Oblivious parallel programming

Vincent Danjean, Bruno Raffin,
Jean-Louis Roch, Marc Tchiboukdjian

MOAIS team-project
Lab. d'Informatique de Grenoble

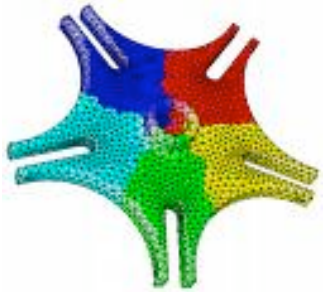
Louvre, Musée de l'Homme
Sculpture (Tête)
Artist : Anonyme
Origin: Rapa Nui [Easter Island]
Date : between the XIst and the XVth century
Dimensions : 1,70 m high

http://moais.imag.fr

INRIA INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
LIG
CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
GRENOBLE UNIVERSITÉS

Relation machine model / program

- One single (simple) computation for a given (simple) machine
  - Example: domain decomposition on the TERA computer :
    - « static parallelization» (MPI)



- But, concurrently several different (simple) computations
  - Example: multi-physic domains with adaptive irregular refinement

=> composition of parallel computations is difficult

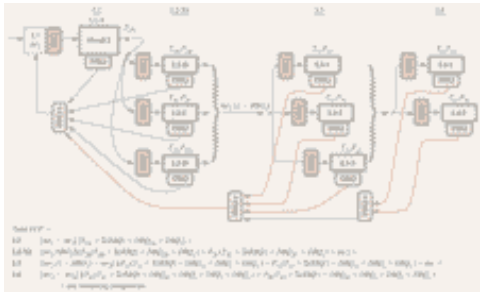
Need an abstract machine model

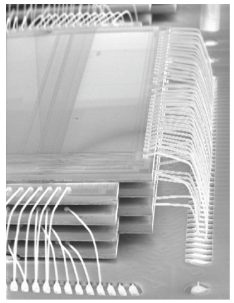
- To enable composition of parallel programs, by abstracting the resources at the programming level
  - Ideally: each computation performance should related to the effective allocated speed : Π_{tot}

$$\Pi_{tot} = p.\Pi_{ave} \qquad \Pi_{ave} = \frac{\sum_{t=1}^{T}\sum_{i=1}^{p}\Pi_i(t)}{T.p}$$

[Bender&al 2002]

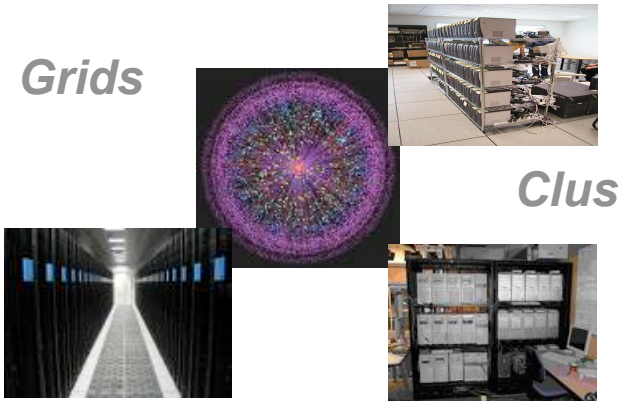
- And yet to fit next machine generation
  - Future MPSoC will have hundred of specialized units, with different frequencies (fixed but non predictable)
  - Memory hierarchy

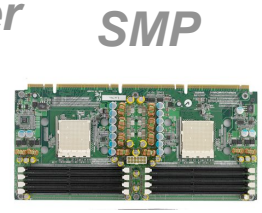


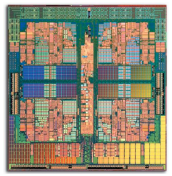


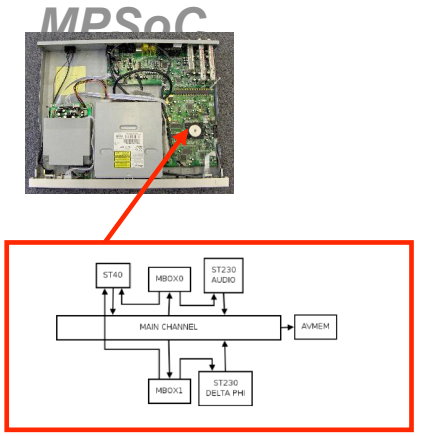
Evolution of parallel programming

- Parallelism everywhere
  - Distributed, Heterogeneous









MPI OpenMP

MapReduce [Google]

Cuda [NVidia]

TBB [Intel]

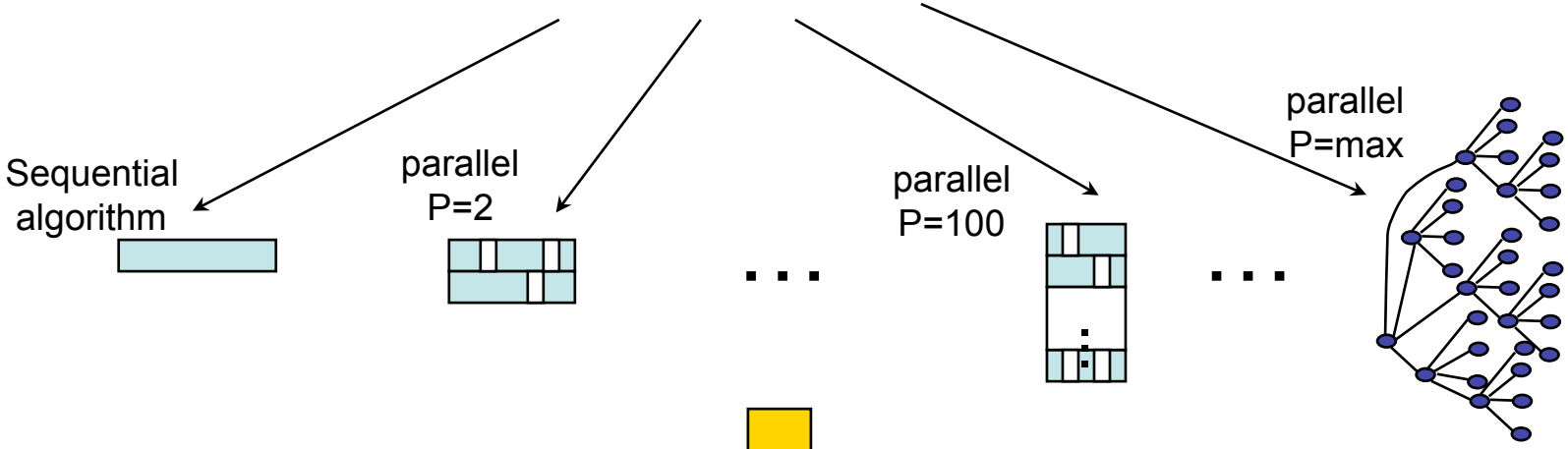
Cilk++ [CilkArts]

...*SPIRIT*

Fortress [Sun]

Towards oblivious algorithms

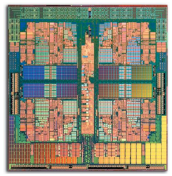
To design a single efficient algorithm
with provable performances on an arbitrary architecture



Which algorithm
to choose ?

?

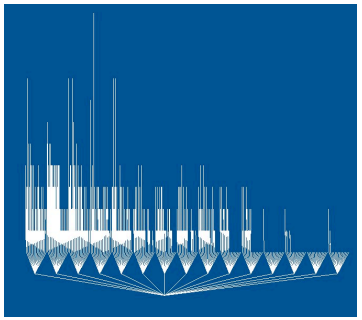


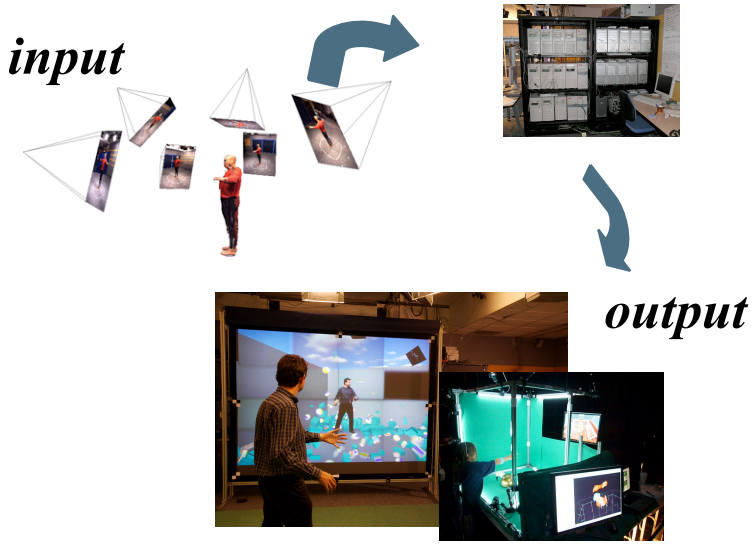

The MOAIS team-project

- Objective: End-to-end parallel programming solutions for high-performance interactive computing with provable performances.

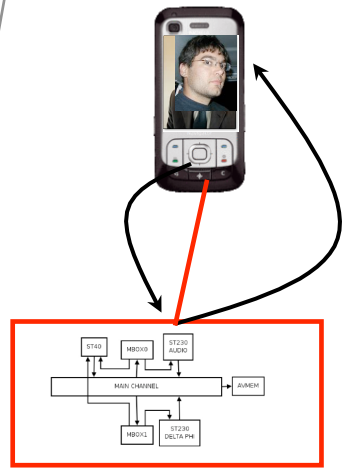
optimization **computational steering, VR** **embedded**



QAP/Nugent on Grid'5000
[PRISM, GSCOP, DOLPHIN]



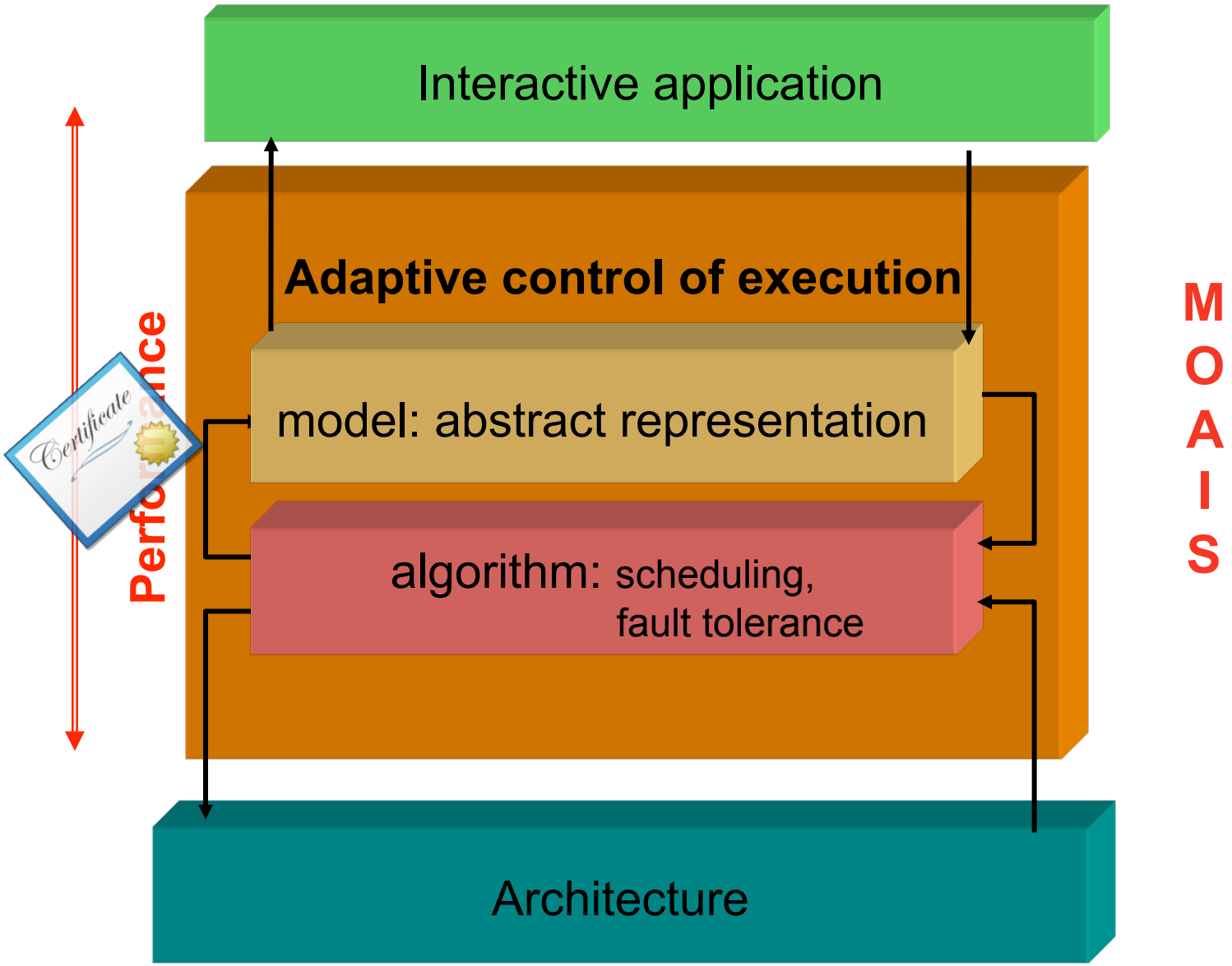
INRIA Grimage platform
[MOAIS, PERCEPTION, EVASION]



Streaming on MPSoCs
[ST]

- Performance is multi-objective
- Adaptive to the platform

Adaptation: from application to architecture



Outline

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

• Analysis on a given (abstract) architecture which proves optimality :
behaves as well as an optimal (off-line, non-oblivious) algorithm

•**Talk: Basic techniques to design oblivious algorithms**

- 1. Introduction - Motivation for obliviousness
- 2. Processor oblivious
- 3. Cache oblivious
- 4. Conclusion - Towards cache and processor oblivious

- 1. Introduction - Motivation for obliviousness
- **2. Processor oblivious**
- 3. Cache oblivious
- 4. Towards cache and processor oblivious

Example: Parallel prefix

- **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
(and minimal cache misses)

- Optimal parallelization on **p identical processors**:

Optimal time T_p = 2n / (p+1)

