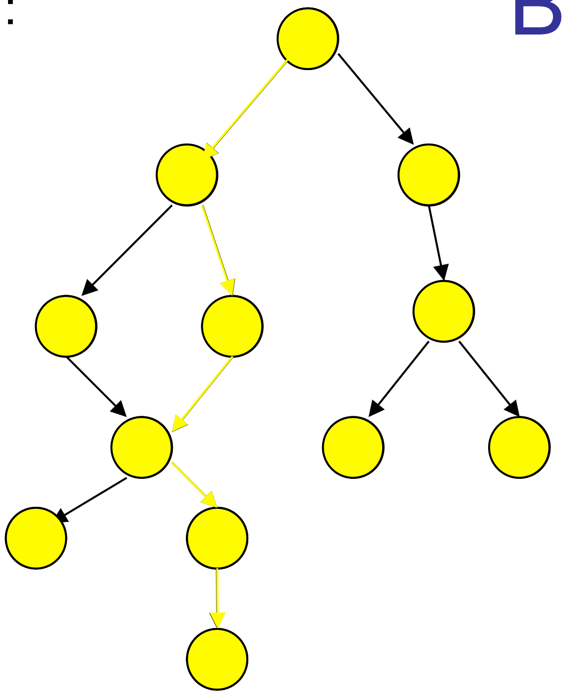
Work-stealing technology

[Cilk90, Athapascan/Kaapi, … CilkArts08,… X10, TBB]



- When idle, resource steals a ready task from another one:
  - "Greedy scheduling" (*list-scheduling)*
  - Distributed implementation (randomized)

:



Basic notations: Work and depth

"**Work**" W = #total number operations performed

"**Depth**" **D** = #operations on a critical path

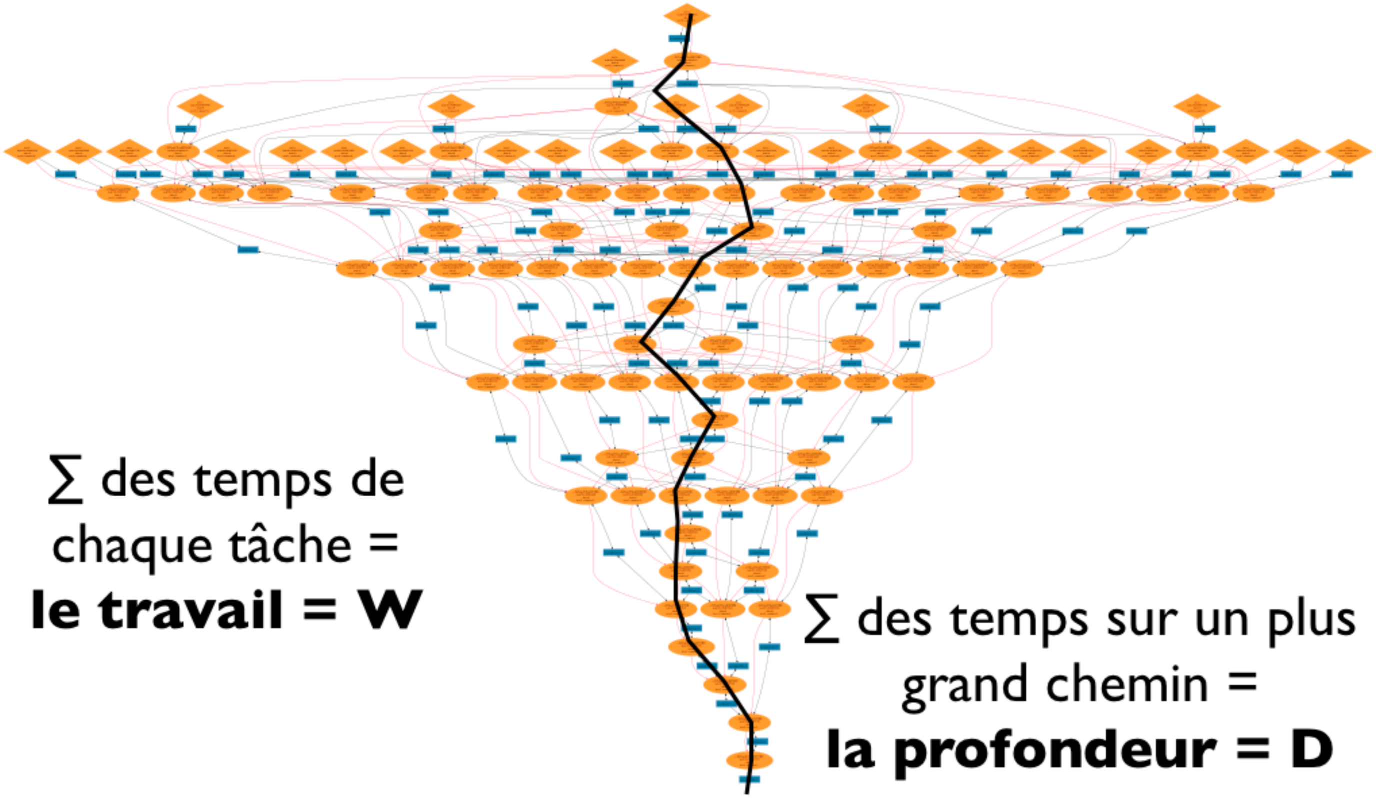
(~parallel "time" on ∞ resources)

