Introduction au calcul haute performance

P. Fortin
Programmation parallèle avancée (PPA)
Master 2 Informatique SAR-STL, UPMC

Plan du cours 1

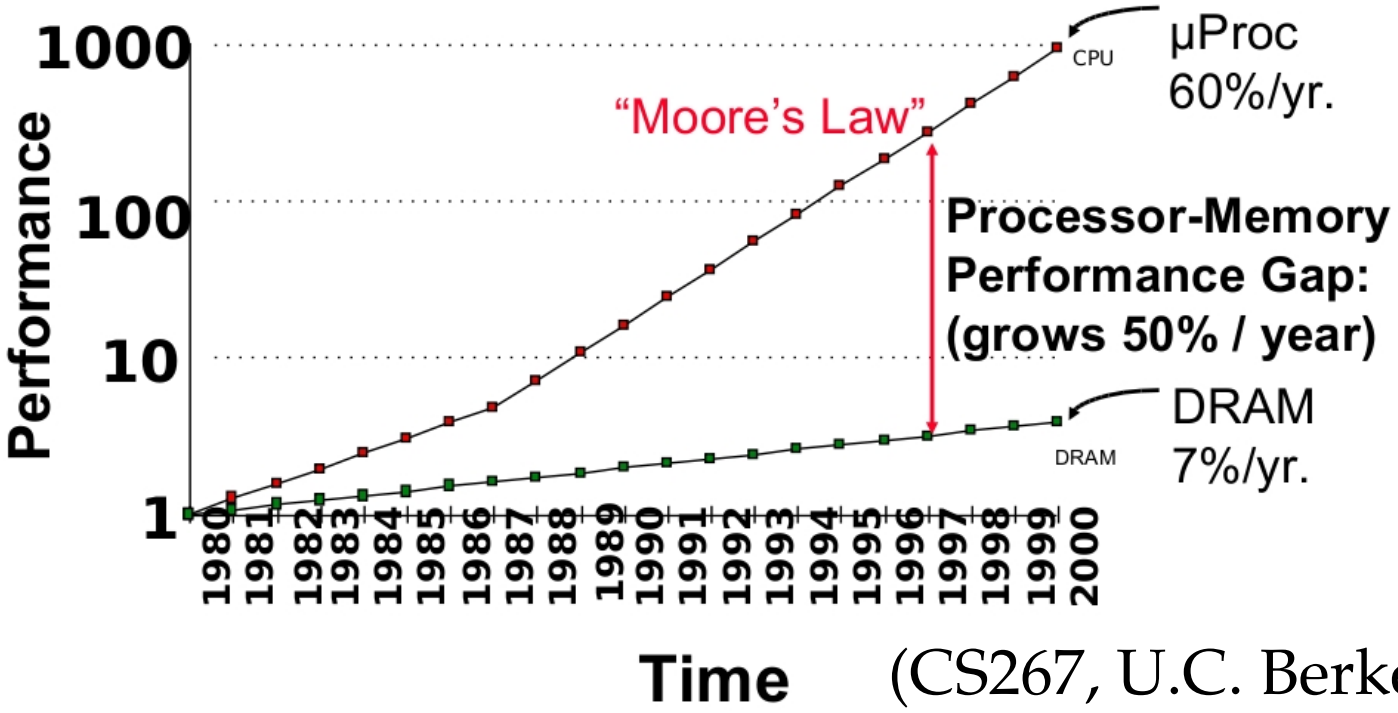
- Architectures monocoeurs
- Architectures multicoeurs homogènes
- Architectures multicoeurs hétérogènes
- Applications de référence

Bibliographie et remerciements

- U.C. Berkeley CS267 (J. Demmel, http://www.cs.berkeley.edu/~demmel/cs267_Spr09)
- Parallel Programming in C with MPI and OpenMP (M. J. Quinn)
- Architectures et Systèmes des Calculateurs Parallèles (F. Pellegrini, http://uuu.enseirb.fr/~pelegrin/enseignement/enseirb/archsys/cours/c.pdf)
- Univ. Tennessee Knoxville CS 594 (J. Dongarra, http://www.cs.utk.edu/~dongarra/WEB-PAGES/cs594-2009.htm)

Remarque préliminaire : le « Memory Gap »

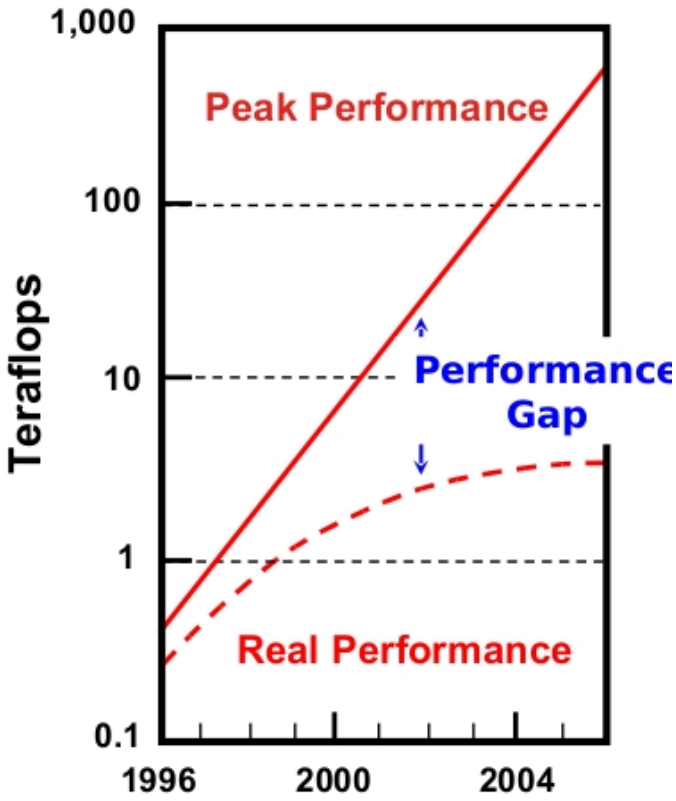
- Ecart processeur / latence mémoire
  (croissance de la bande passante : 23% / an)
- Nécessite algorithmes adaptés : minimiser les accès mémoire coûteux plutôt que les calculs...
- Compensé (partiellement) par la hiérarchie mémoire



(CS267, U.C. Berkeley)

Remarque préliminaire (2) : performance crête et performance réelle

- Performance crête augmente exponentiellement (loi de Moore)
- Mais l'efficacité (= performance d'un programme par rapport à la perf. crête) décline !
  - ➤ 40-50 % sur ordinateurs vectoriels (années 90)
  - ➤ 5-10 % sur les ordinateurs parallèles actuels
- Solutions :
  - ➤ Mathématiques et algorithmique pour haute performance sur 1 processeur et pour passage à l'échelle en parallèle
  - ➤ Outils et modèles de programmation plus efficaces pour le parallélisme (massif)