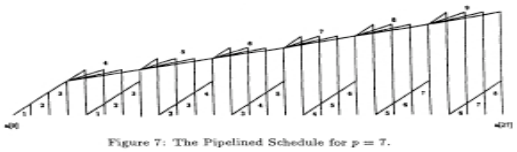
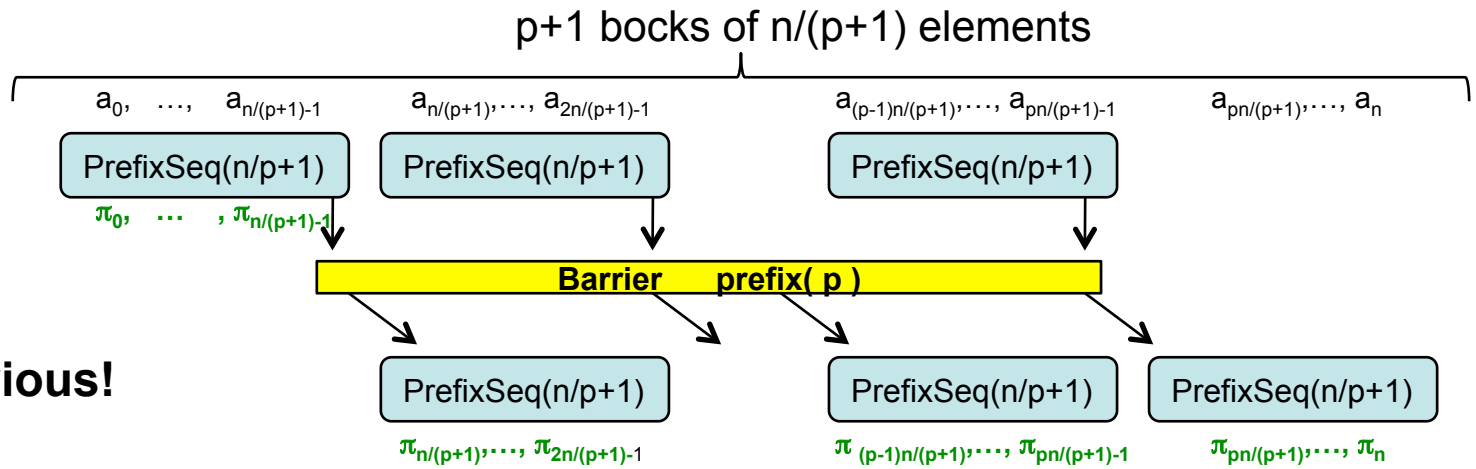
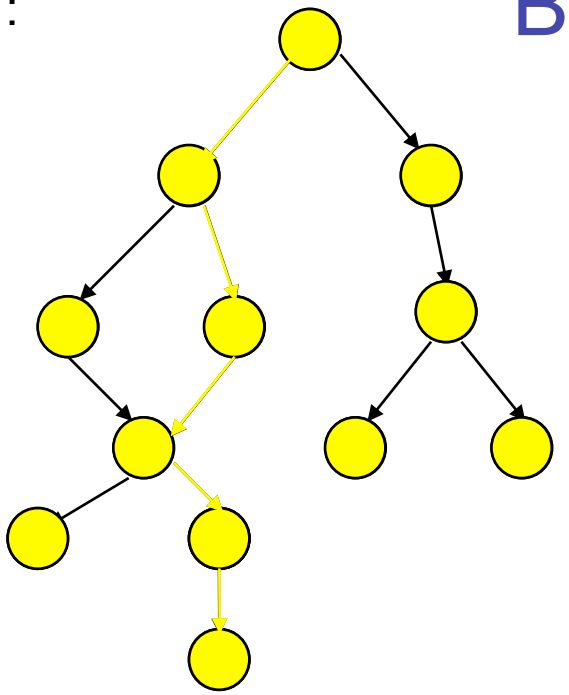


Figure 7: The Pipelined Schedule for $p = 7$.



- **Non oblivious!**

:



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)

For any greedy *maximum utilization* schedule Graham69, Brent70, Jaffe80, Bender-Rabin02]

$$\frac{1}{\Pi_{ave}} Max\left(\frac{W}{p};D\right) \leq Time(p,\Pi) \leq \frac{1}{\Pi_{ave}}\left(\frac{W}{p}+D\right)$$

- **There exist schedulers that reach the bound :** $\frac{1}{\Pi_{ave}}\left(\frac{W}{p}+O(D)\right)$

Work-Stealing: a basis to design processor-oblivious algorithm

- **Work-stealing = oblivious schedulers** that reach: $\frac{1}{\Pi_{ave}}\left(\frac{W}{p}+O(D)\right)$

  "A decentralized thread scheduler: whenever a processor runs out of work, it steals work from a randomly chose processor."

Moreover: if D small, few steals request [O(p.D) w.h.p.]

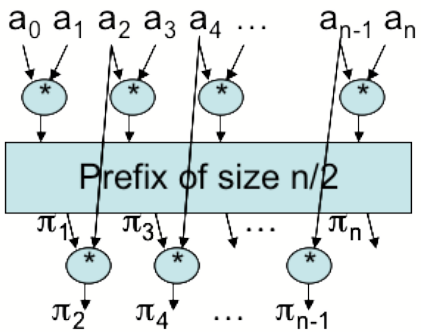
- *Then, if both the work **W >> D** **(i.e. D very small)** work-stealing ensures **provable performances,** both **theoretical** and **practical** [Cilk, TBB, Kaapi, …]*
  - *And if W ~ W_{seq} : optimal processor oblivious performance*

Application to parallel prefix

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

- **Oblivious parallel** algorithm : recursive to minimize the depth D

[Ladner-Fisher-81]



Depth D = 2. *log* n
but performs W = **2.n ops**

- Non oblivious tight lower bound on **p identical processors**:

[Nicolau&al. 1996]

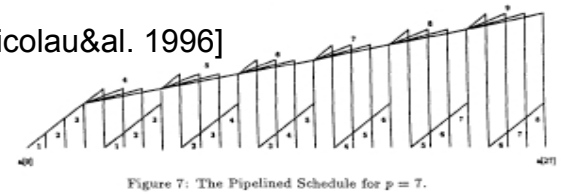


Figure 7: The Pipelined Schedule for $p = 7$.

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

Oblivious

but

non optimal

Adaptive scheme to simultaneously minimize D and W

① To minimize depth D

– By enabling, at each steal, extraction of a fraction of the remaining work on the victim => D = O (log W)
  - => only O(log W) steals per proc
  - small scheduling overhead

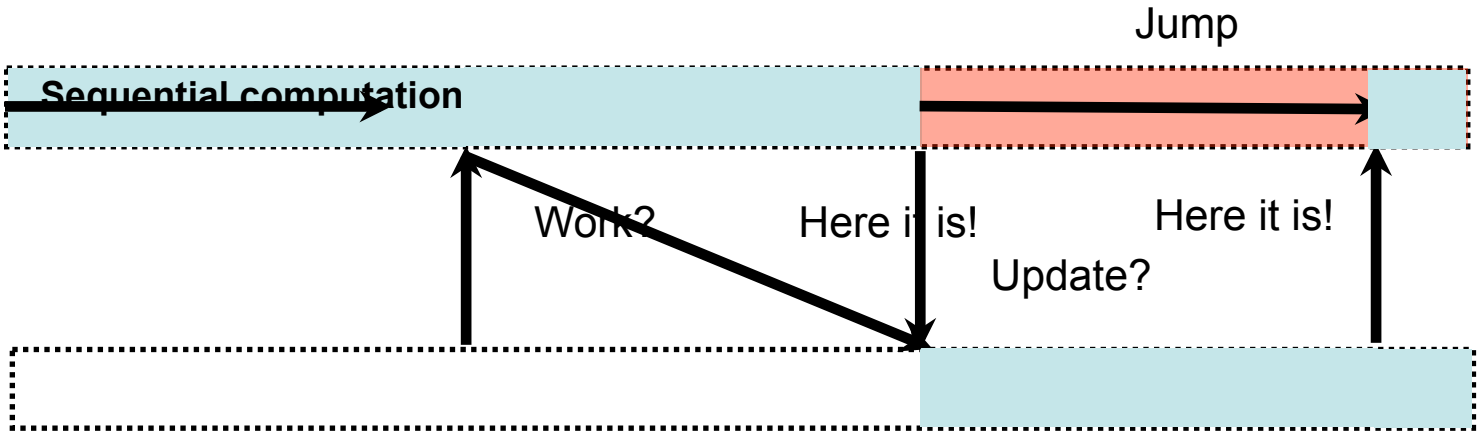
② To minimize work W => "*work first*" principle

Optimize the sequential local execution, that mostly occurs

– the overhead of a stack can be avoided [Roch&al 08]
  - contention between the local processor and its -potential- stealers can be made neglected

Adaptive parallel extraction

- Each resource performs a specialized sequential algorithm
  - other resources act as co-processors
  - at any time, sequential computation is in progress



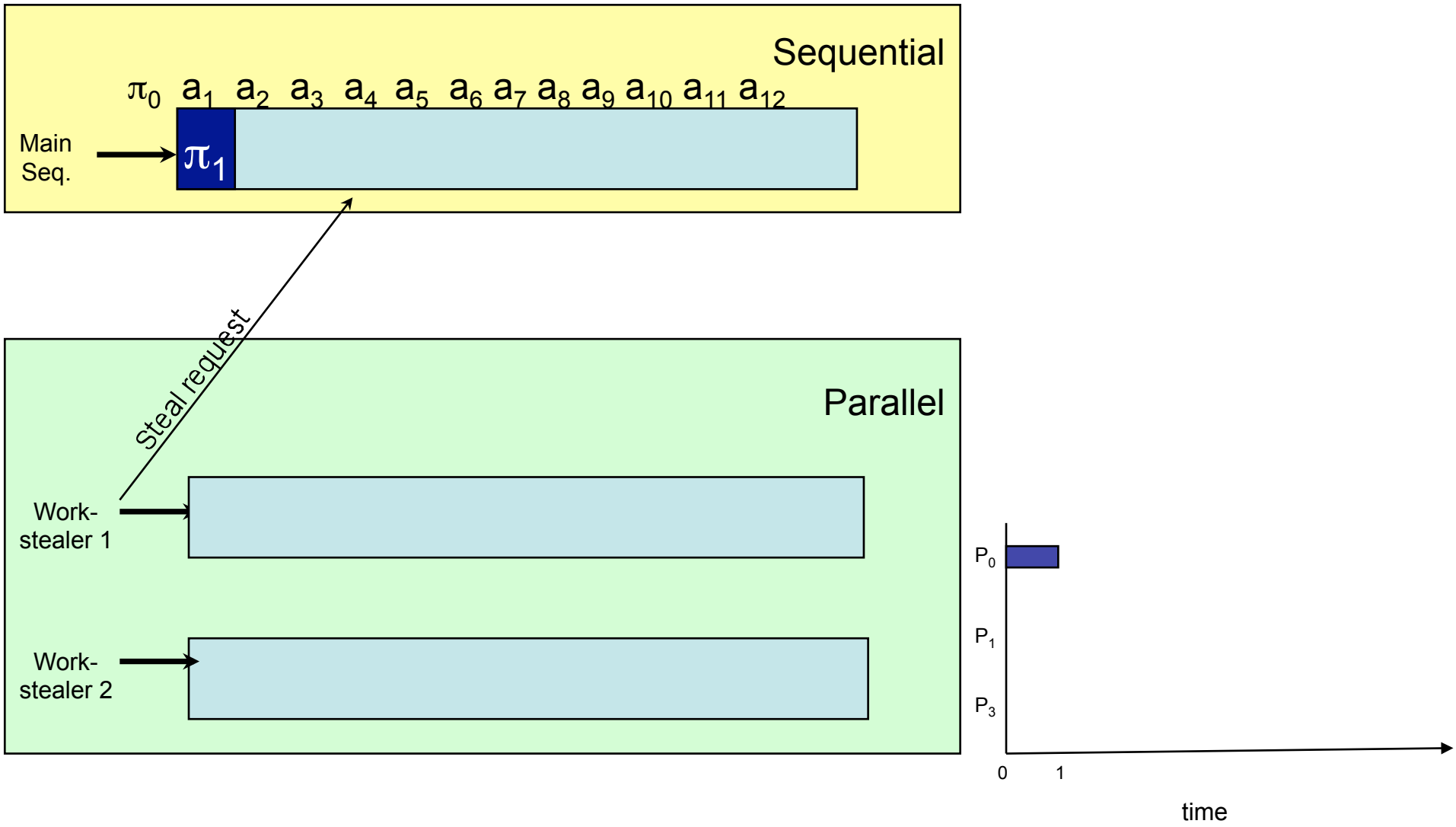
- When a stealer appears, it extracts some work (steal).
- When the work is completed, partial results are merged (eventually, preemption of the stealer).

Processor oblivious design

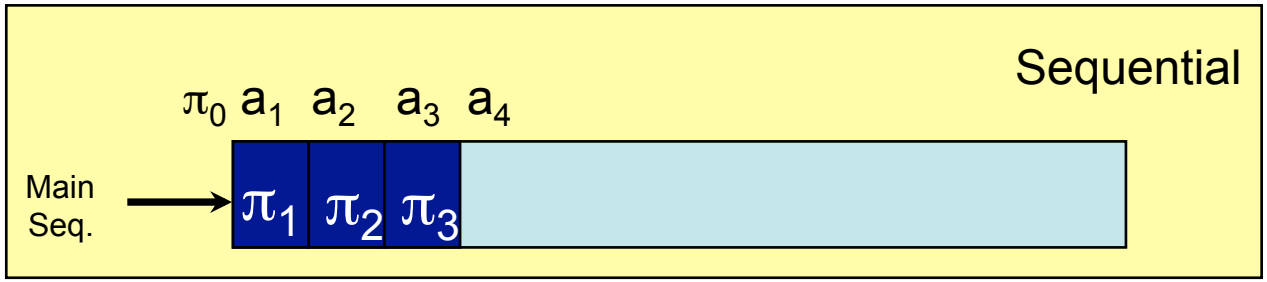
- implement the best (preemptive) sequential code
- add the parallelism extraction and merge
- use a work-stealing engine for the coupling
  - Performance guarantee by a three nested loops scheme
    - Local sequential computation always active [Danjean&al07]

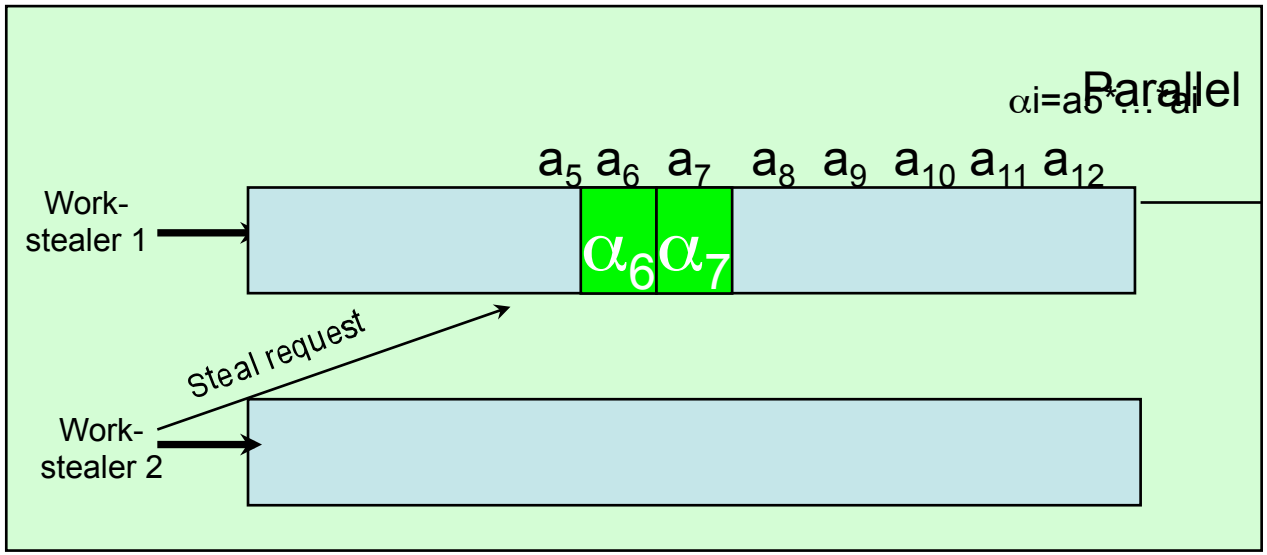
- Extraction/merge to define for each new program
- Provable performances related to those functions

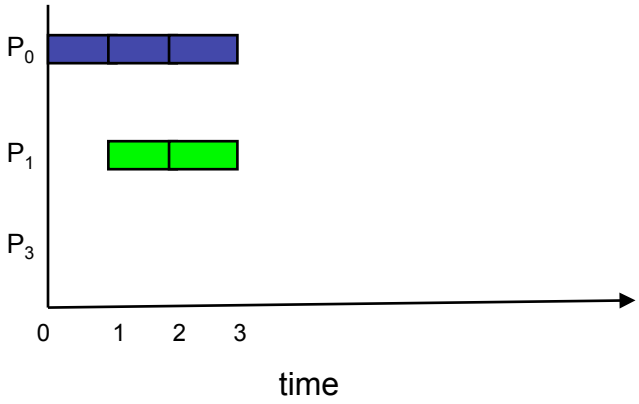
P-Oblivious Prefix on 3 proc.



