M1 MIAGE Option IFD
Data Mining et Parallélisme

Alexandre Termier

2011-2012 S2

Besoin

- Data Mining doit...
  - traiter de gros volumes de données
  - pouvoir effectuer des analyses complexes (gros calculs)
  - souvent les deux en même temps
- Très consommateur en ressources
  - puissance de calcul
  - mémoire
- Difficulté de passage à l'échelle sur des données réelles
  - données génomiques ?
  - recherches google ?
  - analyse de données de réseaux sociaux facebook ?
  - analyse en ligne des transactions amazon ?

Avant (pré-2005)

Loi de Moore

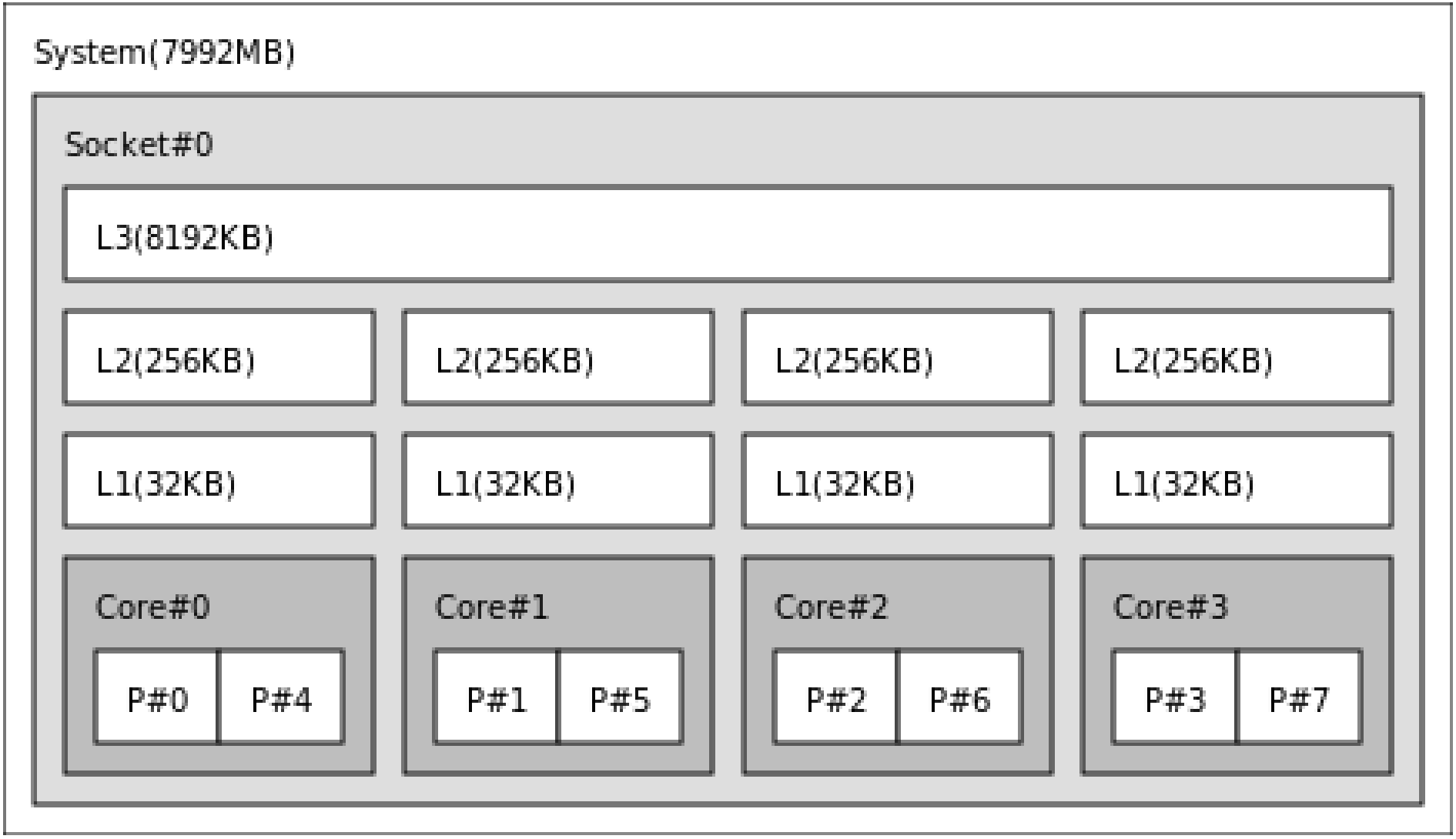
La quantité de transistors que l'on peut mettre sur un circuit intégré double tous les 18 mois.

