BIG DATA ET SCIENCE DE DONNÉES

CONCEPTS DE BASE

Master 1 Intelligence Artificielle

Université de M'sila, Département d'Informatique

Dr Mehenni Tahar

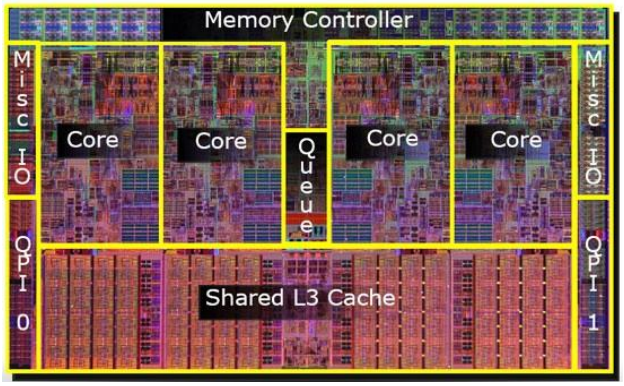
2020-2021

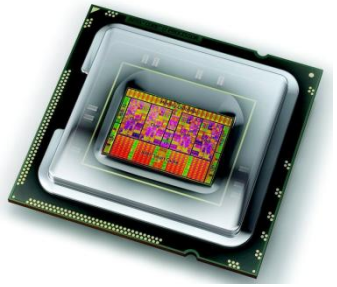
Composants matériels

Le processeur et ses unités de calcul

- Un processeur standard:
  - cœurs physiques (cores) fonctionnant en parallèle,
  - mémoires caches internes,
  - unités de calcul vectoriel dans chaque coeur.

- Un coeur (core): processeur autonome qui exécute un code différent de ceux des autres coeurs, à une fréquence d'horloge éventuellement différente. De plus, chaque cœur contient et exploite une petite mémoire cache locale, et des unités de calcul vectoriel (typiquement 4 ou 8), qui lui permettent d'exécuter une même instruction en parallèle sur des données différentes

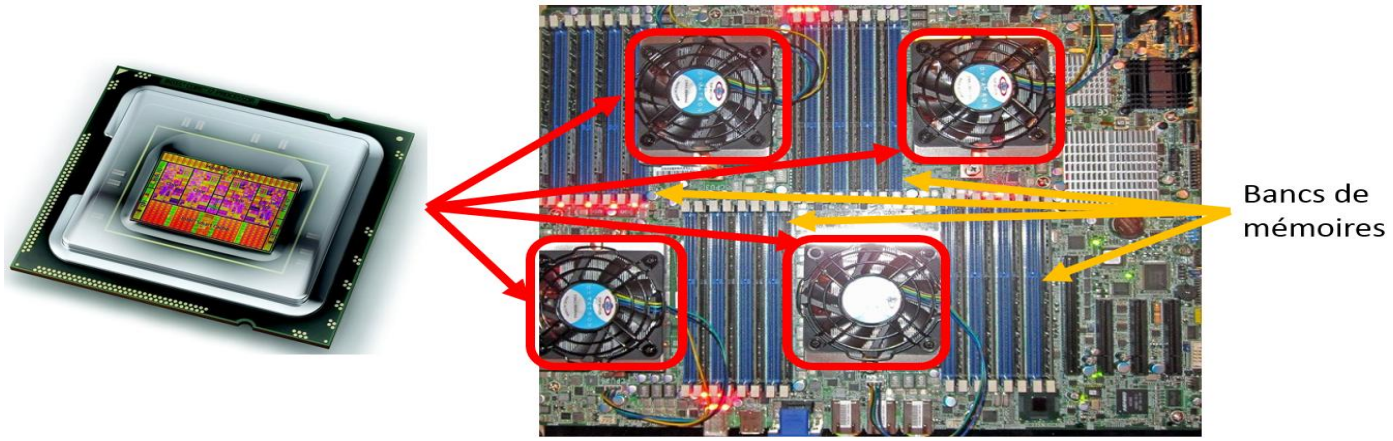




Composants matériels

Le noeud de calcul et sa mémoire partagée

- **Un noeud de calcul:** un ordinateur, un PC, avec son/ses processeurs, sa mémoire RAM, son/ses accès réseau, et éventuellement un/des disques locaux.
- Un noeud est aujourd'hui une machine parallèle à mémoire partagée.
- **Catégories de nœuds:**
  - Architecturte à accès mémoire uniforme qui se fait toujours avec le même délai.
  - Architecture à accès mémoire non uniforme ou NUMA (non uniform memory architecture): où chaque processeur est plus proche de certains bancs mémoire.
- La tendance actuelle est aux noeuds NUMA



Composants matériels

Le cluster de calcul et son réseau d'interconnexion

- **cluster de calcul:** un ensemble de noeuds de calculs, installés dans des baies et reliés par un réseau local d'interconnexion (l'interconnect).
- Il existe des clusters avec des noeuds très rapides et beaucoup de coeurs, avec beaucoup de mémoire et beaucoup d'espace disque par noeud,...
- il existe aussi des clusters avec des noeuds ressemblant à des serveurs standards.