Relation to execution time T(p, Π)

$$\frac{1}{\Pi_{ave}}\left(\frac{W}{p}+O(D)\right)$$

=> *Near optimal if Work >> Depth*

Cas trivial



∑ des temps de chaque tâche = **le travail = W**

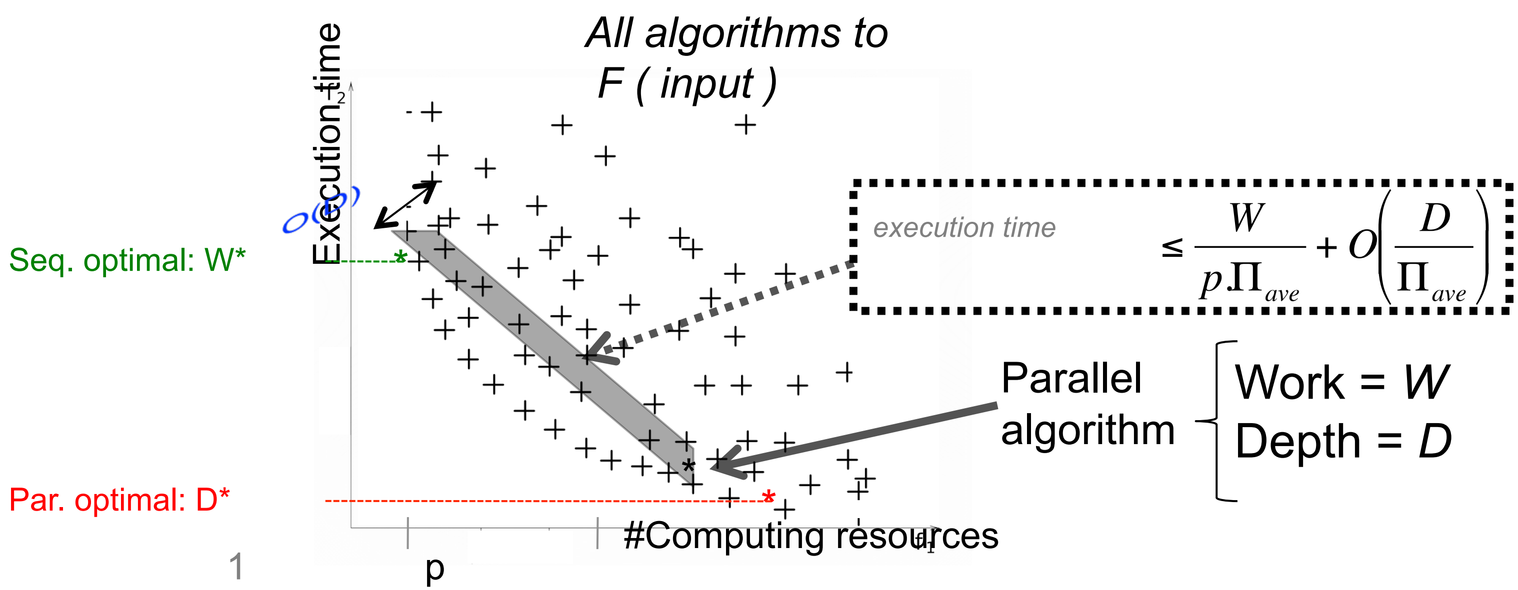
∑ des temps sur un plus grand chemin = **la profondeur = D**

