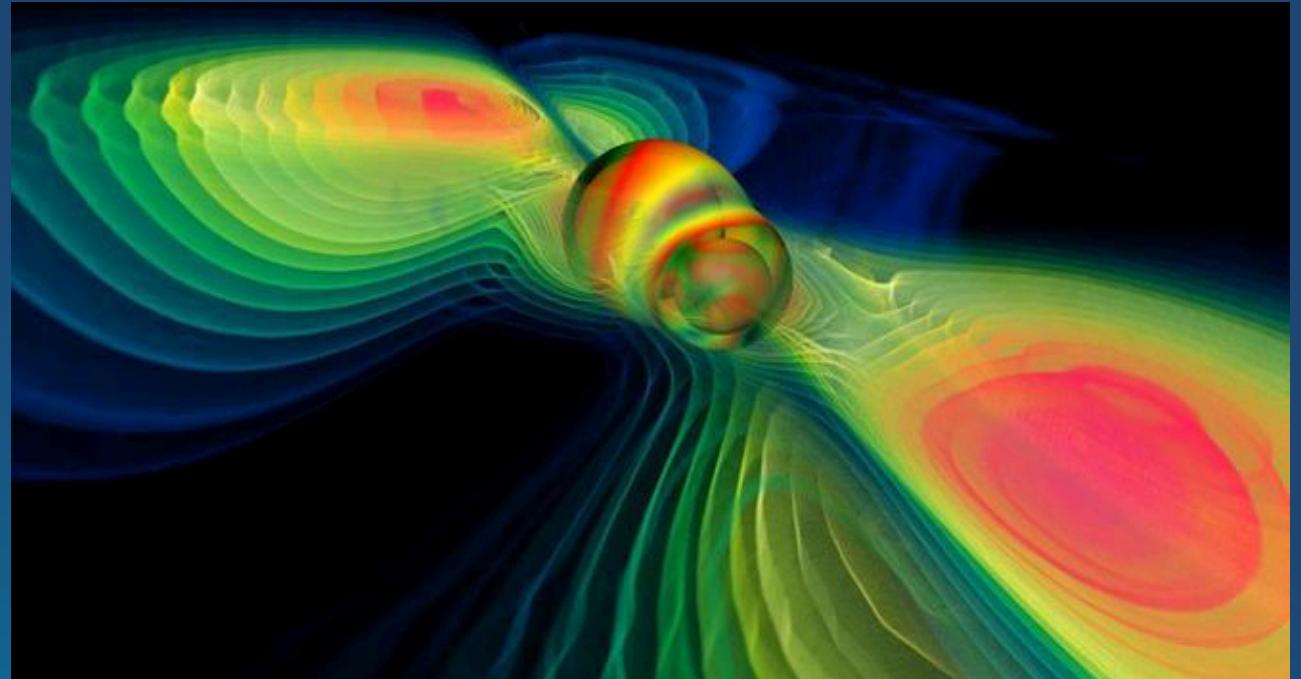


# HPC, Architectures et Performances

Frédéric Desprez  
INRIA  
Equipe Corse

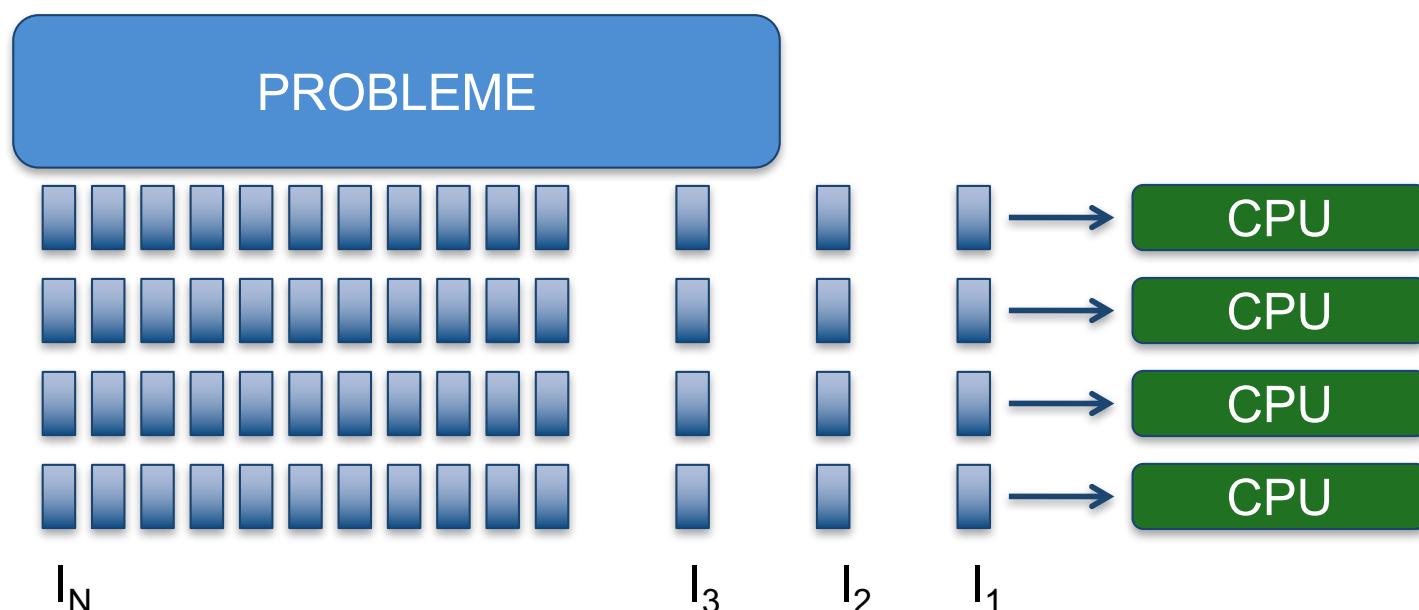
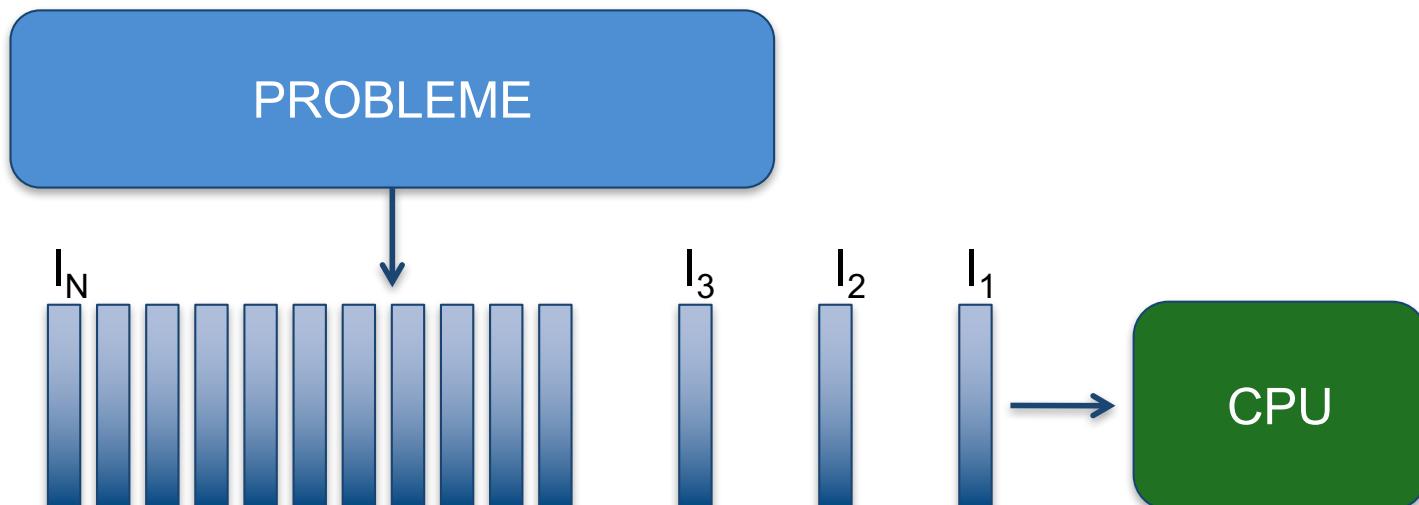


Simulation par ordinateur des ondes gravitationnelles produites lors de la fusion de deux trous noirs. Werner Benger, CC BY-SA

# Parallélisme ??



# Le parallélisme c'est



# Pourquoi s'intéresser au parallélisme ?

Parce qu'il est partout !



Tianhe-2 (3,120,000 cores)

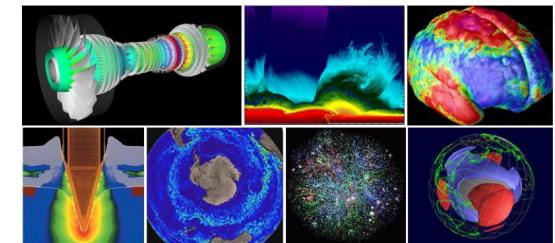


iPad

# Parce qu'on en a besoin !

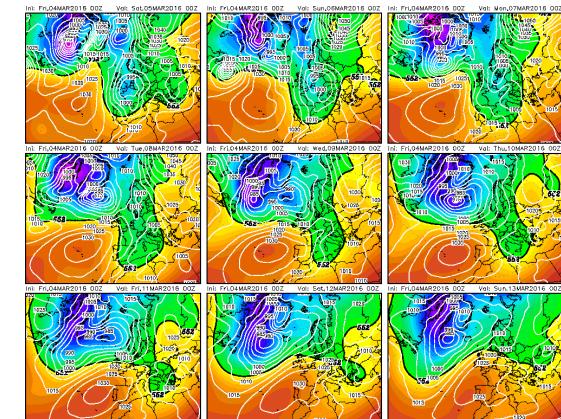
- **Résoudre des problèmes plus rapidement**

- Traiter plus de requêtes à la seconde (exemple Google)
- Améliorer les temps de réponse des applications interactives



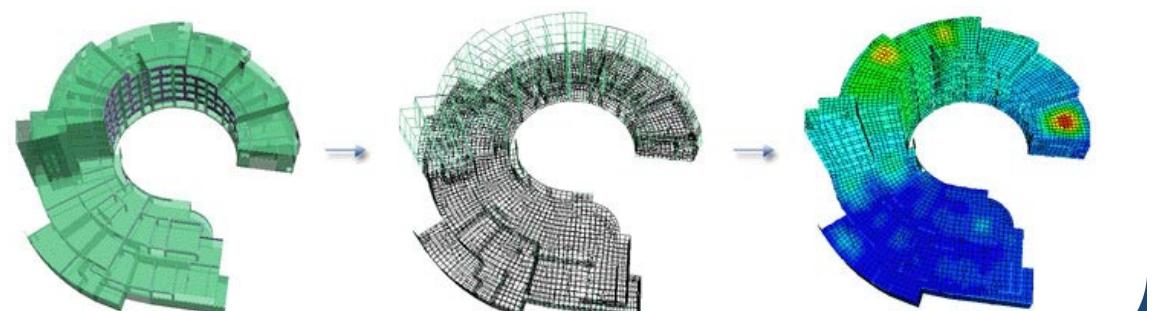
- **Obtenir des meilleurs résultats dans le même temps**

- Raffiner les modèles (exemple Météo France)
- Compliquer les modèles (multi-physiques)



- **Traiter des problèmes de plus grande taille**

- Maillages plus importants
- Données des réseaux de capteurs, des réseaux sociaux, ...
- Recherche dans les pages web (Google)
- Réseaux sociaux
- LHC (CERN)