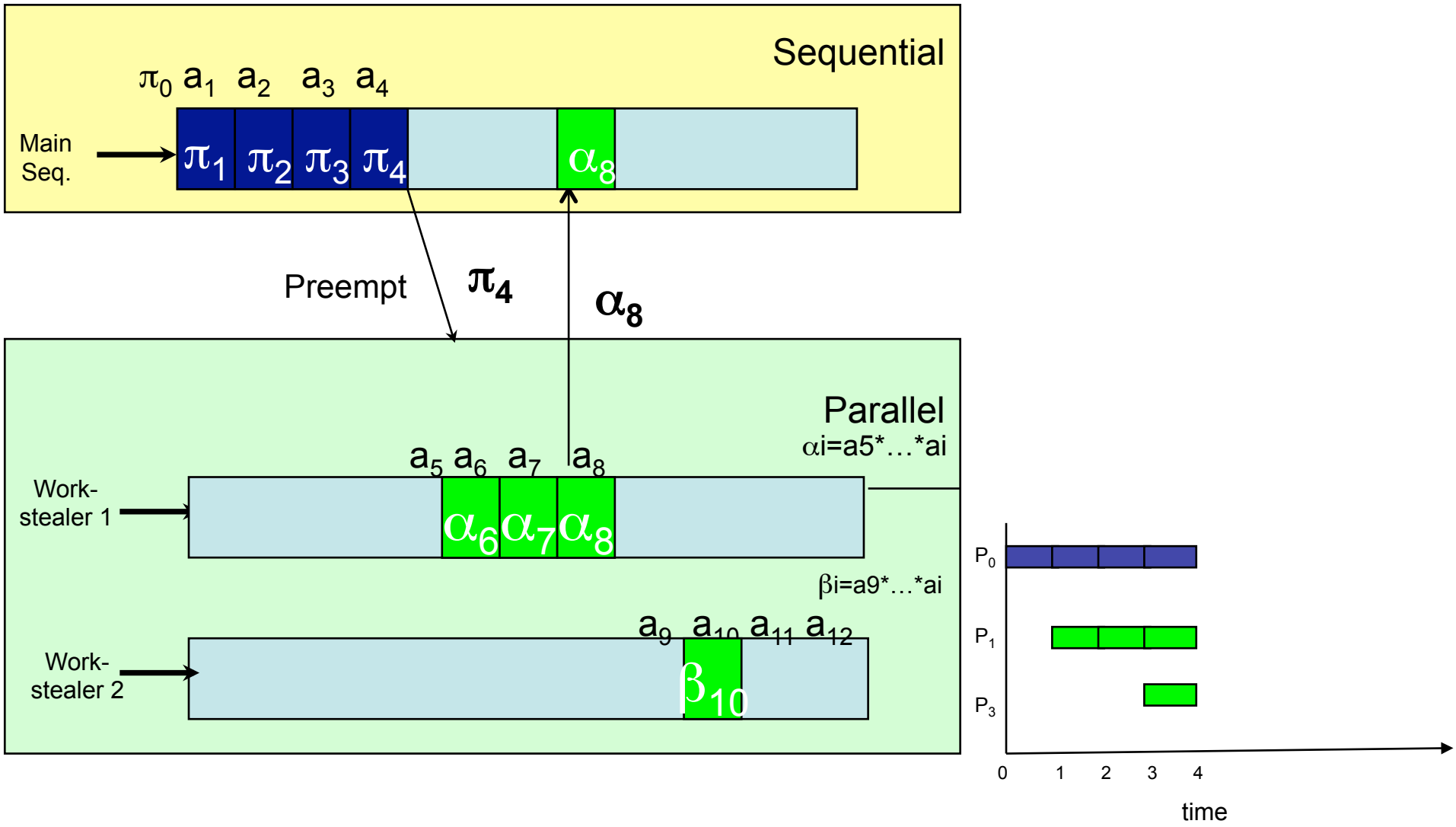
P-Oblivious Prefix on 3 proc.



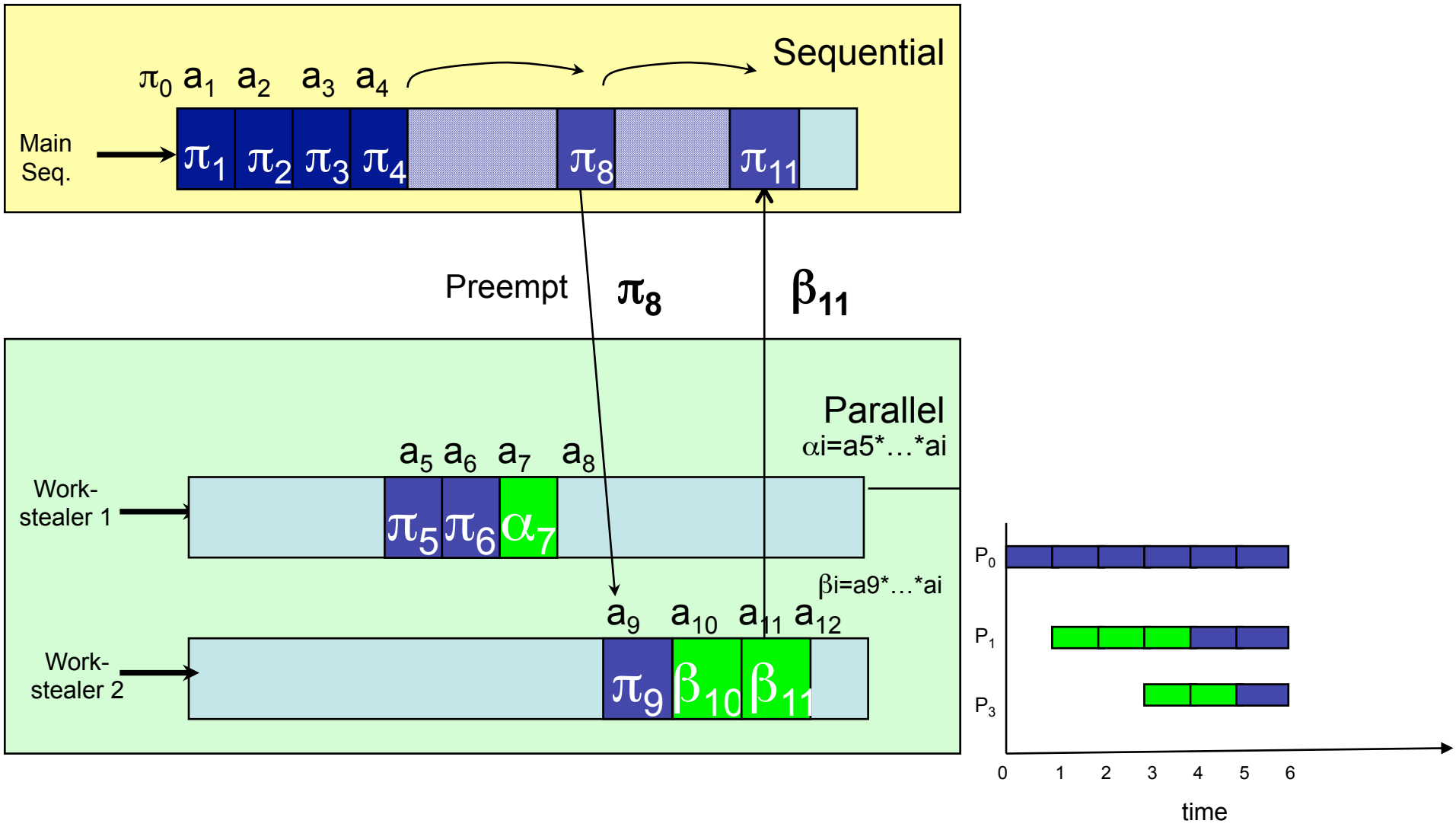




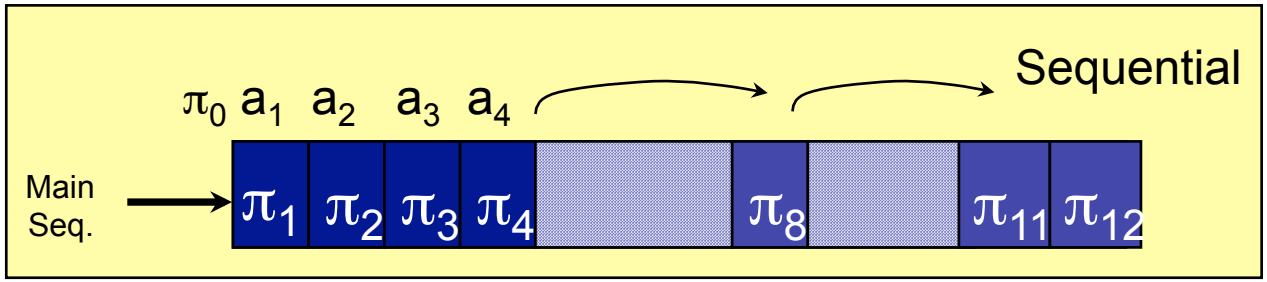
P-Oblivious Prefix on 3 proc.

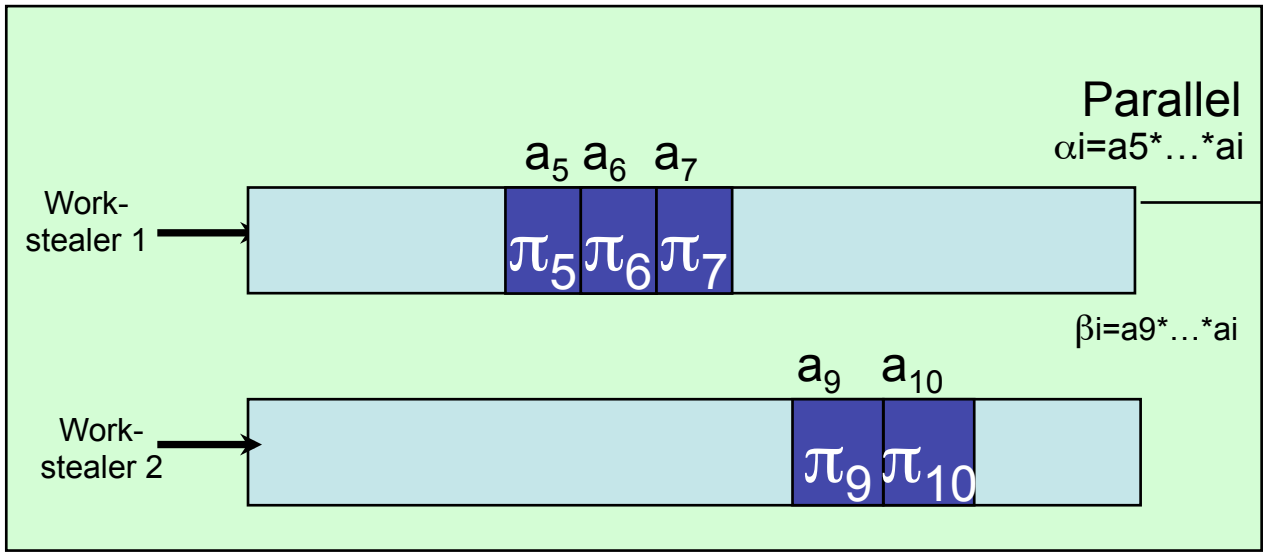


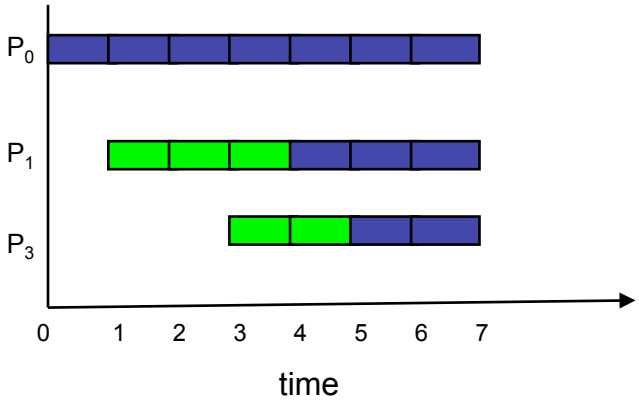
P-Oblivious Prefix on 3 proc.



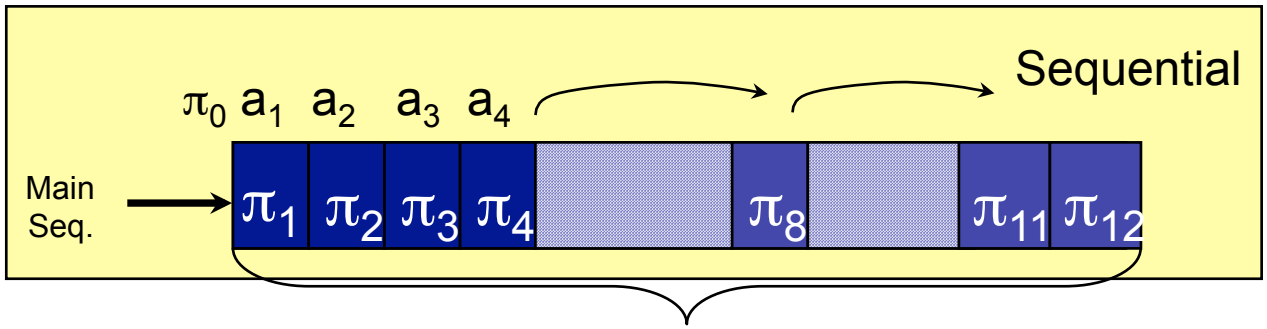
P-Oblivious Prefix on 3 proc.