- Exemple: calcul de $a_0 * a_1 * \dots * a_n$
  - Découpe en blocs élémentaires (grain fixé, dépendant de n => ~optimal)

Work-stealing technology

[Cilk90, Athapascan/Kaapi, … CilkArts08,… X10, TBB]

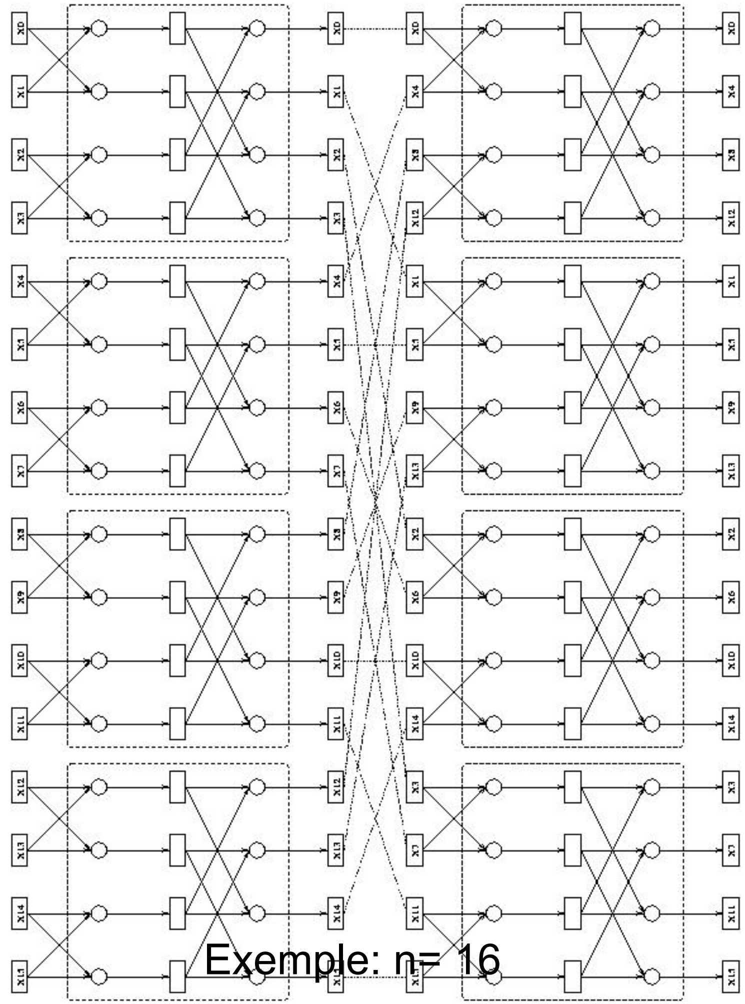
- For a fixed algorithm " * " :
  Provable performances including cost of scheduling control.
  - Uniform resources, variable speeds



- Note: some variant to cope with multi-objective and various implementation features [mixed push/pull…]

FFT « processor+cache oblivious »

- Avec une découpe récursive de taille √n :
  1. Calcul de √n FFTde taille √n
  2. Permutation par blocs : $V_i[j] \leftrightarrow V_j[i]$
  3. Calcule √n FFTde taille √n



Exemple: n= 16

- si n>Z: Q(n)=2√n.Q(√n)+O(n/L)
  sinon Q(n) = n.