(CS267, U.C. Berkeley)

Architectures monocoeurs : la hiérarchie mémoire

- But : exploiter la localité des accès aux données présente dans la plupart des programmes
  - Localité spatiale : accès à de nouvelles données « proches » en mémoire des données précédentes → *ligne* de cache, page mémoire (TLB)
  - Localité temporelle : réutilisation de données déjà accédées
- Hiérarchie mémoire d'un processeur :
  - Registres : vitesse ~ processeur / taille ~ quelques octets
  - Caches L1 : ~ 1ns / ~ Ko
  - Caches L2 : ~ 10ns / ~ Mo (parfois cache L3)
  - Mémoire principale : 100 ns / ~Go
  - Disque dur : 10 ms / ~To
  - Et aussi TLB (*Translation Lookaside Buffer*) : cache de la table des pages (traduction adresse virtuelle → adresse physique)
- Hiérarchie mémoire **implicite** pour le programmeur !

Vectorisation

- Processeur scalaire : au plus une instruction / cycle
- Processeurs superscalaires : 3 à 5 instructions / cycle (plusieurs pipelines d'instruction)
- Processeurs vectoriels : SIMD
- Instructions vectorielles :
  - SSE / SSE{2,3,4,5} (128 bits) et AVX (256 bits) pour ix86 et x86_64 (Intel et AMD)
  - Altivec pour powerpc* (IBM)
  - SPE du processeur Cell :
    - 128 registres vectoriels de 128 bits : 4 floats ou 2 double
    - exemple d'instruction vectorielle pour le SPE :
      spu_madd: vector multiply and add
      d = spu_madd(a, b, c)
      avec d, a, b et c de type *vector float* (ou *vector double*)

Optimisation au niveau programmation

- Quelques bonnes pratiques dans les sections de code à optimiser (corps des boucles internes) :
  - Éviter les tests (pipelines d'instructions)
  - Minimiser le nombre de calculs quitte à utiliser plus de variables temporaires (optimisation de l'utilisation des registres par le compilateur, attention au « *register spilling* »)
  - Indexation simple des tableaux et des boucles
  - Eviter les indirections multiples (pointeurs, tableaux...) : privilégier les accès mémoires contigus
  - Déroulage (modéré) des boucles
  - Utilisation de fonctions « inlines » ou de macros
  - En C++ :
    - Templates : ok (génération du code à la compilation)
    - Polymorphisme : à éviter (gestion à l'exécution du code)

Optimisation à la compilation

- Optimisation à la compilation :
  - Exemple de gcc :
    - -O / -O1 : réduit taille du code et temps d'exécution
    - -O2 : augmente temps de compilation et performance du code
    - -O3 : optimise encore plus
      - « inlining » des fonctions
      - vectorisation automatique (avec -maltivec sur powerpc*, et -msse/-msse2 sur ix86 and x86_64)
    - -funroll-loops : déroulage des boucles dont le nombre d'itérations est connu → code plus gros, et plus rapide (ou pas !) sur CPU
    - -ffast-math : résultat du programme potentiellement différent de la norme IEEE
    - -mtune=cpu_type / -march=cpu_type : optimisation pour l'architecture spécifiée (dont (ré-)ordonnancement des instructions)

Optimisation à l'exécution

- Exécution d'un code sur un processeur moderne :
  - Prédiction de branchement (statiques ou dynamiques)
  - Ré-ordonnancement dynamique des instructions (exécution *out-of-order*)
  - Simultaneous multithreading (SMT) ou Hyperthreading :
    - Partager des ressources du processeur par 2 threads en même temps (ex. : pipelines d'un processeur superscalaire)
    - Recouvrement des accès mémoire et accroissement de l'utilisation du processeur et du « débit d'instructions »
    - Le plus souvent : *2-way SMT*, mais aussi possible : *4-way SMT* (Intel Xeon Phi, IBM POWER7)
  - Politique de gestion des caches et des pages mémoires

Exemple d'optimisation d'algorithme sur architecture monocoeur : le produit matriciel (d'après J. Demmel, U.C. Berkeley)

- Pourquoi le produit matriciel ?
  - Goulot d'étranglement pour l'algèbre linéaire dense
  - 1 des 13 motifs (voir plus loin)
  - Proches d'autres algorithmes (réutilisation des mêmes idées)
  - Meilleur exemple pour les gains d'optimisation (intensité de calcul – voir plus loin, FMA...)
  - L'algorithme de base du calcul haute performance

Modèle de mémoire simplifié

- 2 niveaux dans la hiérarchie mémoire : lent et rapide
- Toutes les données initialement dans la mémoire lente
- Variables :
  - m = nombre de mots mémoire déplacés entre la mémoire lente et la mémoire rapide
  - tm = temps pour 1 accès à la mémoire lente
  - f = nombre d'opérations arithmétiques
  - tf = temps pour 1 opération arithmétique << tm
  - **Intensité de calcul q = f/m** : nombre moyen de flops par accès mémoire (lente)
    → clé de l'efficacité de l'algorithme

Modèle de mémoire simplifié (2)

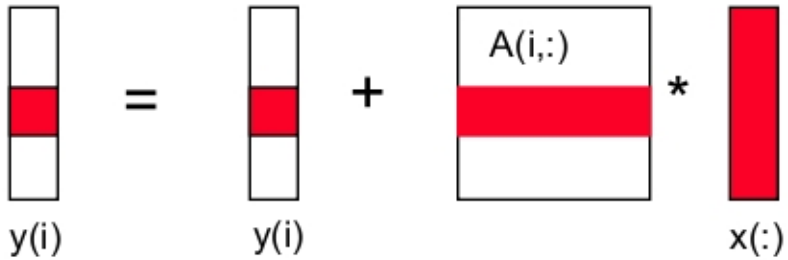
- Temps minimum si toutes les données sont dans la mémoire rapide → temps min = f * tf
- Temps réel :

$$f * tf + m * tm = f * tf * (1 + tm/tf * 1/q)$$

- **tm/tf =** ***Machine balance*** (« proportion » de la machine) → clé de l'efficacité de la machine
- Plus q est grand → plus le temps est proche du min
  - ➤ q >= tm/tf nécessaire pour obtenir au moins la moitié du meilleur temps (i.e. la moitié de la performance crête)