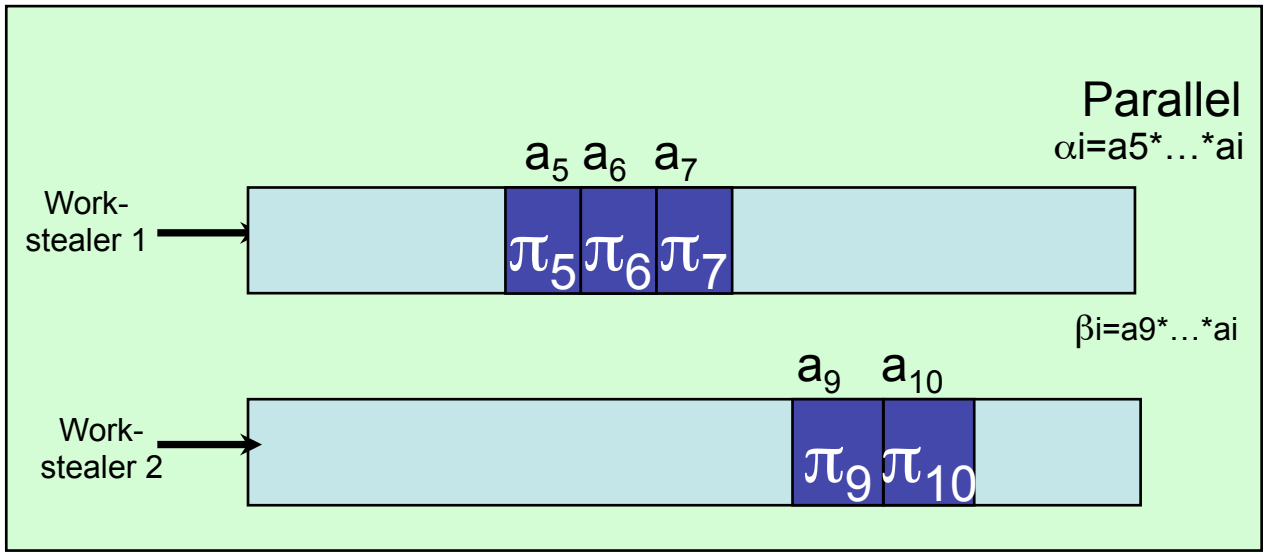


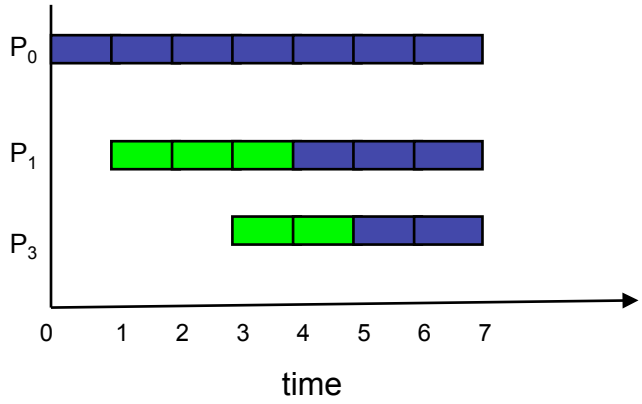
P-Oblivious Prefix on 3 proc.



Implicit critical path on the sequential process

$T_p = 7$ $T_p^* = 6$



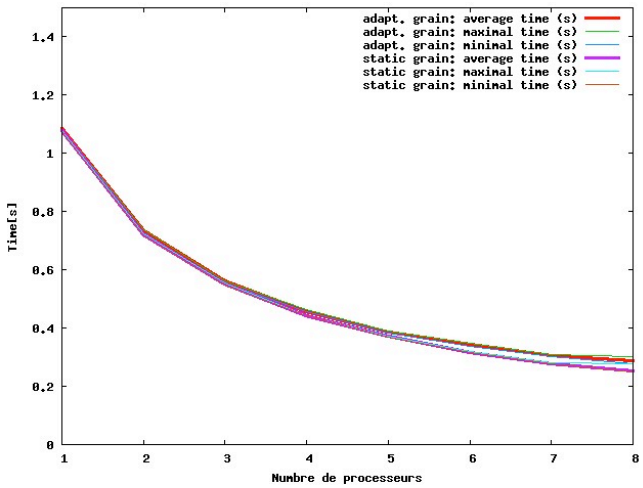


Provable perfomance of the P-obivious parallel prefix

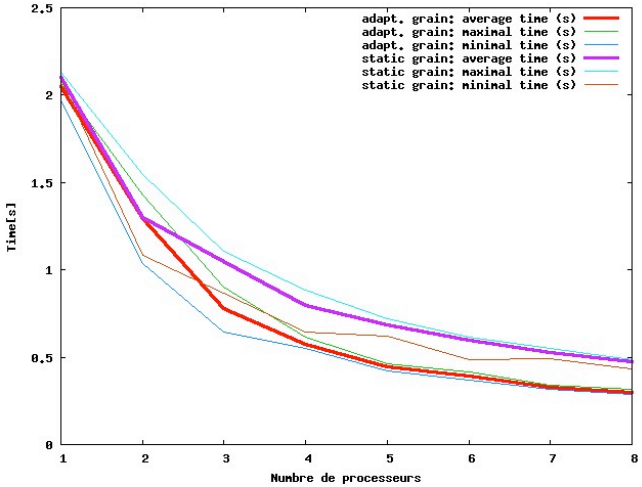
Execution time $\leq \frac{2n}{(p+1).\Pi_{ave}} + O\left(\frac{\log^2 n}{\Pi_{ave}}\right)$

Lower bound

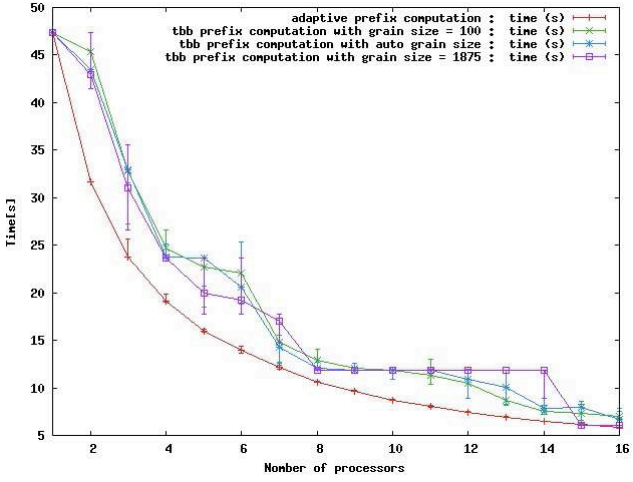
Practical performance [* = double, finest “grain” = 2048 double]



Single user



Processors with small variations



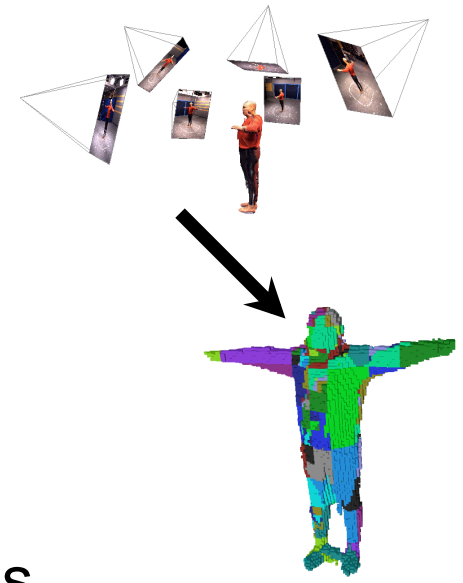
Comparison with TBB

P-Oblivious + good cache locality from the sequential algorithm

Some instances at Moais

- Parallel Prefix [Traore&al]
  - Processor oblivious, reaches the lower bound:
    $2n/(p+1)\Pi_{ave}$ $[+O(p \log^2 n)]$

- Oct-tree computation [Raffin&al]
  - Linear speed-up on CPUs + GPUs
  - Adaptation to realtime constraint



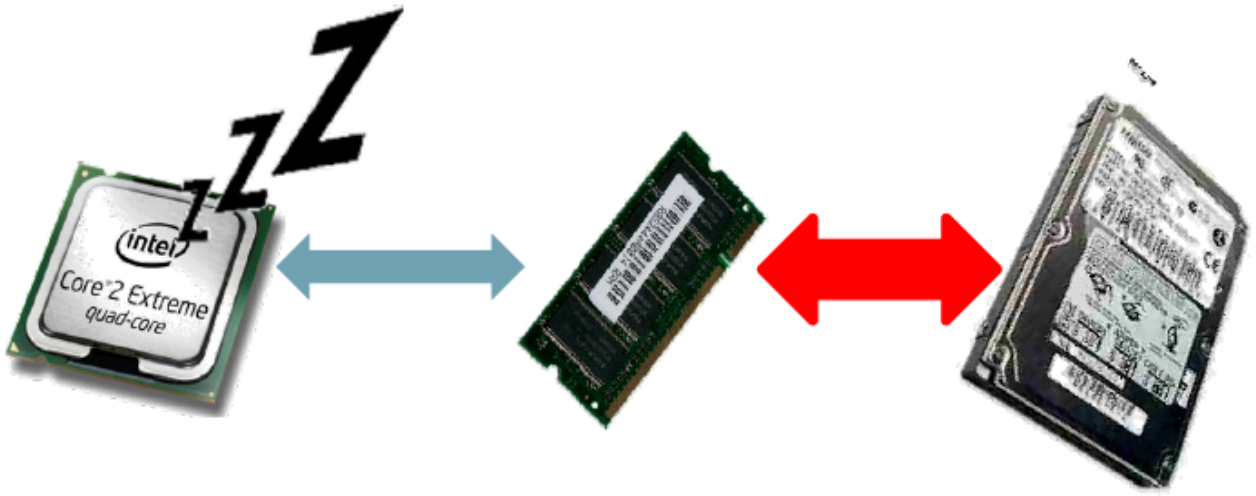
- Stream computations [Bernard&al]
  - compression, noise filtering on MPSoCs

- STL, Merging and sorting [Traore&al]

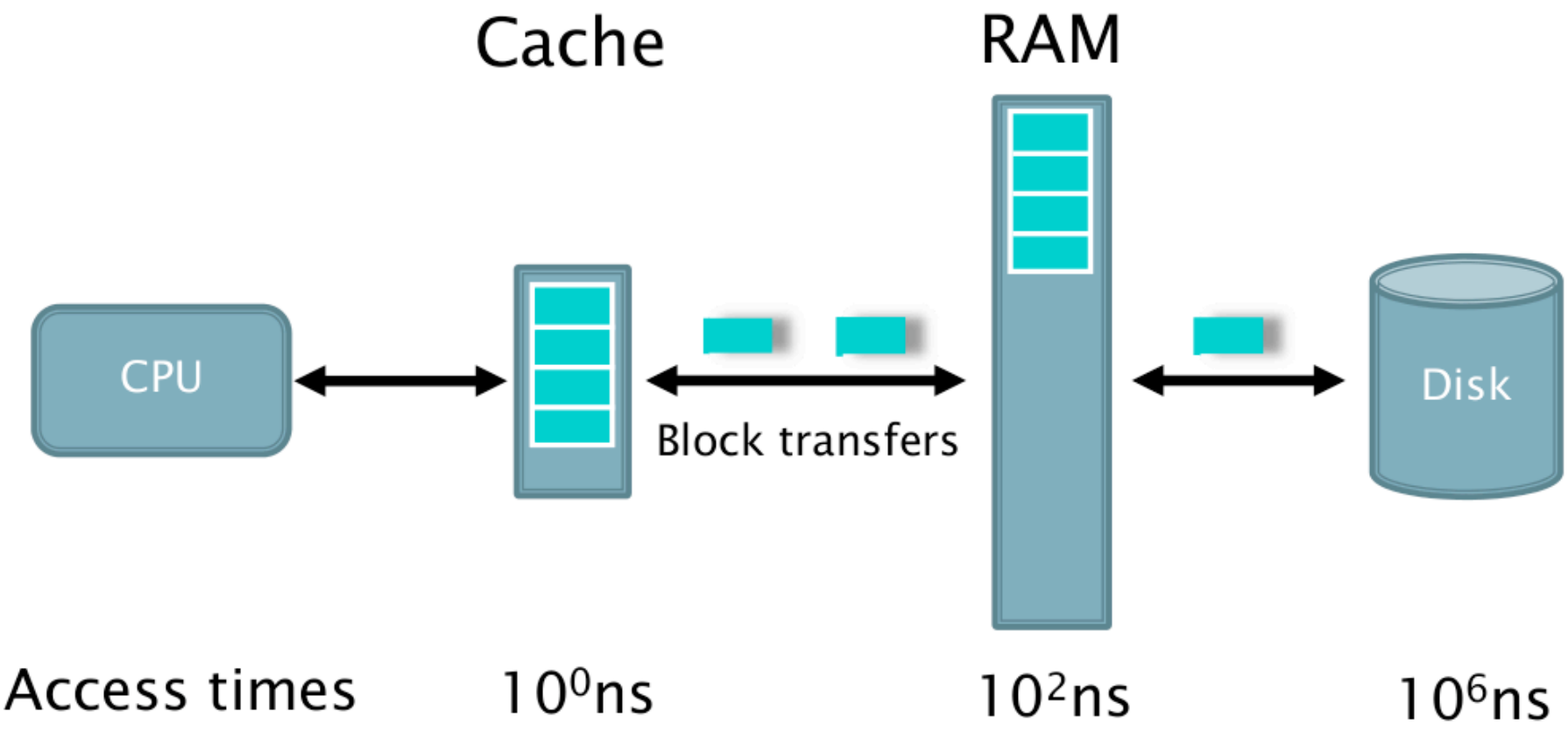
- 1. Introduction - Motivation for obliviousness
- 2. Processor oblivious
- **3. Cache oblivious**
- 4. Towards cache and processor oblivious mesh partitioning

Why are we interested in cache performance ?