- Q(n) = O(n/L . $\log_Z$ L) : optimal

- Cache+parallélisme :
  algorithmique portable

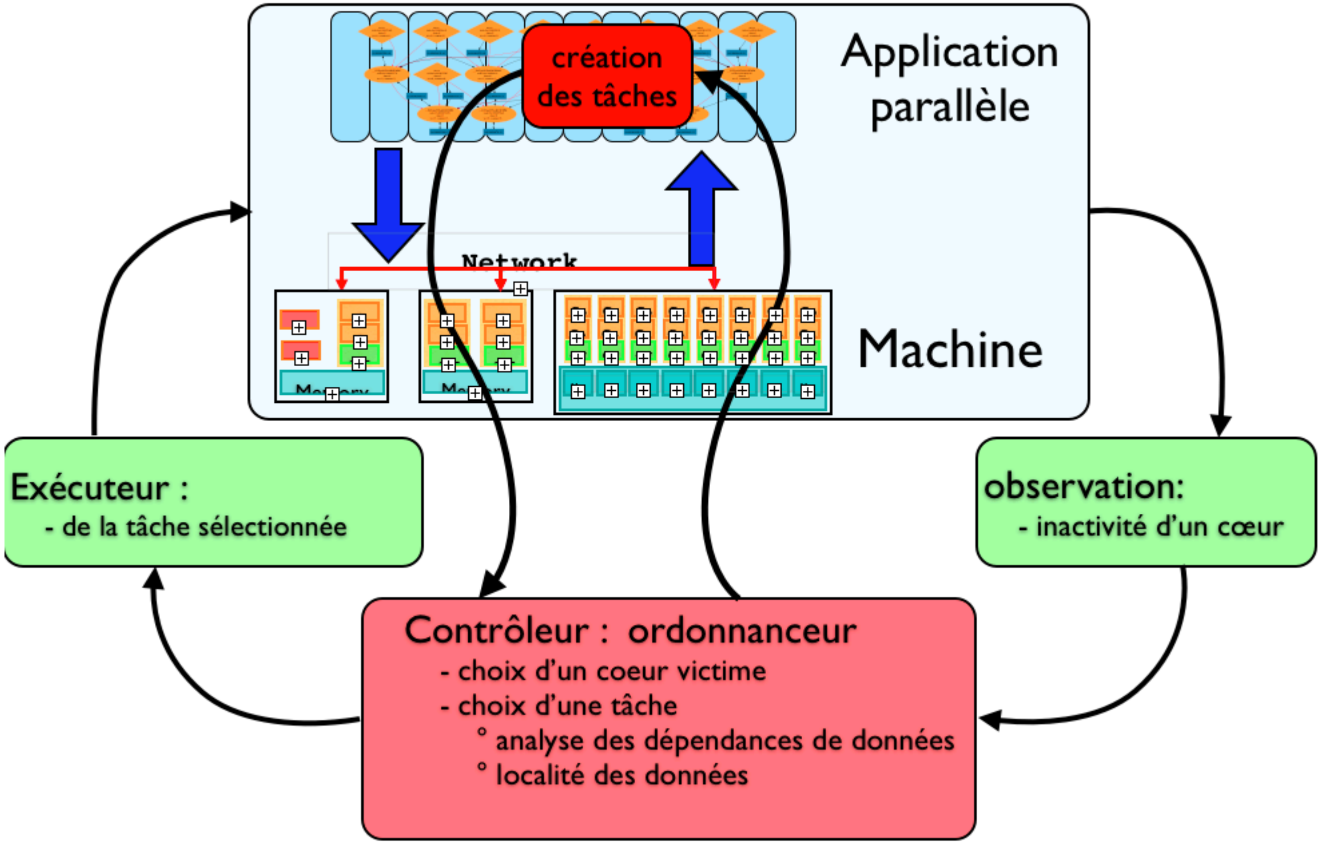
But often parallelism has a cost !