Echauffement : produit matrice-vecteur (1)

```
{implements y = y + A*x}
for i = 1:n
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



Hypothèse : la mémoire rapide peut contenir 3 vecteurs, mais pas 1 matrice entière.

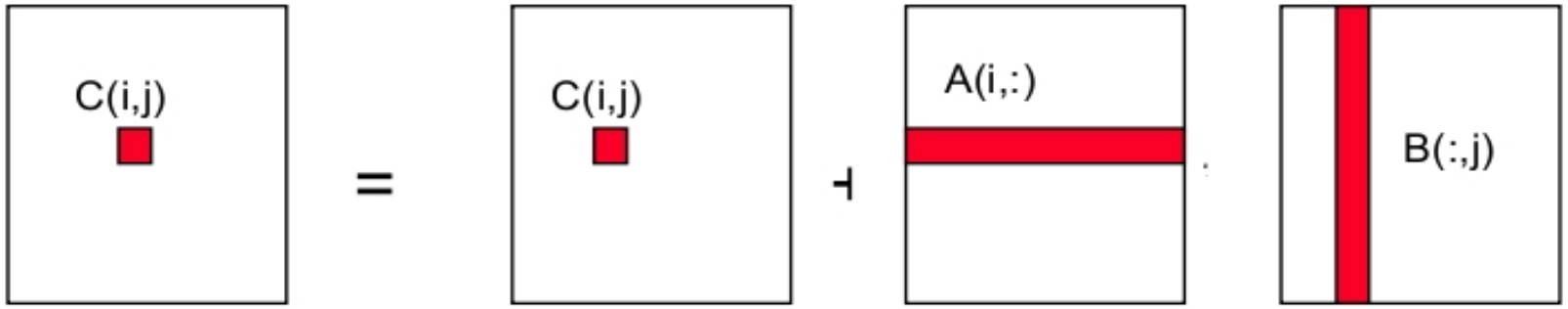
Echauffement : produit matrice-vecteur (2)

```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
      {read row i of A into fast memory}
      for j = 1:n
            y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

- m = 3*n + n*n
- f = 2 * n *n
- q = f/m ~ 2 (asymptotiquement)

  → produit matrice vecteur limité par la vitesse de la mémoire lente (*memory-bound*)