- CPU-bounded vs I/O-bounded



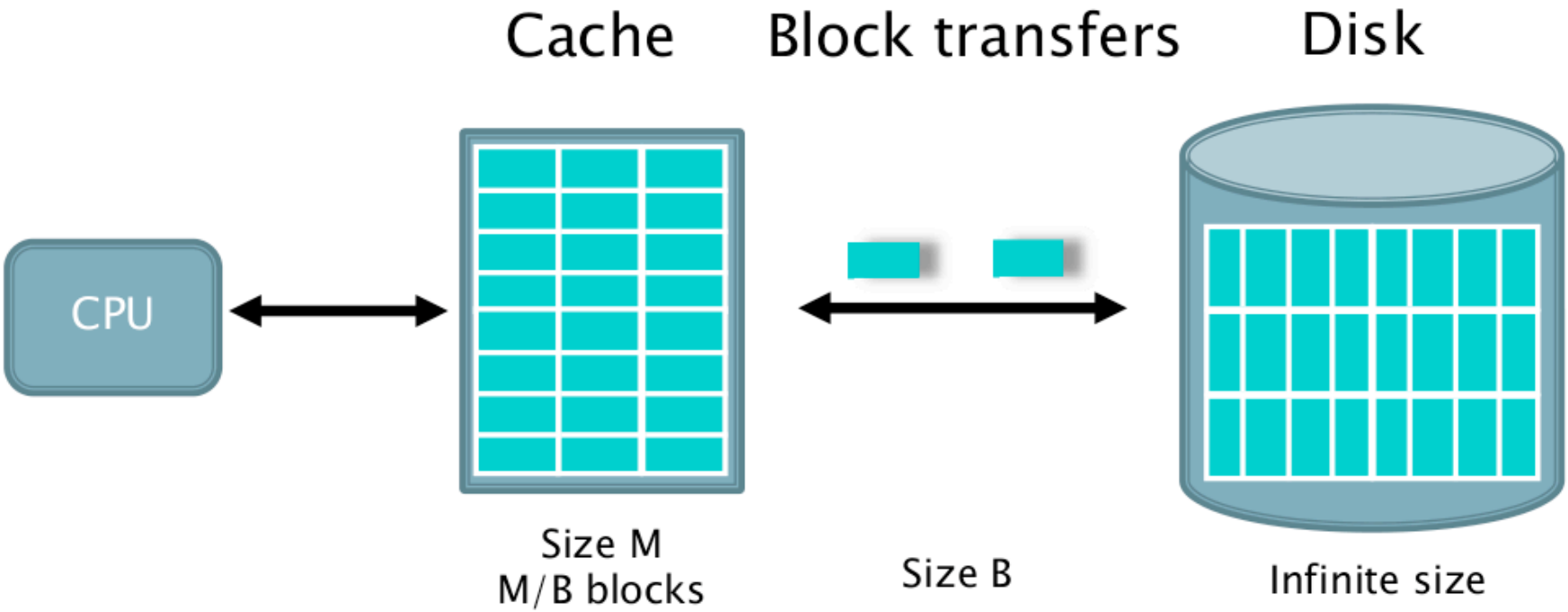
Memory Hierarchy



Cache-aware model (CA)

[Aggarwal & Vitter 1988]

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



W: #operations

Q: #block transfers

Multiplying in the CA model

NxN matrices in row-major order : naive doesn't work

Using the naive N^3 algorithm:

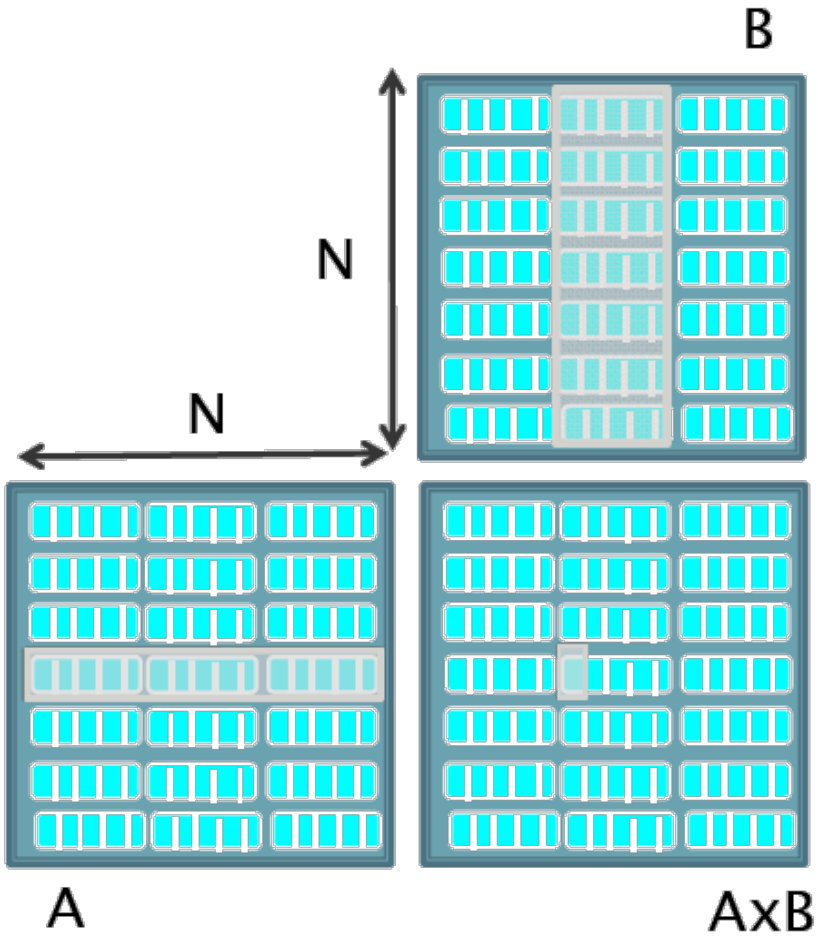
$$W(N) = O(N).N^2$$

$$W(N) = O(N^3)$$

Memory accesses in B are suboptimal:

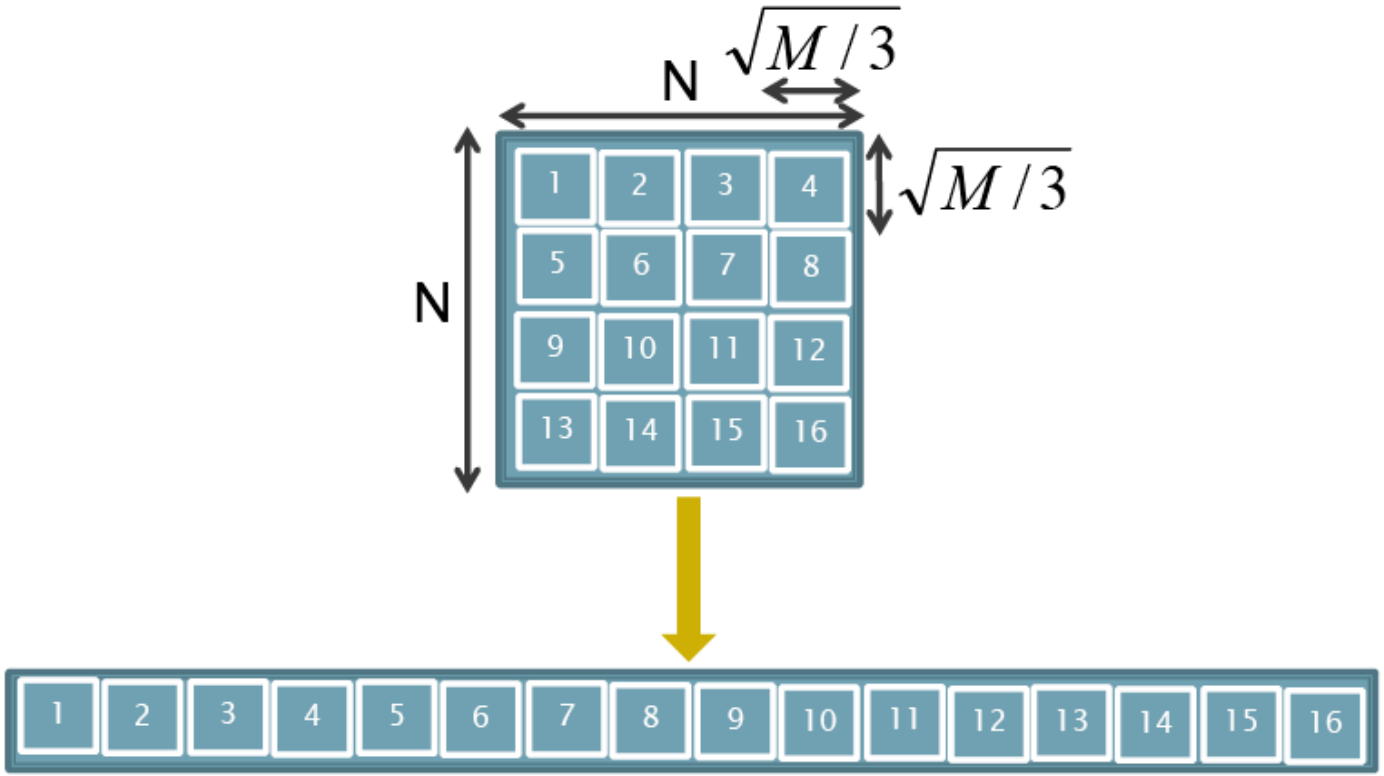
$$Q(N) = O\left(\frac{N}{B} + N\right).N^2$$

$$Q(N) = O(N^3)$$



Multiplying in the CA model

NxN matrices in submatrices



Multiplying in the CA model

NxN matrices in submatrices

- Cost for two sub-matrices

$$W(N) = O\left(\sqrt{M}^3\right) \quad Q(N) = O\left(\frac{M}{B}\right)$$

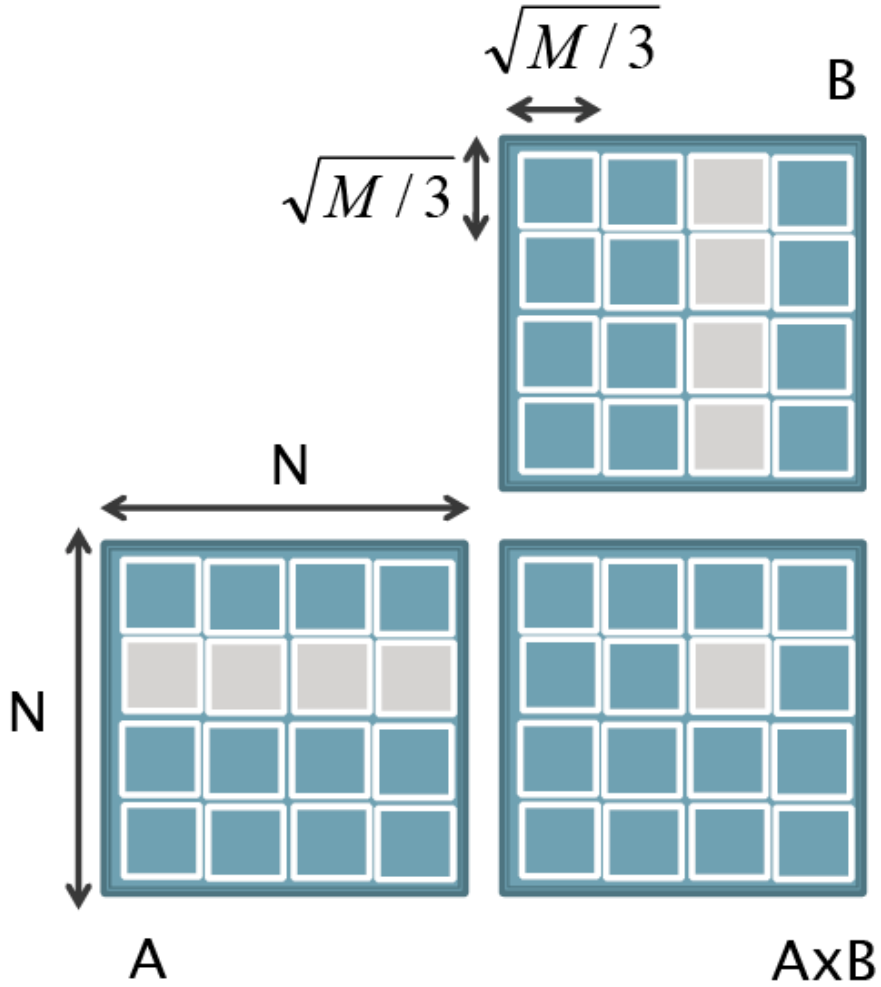
- Total cost

$$W(N) = O\left(\sqrt{M}^3\right).O\left(\frac{N}{\sqrt{M}}\right).O\left(\frac{N^2}{M}\right)$$

$$W(N) = O\left(N^3\right)$$

$$Q(N) = O\left(\frac{M}{B}\right).O\left(\frac{N}{\sqrt{M}}\right).O\left(\frac{N^2}{M}\right)$$

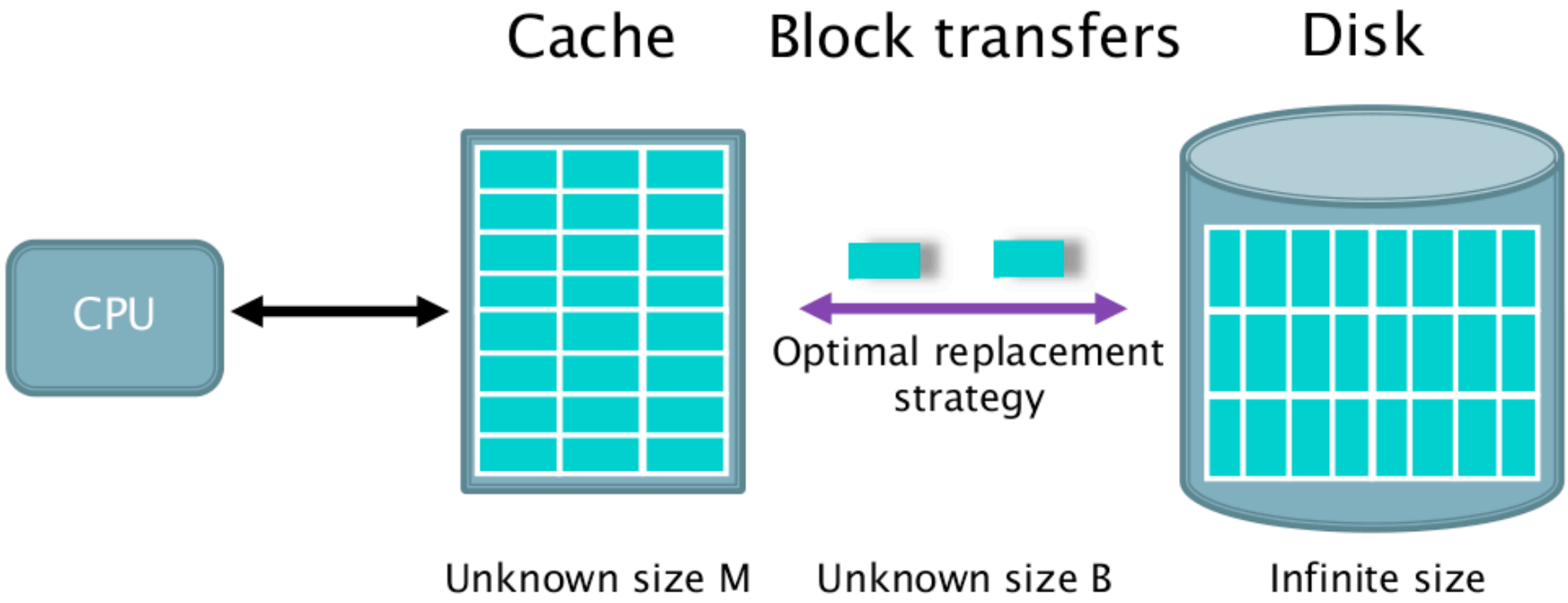
$$Q(N) = O\left(\frac{N^3}{B\sqrt{M}}\right)$$