- Solution: to **mix** both a **sequential** and a **parallel** algorithm

- ***Adaptive granularity*** : dual approach :
  - Parallelism is ***extracted*** at run-time from ***any*** *sequential task*

Work-stealing and adaptive algorithms

[Thèses Tarore09, Tchiboukdjian10, Quintin11]



- When idle, resource steals a ready task from another one:
  - "Greedy scheduling" (*list-scheduling)*
  - Distributed implementation (randomized)

Adaptive work-stealing: concurrently sequential and parallel

*Based on the work-stealing and the **Work-first** principle* :

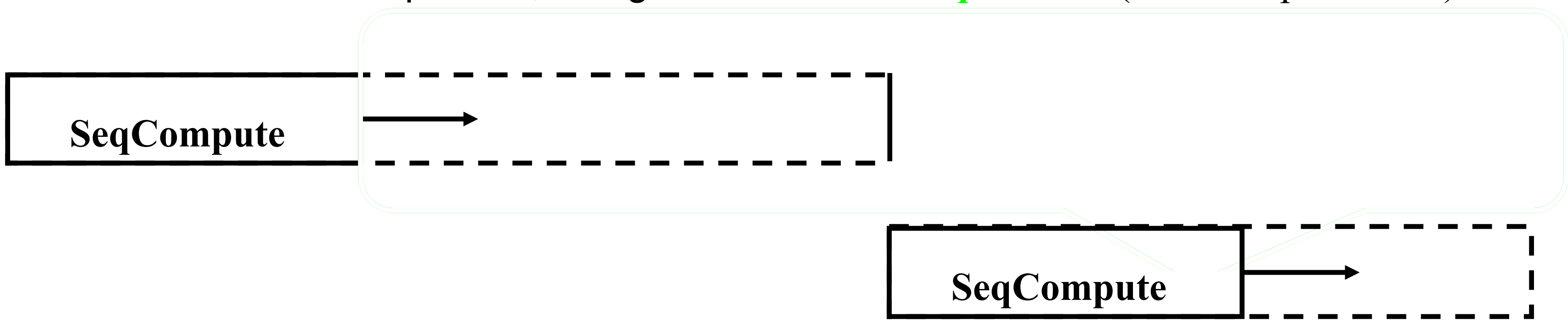
Instead of optimizing the **sequential execution** of the **best parallel** algorithm, let optimize the **parallel execution** of the ***best sequential*** algorithm

Execute always a sequential algorithm to reduce parallelism overhead

⇒ parallel algorithm is used only if a processor becomes **idle** (ie *workstealing*) [Roch&al2005,…] to ***extract parallelism*** from the remaining work a sequential computation

Assumption : two concurrent algorithms that are complementary:

- - one sequential : ***SeqCompute*** (always performed, the priority)
  - the other parallel, fine grain : ***LastPartComputation*** (often not performed)



Adaptive work-stealing : concurrently sequential and parallel

Based on the work-stealing and the ***Work-first*** *principle* :

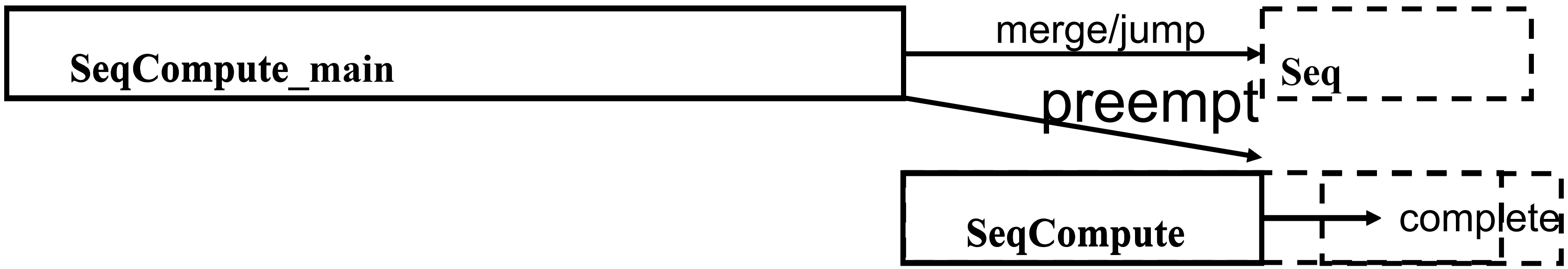
Instead of optimizing the **sequential execution** of the **best parallel** algorithm, let optimize the **parallel execution** of the ***best sequential*** algorithm

Execute always a sequential algorithm to reduce parallelism overhead

⇒ parallel algorithm is used only if a processor becomes **idle** (ie *workstealing*) [Roch&al2005,…] to ***extract parallelism*** from the remaining work a sequential computation