Produit matriciel naïf

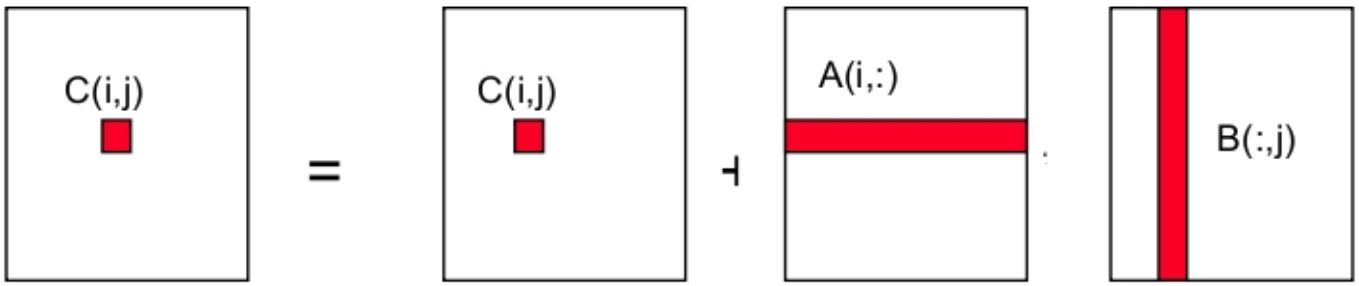



```
{implements C = C + A*B}
for i = 1 to n
      for j = 1 to n
            for k = 1 to n
                  C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

- 2*n^3 flops = O(n^3) pour 3*n^2 données
- q = 2*n^3 / 3*n^2 = O(n) asymptotiquement
  → potentiellement *compute-bound* pour n grand

Produit matriciel naif (2)

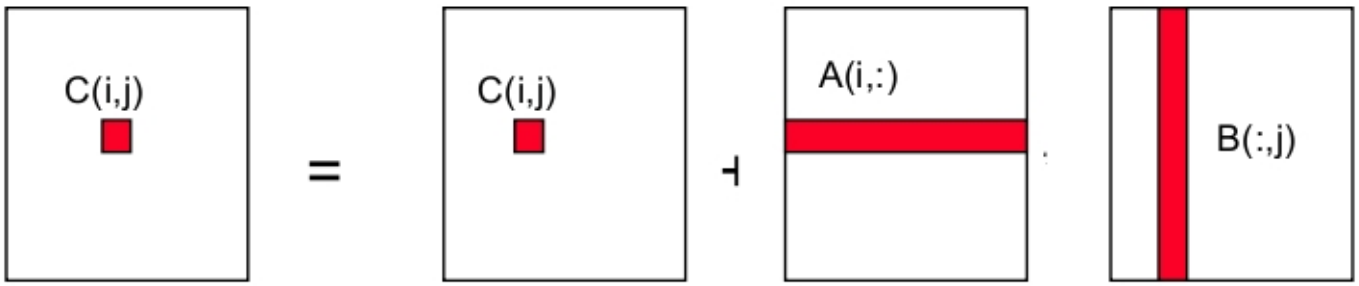
```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write C(i,j) back to slow memory}
```



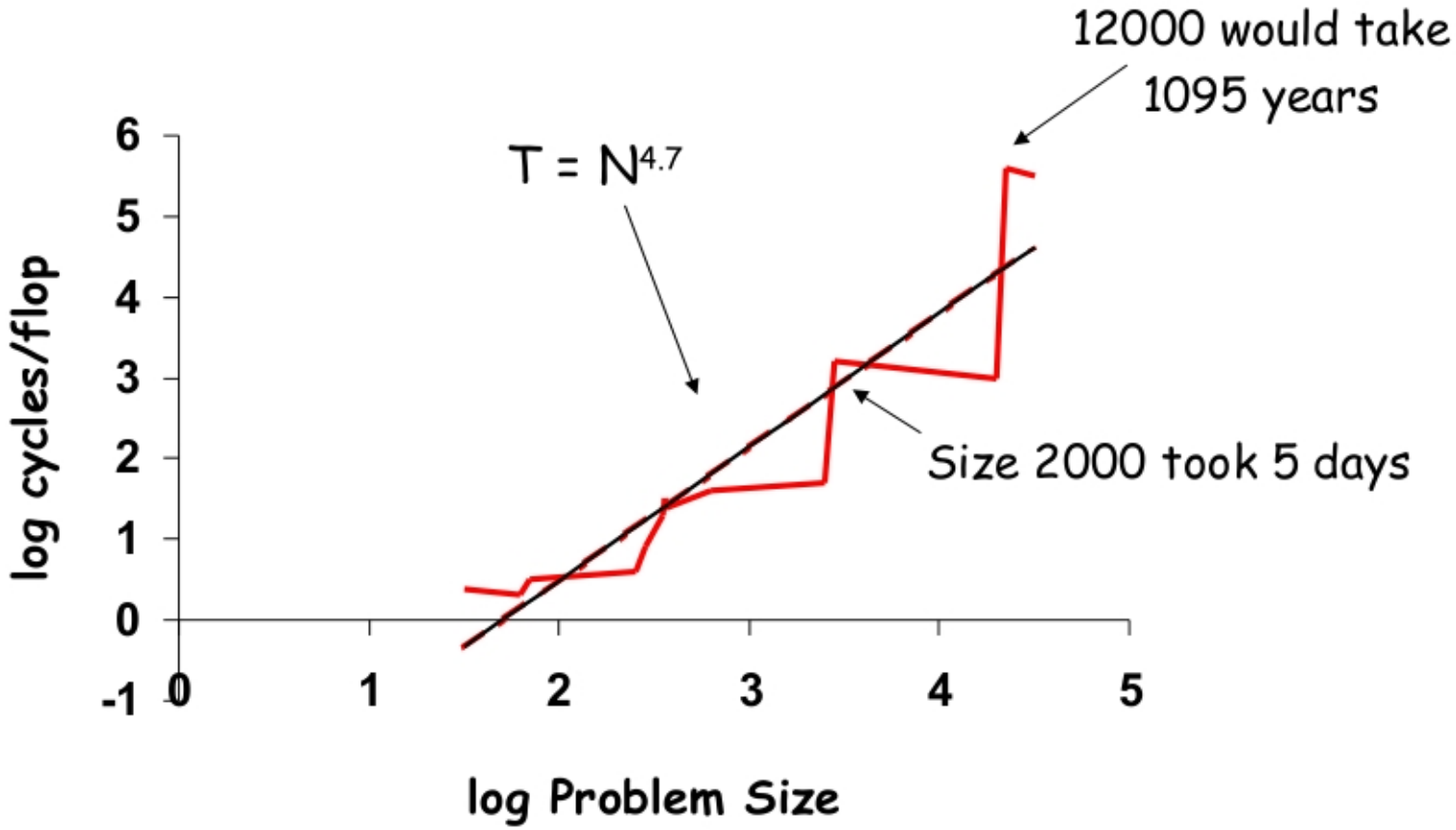
Produit matriciel naif (3)

- m = n^3 pour lire n^2 fois chaque colonne de B
  + n^2 pour lire A 1 fois
  + 2 n^2 pour lire et écrire chaque élément de C
  = n^3 + 3 n^2
- q = f/m = 2n^3 / (n^3 + 3n^2)

  q ~ 2 pour n grand → pas de gain / produit matrice-vecteur

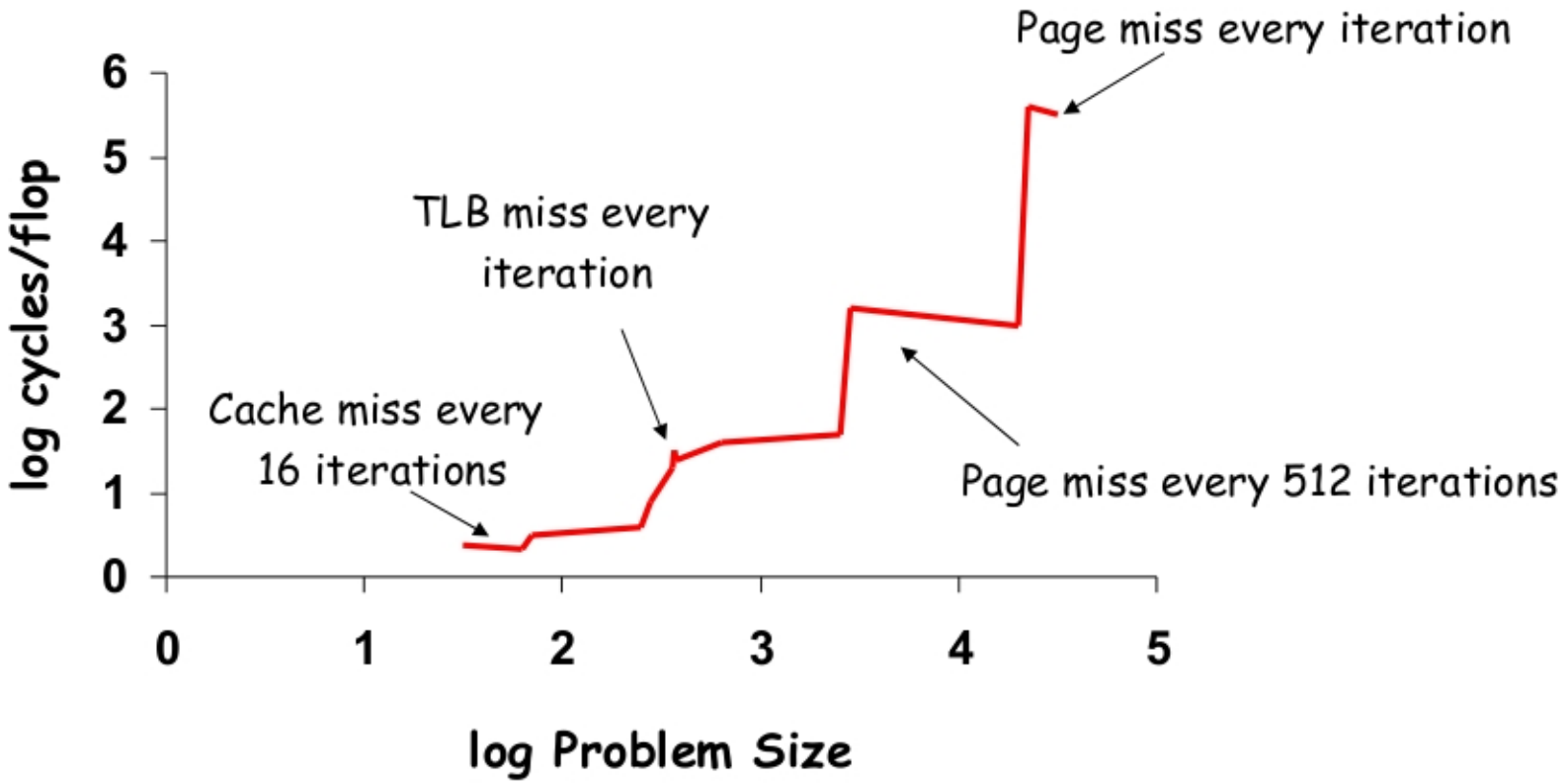


Performance sur RS/6000



Algorithme en O(N^3) → devrait avoir un nombre constant de cycles/flop. Mais ici performance en O(N^4.7) ...

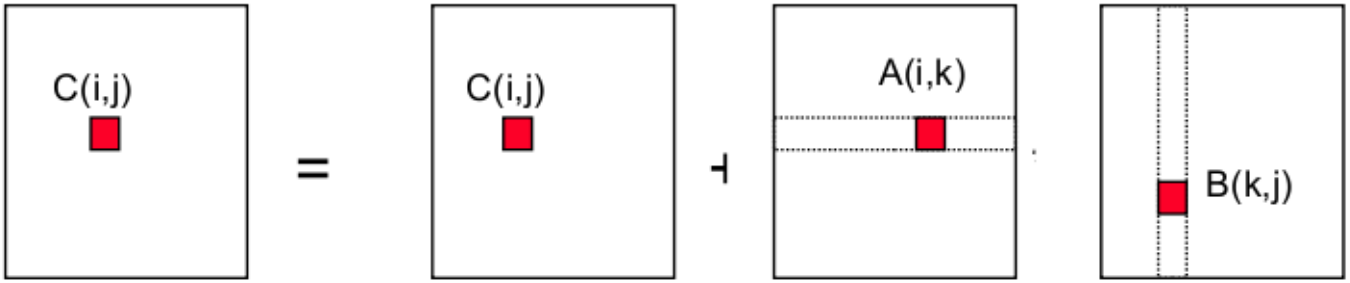
Performance sur RS/6000



Produit matriciel par blocs (*blocked/tiled*)

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where
b=n / N is called the block size