CA

- Only two levels of the memory hierarchy
- Fixed values of B and M
- Machine-dependent

Cache-oblivious model (CO) [Frigo & al 1999]



CA vs CO

– Only two levels of the memory hierarchy

– Fixed values of B and M

– Machine-dependent

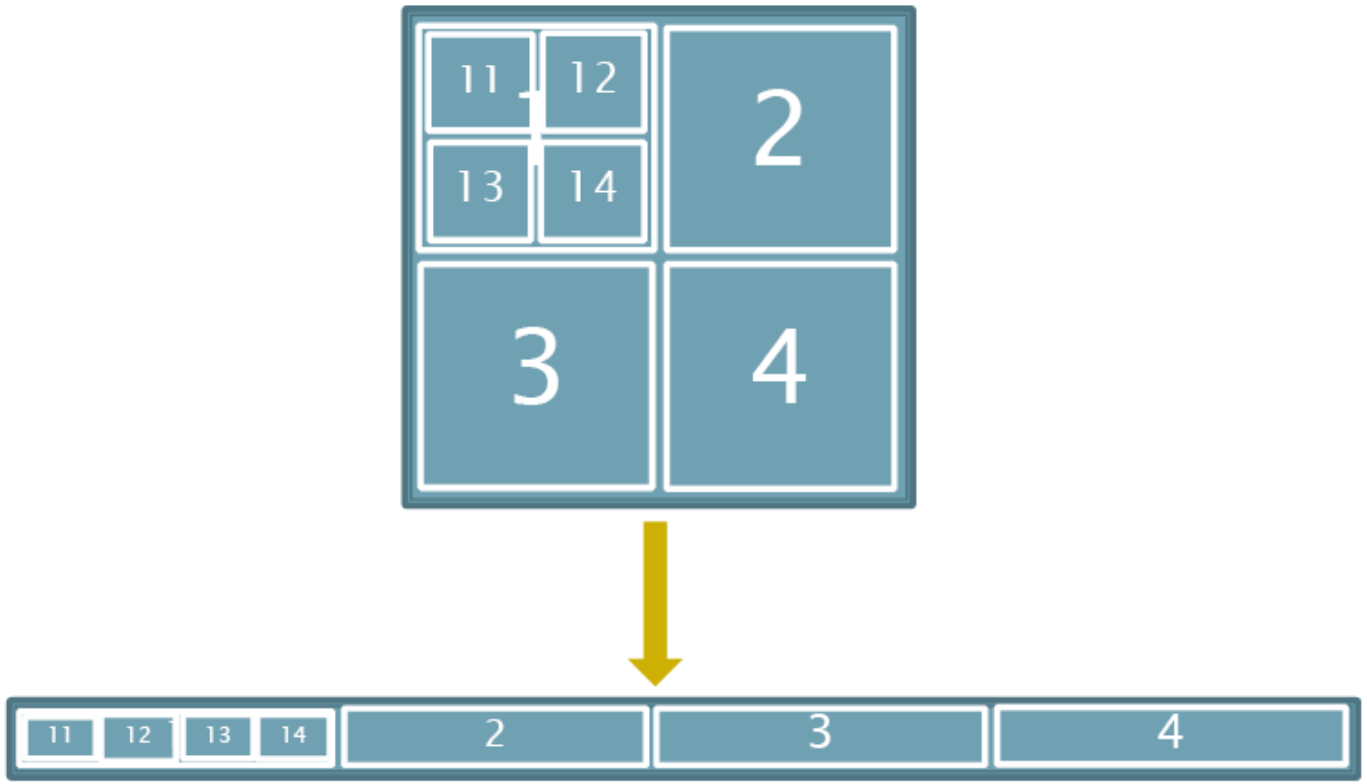
+ Efficient with all levels of the memory hierarchy

+ Adapt to varying values of B & M
  - multi-process scheduling
  - disk seek time

+ Machine-independent

Multiplying in the CO model

D&C matrix multiplication using a recursive layout



Multiplying in the CO model

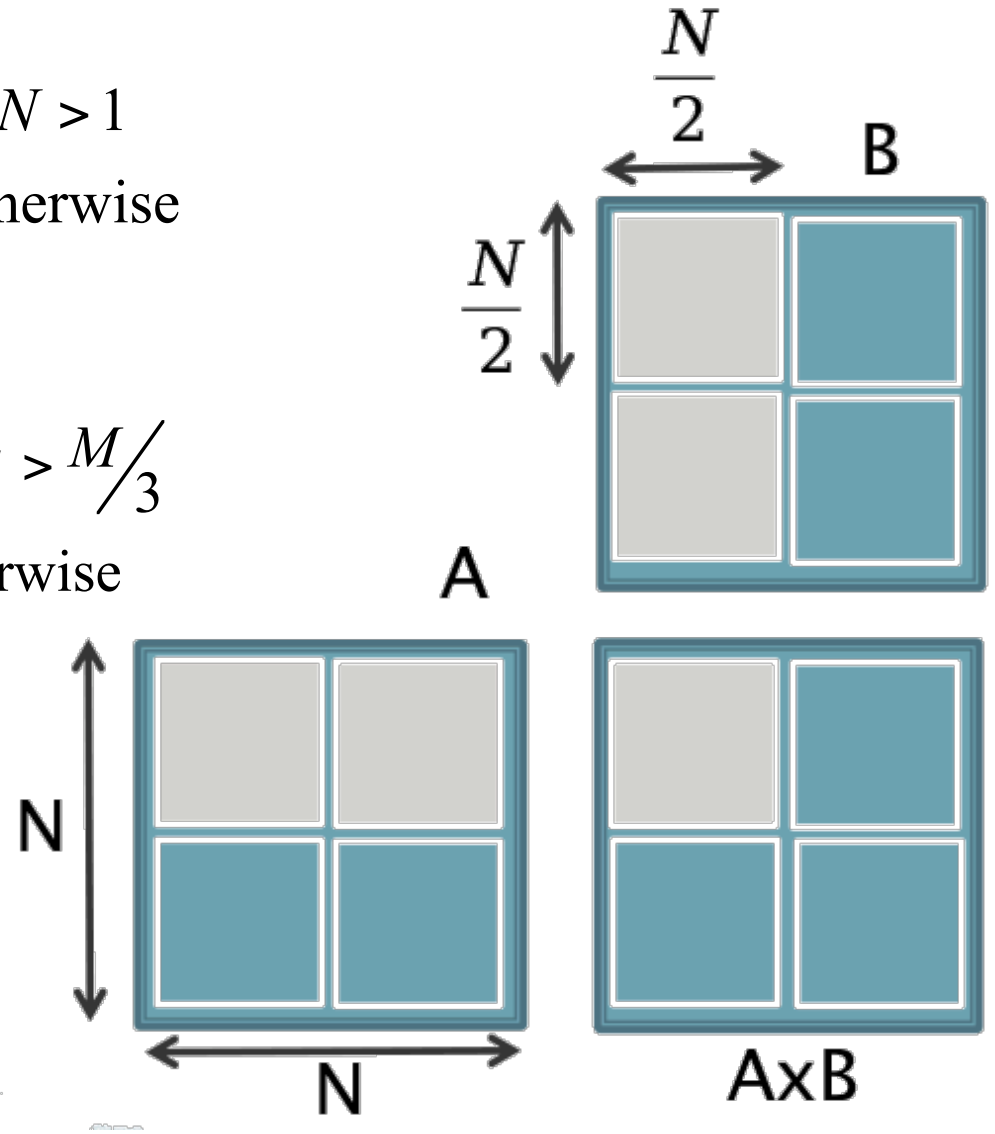
D&C matrix multiplication using a recursive layout

$$W(N) = \begin{cases} 8W\left(N/2\right) + O\left(N^2\right) & \text{if } N > 1 \\ O(1) & \text{otherwise} \end{cases}$$

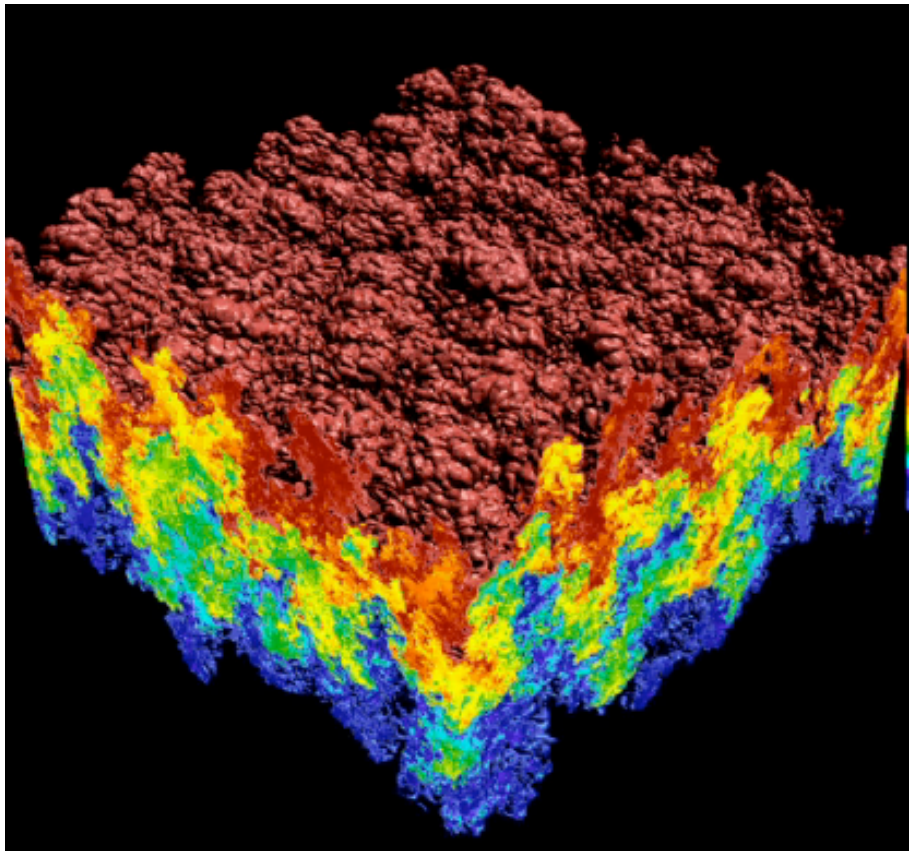
$$W(N) = O\left(N^3\right)$$

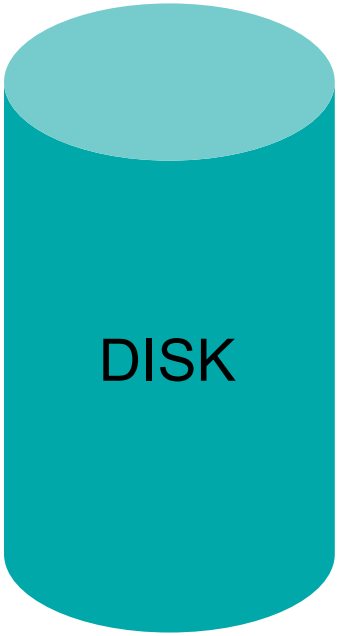
$$Q(N) = \begin{cases} 8Q\left(N/2\right) + O\left(N^2/B\right) & \text{if } N^2 > M/3 \\ O\left(N^2/B\right) & \text{otherwise} \end{cases}$$

$$Q(N) = O\left(N^3/B\sqrt{M}\right)$$



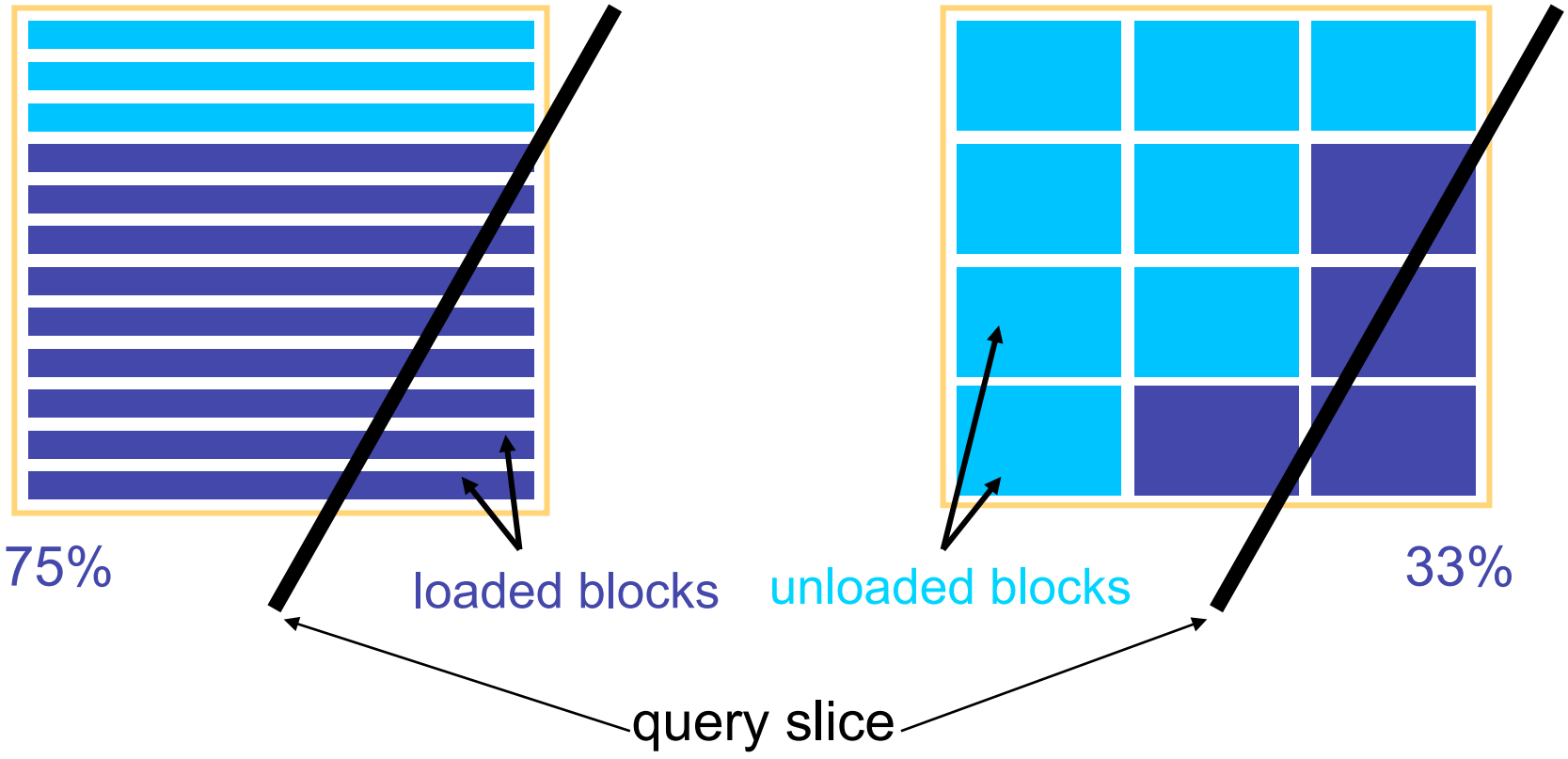
How to store efficiently a mesh ?