Assumption : two concurrent algorithms that are complementary:

- - one sequential : ***SeqCompute*** (always performed, the priority)
  - the other parallel, fine grain : ***LastPartComputation*** (often not performed)



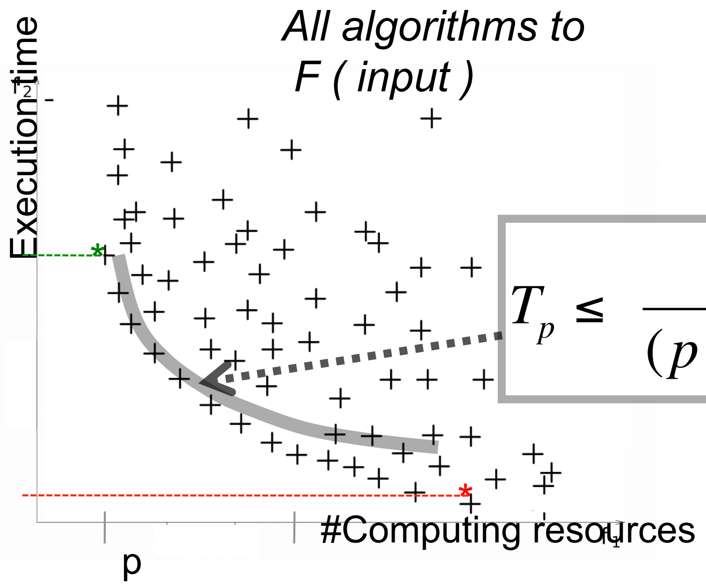
Note:

- **merge and jump** operations to ensure non-idleness of the victim
- Once *SeqCompute_main* completes, it becomes a work-stealer

Work-stealing and algorithms choice

•

$$T_p \le \frac{W_p}{p\,\Pi_{ave}} + O\left(\frac{D_p}{\Pi_{ave}}\right)$$



Sequential optimal: W^*_{seq}

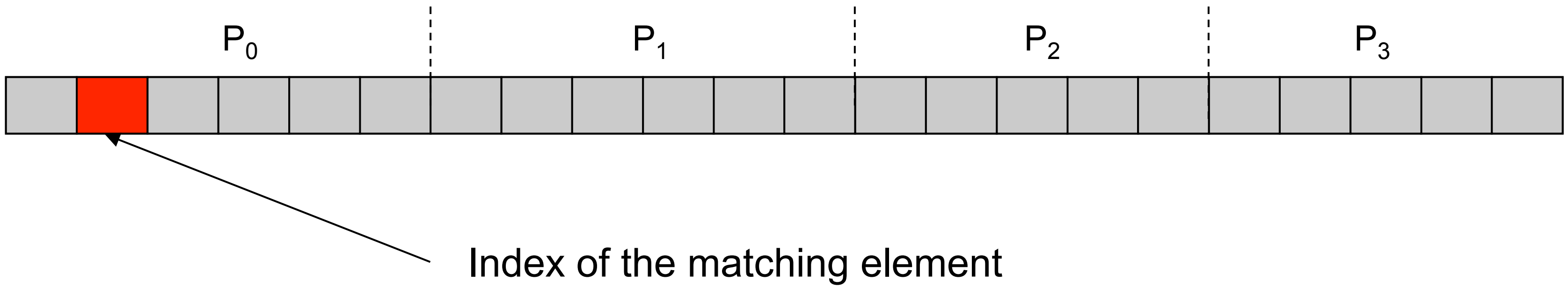
Parallel optimal: D^*_{par}
but $W^*_{par} = \alpha W^*_{seq}$

$$T_p \le \frac{\alpha . W^*_{seq}}{(p+\alpha-1)\,\Pi_{ave}} + O\left(\frac{D^*_{par}}{\Pi_{ave}}\right)$$

- **On-line recursive cascading sequential/parallel**
  - Provable performances, both theory and practice
  - *Convexity of speed-up provides sufficient conditions for optimality while the sequential process is active* (Amortized loop)

Amortizing Parallel Arithmetic overhead: example: find_if

- find_if : returns the index of the first element that verifies a predicate.