```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
            C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```



Produit matriciel par blocs (2)

- NxN blocs de taille bxb (avec b=n/N)
- f = 2*n^3
- m = N*n^2 lecture de N^3 blocs de B
  + N*n^2 lecture de N^3 blocs de A
  + 2n^2 lecture et écriture de chaque bloc de C
  = (2N + 2) * n^2
- q = f/m = 2n^3 / ((2N + 2) * n^2)

  q ~ n/N = b asymptotiquement
- **Augmentation de b → augmentation de la performance**
- Contrainte : la mémoire rapide doit pouvoir contenir 3 blocs
  Taille mémoire rapide Mf :
  3b^2 <= Mf soit **q ~ b <= sqrt(Mf/3)**

Routines BLAS

- Interface standard d'opérations d'algèbre linéaire

  www.netlib.org/blas, www.netlib.org/blas/blast-forum
- Historique :
  - BLAS1 (années 70) :
    - Opérations entre vecteurs : produit scalaire, saxpy (y=a*x+y), etc.
    - m=2*n, f=2*n → q~1 ou moins
  - BLAS2 (milieu des années 80) :
    - Opérations matrice-vecteur : produit matrice-vecteur, etc.
    - m=n^2, f=2*n^2 → q~2
    - Plus rapide que les BLAS1
  - BLAS3 (fin des années 80) :
    - Opérations matrice-matrice : produit matriciel, etc.
    - m<=3n^2, f=O(n^3) → q=f/m~n (quand n→+inf)
    - BLAS3 potentiellement bien plus efficace que BLAS2

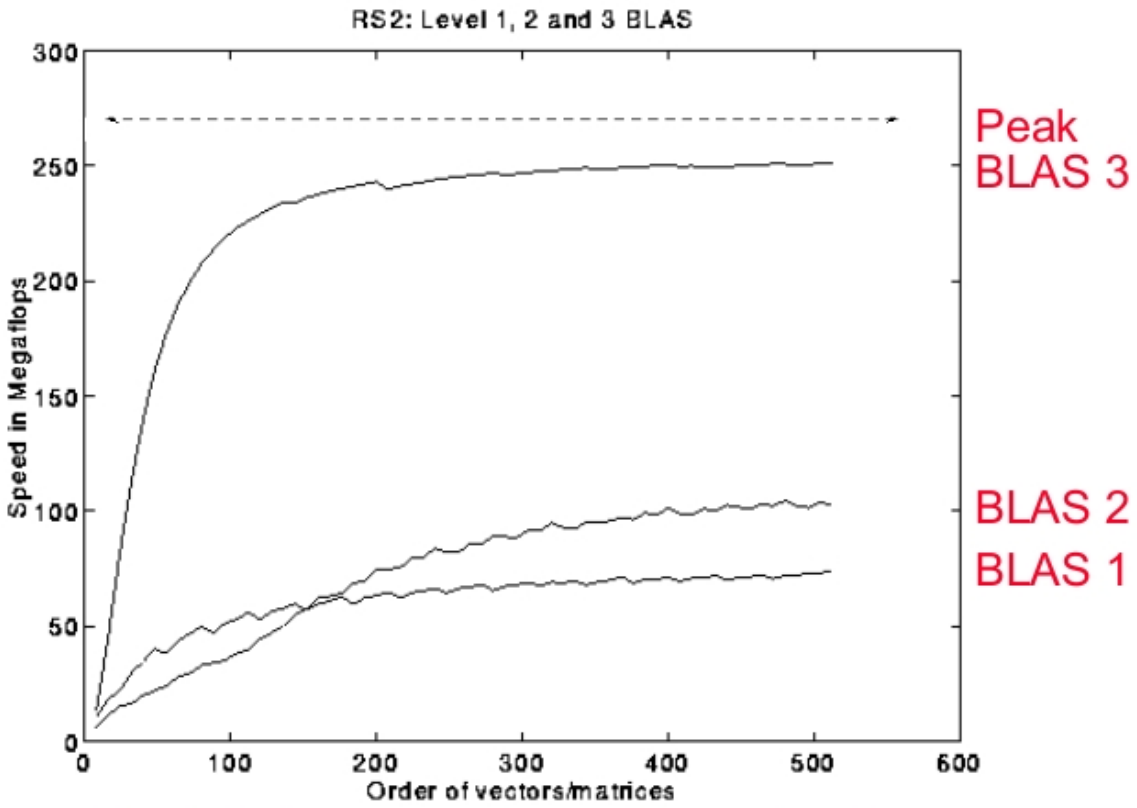
Routines BLAS (2)

- Implémentation de référence (non optimisée) en FORTRAN77 : http://www.netlib.org/blas/
- Implémentations constructeurs
  - IBM : ESSL, Intel : MKL, ...
- Implémentations génériques :
  - ATLAS (Automatically Tuned Linear Algebra Software) : configuration automatisée à l'architecture de la machine courante
  - GOTO BLAS : minimisation des *TLB misses*
- Utilisation des BLAS :
  - LAPACK (Linear Algebra PACKage) : bibliothèque standard d'algèbre linéaire (machines séquentielles ou à mémoire partagée)
  - ScaLAPACK (Scalable LAPACK) : version parallèle de LAPACK pour machines à mémoire distribuée

Routines BLAS (3)

Performance sur 1 IBM RS6000/590 (CS267, U.C. Berkeley)