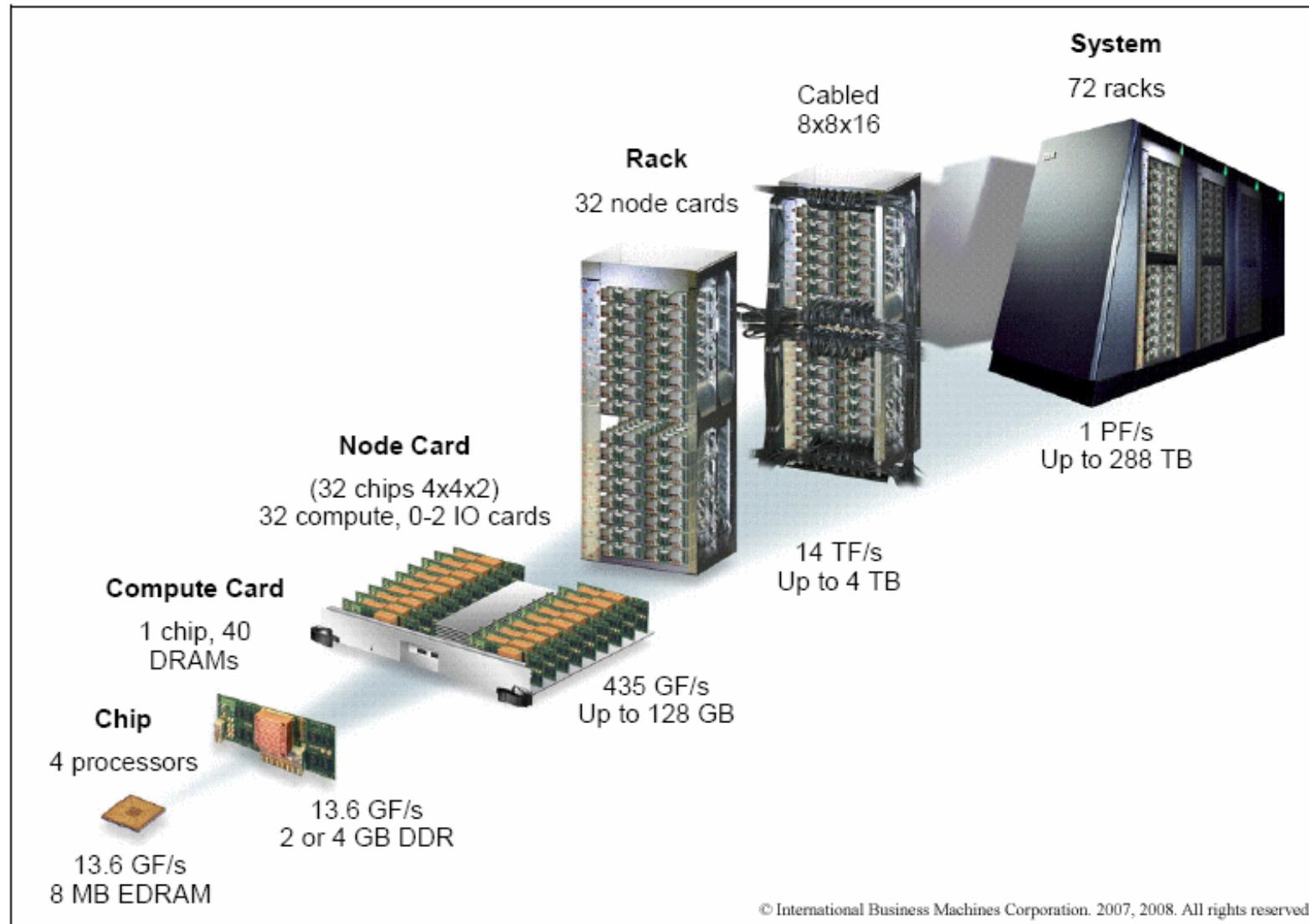
# Unités de mesure des performances

- Les unités en HPC
  - Flop: floating point operation, généralement double précision
  - Flop/s: opérations flottantes par seconde
  - Octets: taille des données (8 pour un nombre en double précision)

- Des tailles typiques millions, milliards, trillions...

Mega	$Mflop/s = 10^6 \text{ flop/sec}$	$Moctet = 2^{20} = 1048576 \sim 10^6 \text{ octets}$
Giga	$Gflop/s = 10^9 \text{ flop/sec}$	$Goctet = 2^{30} \sim 10^9 \text{ octets}$
Tera	$Tflop/s = 10^{12} \text{ flop/sec}$	$Toctet = 2^{40} \sim 10^{12} \text{ octets}$
Peta	$Pflop/s = 10^{15} \text{ flop/sec}$	$Poctet = 2^{50} \sim 10^{15} \text{ octets}$
Exa	$Eflop/s = 10^{18} \text{ flop/sec}$	$Eoctet = 2^{60} \sim 10^{18} \text{ octets}$
Zetta	$Zflop/s = 10^{21} \text{ flop/sec}$	$Zoctet = 2^{70} \sim 10^{21} \text{ octets}$
Yotta	$Yflop/s = 10^{24} \text{ flop/sec}$	$Yoctet = 2^{80} \sim 10^{24} \text{ octets}$

- La machine la plus puissante au monde actuellement ~ 54 Pflop/s
  - Liste mise-à-jour deux fois par an: [www.top500.org](http://www.top500.org)



# ARCHITECTURES

# Pourquoi ne pas accélérer les processeurs séquentiels ?

- **Soit une machine séquentielle à 1 Tflop**

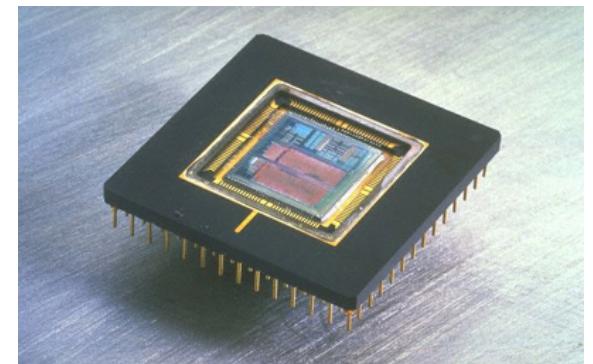
- Les données doivent aller de la mémoire au CPU (distance  $r$ )
- Pour récupérer une donnée par cycle ( $10^{12}$  fois par seconde) à la vitesse de la lumière ( $c = 299\ 792\ 458\ \text{m/s} \approx 3\text{e}8\ \text{m/s}$ )
- Donc  $r < c/10^{12} = .3\text{mm}$

- **Il faut mettre 1 Tera-octet de données dans  $0.3\ \text{mm}^2$**

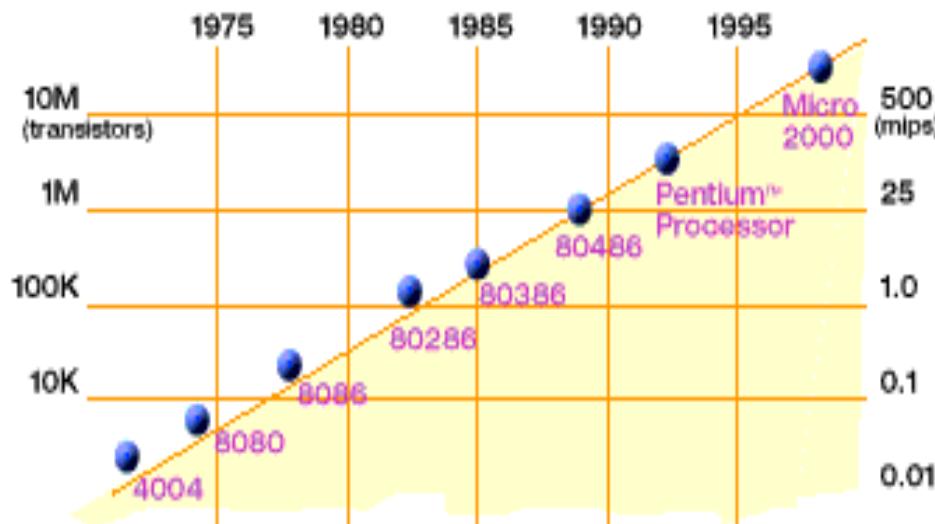
- Chaque mot occupe  $\approx 3\ \text{Angstroms}^2$ , soit la taille d'un petit atome

- Impossible à réaliser avec la technologie actuelle

- Attention aussi à la chaleur dégagée par un tel processeur !

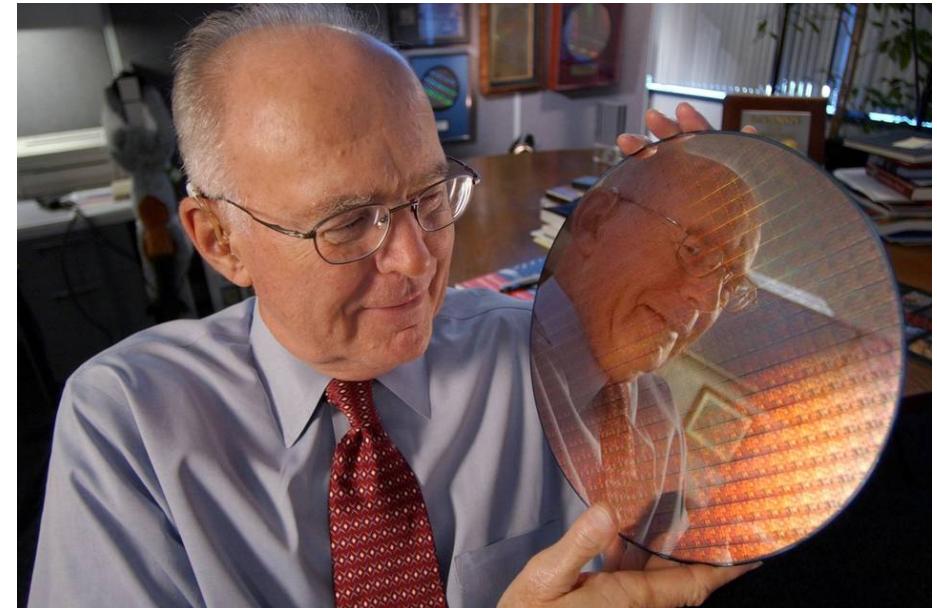


# Loi de Moore



2X transistors/puce tous les 18 mois  
(60% par an d'augmentation)

Les microprocesseurs sont devenus plus denses, plus petits et surtout plus puissants.



Gordon Moore (co-fondateur d'Intel) a prédit en 1965 que la densité des processeurs doublerait environ tous les 18 mois.

Source: Jack Dongarra

19/02/2017 - 9

# Loi de Moore, suite

- En 1965, raisonnement empirique basé sur une relation entre la complexité des circuits et le temps
- Loi qui a perduré à travers les années
- **Croissance due à plusieurs facteurs**
  - Augmentation de la complexité des processeurs (densité en transistors, augmentation de la taille)
  - Ajout de fonctionnalités (caches internes, buffers d'instructions plus grands, lancement de plusieurs instructions par cycles, multithreading, profondeur des pipelines, ré-arrangement des instructions)

# “Free lunch is over”, Herb Sutter

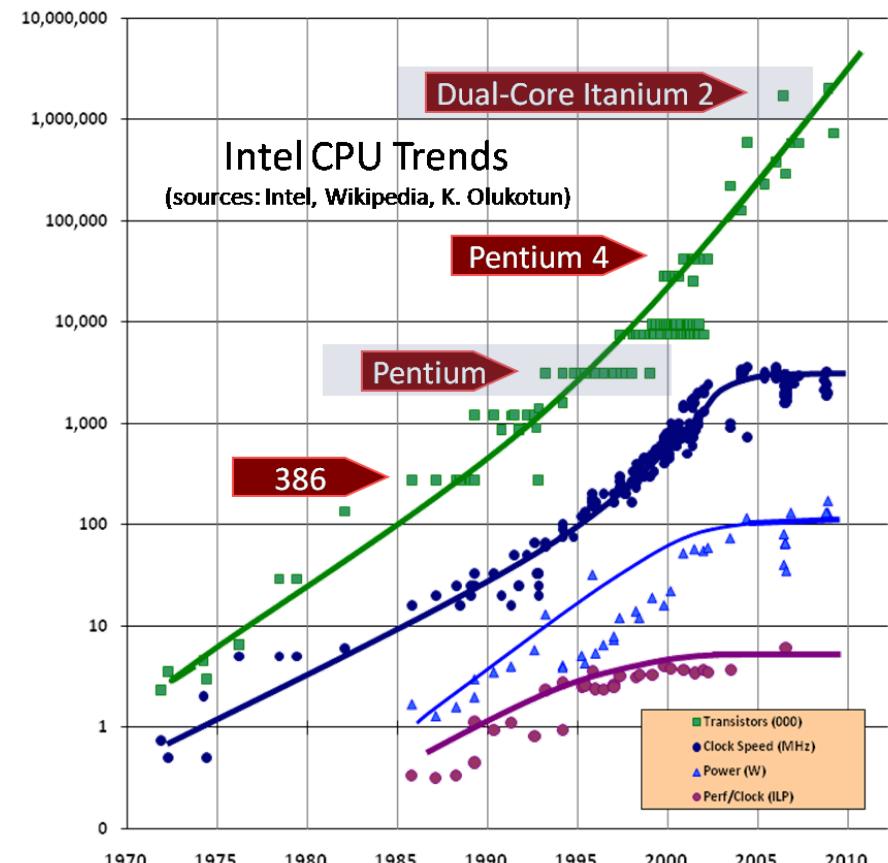
- La vitesse d'horloge ne va plus continuer à doubler ...
- ... mais on veut que les applications continuent à accélérer !