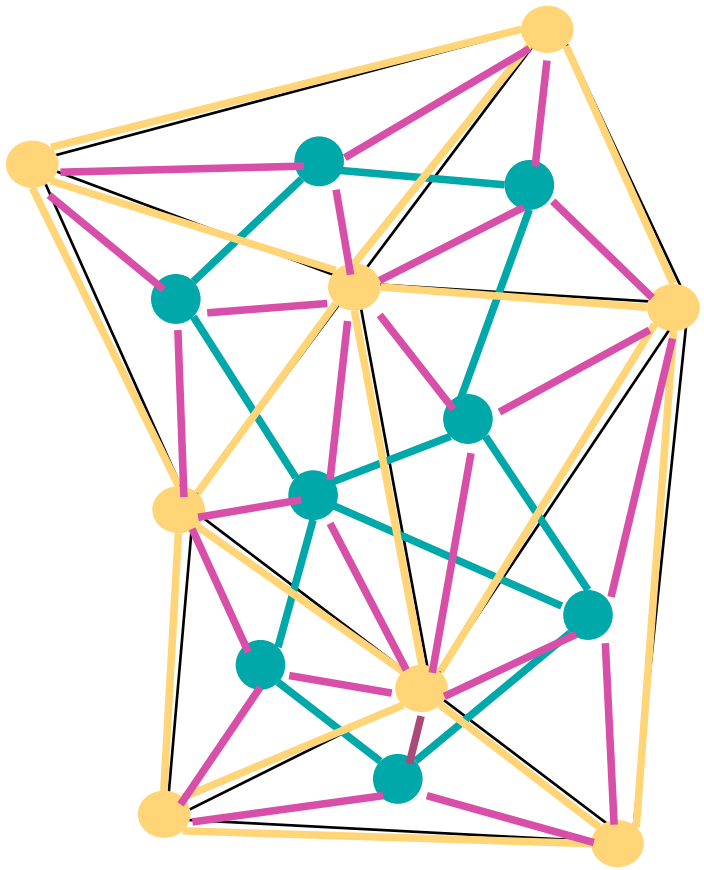


Idea

- Triangles (or vertices) that are most likely to be accessed sequentially should be stored nearby



Graph of sequential accesses



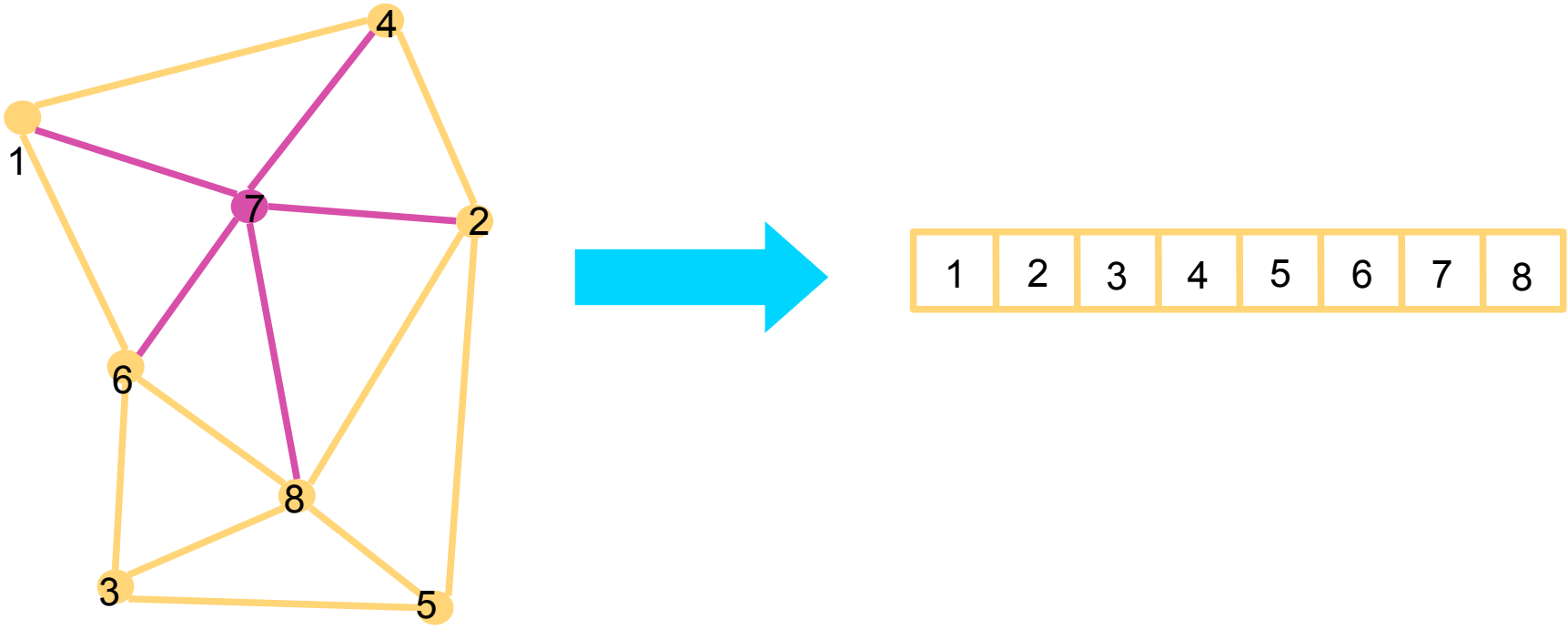
G_1: sequential access between triangles

G_2: sequential access between vertices

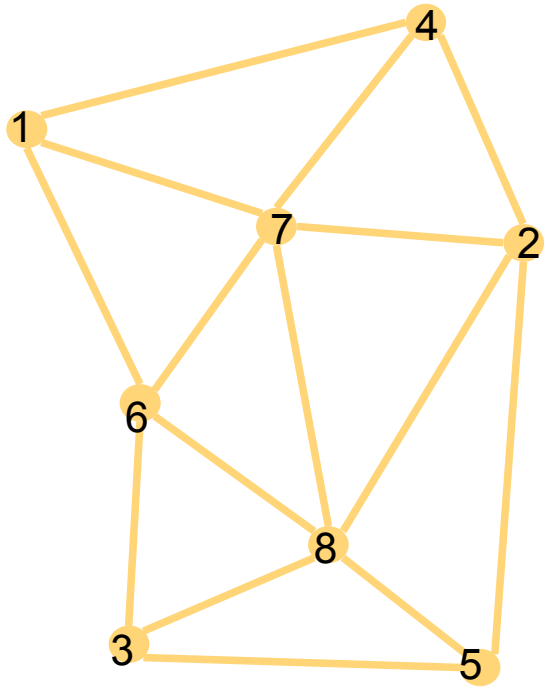
G_3: both

Mesh layout problem:

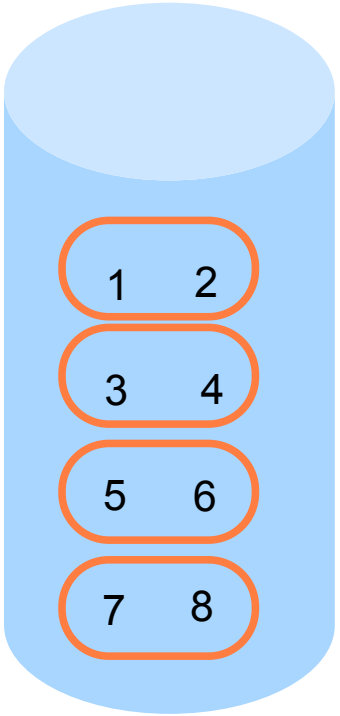
Minimize # of cache misses if each node touches all its neighbors ?



Example



cache (B=2,M=4)



disk

Previous work on mesh layouts

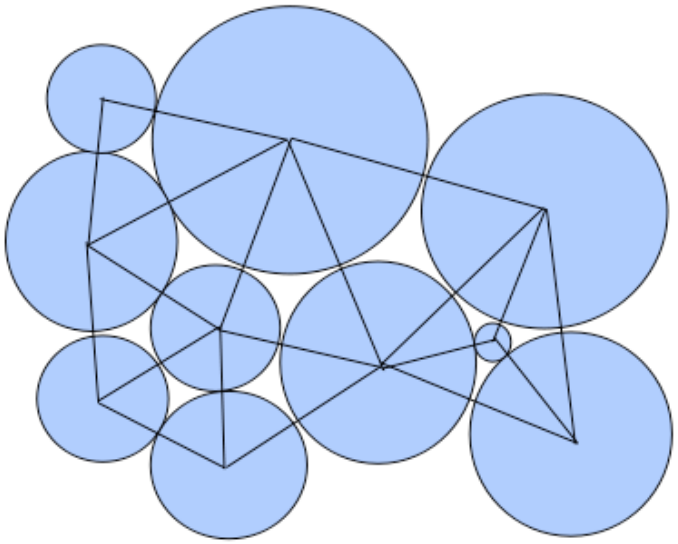
[Pascucci & al 2005]

- Heuristic algorithm based on multi-level optimization
- Good experimental results (2-5x improvement)
- But no guarantee on :
  - time to compute the layout
  - layout quality

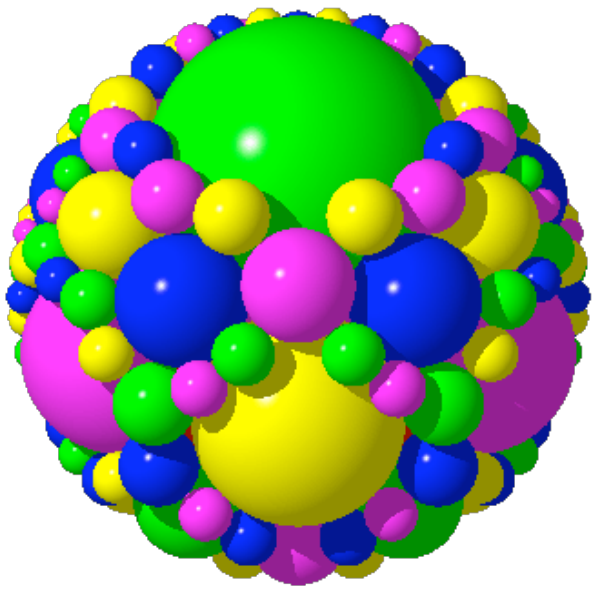
Overlap graphs

[Miller & al 98]

- Generalize planar graphs
- Contain well-shaped meshes



circles



d-dim spheres

Separator for overlap graphs

[Miller & al 98]

- Separate the mesh into two roughly equal-size pieces cutting few edges

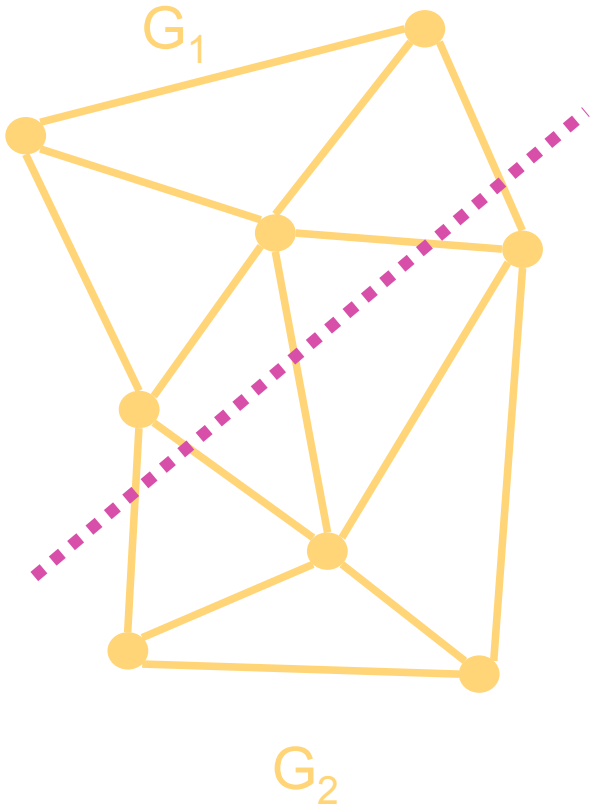
- Planar graphs [Lipton-Tarjan]

$$|G_1|, |G_2| \le \frac{2}{3}|G| \qquad E(G_1, G_2) = \sqrt{8|G|}$$

- Overlap graphs (randomized linear time)

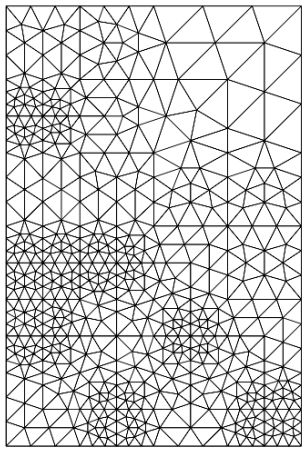
$$|G_1|, |G_2| \le \frac{d+1}{d+2}|G|$$

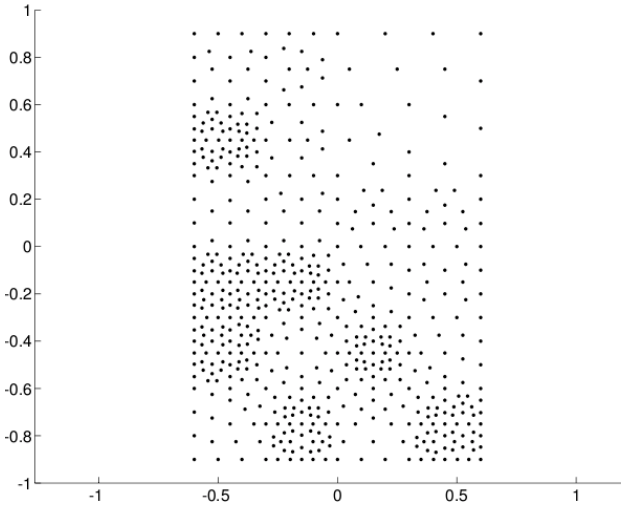
$$E(G_1, G_2) = O\left(|G|^{1-1/d}\right)$$