Peak speed = 266 Mflops

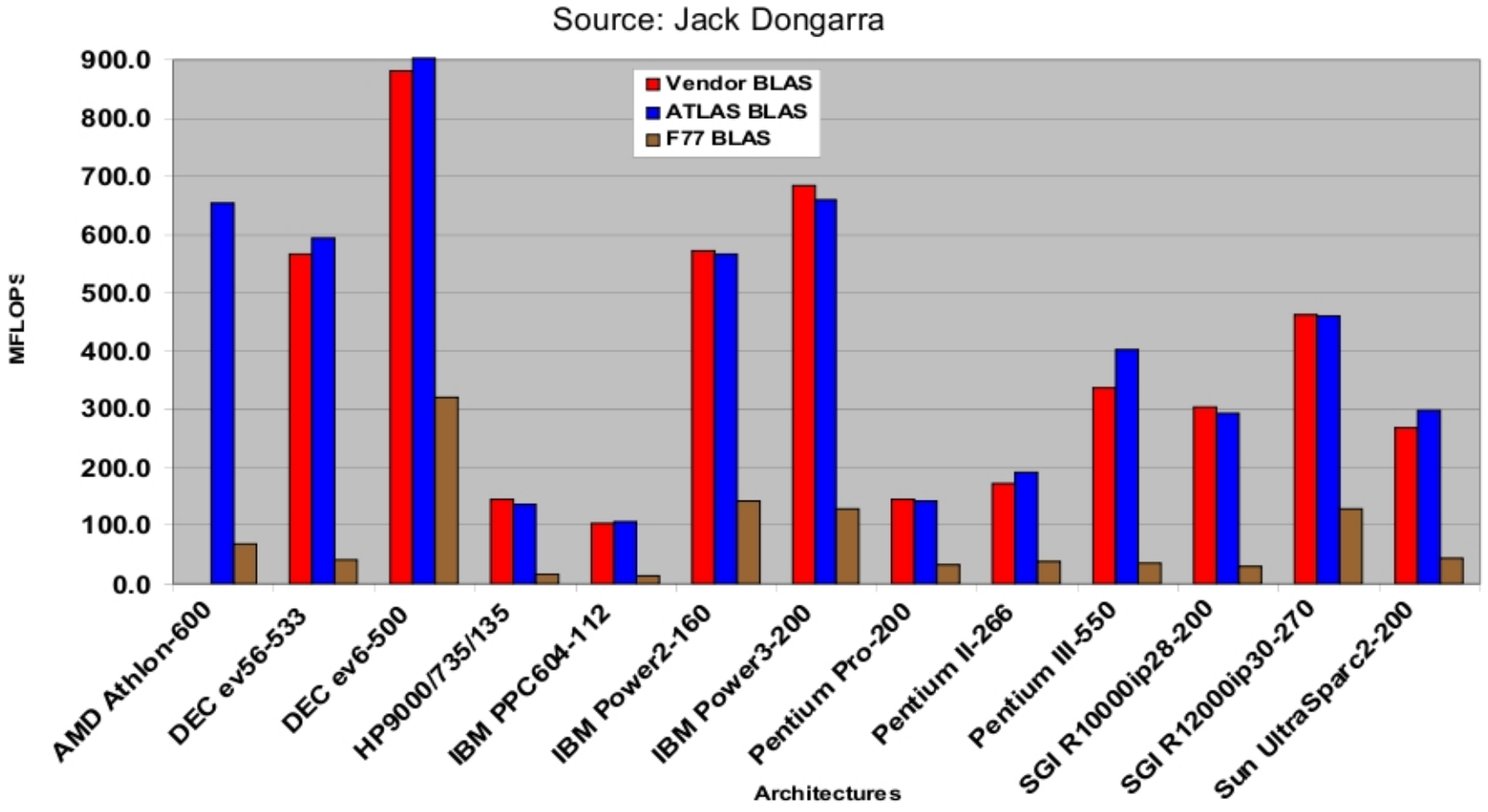


BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)

Routines BLAS (4)

Performance d'ATLAS (DGEMM n=500)

(CS267, U.C. Berkeley)



Autre exemple : la FFT et FFTW

- FFT (*Fast Fourier Transform*) : transformée discrète de Fourier en O(N ln N) (au lieu de O(N^2))
- Exemple :
  - convolution en O(N*N) remplacée par :
  - FFT en O(N ln N) + produit terme à terme en O(N) + FFT inverse en O(N ln N)
- FFTW (*Fastest Fourier Transform in the West)*
  - Optimisée pour
    - la version de FFT utilisée (réelle, complex, 1D, 2D...)
    - la valeur de N
    - l'architecture et sa hiérarchie mémoire
  - Utilise récursion → améliore la localité des données (pour les caches)

Architectures multicoeurs homogènes

- Rappel : depuis 2004-2005 faible augmentation de la fréquence des processeurs
  - ➤ chaleur dissipée trop importante,
  - ➤ consommation électrique trop importante,
  - ➤ degré de complexité trop important
- Loi de Moore toujours vérifiée
  → augmentation du nombre de coeurs (*cores*)
  → architectures multiprocesseur et/ou multicoeur
- Différence coeur/processeur : les coeurs partagent des élements du processeur (cache L2 par ex.)
- Intérêt multicoeur/multiprocesseur → coeurs plus proches :
  - ➤ Connections avec fréquence d'horloge plus rapide
  - ➤ Plus compact
  - ➤ Moins de puissance requise

Architectures multicoeurs homogènes (2)

- Programmation : voir cours suivant
- Problème crucial de l'accès à la mémoire :
  - ➤ Devient le goulot d'étranglement :
    - la bande passante augmente moins vite que le nombre de coeurs
    - hiérarchie mémoire critique pour performance : plus qu'en séquentiel à cause des transferts liés à la cohérence des données entre processeurs/coeurs
  - ➤ Effets NUMA (plusieurs sockets multicoeurs)
- Néanmoins le nombre de coeurs continue d'augmenter !
  - ➤ Bus : *Intel QuickPath Interconnect (QPI)* et *AMD HyperTransport*
  - ➤ Intel Nehalem – Westmere-EX 10 coeurs, Intel Xeon Phi, IBM Power 7 avec 8 coeurs /chip et « 4-way » SMT par cœur

Architectures multicoeurs hétérogènes

- Architectures monocoeurs / multicoeurs homogènes : part importante de la surface du processeur consacrée
  - au cache (L1),
  - au contrôle de l'exécution (par ex. : réordonnancement dynamique des instructions)
  - à la comptatibilité avec les architectures antérieures (ix86 par ex.)
- Nouvelles architectures : processeurs avec coeurs hétérogènes ou coprocesseurs spécialisés sur carte séparée (HWA : *HardWare Accelerator)*
- Architectures différentes pour
  - Gagner en puissance de calcul : augmentation du nombre d'unités de calcul et unités de calcul vectorielles
  - Améliorer l'accès à la mémoire
  - Utilisation restreinte à certains types d'applications :
    - Imagerie, jeux vidéos : GPU, processeur Cell (PS3)
    - Finance
    - Calcul scientifique : quelles applications ?