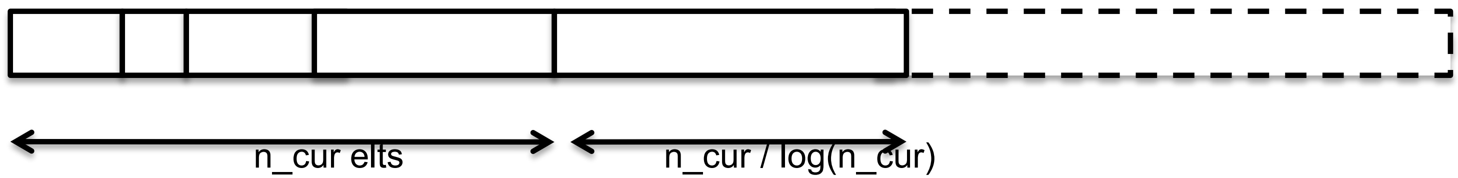


- Sequential time is $T_{seq} = 2$
- Parallel time= time of the last processor to complete: here, on 4 processors: $T_4 = 6$

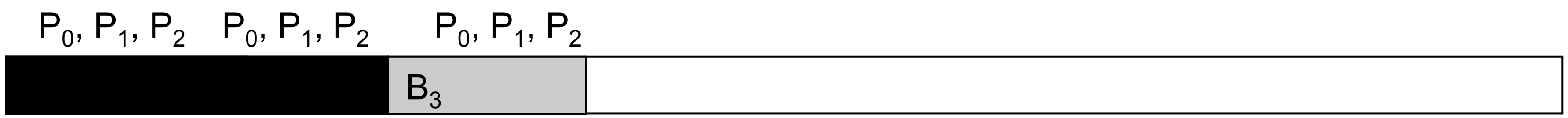
Amortizing Parallel Arithmetic overhead: example: find_if

- To adapt with provable performances ($W_{par} \sim W_{seq}$) : compute in parallel no more work thant the work performed by the sequential algorithm

Amortized scheme similar to Floyd's algorithm : **Macro-loop** [Danjean, Gillard, Guelton, R., Roche, PASCO'07]),