- Quelques problèmes dus à l'augmentation de la vitesse d'horloge

- Consommation électrique
- Dissipation de la chaleur
- Fuites

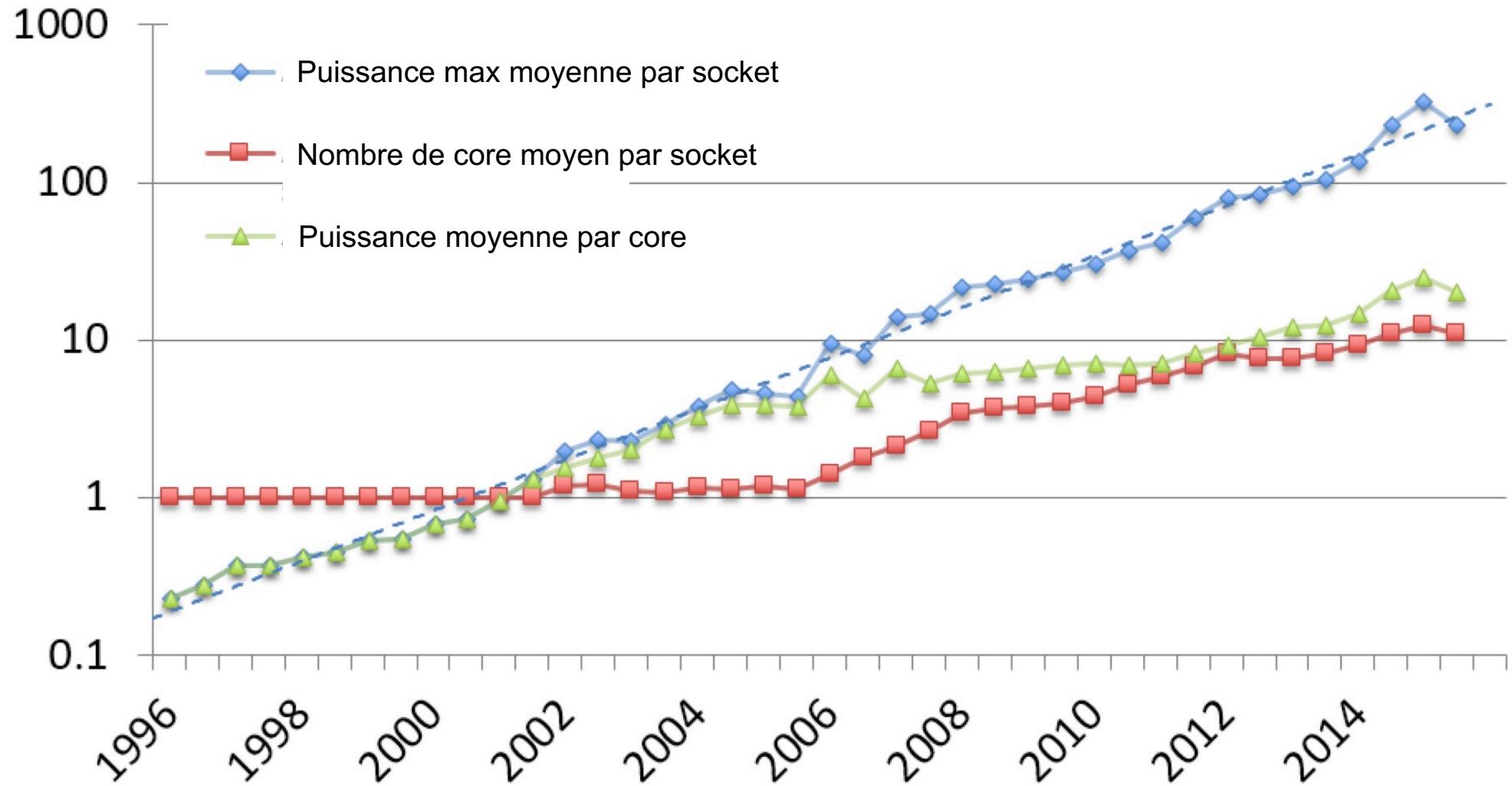
Mais aussi

- Limite physique due à la vitesse de la lumière (propagation des signaux)

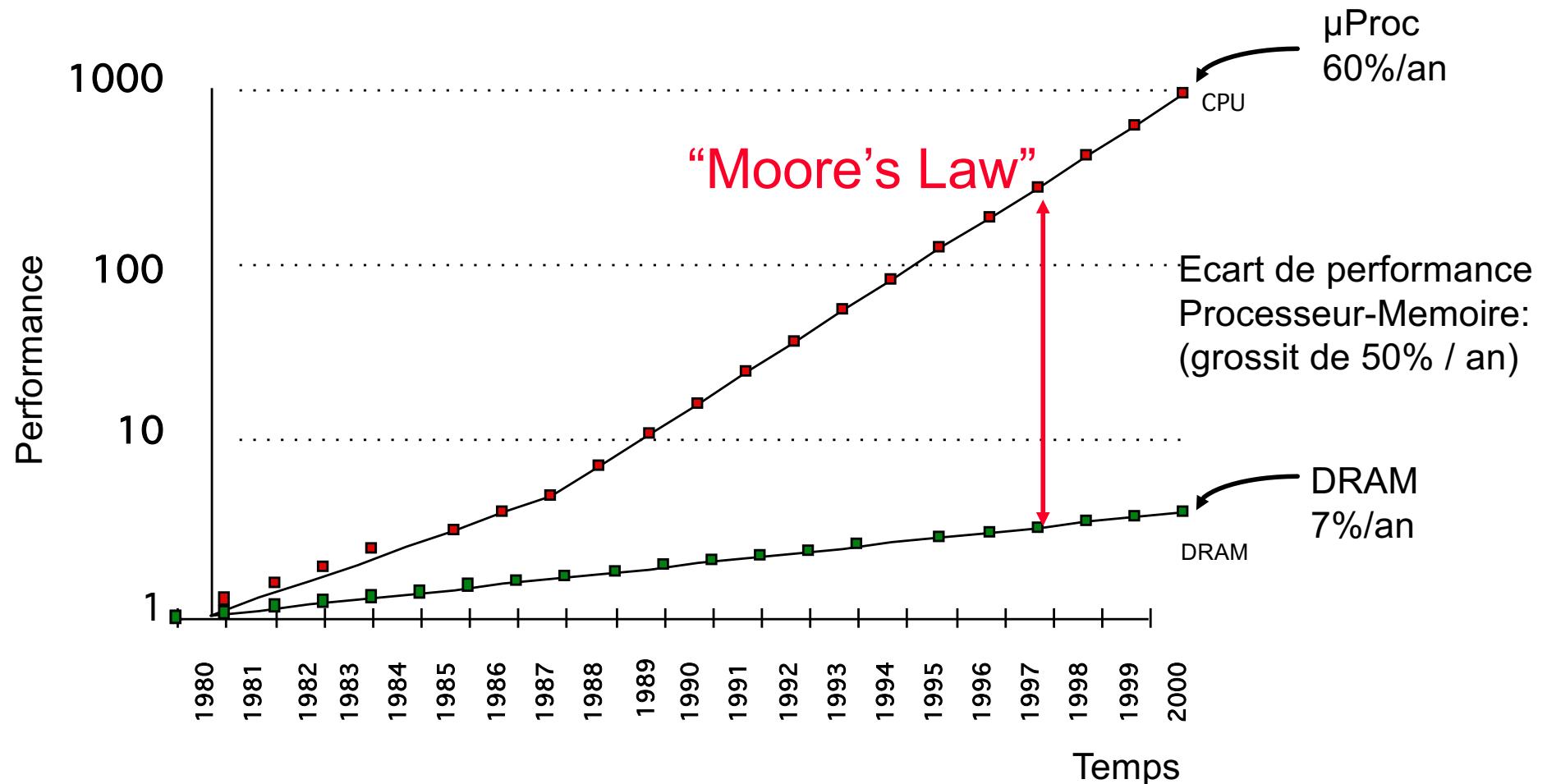


***The Free Lunch Is Over, A Fundamental Turn Toward Concurrency in Software***, Herb Sutter,  
Dr. Dobb's Journal, 30(3), March 2005. <http://www.gotw.ca/publications/concurrency-ddj.htm>

# Loi de Moore, suite



# Ecart Processeur-DRAM (latence)



**But:** trouver des algorithmes qui minimisent les transferts de données, pas forcément les calculs

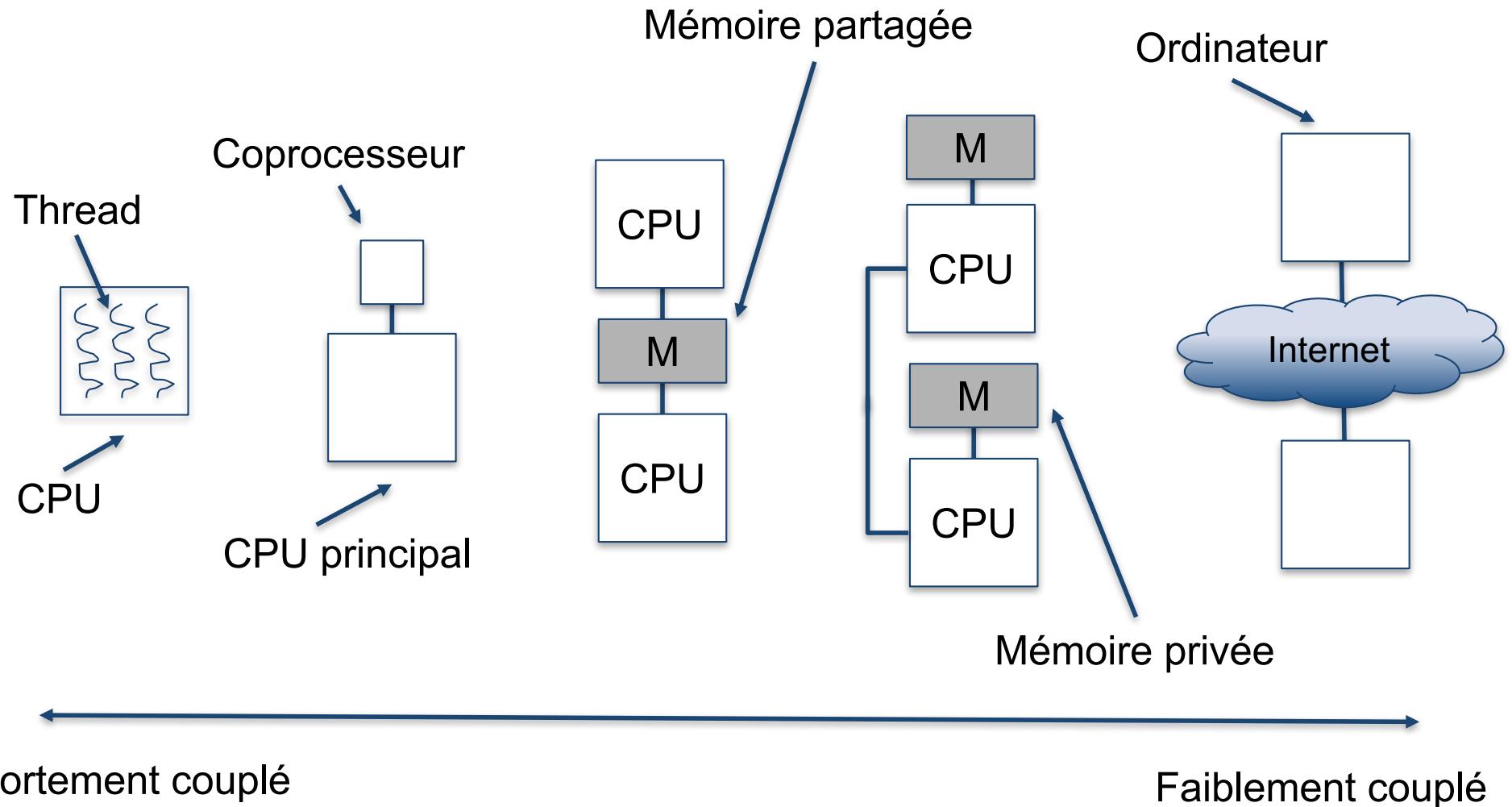
# Qu'est-ce qu'une machine parallèle ?

- Une collection d'éléments de calcul capables de communiquer et de coopérer dans le but de résoudre rapidement des problèmes de grande taille
- **Une collection d'éléments de calcul**
  - Combien ?
  - De quelle puissance ?
  - Que sont-ils capables de réaliser?
  - Quelle est la taille de leur mémoire associée ?
  - Quelle en est l'organisation ?
  - Comment les entrées/sorties sont-elles réalisées ?
- **... capables de communiquer ...**
  - Comment sont-ils reliés entre eux ?
  - Que sont-ils capables d'échanger ?
  - Quel est leur protocole d'échange d'information ?