- Conséquence générale
  - processeurs plus rapides :
    transistors plus petits $\rightarrow$ moins de consommation d'énergie
    $\rightarrow$ augmentation de la fréquence
  - mémoires plus volumineuses
  - plus de pixels dans un appareil photo numérique...
- Conséquence pour le data mining
  - On a un algo
  - Il ne va pas assez vite
  - Dans deux ans il ira deux fois plus vite

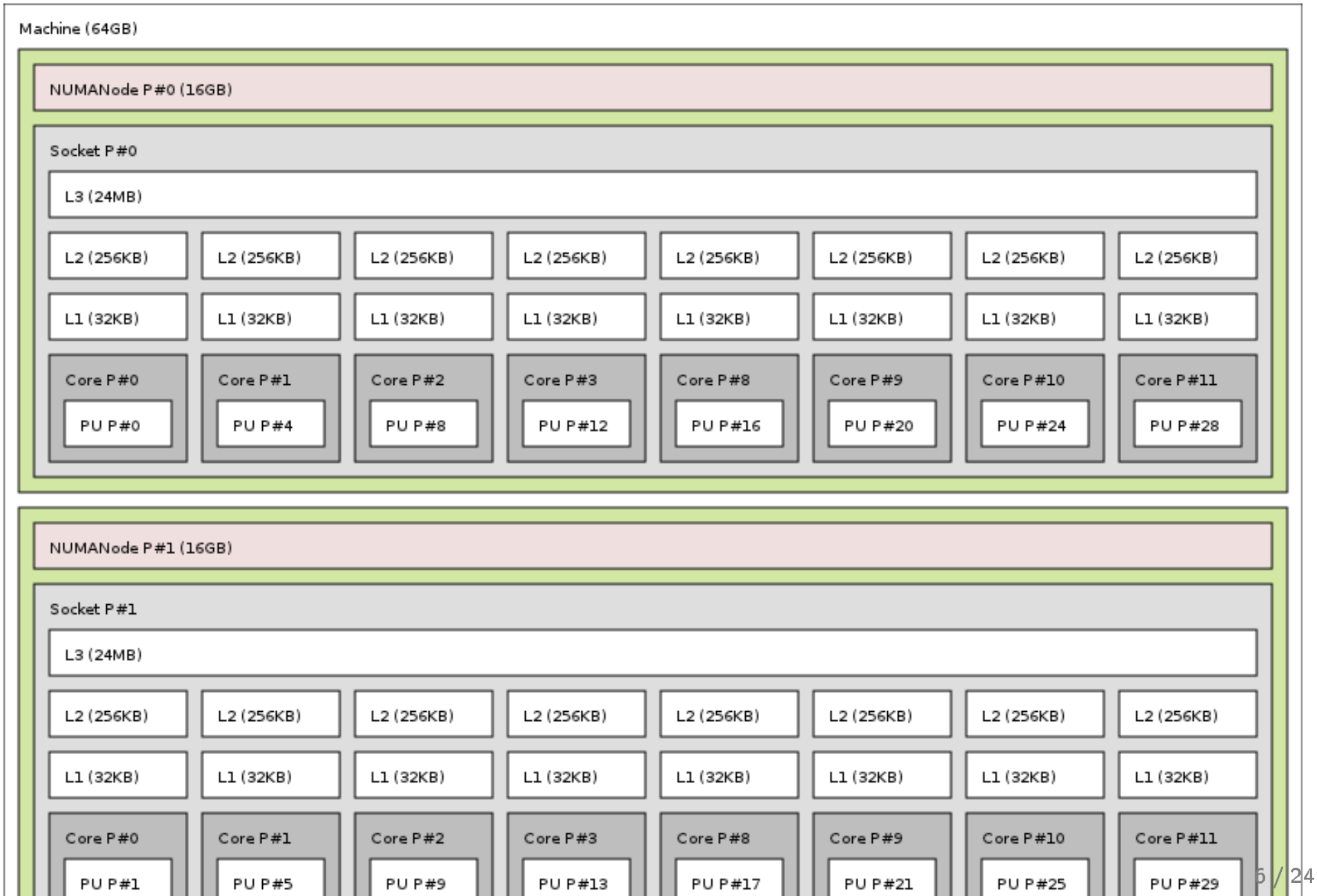
Maintenant (post-2005)