Deux métriques pour évaluer la qualité d'un interconnect :

- la bande passante entre deux noeuds de calculs
- la latence entre deux nœuds: temps écoulé entre le début de l'envoie d'un message et le début de son arrivée sur le noeud destination.

Composants matériels

Les accélérateurs matériels

- Un accélérateur matériel est une carte fille installée dans un PC, avec sa propre mémoire, ses nombreuses petites unités de calcul, et une architecture nécessitant une programmation massivement parallèle.
- Un accélérateur peut-être vu comme un co-processeur de calcul scientifique. Mais depuis peu, certains accélérateurs peuvent devenir des processeurs à part entière sur la carte mère.
- Très utilisés pour paralléliser des calculs de machine learning, comme le deep learning.

Intel Xeon-phi 80 cores
(« Knight Landing »)

GPU NVIDIA (« TESLA »)

Intel FPGA accelerator
(« PSG Arria »)

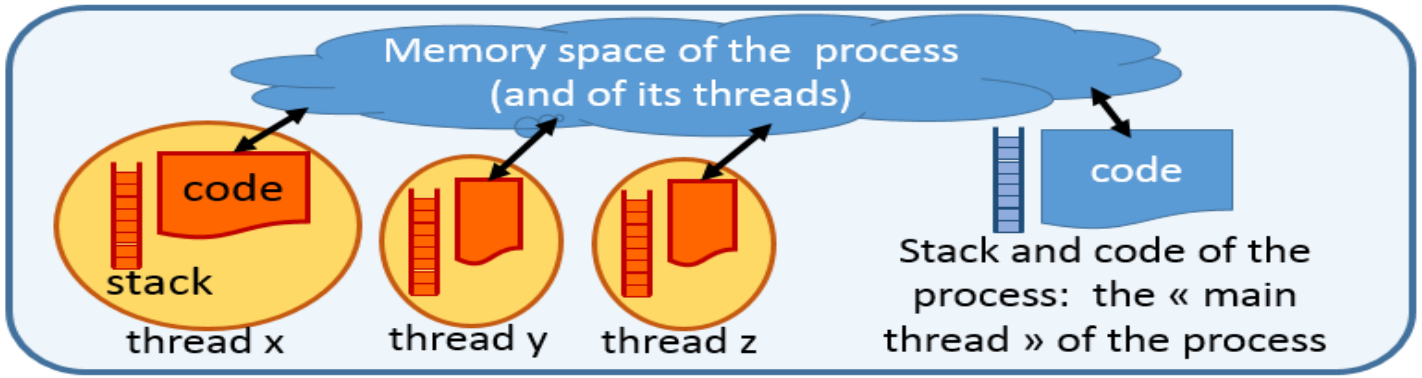
Composants logiciels

Les threads et processus

- **Les processus (*concept ancien*):** Un programme qui s'exécute, comme un éditeur de texte ou un compilateur, est un processus. Il possède son espace mémoire propre, où il stocke ses données et sa pile (qu'il utilise notamment pour enchaîner ses appels de fonctions). Deux processus peuvent partager de la mémoire en demandant explicitement au système d'exploitation de leur allouer un espace mémoire partagée. Par défaut les espaces mémoires des processus ne sont pas partagés.
- **Les threads (*concept récent*):** Chaque thread a bien une pile propre, mais il s'exécute dans l'espace mémoire de son processus hôte. En fait, un processus peut créer plusieurs threads, et par défaut tous les threads de ce processus partagent son espace mémoire et ses données. Pour le système d'exploitation, passer d'un thread à un autre est plus rapide que de changer de processus (l'espace d'adressage restant notamment le même) et l'on désigne un thread comme étant un processus léger.