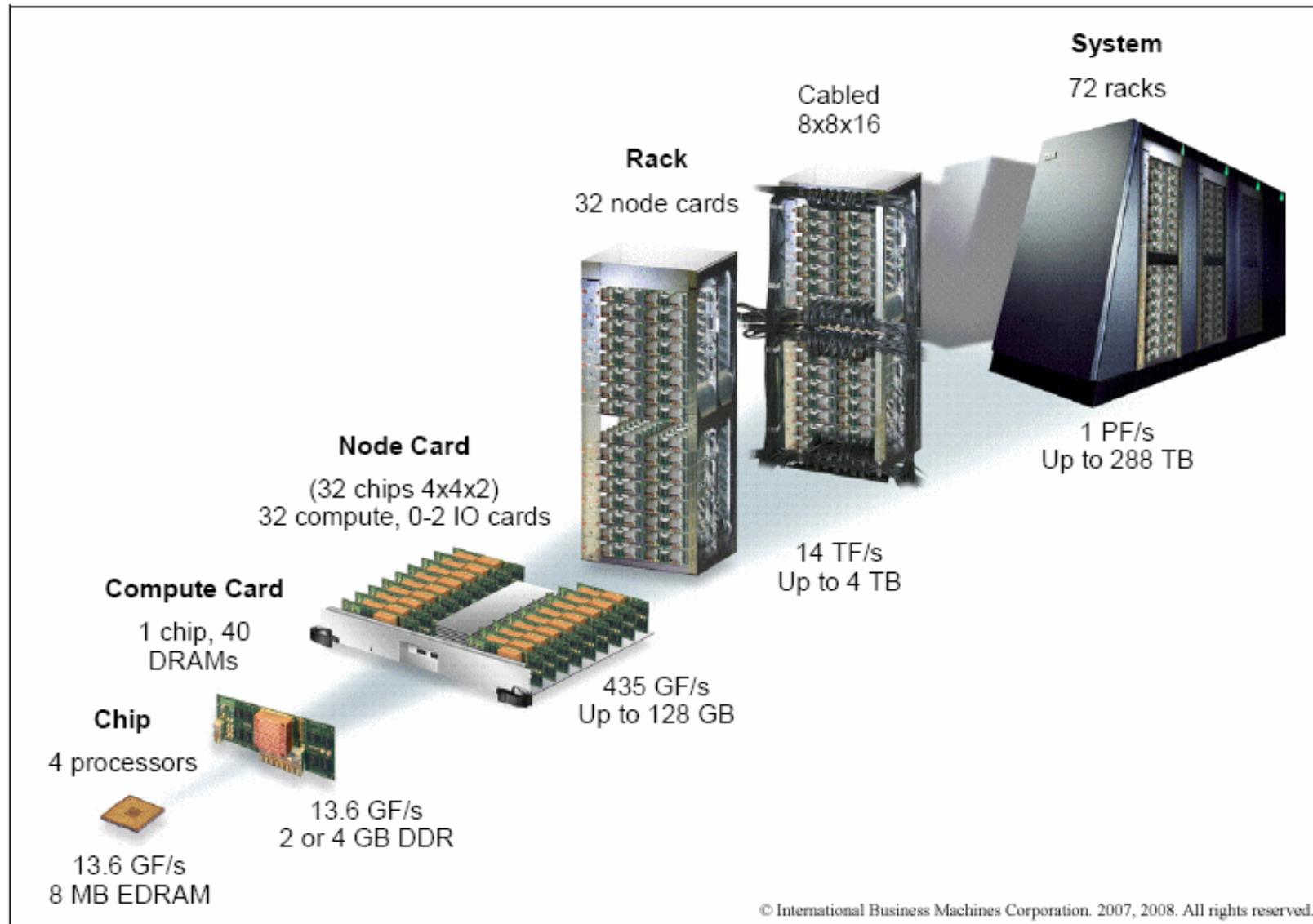
# Qu'est-ce qu'une machine parallèle ?, suite

- **... et de coopérer ...**
  - Comment les éléments de calcul synchronisent ils leurs efforts ?
  - Quel est leur degré d'autonomie ?
  - Comment sont-ils pris en compte par le système d'exploitation ?
- **... dans le but de résoudre des problèmes de grande taille.**
  - Quels sont les problèmes à fort potentiel de parallélisme ?
  - Quel est le modèle de calcul utilisé ?
  - Quel est le degré de spécialisation des machines à un problème donné?
  - Comment choisir les algorithmes ?
  - Quelle efficacité peut-on espérer ?
  - Comment ces machines se programment elles ?
  - Quels langages faut-il ?
  - Comment faut-il exprimer le parallélisme ?
  - Le parallélisme est il implicite ou explicite ?
  - L'extraction du parallélisme est-elle automatique ou manuelle ?

# Architectures parallèles



# Les machines parallèles d'aujourd'hui

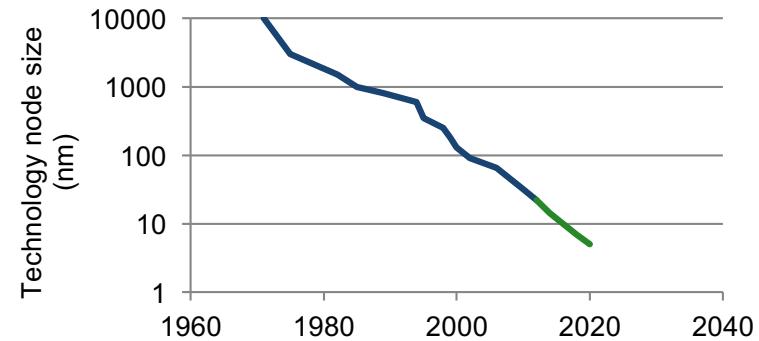


<https://computing.llnl.gov/tutorials/bgp/images/bgpScalingArch.gif>

# Quelques révolutions technologiques à venir

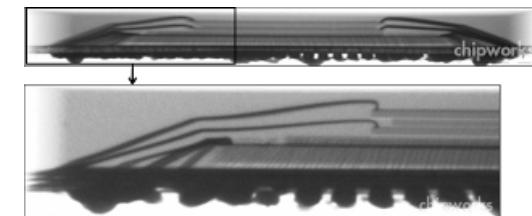
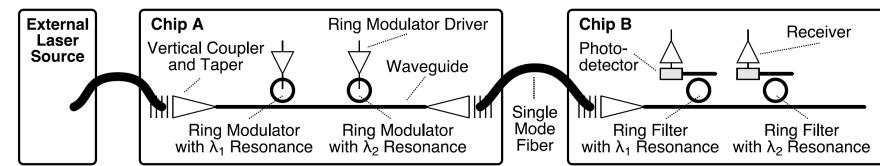
## • CPU

- Plus de cores (500 ?) plus lents
  - 60 cores @22nm => ~500 cores @8nm
  - 1 Tflops @22nm => ~8 Tflops @ 8nm
- Réduire la consommation d'énergie
- Problème de Fréquence/voltage



## • Déplacer les données

- Photonics
  - Modulateur, Récepteur, Multiplexeur
  - Distances plus longues, réduction de la consommation
- Circuits intégrés tridimensionnels
  - CPU au dessus de la mémoire
  - Réduire la latence, réduction de la consommation
- *Non-Volatile RAM (NVRAM)*
  - PCRAM, memristor
  - *Non-Volatile Dual In-line Memory Module (NVDIMM)*



	Bande-passante	Energie
DDR-3	10,66 GB/s	50-70 pJ/bit
HMC	128GB/s	8 pJ/bit

# Architectures des Clusters

## Architectures dédiées

- YarcData's Urika (Cray)
  - Mémoire partagée de grande taille (jusqu'à 512 To)
  - Processeur Multithreadé (65000 threads dans un système à 512 processeurs)
  - Système d'entrées/sorties à grande vitesse (jusqu'à 350 To/s), réseau en tore 3-D
- Plate-forme basée sur Hadoop

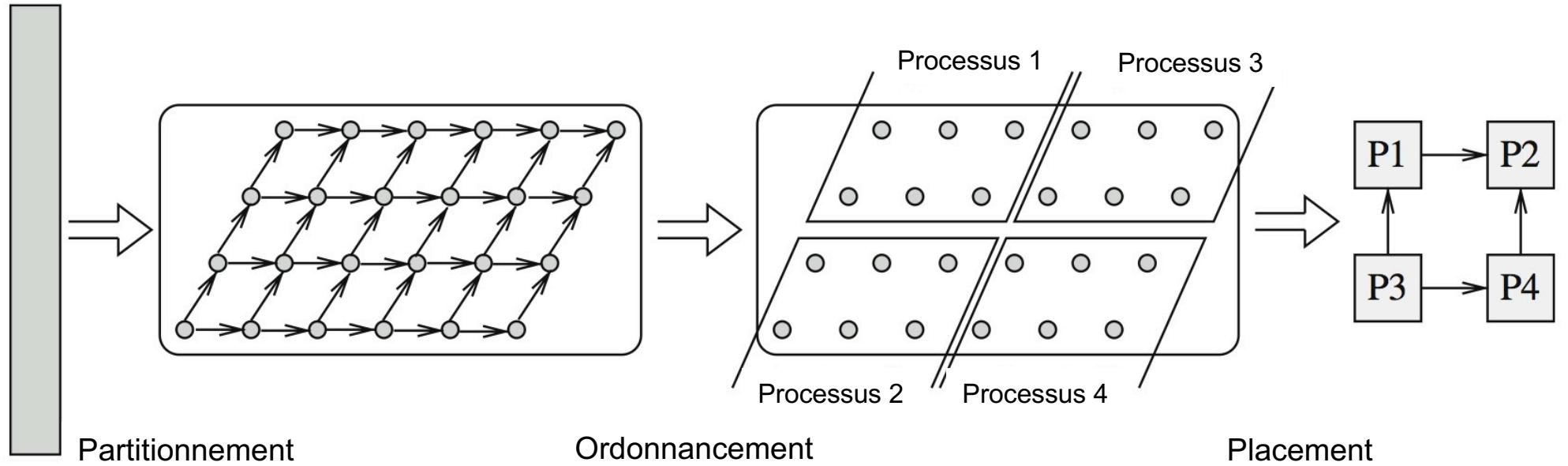


## Mémoire universelle

- Réduire la complexité de la pile mémoire

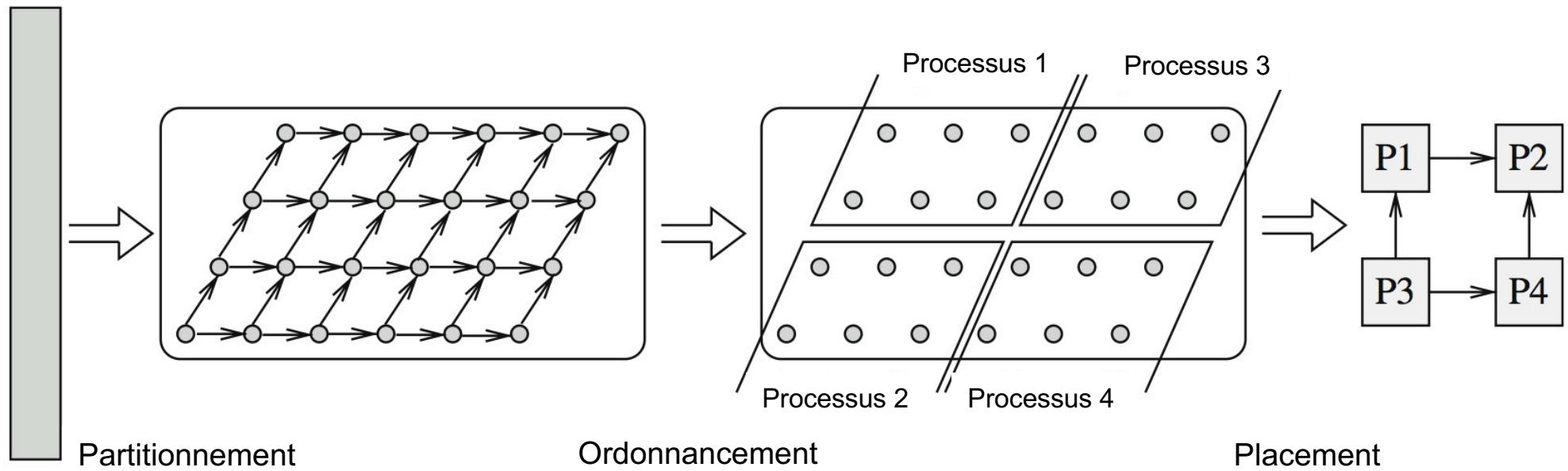
## Clusters sans disques

- Processeurs à grande mémoire utilisant des NVRAM



# PARALLELISATION

# Etapes de parallélisation

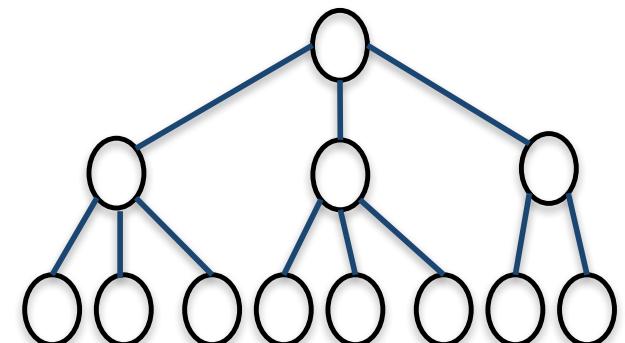


# Parallélisation des algorithmes

- Si l'architecture et/ou le modèle de programmation ont une influence sur la manière d'implanter un algorithme parallèle et de l'écrire
- Il faut déjà concevoir l'algorithme parallèle en fonction des caractéristiques de l'algorithme séquentiel
  - Types de données manipulées (tableaux, graphes, structure de données, ...)
  - Accès aux données (régulier, aléatoire, ...)
  - Granularité
  - Dépendances (de contrôle, de données)
  - ...
- **Buts**
  - Obtenir le même résultat en parallèle qu'en séquentiel !
  - Gagner du temps
  - Réduire la consommation de stockage (mémoire, disque)

# Décomposition des calculs

- Les calculs d'un programme séquentiels sont décomposés en tâches (avec des dépendances entre elles)
- Chaque tâche est la plus petite unité de parallélisme
- **Différents niveaux d'exécution**
  - Parallélisme d'instructions
  - Parallélisme de données
  - Parallélisme fonctionnel
- La décomposition peut être statique (avant exécution) ou dynamique (à l'exécution)
- A tout moment de l'exécution, le nombre de tâches « exécutables » donne une borne supérieure du degré de parallélisme du programme
- La décomposition en tâches consiste à générer assez de parallélisme pour avoir une occupation maximum des cores/processeurs/machines



# Décomposition des calculs

- Attention à la **granularité** !
  - Il faut que le temps d'exécution d'une tâche soit grand par rapport au temps qu'il faut pour l'ordonnancer, la placer, la démarrer, acquérir ses données, renvoyer les données calculées
    - Tâches à gros grains avec de nombreux calculs
    - Tâches à grain fin avec peu de calculs
  - Compromis à trouver en fonction des performances (calcul, accès aux données, surcoût du système, communications, disque, ...)