Architectures multicoeurs hétérogènes (2)

- Exemples d'architectures hétérogènes : (CPU+)GPU, APU (PS4, Xbox One), processeur Cell (PS3), Xeon Phi, ARM+GPU, CPU+FPGA...
- Programmation :
  - OpenCL
  - Directives de compilation
    - OpenACC
    - OpenMP 4.0
    - HMPP...
  - HSA (Heterogeneous System Architecture) : AMD, ARM...
  - ...

OpenCL sur CPU

- Actuellement supporté par Apple, NVIDIA, AMD-ATI, Intel, IBM...
- Exécution d'un kernel OpenCL sur CPU multicoeur :
  - Un thread est créé pour chaque cœur
  - Ces threads recoivent dynamiquement un bloc (=work-group) de work-items à exécuter
  - Exécution *possible* : le thread exécute chaque work-item un par un
    → exécution du work-item jusqu'à la fin de son kernel ou jusqu'à une barrière de synchronisation entre work-items
- Code SIMD :
  - Soit vectorisation *explicite* : via instructions vectorielles OpenCL (float4)
  - Soit vectorisation *implicite* (SDK Intel) :
    - Work-items scalaires
    - Regroupement de 2/4/8... work-items consécutifs pour exploiter les unités vectorielles (SSE, AVX, Xeon Phi)
    - Modèle « SPMD sur SIMD »

Nouvelles architectures hétérogènes : convergence CPU-GPU

- AMD = AMD+ATI → AMD *Fusion* (2011) : CPU+GPU=APU (*Accelerated Processing Unit*)
  - Premiers modèles destinés à l'embarqué
  - Exemple du Llano (A8-série) : CPU = 4 cœurs @2.9 GHz, GPU = 400 « coeurs » @600 MHz (27 Go/s de bande passante mémoire interne), TDP de 100 W
  - Possibilité de combiner 1 APU (CPU+GPU) avec 1 GPU discret sur un même nœud
- Intel Xeon Phi (ou *MIC - Many Integrated Core*) :
  - Suite de *Larrabee* et du *Single Chip Cloud Computer*
  - Prototype *Knight's Ferry* (2010), puis *Knight's Corner* (2012-2013) avec 60 cœurs x86 @1,053 GHz, *4-way* SMT, exécution *in-order*, unités SIMD de 512 bits et architecture cache-cohérente (au niveau des caches L2), 8 Go de mémoire → toujours via bus PCI !

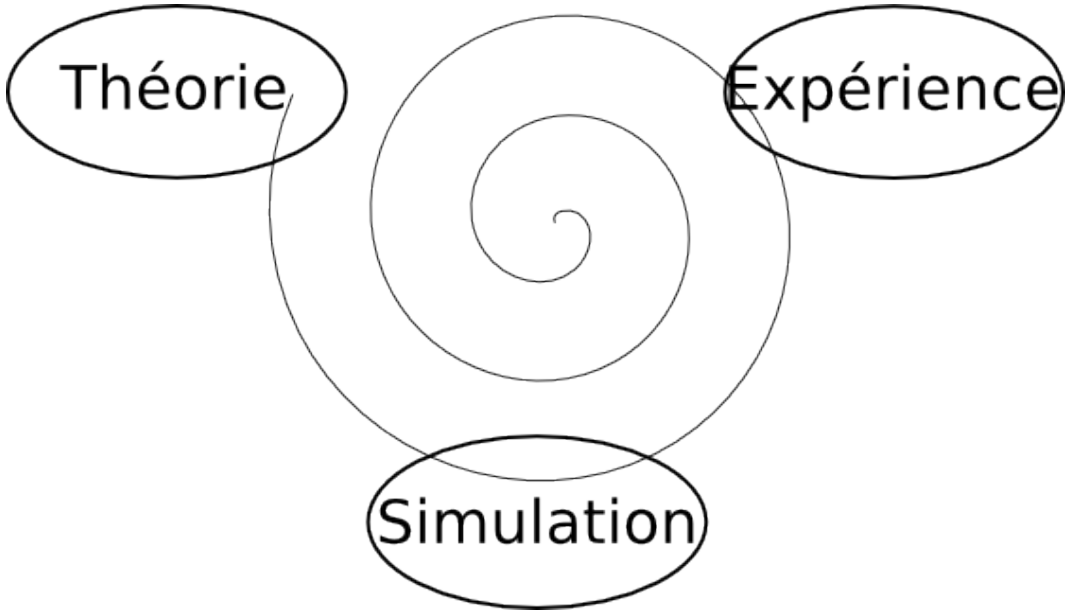
Efficacité en terme de puissance consommée

- Mesure : Flop/s/W (Watt)
- Meilleure utilisation du matériel dans les multicoeurs hétérogènes
  - Cell : 4 premiers du Green500 (www.green500.org, juin 2009) = clusters de QS22 (PowerXCell 8i)
  - Green500 actuels : BlueGene/Q et nœuds CPU-GPU ou nœuds CPU+Xeon Phi
  - GPU :
    - Tesla C2050 : 1030 Gflop/s (crête) pour 238 W max → 4,33 Gflop/s/W
    - Intel Westmere Xeon X5650 : 64,08 x2 (add+mul) = 128,16 Gflop/s pour 95 W (TDP) → 1,35 Gflop/s/W
      → Rapport C2050/X5650 = 3,21
  - FPGA : matériel « sur mesure » pour l'application

Importance du calcul scientifique

Simulation : le 3ième pilier de la science (J. Demmel)

- Méthode scientifique traditionnelle :
  - ➤ 1/ Elaboration d'une théorie
  - ➤ 2/ Expérimentation pour valider la théorie
- Limites : l'expérimentation peut être trop difficile, trop chère, trop lente ou trop dangereuse
- Grâce au calcul scientifique :
  - ➤ 3/ Utilisation d'ordinateurs pour simuler et analyser le phénomène
    - à partir des lois physiques connues
    - grâce à des méthodes numériques efficaces
    - Besoin d'ordinateurs toujours plus puissants (puissance de calcul, mémoire)



Applications de références

- *The Landscape of Parallel Computing Research : A View from Berkeley* (Asanovic et al., U.C. Berkeley, 2006)
- Identifier des noyaux de calculs et des schémas de communications représentatifs de l'ensemble des applications de calcul intensif
  - ➤ Calcul scientifique
  - ➤ Calcul embarqué
  - ➤ Applications classiques sur serveurs et PC personnels :
    - SPEC (Standard Performance Evaluation Consortium)
    - Bases de données
    - Imagerie et jeux vidéos
    - Apprentissage automatique (*machine learning*)
- Résultats : 13 *dwarfs/motifs* = 7 pour calcul scientifque + 6 motifs supplémentaires

7 motifs pour le calcul scientifique

- Algèbre linéaire dense
  - ➤ Opérations matricielles, routines BLAS
- Algèbre linéaire creuse
  - ➤ Matrices creuses : les nombreux zéros ne sont ni stockés, ni calculés
- Méthodes spectrales
  - ➤ Exemple : Fast Fourier Transform (FFT)
- Problèmes à N-corps
  - ➤ (Très) Nombreuses "particules" (atomes, étoiles...) déplacées en fonction des interactions avec tout ou partie des autres particules (forces éléctrostatiques, gravitationnelles...)