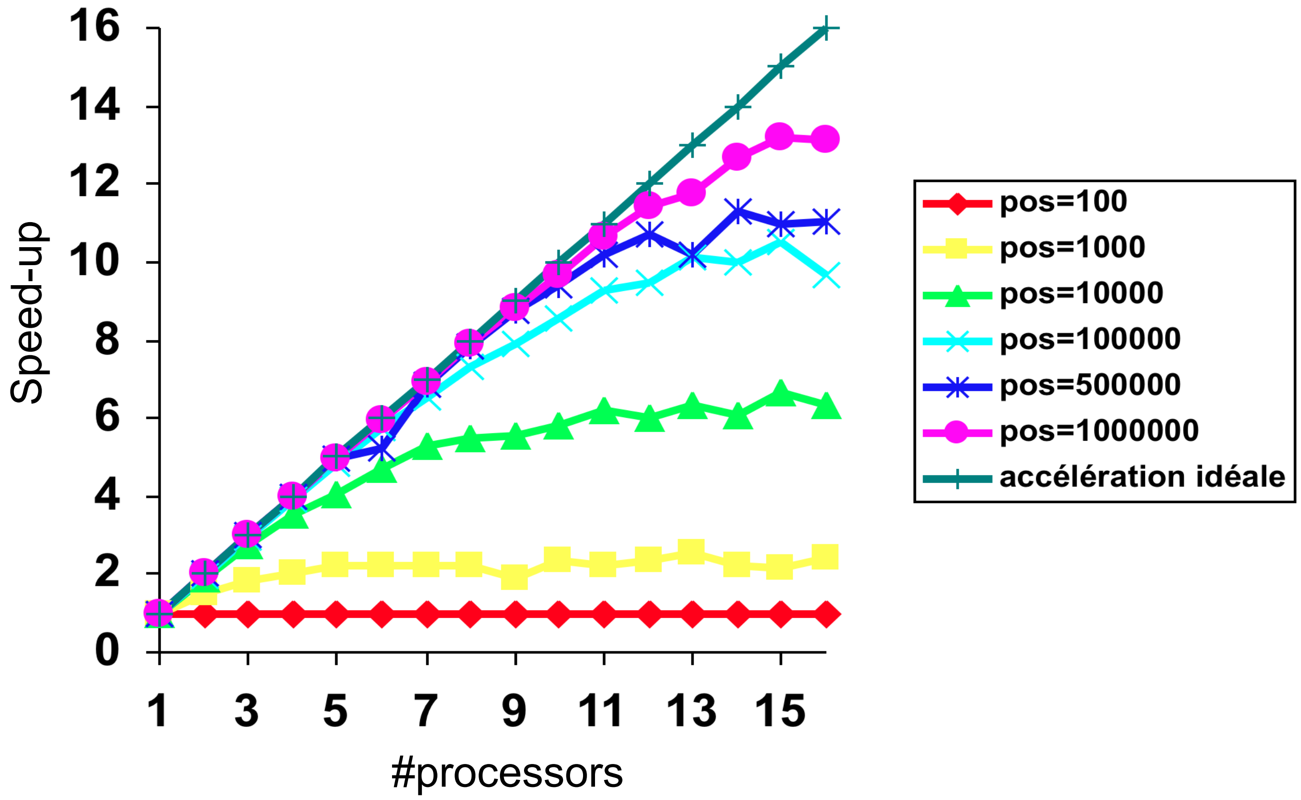
- Example : **find_if**



Amortizing Parallel Arithmetic overhead: example: find_if [Daouda Traore 2009]

- Example : find_if STL
  - Speed-up w.r.t. STL sequential tim and the position of the matching element.

Machine :
AMD Opteron (16 cœurs);
Data: doubles;
Size Array: 10^6;
Predicate time≈ 36µ



Parallelism induces overhead :
e.g. Parallel prefix on fixed architecture

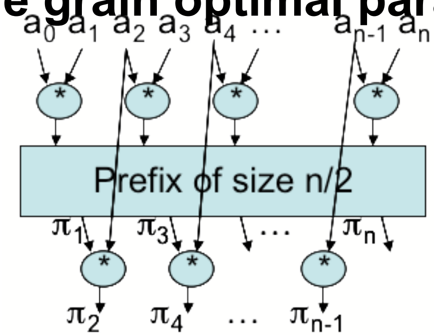
- **Prefix problem** :
  - input : $a_0, a_1, \ldots, a_n$
  - output : $\pi_1, \ldots, \pi_n$ with $$\pi_i = \prod_{k=0}^{i} a_k$$

- **Sequential** algorithm :
  - for (π[0] = a[0], i = 1 ; i <= n; i++) π[i] = π[i – 1] $*$ a [i] ;

*performs only **n** operations*

- **Fine grain optimal parallel** algorithm :

[Ladner-Fisher-81]



Critical time = 2. *log* n
but performs 2.n ops

Parallel requires twice more operations

$$\leq \frac{2n}{(p+1).\Pi_{ave}} + O\left(\frac{\log n}{\Pi_{ave}}\right)$$

- Tight lower bound on **p identical processors**:

[Nicolau&al. 1996]

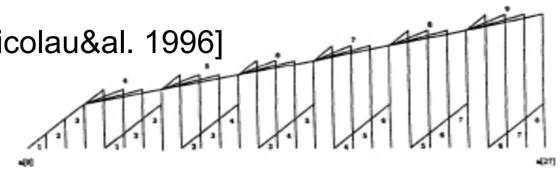


Figure 7: The Pipelined Schedule for p = 7.

Optimal time T_p = 2n / (p+1)
but performs 2.n.p/(p+1) ops

Conclusion

- Abstraction technology « system » :
  - Cache : LRU
  - CPUs : work-stealing
- Advanced algorithmic designs
  - « arithmetic » control flow from the coupling of the system and the algorithm
- New algorithmic analysis : multicriteria performance
  - Various applications: interactive
- Perspectives, open questions / work in progress ;
  - Both processor and cache – oblivious : limit
  - Distributed memory (communication)
  - On-line adaptive scheduling (number of resources, energy)

Questions