# Affectation aux processus ou threads

- Un **processus** ou un **thread** représente un **flot de contrôle** exécuté par un **processeur** ou un **core**
  - Exécute les tâches les une à la suite des autres
  - Pas forcément le même nombre que le nombre de processeurs (ou de cores)
- Affecter les tâches pour assurer un bon équilibrage (i.e. tous les processeurs ou les cores doivent avoir le même volume de calcul à exécuter)
  - Attention à prendre en compte si possible
    - les **accès mémoire** (pour les mémoires partagées)
    - les **échanges de messages** (pour les mémoires distribuées)
  - Réutilisation des données si possible
  - Aussi appelé **ordonnancement**.
    - Si exécuté durant la phase d'initialisation du programme: **ordonnancement statique**
    - Si exécuté durant l'exécution du programme: **ordonnancement dynamique**
- **But principal:** utilisation égale des processeurs tout en conservant un volume de communications entre les processeurs le plus petit possible

# Placement des processus sur les processeurs

- Généralement placement de chaque processus (ou thread) sur des processeurs (ou core) séparés
- Si plus de processus que de processeurs alors **placement** de plus de processus sur les processeurs
  - Effectué par le système et/ou des directives du programme
  - But: assurer l'équilibre et réduire les communications
- **Algorithme d'ordonnancement**
  - Méthode pour assurer une exécution efficace d'un ensemble de tâches pour une durée déterminée sur un ensemble d'unités d'exécutions déterminé
  - Satisfaire les dépendances entre les tâches (contraintes de précédence)
  - Contraintes de capacités (car nombre d'unités d'exécution limité)
  - Tâches séquentielles ou parallèles
  - **But**
    - définir le **temps de départ** et l'**unité d'exécution** cible pour chaque tâche
    - Minimiser le **temps d'exécution** (*makespan*) = temps entre début de la 1<sup>ère</sup> tâche et la fin de la dernière

# Niveaux de parallélisme

- Les calculs exécutés par un programme fournissent des opportunités d'exécution parallèle à différents niveaux

- Instruction
- Boucle
- Fonctions

```
for i = 0 to n-1 {  
    y[i] = 0;  
    for j = 0 to n-1  
        y[i] = y[i] + A[i,j] * x[j];  
}
```

- Différents niveaux de granularité (fine, moyenne, large)
  - Possibilité de grouper les différents éléments pour augmenter la granularité
  - Différents algorithmes suivant les différentes granularités

# Parallélisme au niveau instruction

- Les instructions d'un programme peuvent s'exécuter en parallèle si elles sont indépendantes
- Il y a plusieurs types de dépendances
  - **Dépendances de flot** (*flow dependencies* ou *true dependencies*): il y a une dépendance de flot entre  $I_1$  et  $I_2$  si  $I_1$  calcule une valeur résultat dans un registre ou une variable utilisée par  $I_2$  comme opérande
  - **Anti-dépendance** (*anti-dependency*): il y a une anti-dépendance entre  $I_1$  et  $I_2$  si  $I_1$  utilise un registre ou une variable en opérande qui est utilisé par  $I_2$  pour stocker un résultat
  - **Dépendance de sortie** (*output dependency*): il y a une dépendance de sortie entre  $I_1$  et  $I_2$  si  $I_1$  et  $I_2$  utilisent le même registre ou la même variable pour stocker un résultat d'un calcul

$$I_1: \underline{R_1} \leftarrow R_2 + R_3$$

$$I_2: R_5 \leftarrow \underline{R_1} + R_4$$

flow dependency

$$I_1: R_1 \leftarrow \underline{R_2} + R_3$$

$$I_2: \underline{R_2} \leftarrow R_4 + R_5$$

anti dependency

$$I_1: \underline{R_1} \leftarrow R_2 + R_3$$

$$I_2: R_1 \leftarrow R_4 + R_5$$

output dependency

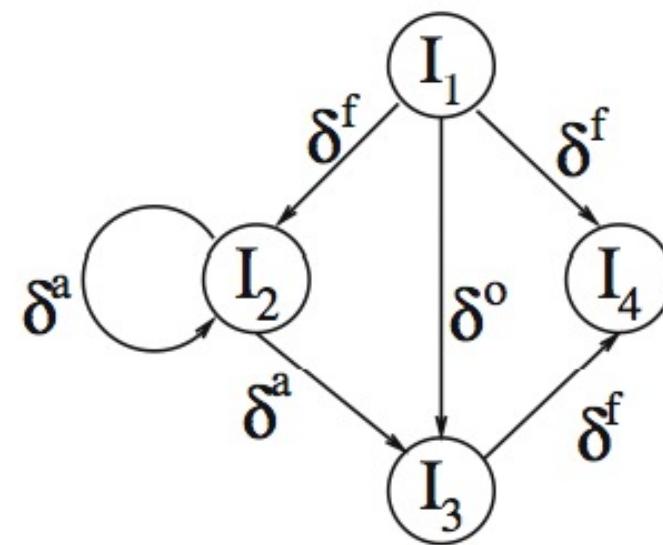
# Graphe de dépendances

$I_1: R_1 \leftarrow A$

$I_2: R_2 \leftarrow R_2 + R_1$

$I_3: R_1 \leftarrow R_3$

$I_4: B \leftarrow R_1$



# Utilisation du parallélisme d'instruction

- Utilisé par les processeurs superscalaires
  - Ordonnancement dynamique d'instructions au niveau matériel avec extraction des instructions indépendantes automatiquement
  - Affectées ensuite à des unités indépendantes
- Sur les VLIW, ordonnancement statique réalisé par le compilateur avec arrangement des instructions dans des instructions longues avec affectation aux unités
- En entrée un programme séquentiel sans parallélisme explicite

# Parallélisme de données

- **Même opération** effectuée sur **différents éléments** d'une structure de données de plus grande taille (tableau par exemple)
  - Si ces opérations sont indépendantes alors elles peuvent être effectuées en parallèle
  - Les éléments de la structure de donnée sont répartis de manière équilibrée sur les processeurs
  - Chaque processeur exécute l'opération sur les éléments qui lui ont été affectés
- **Extension de langage de programmation** (langages data-parallèles)
  - Un seul flot de contrôle
  - Constructions spéciales pour exprimer les opérations data-parallèles sur les tableaux (modèle SIMD)

```
a(1:n) = b(0:n-1) + c(1:n)
```

```
for (i=1:n)
    a(i) = b(i-1) + c(i)
endfor.
```

- Owner compute rule

# Etapes de l'algorithme de Fox

$C_{00}$	$C_{01}$	$C_{02}$	$C_{03}$
$C_{10}$	$C_{11}$	$C_{12}$	$C_{13}$
$C_{20}$	$C_{21}$	$C_{22}$	$C_{23}$
$C_{30}$	$C_{31}$	$C_{32}$	$C_{33}$

$C_{00}$	$C_{01}$	$C_{02}$	$C_{03}$
$C_{10}$	$C_{11}$	$C_{12}$	$C_{13}$
$C_{20}$	$C_{21}$	$C_{22}$	$C_{23}$
$C_{30}$	$C_{31}$	$C_{32}$	$C_{33}$

$C_{00}$	$C_{01}$	$C_{02}$	$C_{03}$
$C_{10}$	$C_{11}$	$C_{12}$	$C_{13}$
$C_{20}$	$C_{21}$	$C_{22}$	$C_{23}$
$C_{30}$	$C_{31}$	$C_{32}$	$C_{33}$

$A_{00}$	$A_{01}$	$A_{02}$	$A_{03}$
$A_{10}$	$A_{11}$	$A_{12}$	$A_{13}$
$A_{20}$	$A_{21}$	$A_{22}$	$A_{23}$
$A_{30}$	$A_{31}$	$A_{32}$	$A_{33}$

$A_{00}$	$A_{00}$	$A_{00}$	$A_{00}$
$A_{11}$	$A_{11}$	$A_{11}$	$A_{11}$
$A_{22}$	$A_{22}$	$A_{22}$	$A_{22}$
$A_{33}$	$A_{33}$	$A_{33}$	$A_{33}$

$A_{00}$	$A_{00}$	$A_{00}$	$A_{00}$
$A_{11}$	$A_{11}$	$A_{11}$	$A_{11}$
$A_{22}$	$A_{22}$	$A_{22}$	$A_{22}$
$A_{33}$	$A_{33}$	$A_{33}$	$A_{33}$

$B_{00}$	$B_{01}$	$B_{02}$	$B_{03}$
$B_{10}$	$B_{11}$	$B_{12}$	$B_{13}$
$B_{20}$	$B_{21}$	$B_{22}$	$B_{23}$
$B_{30}$	$B_{31}$	$B_{32}$	$B_{33}$

$B_{00}$	$B_{01}$	$B_{02}$	$B_{03}$
$B_{10}$	$B_{11}$	$B_{12}$	$B_{13}$
$B_{20}$	$B_{21}$	$B_{22}$	$B_{23}$
$B_{30}$	$B_{31}$	$B_{32}$	$B_{33}$

$B_{00}$	$B_{01}$	$B_{02}$	$B_{03}$
$B_{10}$	$B_{11}$	$B_{12}$	$B_{13}$
$B_{20}$	$B_{21}$	$B_{22}$	$B_{23}$
$B_{30}$	$B_{31}$	$B_{32}$	$B_{33}$

Etat initial

Diffusion de la 1ère diag. de A (stockée dans un tampon séparé)

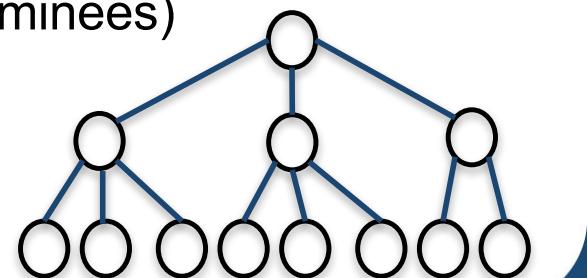
Calculs locaux

# Parallélisme de boucles

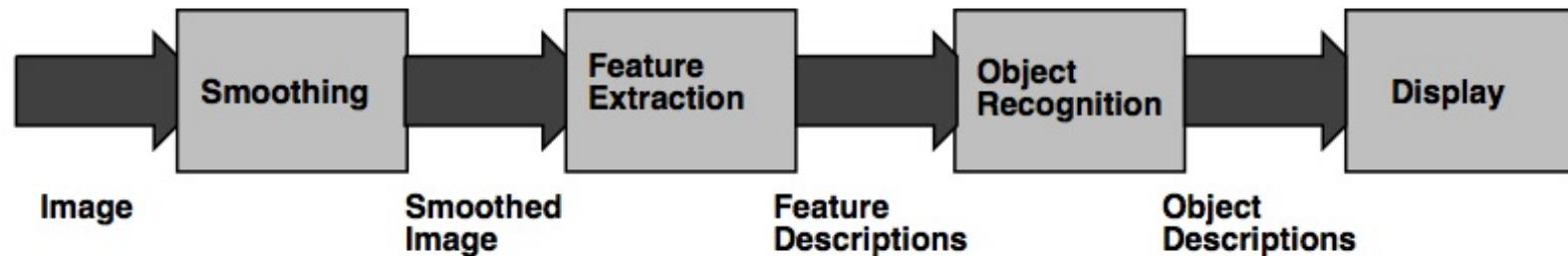
- Nombreux algorithmes traversent une structure de donnée de grande taille (tableau de données, matrice, image, ...)
- Exprimé par une boucle dans les langages impératifs
- S'il n'y a pas de dépendances entre les différentes itérations d'une boucle, on aura une boucle parallèle
- Plusieurs types de boucles parallèles
  - FORALL
  - DOPAR
- Possibilité d'avoir
  - plusieurs instructions dans une boucle
  - Boucles imbriquées
- Transformation de boucles pour trouver du parallélisme