- Limites physiques : on ne peut plus augmenter la fréquence
  $\rightarrow$ les processeurs ne peuvent plus aller plus vite !
- La loi de Moore est toujours vraie
- Introduction des processeur multicoeurs
  - Plusieurs processeurs sur un même circuit intégré
  - Caches indépendants ou partagés
  - Mémoire RAM "partagée"
    - Architectures UMA : Uniform Memory Access
    - Architectures NUMA : Non Uniform Memory Access

Exemple : Intel Core i9



Exemple : 4 x Intel Xeon X7560



Autres machines parallèles

- **GPU** : processeur spécialisé avec beaucoup de coeurs (>200), chaque coeur est assez simple
- **Cluster** : plusieurs machines reliées par un réseau local
- **Grille** : plusieurs clusters reliés par Internet
- **Cloud** : buzzword pour désigner une grille munie de nombreux services
  - cette grille est partagée entre beaucoup d'utilisateurs
  - pour chaque utilisateur elle apparaît comme une grosse machine à sa disposition exclusive

Exploiter les processeurs multicoeurs

- Programmes classiques : **séquentiels**

  N'exploitent qu'un des coeurs $\rightarrow$ pas de gain de vitesse
- Solution : programmes parallèles
  - Ecrire le programme de manière à exploiter plusieurs coeurs
  - Gains de vitesses possibles, mais pas garantis
  - Difficulté : **synchronisation**

Concurrence vs Parallélisme

- Concurrence
  - Programme divisé en “tâches” qui peuvent être exécutées dans n’importe quel ordre
  - Les tâches peuvent interagir entre elles : synchronisation nécessaire
  - Programme peut être exécuté par un processeur monocoeur (entrelacement des tâches)
- Parallélisme
  - Plusieurs calculs/tâches effectuées simultanément
  - Peut avoir besoin des techniques de concurrence pour synchroniser ces tâches

Programmation parallèle

- Beaucoup d'environnements spécialisés
- Facilité d'utilisation dépend du niveau d'abstraction fourni
  - PThreads : gestion des threads et des synchronisations manuelle
  - Haskell/Strategies : parallélisme "invisible"
- Automatiser la parallélisation avec un bon compilateur ?
  - Saint Graal des chercheurs en compilation
  - Rêve lointain pour nous autres

PThreads

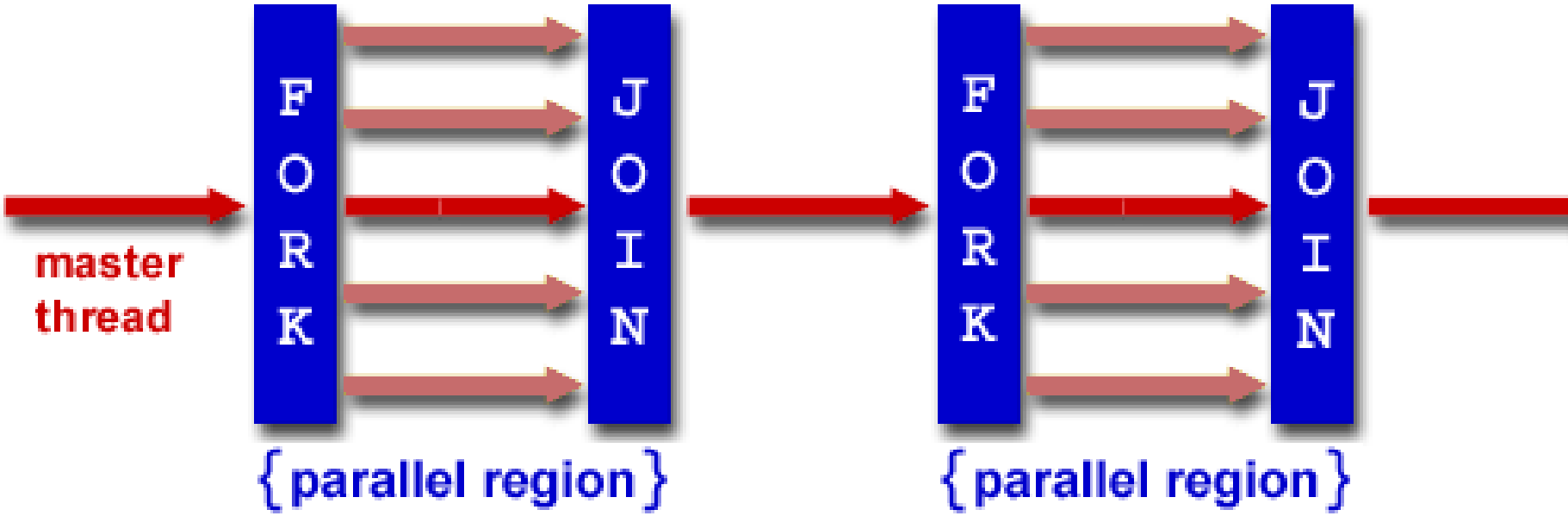
- Rappels du cours de système :
  - Threads : processus "légers"
  - Création explicite des threads
  - Synchronisation explicite par mutex/sémaphores
- Contrôle maximal sur le parallélisme
- Difficile à (bien) écrire