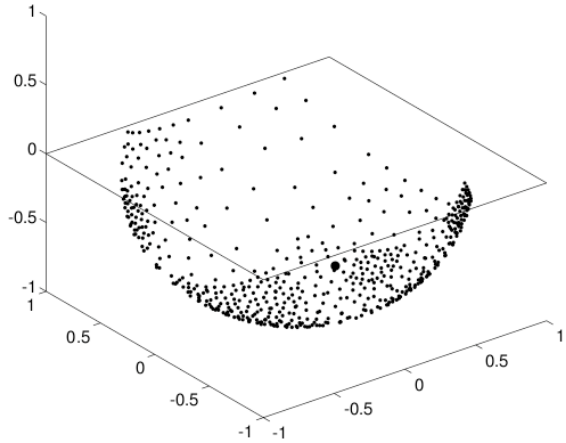


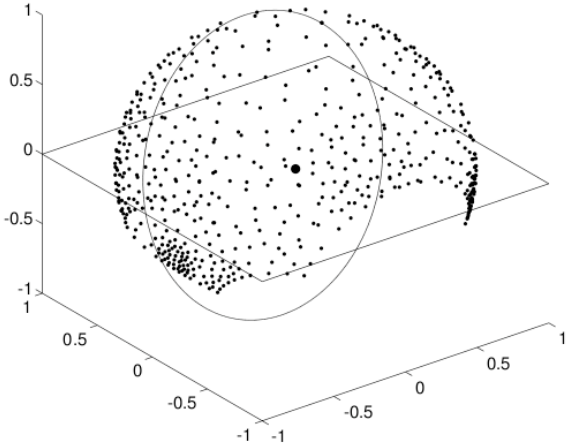
Separator for overlap graphs

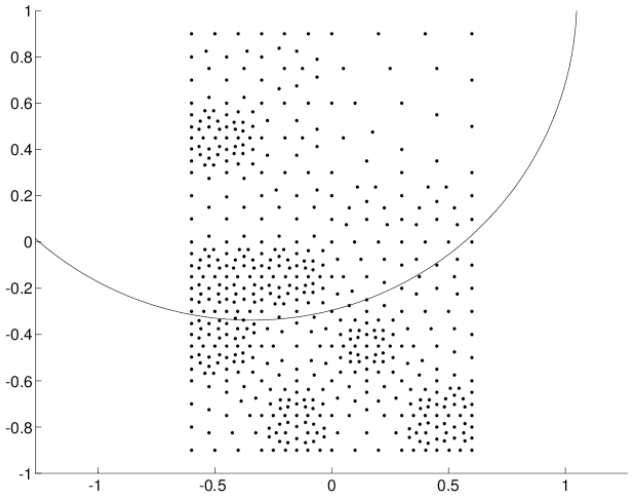
[Miller & al 98]

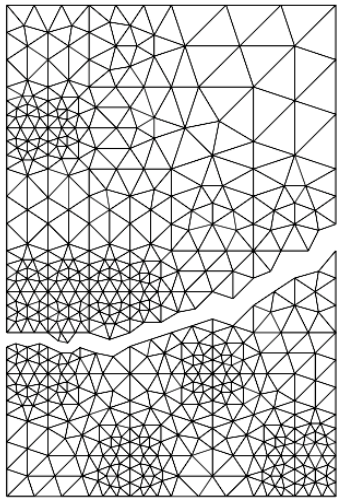






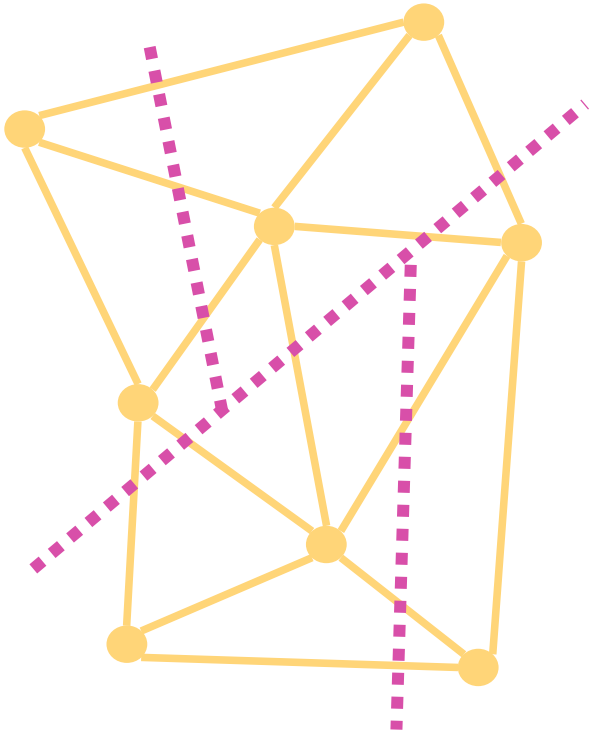


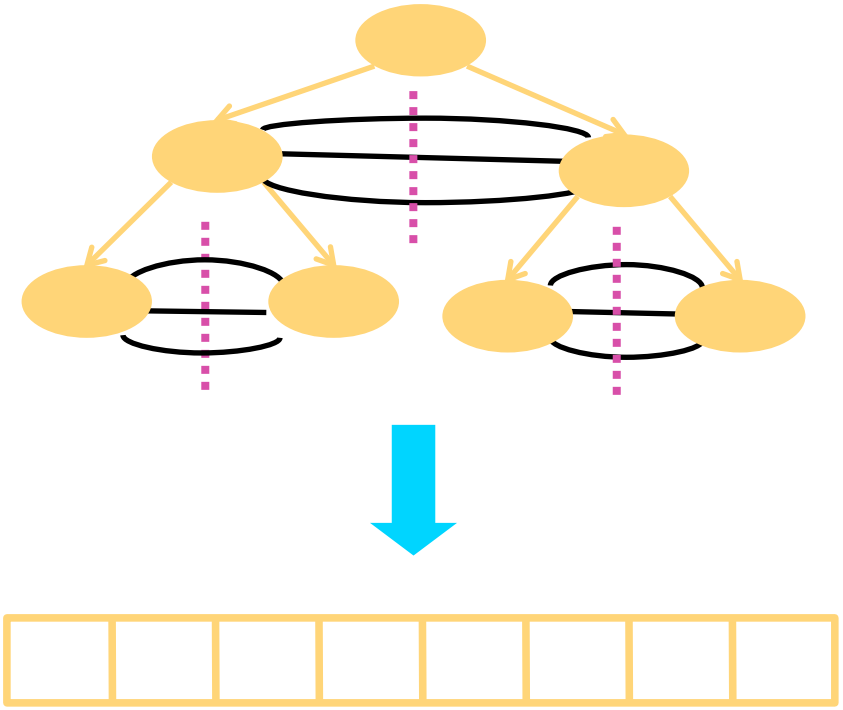




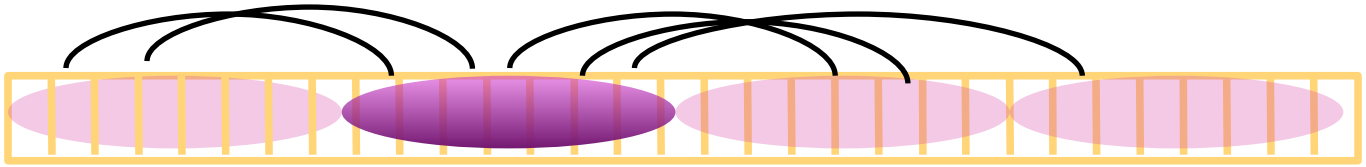
Our layout

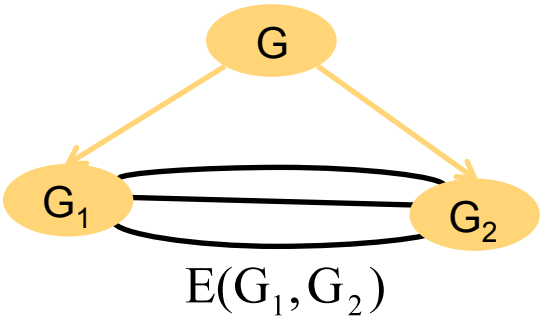
- Recursively cut the mesh $W(N) = O(N \log N)$
- The order of the leaves gives the layout





Analysis of the layout





$$\#\text{block transfers} = O\left(\frac{N}{B} + \#\text{outgoing edges}\right)$$

$$\text{Out}(G) = \text{Out}(G_1) + \text{Out}(G_2) + \left|E(G_1, G_2)\right|$$

$$K(N) \le \max_{\frac{1}{2} \le \delta \le \frac{d+1}{d+2}} \left\{K(\delta N) + K((1-\delta)N)\right\} + cN^{1-1/d}$$

$$K(M) = 0$$

$$\#\text{block transfers} = O\left(\frac{N}{B} + \frac{N}{M^{1/d}}\right)$$

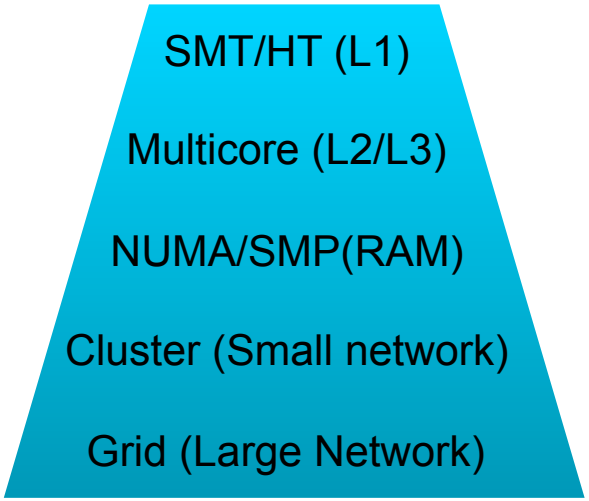
To come

- Use the layout
  - Experiments with vtk
  - Develop CO visualization algorithms
- Improve the layout
  - Can we make it better/faster for AMR ?
  - What if only part of the mesh is accessed ?
  - Dynamic
  - Space partitioning (e.g. octree)
- Develop PO & CO visualization algorithms

- 1. Introduction - Motivation for obliviousness
- 2. Processor oblivious
- 3. Cache oblivious
- **4. Conclusion**
  Towards processor & cache oblivious

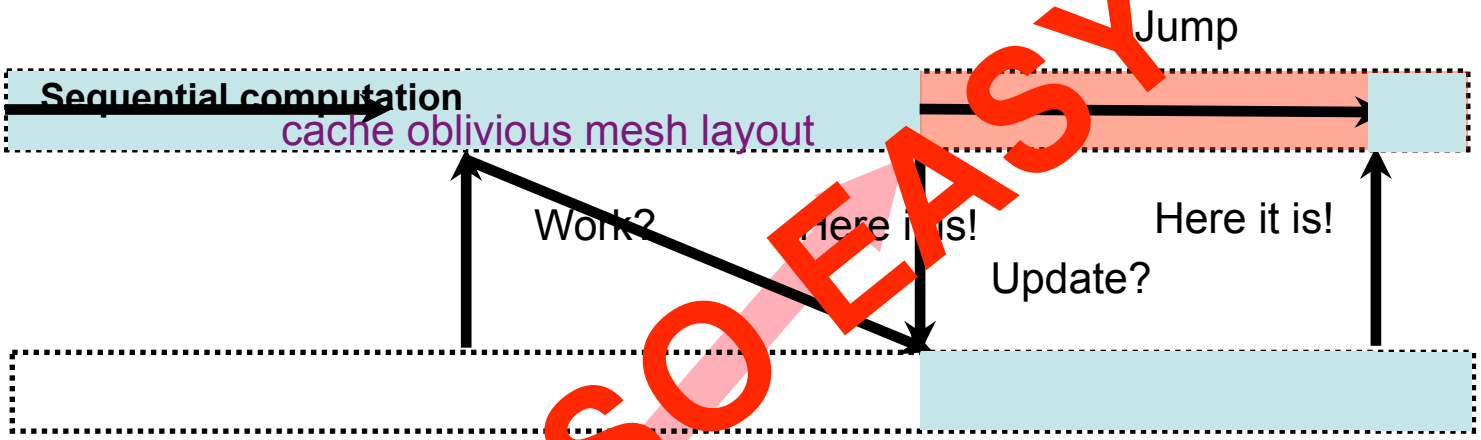
A processor & cache oblivious model

- Deeper memory hierarchy



- Processor + Cache → Communications
  - sharing a cache ↔ low cost communication

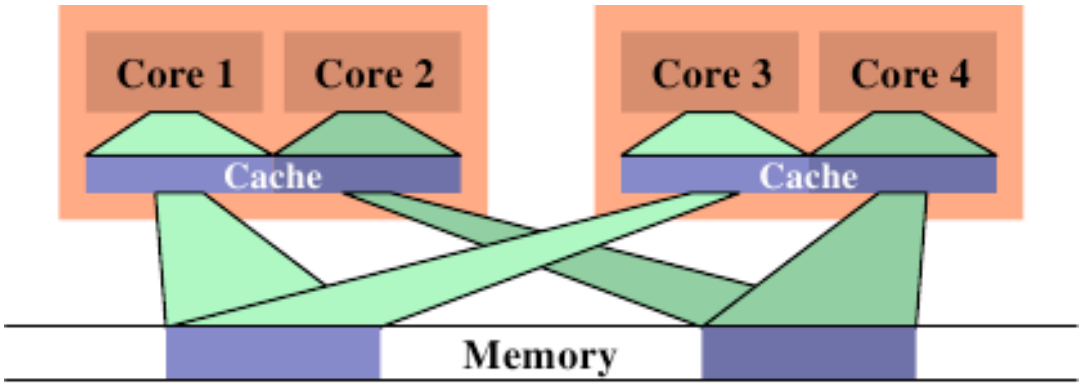
PCO mesh layout



NOT SO EASY

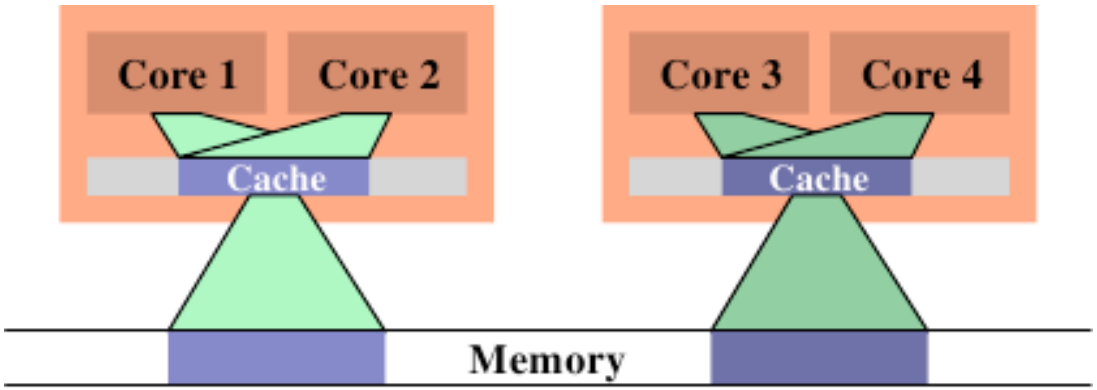
Cut few edges so few cache misses/synchro

Interaction cache / scheduling



Can be worse than
each core has a cache of half size

Cache sharing



Can be as good as
each core has a cache of full size

Conclusion

- PO+CO: Need a scheduler that is fully aware of the memory hierarchy [Blelloch & al 2008]
- Tradeoff between full parallelism and good cache complexity

Questions?