# Parallélisme fonctionnel

- De nombreux programmes parallèles ont des **parties indépendantes** les unes des autres qui peuvent être **exécutées en parallèle**
  - Instructions simples, blocs, boucles ou appels de fonctions
- On peut voir du **parallélisme de tâches** en considérant les différentes parties d'un programme comme des tâches
  - **Parallélisme fonctionnel** ou **parallélisme de tâches**
- Pour utiliser le parallélisme de tâches, les tâches et leurs dépendances sont représentées sous forme d'un graphe où les nœuds sont des tâches et les connexions les dépendances entre les tâches
  - Tâches **séquentielles** ou tâches **parallèles** (parallélisme mixte)
- Pour ordonner les tâches, il faut affecter une date de départ à chaque tâche qui permette de satisfaire les dépendances (une tâche ne peut être démarrée tant que toutes les tâches dont elle dépendent ont été terminées)
  - Minimiser le temps d'exécution global
  - Algorithmes statiques et dynamiques



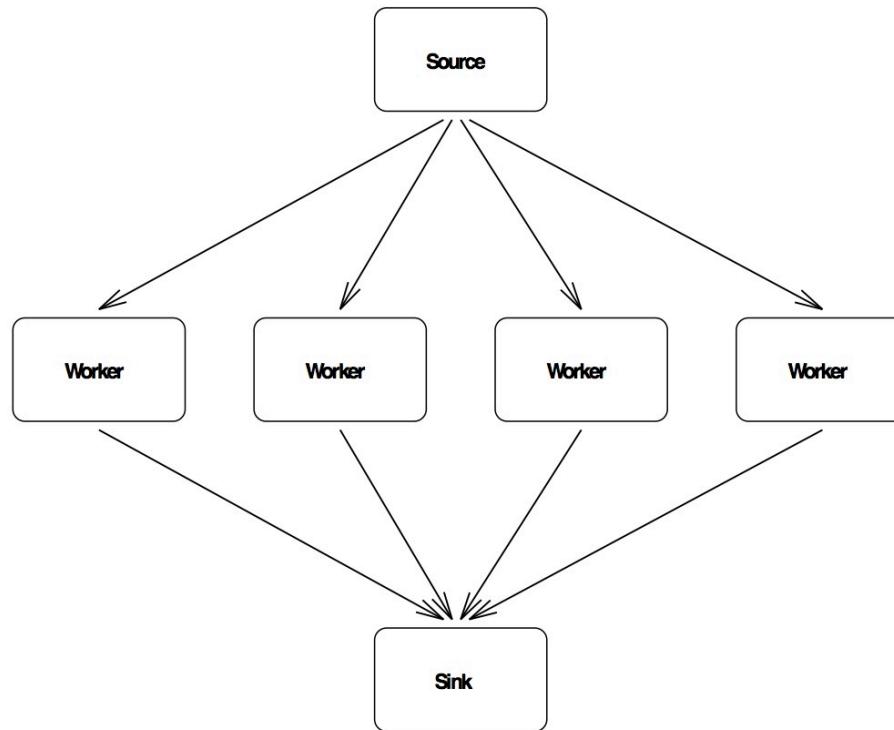
# Parallélisme fonctionnel



- Ordonnancement statique ou dynamique
- **Limitations**
  - **Coût de démarrage**
    - Au départ (et à la fin du pipeline) peu de tâches actives
  - **Equilibrage des charges**
    - Si des tâches ont des coûts plus importants, difficile d'équilibrer le travail entre les processeurs (penser à les re-découper si possible)
  - **Extensibilité**
    - Limitation sur le nombre de processeurs qui peuvent être utilisés

# Task farming

- **Source**
  - Divise les tâches initiales entre les workers et assure l'équilibrage
- **Worker**
  - Reçoit la tâche de la source, effectue le travail et passe le résultat
- **Sink**
  - Reçoit les tâches terminées des workers et collecte les résultats partiels



# Comment trouver le parallélisme ?

## Partir d'un langage séquentiel ?

- L'application a un parallélisme intrinsèque
- Le langage de programmation choisi ne possède pas d'extension "parallèle"

→ Le compilateur, le système d'exploitation et/ou le matériel doivent se débrouiller pour trouver le parallélisme caché!

- Fonctionnement correct pour quelques applications *trivialement* parallèles (parallélisation de boucles imbriquées simples par exemple)
- Mais en général, résultats décevants et problèmes liés à la dynamicité (pointeurs en C par exemple)

# Parallelisme “automatique” dans les processeurs actuels

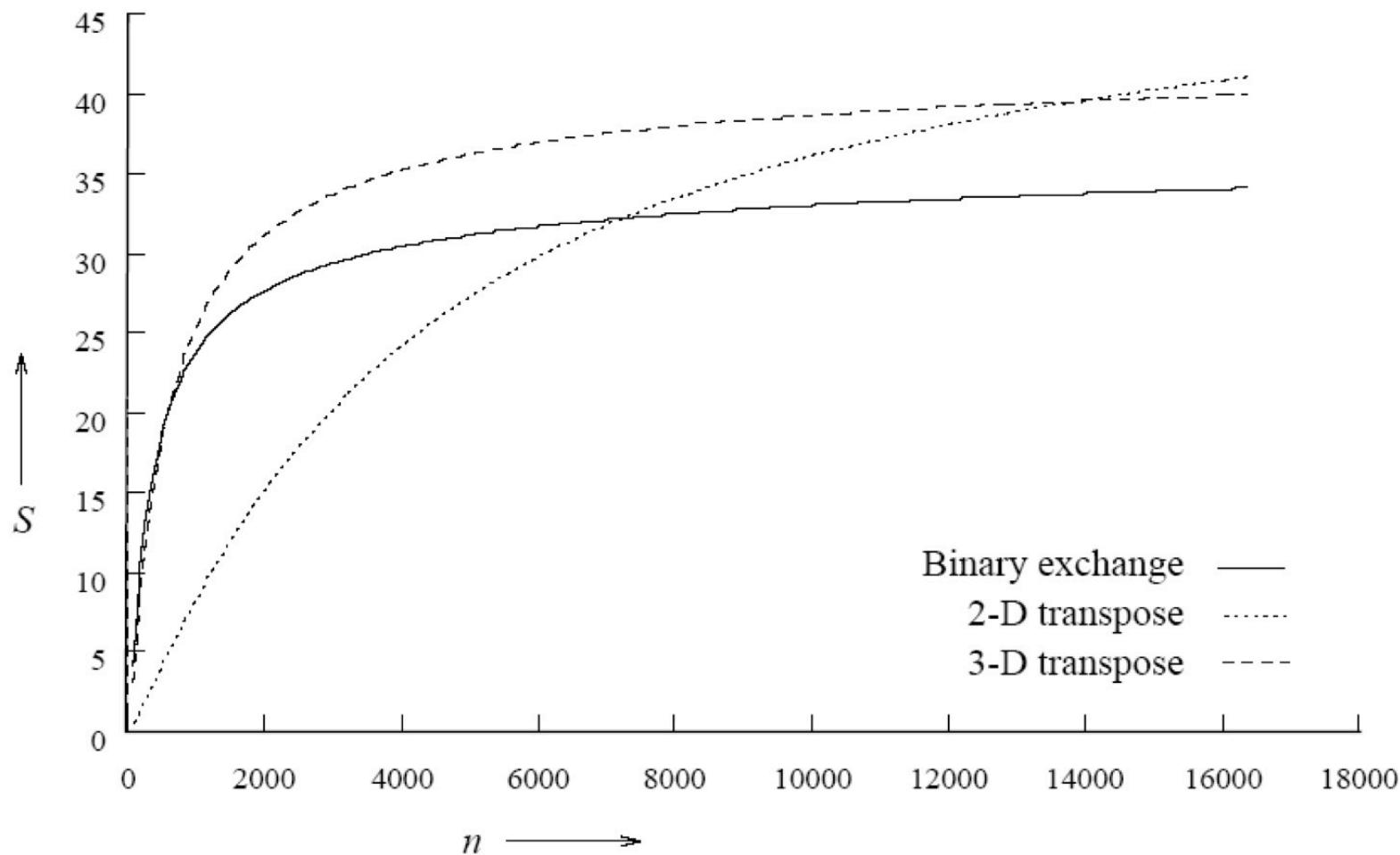
- **Parallélisme au niveau du bit (BLP, *Bit Level Parallelism*)**
  - Dans les opérations flottantes
- **Parallélisme d'instructions (ILP, *Instruction Level Parallelism*)**
  - Exécuter plusieurs instructions par cycle d'horloge
  - Super-scalar, VLIW, EPIC, ThLP (Thread level parallelism: multithreading)
- **Parallélisme des gestionnaires de mémoire**
  - Recouvrir les accès mémoire avec le calcul (prefetch)
  - Opérations vectorielles en parallèle ( $A[*] \leftarrow 3 \times A[*]$ )
- **Parallélisme au niveau du système**
  - Exécuter des tâches différentes sur des processeurs (ou des cores) différents
  - fork [ func1(), func2() ], join [\*]
- **Limites à ce parallélisme “implicite”**
  - Niveau d'intelligence des processeurs et des compilateurs
  - Complexité des applications
  - Nombre d'éléments en parallèle

# Autre approche: la coopération

## Le programmeur et le compilateur travaillent ensemble

- L'application a un parallélisme intrinsèque
- Le langage possède des extensions permettant d'exprimer le parallélisme
- Le compilateur va traduire le programme pour des unités multiples
- Le programmeur donne des conseils au compilateur sur les zones qu'il faut optimiser, quelles sont les boucles parallèles, ...
- Le compilateur, en partant des informations qu'il possède sur le matériel (taille des caches, nombre d'unités parallèles, informations sur les performances), va pouvoir générer un code performant.

```
#pragma omp parallel for private(i,j)
for i = 0 to n-1 {
    y[i] = 0;
    for j = 0 to n-1
        y[i] = y[i] + A[i,j] * x[j];
}
```



# ANALYSE DE PERFORMANCES

# Besoin de modèles analytiques des programmes parallèles

- Un programme séquentiel peut être évalué en fonction de son **temps d'exécution** donné en fonction de la **taille de ses données d'entrées**
- Un programme parallèle a son temps qui dépend d'autres éléments
  - Nombre de processeurs utilisés
  - Leur vitesse relative
  - La vitesse des communications entre eux⇒ Un programme parallèle ne peut être évalué indépendamment de ces éléments
- **Quelques mesures intuitives**
  - Le *wall time* obtenu pour résoudre un problème donné sur une plate-forme parallèle donnée
  - Quel est le gain obtenu en vitesse par rapport au temps séquentiel:  
**l'accélération**

# Facteurs influençant les performances

- L'algorithme doit être parallélisable !
- Le volume des données sur lesquelles il s'applique doit être suffisamment important par rapport au nombre de processeurs utilisés
- Les surcoûts dus aux synchronisations et aux conflits d'accès mémoire peuvent réduire les performances
- L'équilibrage des charges entre les processeurs
- L'utilisation d'algorithmes parallèles peut augmenter la complexité des algorithmes parallèles par rapport aux algorithmes séquentiels
- La répartition des données entre des unités mémoires multiples peut réduire les contentions mémoires et améliorer la localité des données, ce qui peut conduire à des gains en performances

# Sources de surcoûts

- **Interactions entre les processus**

- Un algorithme parallèle non-trivial va nécessiter des interactions entre les processus durant son exécution (synchronisations, échanges de données intermédiaires)
- Les communications sont généralement les sources de pertes de performances les plus importantes

- **L'attente**

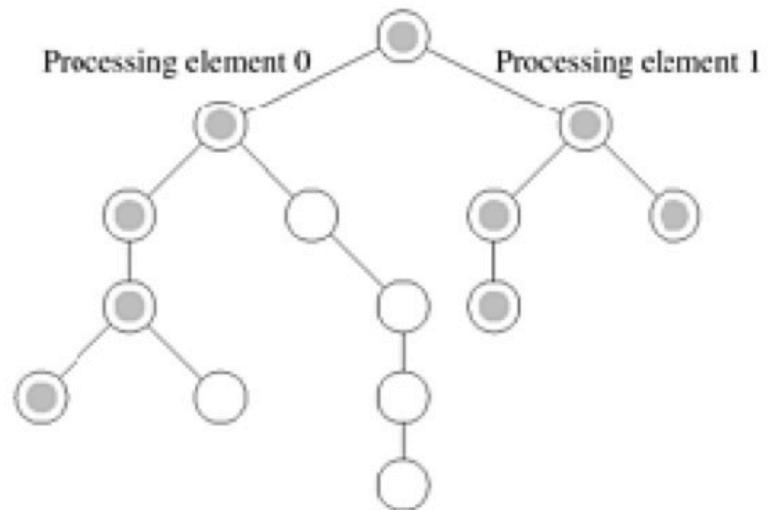
- A cause de nombreuses raisons comme
  - un déséquilibrage des charges,
  - les synchronisations,
  - la présence de parties séquentielles