MPI

- Message Passing Interface
- Pour clusters
- Contrôle de la concurrence et partage des données se fait par passage explicite de messages
- Messages = suites d'octets, le programmeur est en charge de construire ces messages

OpenMP

- API pour C/C++/Fortran
- Interface plus simple pour paralléliser une application
- En particulier : parallélisation efficace de boucles
- Modèle fork-join



Example (boucle parallèle OpenMP)

```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

OpenMP

- Plusieurs options pour contrôler le parallélisme d'une boucle

Example (ordonnancement dynamique)

```
int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */
```

OpenMP

- Synchronisations possibles pour protéger ressources critiques

Example (section critique)

```
float dot_prod(float* a, float* b, intN)
{
  float sum = 0.0;
  #pragma omp parallel for
  for (int i=0; i<N; i++) {
      #pragmaompcritical
      sum += a[i] * b[i];
  }
  return sum;
}
```

Approches fonctionnelles

- Approches impératives : notion d'état "mutable"
- $\rightarrow$ une variable peut changer de valeur
- Impossible dans un langage fonctionnel (pur) !
- $\rightarrow$ fonctionnel "nativement" parallèle

Haskell

- Plusieurs paradigmes de parallélisme
- MVars : variables partagées, synchronisation la plus "manuelle"
- STM : Software Transactional Memory
- Parallélisme semi-implicite : Stratégies

Parallélisme semi-implicite

- Le programmeur **propose** au compilateur des possibilités de parallélisation
- Le RTS (Run Time System) essaie de mettre en oeuvre le parallélisme proposé
- `a 'par' b` : calcule et renvoie `b`, créée une *spark* pour calculer `a` en parallèle
- *spark* = opportunité de parallélisme, peut être prise ou pas

Parallélisme semi-implicite

- `a 'pseq' b` : calcule et renvoie `b`, mais force le calcul de `a` d'abord
- `a 'par' b 'pseq' (f a b)`
  - force l'évaluation de `a 'par' b` pour commencer
  - le thread principal calcule donc `b`
  - une spark est créée pour calculer `a`, elle est forcée d'être exécutée en parallèle
  - à la fin on peut calculer `f a b`

Parallélisme semi-implicite

- Les opérateurs précédents permettent de mettre au point des opérateurs plus complexes
- `parMap` : comme `map`, mais en parallèle

Example (parMap dans ghci)

```
Prelude> :m +Control.Parallel.Strategies
Prelude> (parMap rnf) (+1) [1..1000]
[2,3,4,5,......
```

Map/Reduce

- En Haskell, vous avez vu qu'avec des `map` et des `fold` on pouvait faire beaucoup de choses
- Ces opérateurs se parallélisent très bien
- Google : framework Map/Reduce pour analyser de gros jeux de données sur clusters
  - écrire un/des **map** pour découper les données en sous-problèmes et appliquer des traitements
  - écrire un/des **reduce** pour combiner les réponses aux sous-problèmes en une réponse globale
- Disponible dans le framework Hadoop (Apache), bindings pour de nombreux langages
- Option l'an prochain en M2 MIAGE (F.Jouanot / Gérard Forestier)

Performance parallèle

- Mesure : **speedup** $= \frac{temps\ \ sequentiel}{temps\ \ parallele}$
- Loi d'Amdahl :
  - $P =$ portion du programme parallélisée
  - $N =$ nb processeurs
  - $speedup\ \ max = \frac{1}{(1-P)+\frac{P}{N}}$

Parallélisation des algorithmes de Data Mining

- Clustering, Classification
  - Beaucoup de calculs indépendants sur tous les points des données
  - Comportement assez régulier des algorithmes
  - $\rightarrow$ algorithmes se parallélisent bien
- Patterns fréquents
  - Comportement irrégulier des algorithmes
  - Equilibre délicat calculs / accès mémoire
  - $\rightarrow$ algorithmes difficiles à paralléliser