7 motifs pour le calcul scientifique (suite)

- Grilles structurées
  - Données organisées selon une grille régulière (ex. maillage 2D) et même opération en chaque point de la grille (ex. moyennage pondéré avec les valeurs des voisins – diffusion) : forte localité des données
- Grilles non structurées :
  - Idem mais sur grille non régulière (graphe quelconque) : besoin de récupérer la liste des voisins avant le calcul
- *MapReduce* :
  - Opérations indépendantes (*map*) sur données indépendantes (*embarrassingly parallel*), puis possible réduction finale (*reduce*)
  - Ex. : MonteCarlo, lancer de rayons, Mandelbrot...

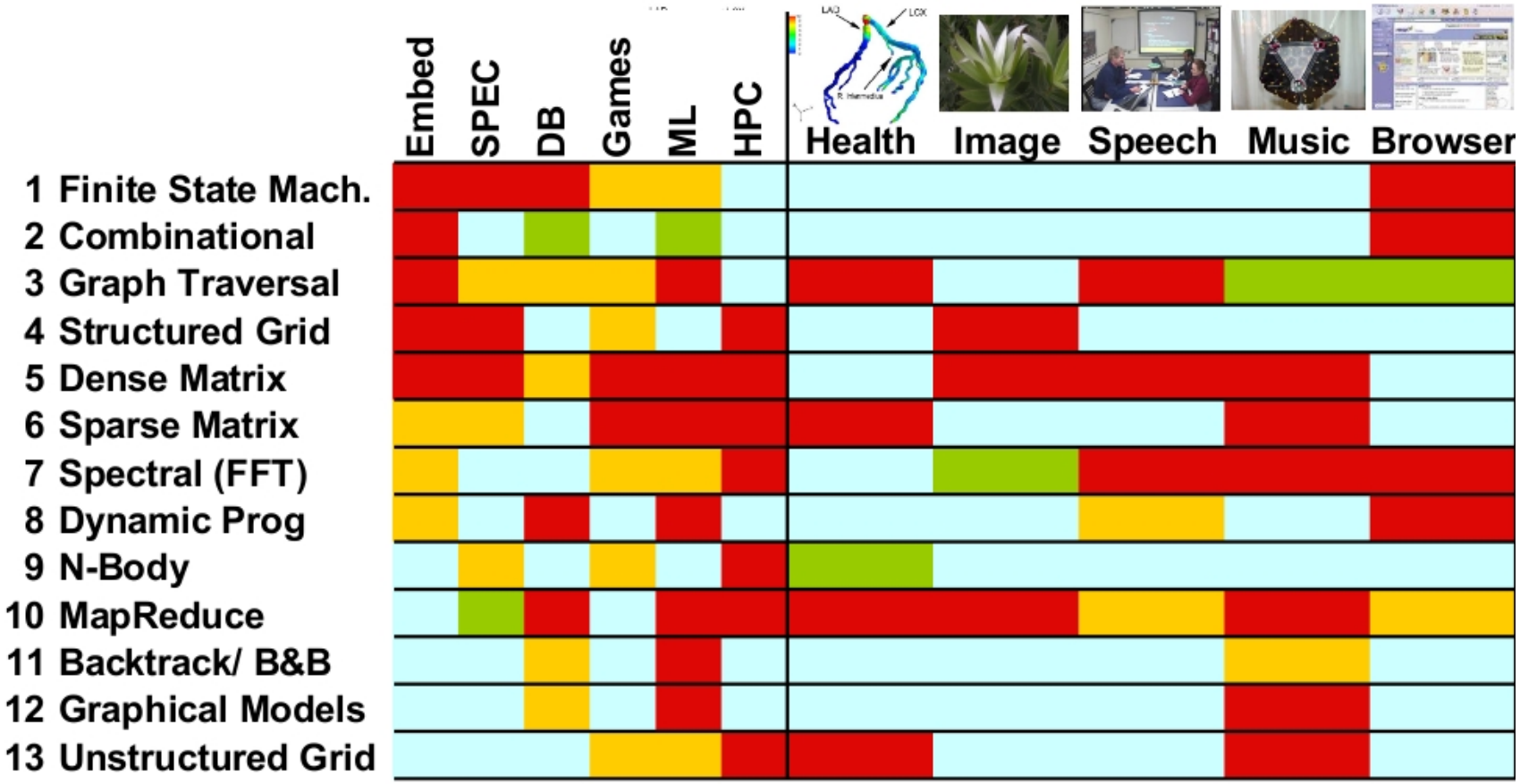
6 motifs supplémentaires

- *Finite State Machines*
  - Transformations XML, Huffman decoding (décompression)
- *Combinational Logic*
  - Op. booléennes sur données volumineuses (cryptographie - AES, DES)
- Traversée de graphe (et opération sur chaque noeud)
  - Arbre de décision, recherche (XML *parsing*, Quicksort...) : nombreux niveaux d'indirection, faible grain de calcul
- *Graphical models*
  - Graphes particuliers pour l'apprentissage automatique (réseau bayésiens, bioinformatique, détection des spams, fouille de données...)
- *Dynamic Programming*
  - Alignement de séquences (génétique)
- *Back-track* et *Branch-and-Bound*
  - divide-and-conquer + élimination des branches sous-optimales (jeu d'échecs...)

Extrait de CS267 (U.C. Berkeley)

What do commercial and CSE applications have in common?

Motif/Dwarf: Common Computational Methods
(Red Hot → Blue Cool)



Quelles limites pour les 13 motifs ?

Extrait de *The Landscape of Parallel Computing Research : A View from Berkeley*

Dwarf	Performance Limit: Memory Bandwidth, Memory Latency, or Computation?
1. Dense Matrix	Computationally limited
2. Sparse Matrix	Currently 50% computation, 50% memory BW
3. Spectral (FFT)	Memory latency limited
4. N-Body	Computationally limited
5. Structured Grid	Currently more memory bandwidth limited
6. Unstructured Grid	Memory latency limited
7. MapReduce	Problem dependent
8. Combinational Logic	CRC problems BW; crypto problems computationally limited
9. Graph traversal	Memory latency limited
10. Dynamic Programming	Memory latency limited
11. Backtrack and Branch+Bound	?
12. Construct Graphical Models	?
13. Finite State Machine	Nothing helps!