# Accélération (speedup)

- Quel gain en performance peut être obtenu en parallélisant une application par rapport à l'implémentation séquentielle ?
- L'accélération est une mesure qui capture le bénéfice relatif à résoudre un problème en parallèle
- L'accélération  $S$  est le ratio entre le temps pour résoudre un problème sur un processeur unique sur le temps pour résoudre un problème sur une machine parallèle à  $p$  processeurs
  - Même type de processeurs entre l'exécution parallèle et séquentielle
  - On doit (normalement) prendre le meilleur algorithme séquentiel pour résoudre le même problème

# Accélération superlinéaire

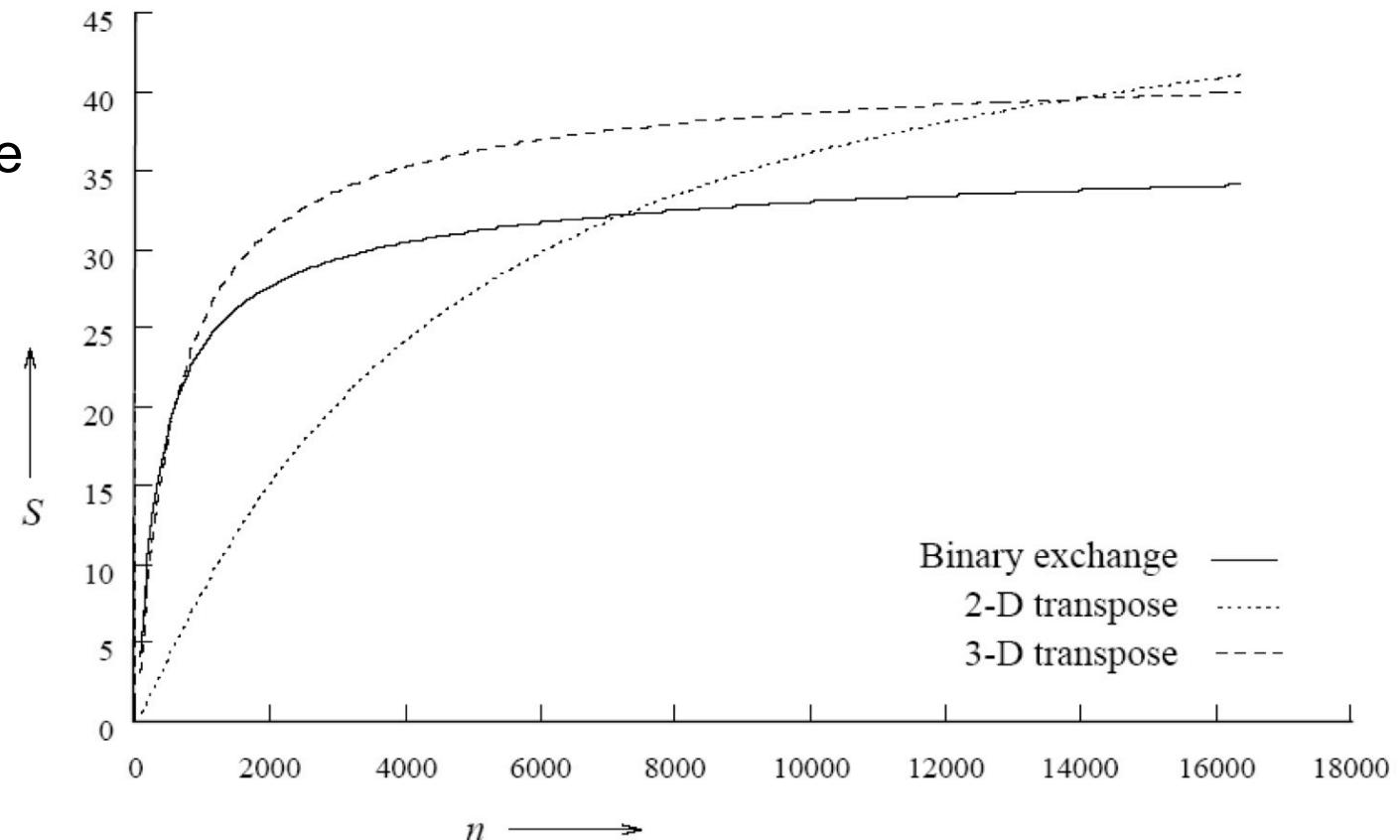
- On constate parfois des accélérations supérieures à p
- Cela arrive lorsque
  - Le travail effectué par un algorithme séquentiel est supérieur à celui de sa version parallèle
  - Exemple: recherche, algorithmes dans les arbres



- A cause de fonctionnalités matérielles qui désavantagent la version séquentielle
  - Si les données entrent dans les caches pour la version parallèle
    - Les performances des mémoires de plus grandes tailles sont moins importantes

# Extensibilité des systèmes parallèles

- **Extrapoler les performances**
  - Comment passer d'un petit problème sur un petit système
  - à un gros problème sur une configuration de taille plus importante
- **Exemples:** 3 algorithmes pour calculer une FFT à  $n$  points sur 64 processeurs
- Choix de l'algorithme en fonction des configurations



# Extensibilité des programmes parallèles

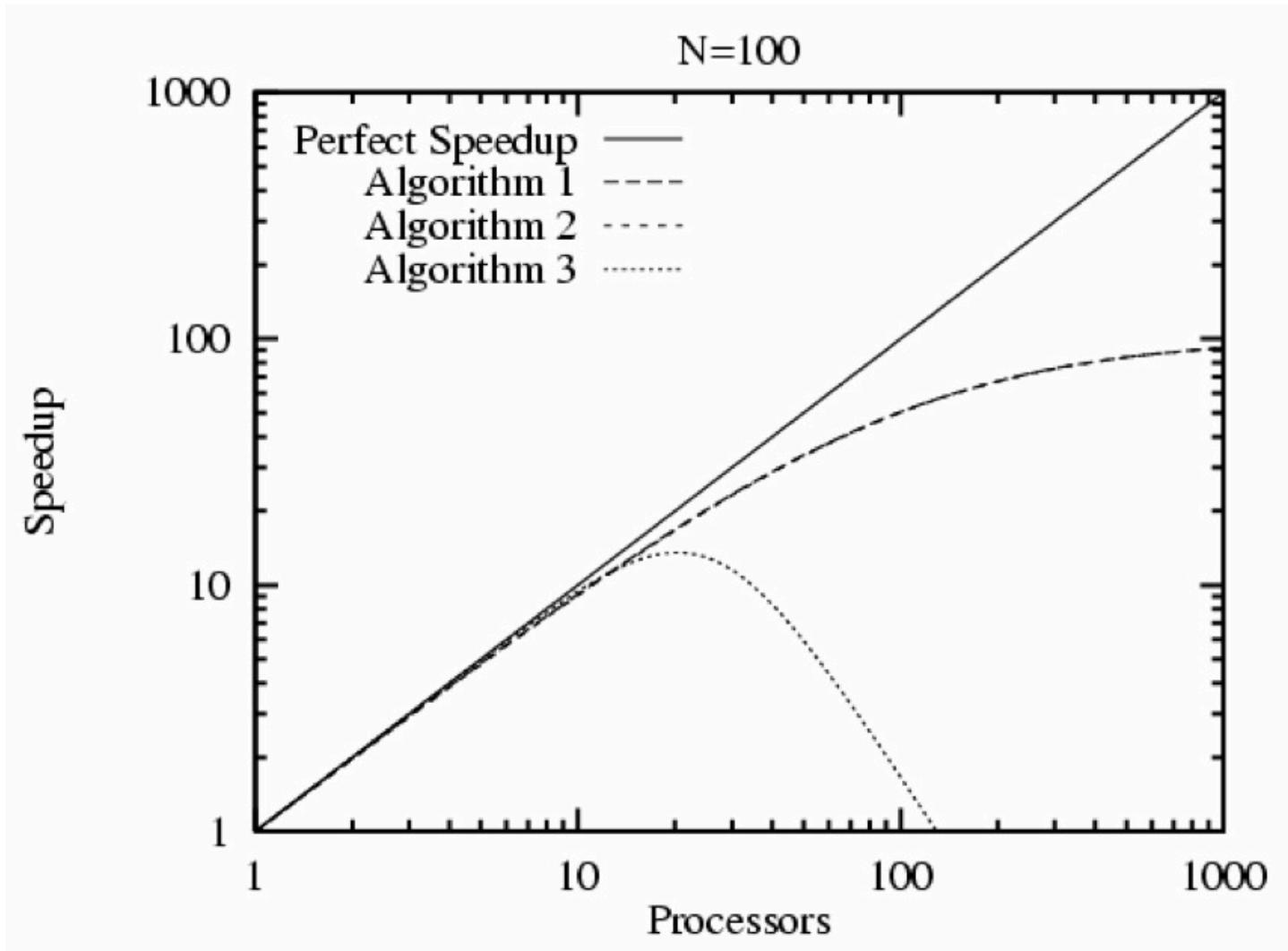
- On lit dans les papiers des observations comme
  - « On a implémenté un algorithme sur la machine parallèle X qui a obtenu une accélération de 10.8 sur 12 processeurs avec une taille de problème égale à 100. »
- Un point sur une courbe !
  - Qu'est-ce qui se passe si on a 100, 1000 processeurs ?
  - Qu'est-ce qui se passe si on a des données de taille 10, 1000 ?

# Extensibilité des programmes parallèles

- Trois modèles théoriques de performances
  - $T = N + N^2 / P$ 
    - Cet algorithme partitionne  $N^2$  calculs mais réplique aussi  $N$  autres calculs
    - Pas d'autres sources de surcoût
  - $T = (N + N^2) / P + 100$ 
    - Cet algorithme partitionne tous les calculs et ajoute un coût supplémentaire de 100
  - $T = (N + N^2) / P + 0.6 P^2$ 
    - Cet algorithme partitionne tous les calculs et ajoute un coût supplémentaire de  $0.6 P^2$
- Tous ces algorithmes ont une accélération de 10.8 sur 12 processeurs pour  $N = 100$  !

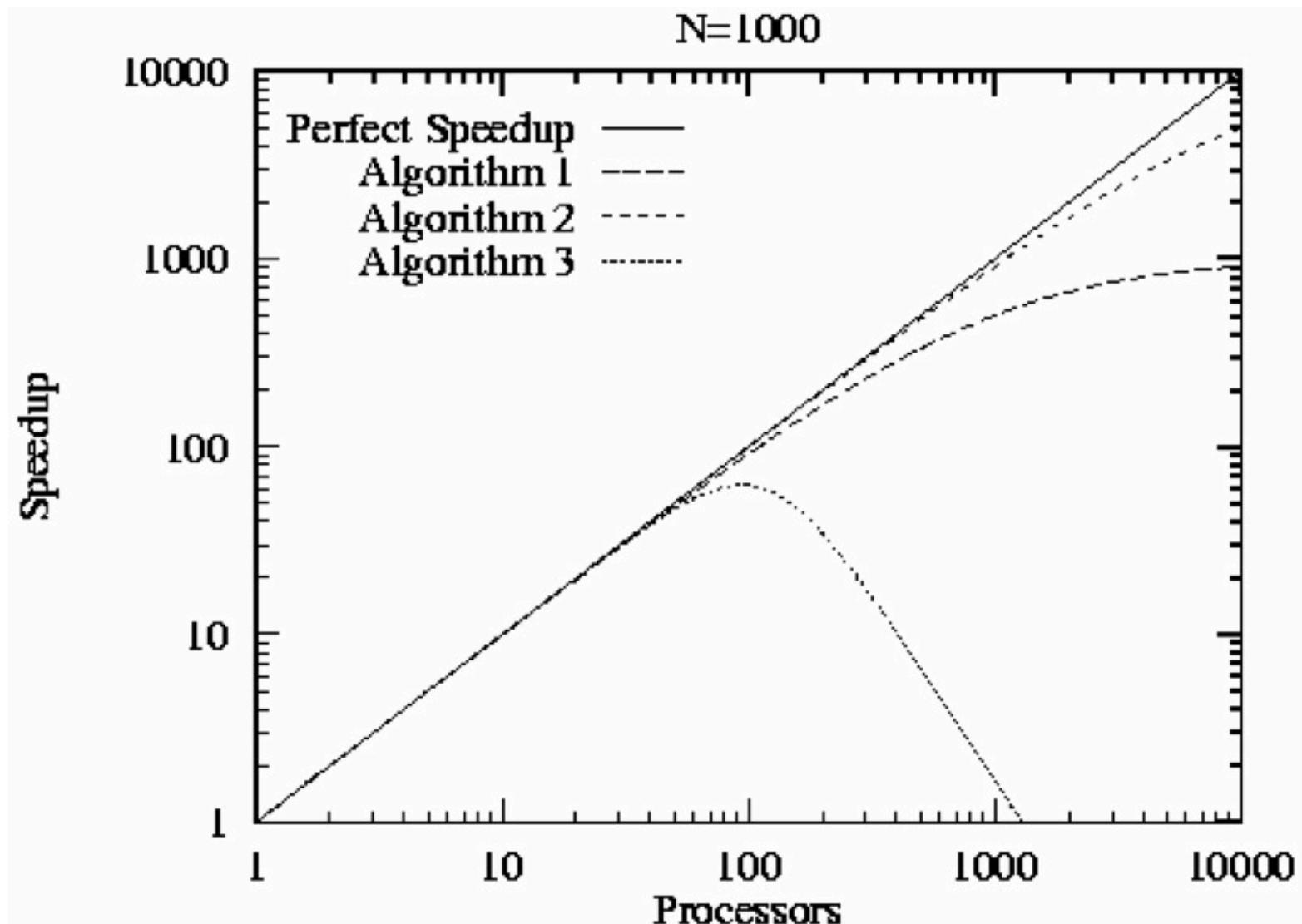
# Extensibilité des programmes parallèles

- Si on augmente le nombre de processeurs pour  $N = 100$



# Extensibilité des programmes parallèles

- Si on augmente le nombre de processeurs pour  $N = 1000$



# Améliorer les performances réelles

## Les performances de crête augmentent exponentiellement

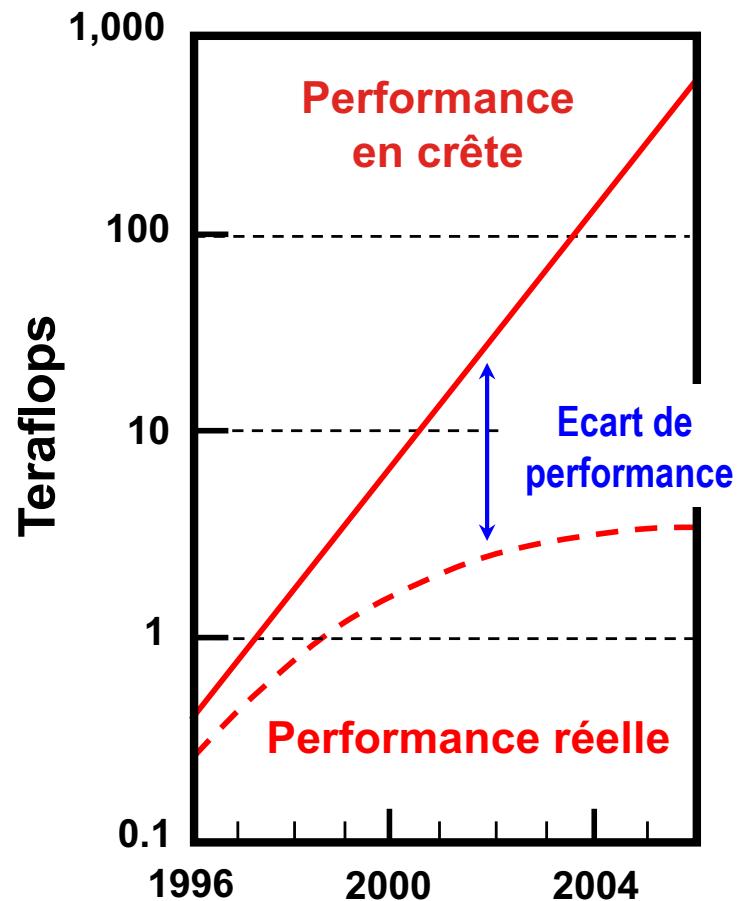
- En 1990's, les performances de crête ont augmenté 100x; dans les années 2000, elles augmenteront 1000x

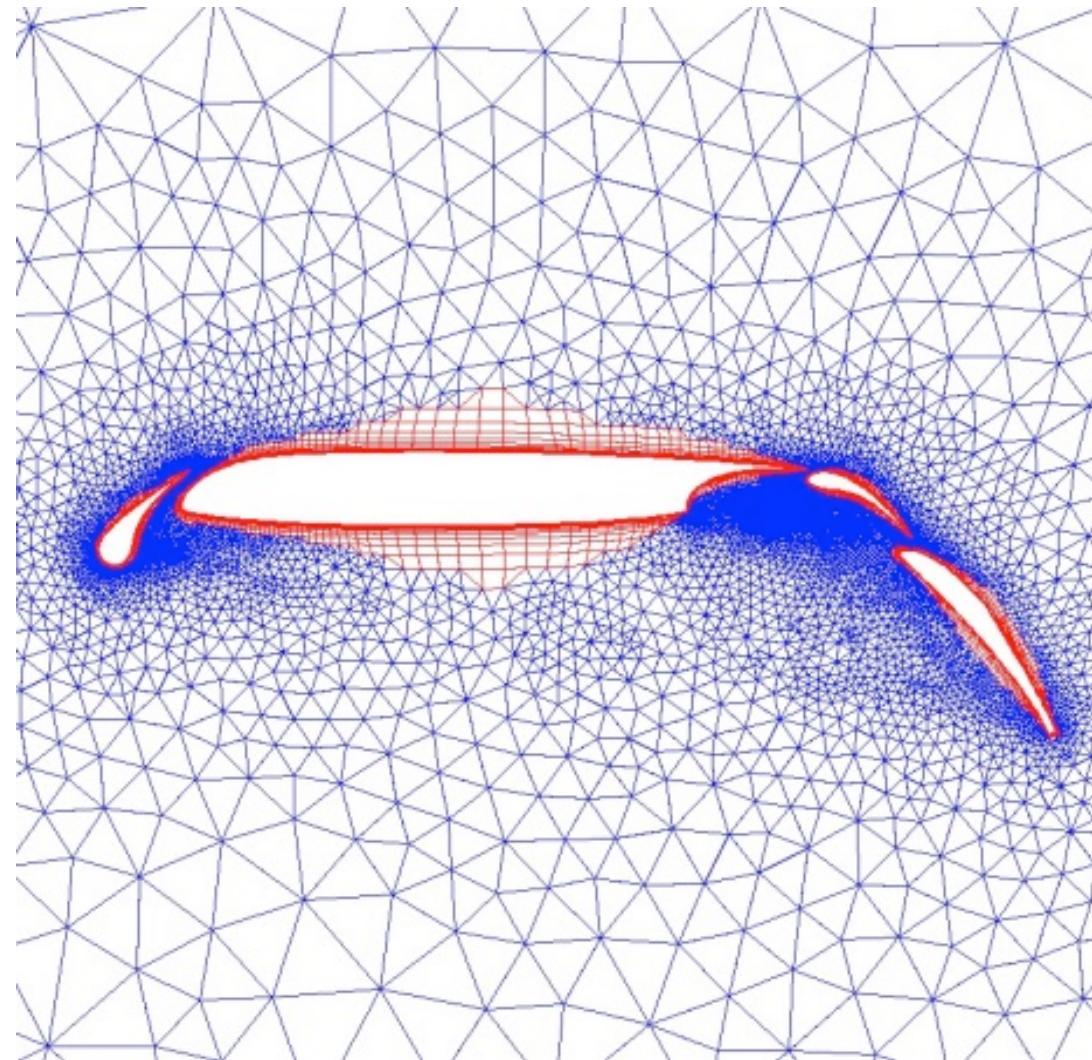
**Mais l'efficacité (les performances par rapport aux performances en crête) ont plutôt déclinées**

- 40-50% sur les supercalculateurs vectoriels des années 1990
- Maintenant proches de 5-10% sur les supercalculateurs d'aujourd'hui

## Réduire l'écart ...

- Algorithmes qui obtiennent des performances sur un seul processeur et sont extensibles sur plusieurs milliers
- Modèles de programmation plus efficaces et des outils pour les machines massivement parallèles





# CONCLUSION

# Parallélisme en 2016

- Tous les vendeurs de processeurs produisent des processeurs multicore
  - Toutes les machines seront bientôt parallèles
  - Pour continuer à doubler la puissance il faut doubler le parallélisme
- Quelles applications vont (bien) tirer partie du parallélisme ?
  - Est-ce qu'il faudra les redévelopper *from scratch* ?
- Est-ce que tous les programmeurs devront être des programmeurs de machines parallèles ?
  - Il faut des nouveaux modèles logiciels
  - Essayer de cacher le parallélisme au maximum
  - Le comprendre !
- L'industrie parie sur ces changements ...
- ... mais encore beaucoup de travail à effectuer

# Challenges à relever

- Les applications parallèles sont souvent très sophistiquées
  - Algorithmes adaptatifs qui nécessitent un équilibrage dynamique
- Le parallélisme multi-niveaux est difficile à gérer
- La taille des nouvelles machines donnent des problèmes d'efficacité
  - Problèmes d'extensibilité
  - Sérialisation et déséquilibrage de charge
  - Goulots d'étranglement et en communication et/ou en entrées/sorties
  - Parallélisation insuffisante ou inefficace
  - Fautes et/ou pannes
  - Gestion de l'énergie
- Difficulté d'obtenir les performances les meilleures sur les nœuds eux-mêmes
  - Contention pour la mémoire partagée
  - Utilisation de la hiérarchie mémoire sur les processeurs multi-cores
  - Influence du système d'exploitation

# Conclusions

- **L'ensemble des machines parallèles est constituée d'un ensemble (très) large d'éléments**
  - depuis des unités parallèles dans les processeurs
  - Jusqu'à des datacenters connectés à travers le monde
- **Champ d'étude du parallélisme**
  - Architectures
  - Algorithmes
  - Logiciels, compilateurs, supports d'exécution
  - Bibliothèques
  - Environnements
- **Changements historiques importants**
  - Jusque dans les années 90, réservées à des gros calculs de simulation
  - Aujourd'hui, parallélisme dans tous les processeurs (des téléphones aux supercalculateurs), parallélisme dans la vie courante (iPad, Google !)
  - Evolution des architectures à venir

# Quelques références

## Parallel Programming – For Multicore and Cluster System

T. Rauber, G. Rünger

## Sourcebook of Parallel Computing

J.J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, A. White

## Parallel Algorithms

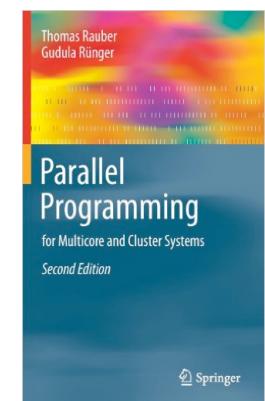
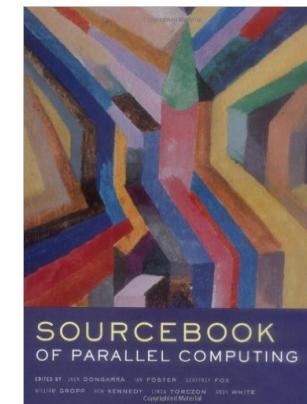
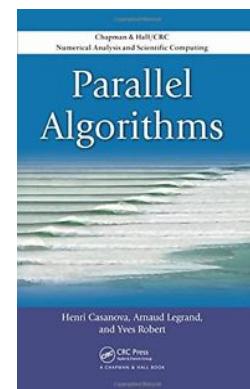
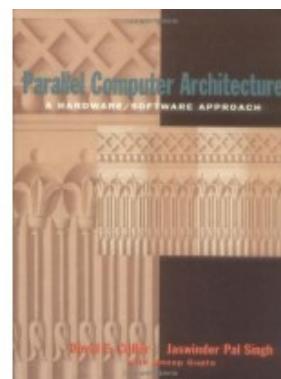
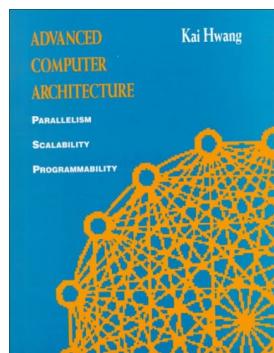
H. Casanova, A. Legrand, Y. Robert

## Parallel Computer Architecture

D.E. Culler, J. Pal Singh

## Advanced Parallel Architecture - Parallelism, Scalability, Programmability

K. Hwang



# Quelques références, suite

- **Les supercalculateurs relèvent le défi - La Recherche**  
Numéro spécial de la recherche, Nov. 2012
- **Super-ordinateurs – Aux extrêmes du calcul**  
Numéro spécial de la recherche, Nov. 2011
- **Cours en ligne**
  - **Why parallel, why now**, Dr Clay Breshears, Intel
  - **Architecture et Système des Calculateurs Parallèles**, F. Pellegrini, LaBRI
  - **Applications of Parallel Computers**, J. Demmel, U.C. Berkeley CS267
  - K. Yelick



Frédéric Desprez  
[Frederic.Desprez@inria.fr](mailto:Frederic.Desprez@inria.fr)