Composants logiciels

- **Les threads et processus**
- Pour paralléliser une application et répartir ses calculs sur plusieurs coeurs d'un même processeur, on préfère créer des threads au sein d'un processus, que de créer plusieurs processus.
- Pour occuper tous les coeurs d'un noeud multi-processeurs on peut aussi utiliser un seul processus et de nombreux threads, mais à cause des architectures NUMA, on préfère souvent déployer un processus par processeur et des threads seulement à l'intérieur de chaque processeur (pour des raisons d'efficacité des accès mémoire).
- Actuellement, la plupart des langages de programmation permettent de créer des threads, et les systèmes d'exploitation peuvent facilement répartir et ordonnancer des milliers de threads (ou plus) sur des dizaines de coeurs.



Composants logiciels

Les outils élémentaires de synchronisation de tâches

- La synchronisation des tâches est indispensable dans un programme parallèle au sein de toute machine à mémoire partagée (comme un simple PC multi-cores).
- Il est important d'implanter des protocoles de synchronisation qui ne re-séquentialisent pas systématiquement les tâches.
- La mutex et le moniteur (sorte de mutex sur des portions de codes) sont notamment à éviter. Leur utilisation conduit souvent à des exécutions quasi-séquentielles des applications multithreads.
- Les protocoles/algorithmes de producteurs/consommateurs, lecteurs/rédacteurs, cohortes et barrières de synchronisation sont beaucoup plus pertinents en calcul parallèle.

Composants logiciels

Les middlewares et frameworks

- Un framework est un environnement de développement visant à simplifier le développement de certains types d'applications ou de composants logiciels.
- Il est plus générique qu'une simple bibliothèque, mais est conçu pour orienter les développements dans certaines voies.
- Un middleware est un environnement d'exécution qui cherche à simplifier la communication entre des composants logiciels, pour faciliter leur assemblage et donc la construction d'applications complexes (et souvent distribuées).
- Le middleware se situe au dessus du système d'exploitation et des couches réseaux,

Composants logiciels

Les middlewares et frameworks

- Il est fréquent de développer des applications distribuées dans un environnement comprenant un framework et un middleware compatibles.
- **Exemple: l'environnement Hadoop du Big Data:**
  - son framework permet de développer facilement en Java des composants destinés à être insérés dans une chaîne logicielle réalisant un Map-Reduce (paradigme de programmation d'Hadoop).
  - L'exécution de l'application distribuée obtenue s'appuiera ensuite sur un middleware comprenant notamment le système de fichiers distribués d'Hadoop (HDFS), et ses mécanismes de distribution et de gestion de tâches.
  - Au final, le développeur applicatif ne percevra pas les détails de mise en oeuvre sur une ou plusieurs machines, ni les complications des mécanismes de tolérance aux pannes.
  - Il sera limité à développer des applications respectant le paradigme Map-Reduce.

Cloud vs Cluster

- Un cluster permet de disposer d'un ensemble de PCs performants, reliés par un réseau à haute performance (en latence et bande passante), sur lequel on pourra exécuter des tâches appartenant à une même application et échangeant de nombreuses données et résultats intermédiaires.
- un cloud permet d'allouer des PCs, ou simplement des coeurs, mais sans aucune hypothèse sur la proximité des ressources allouées ou sur la qualité de l'interconnect.
- Allouer des ressources dans un cloud standard pour y stocker des données et exécuter un code HPC peut s'avérer désastreux en performances, mais être un très bon rapport qualité/prix pour stocker des photos et appliquer un traitement indépendant sur chacune.
- Pour des applications de Big Data selon le schéma de calcul Map-Reduce, où les tâches des phases de Map et de Reduce sont indépendantes, et où les communications entre les deux phases peuvent être partiellement recouvertes, un cloud standard peut présenter un très bon rapport qualité/prix.
- En revanche l'accélération d'algorithmes d'apprentissage (machine learning), sur des données filtrées et rassemblées préalablement par un pré-traitement en Map-Reduce, est habituellement plus indiquées sur des architectures HPC.

Problématique du déploiement ou mapping

- Une application distribuée suit un algorithme qui exploite une topologie distribuée virtuelle.
- Certaines parties d'une architecture distribuée peuvent être en panne de longue durée, ou en maintenance planifiée, ou être tombées en panne peu avant le lancement de l'application à déployer, et d'autres peuvent être déjà très chargées par d'autres tâches provenant d'autres applications.
- Il est donc important de sélectionner des ressources parmi une liste des ressources disponibles maintenue à jour en permanence,
- Ce qui repose en général sur des mécanismes de monitoring par le système d'exploitation de chaque noeud et par une couche de middleware distribué qui fournit une vue d'ensemble sur toutes les ressources.
- Ensuite il faut qu'une tâche (processus ou thread) soit installée sur une ressource de calcul lui donnant accès aux données dont elle à besoin, et lui offrant une capacité de calcul suffisante pour traiter ses données.

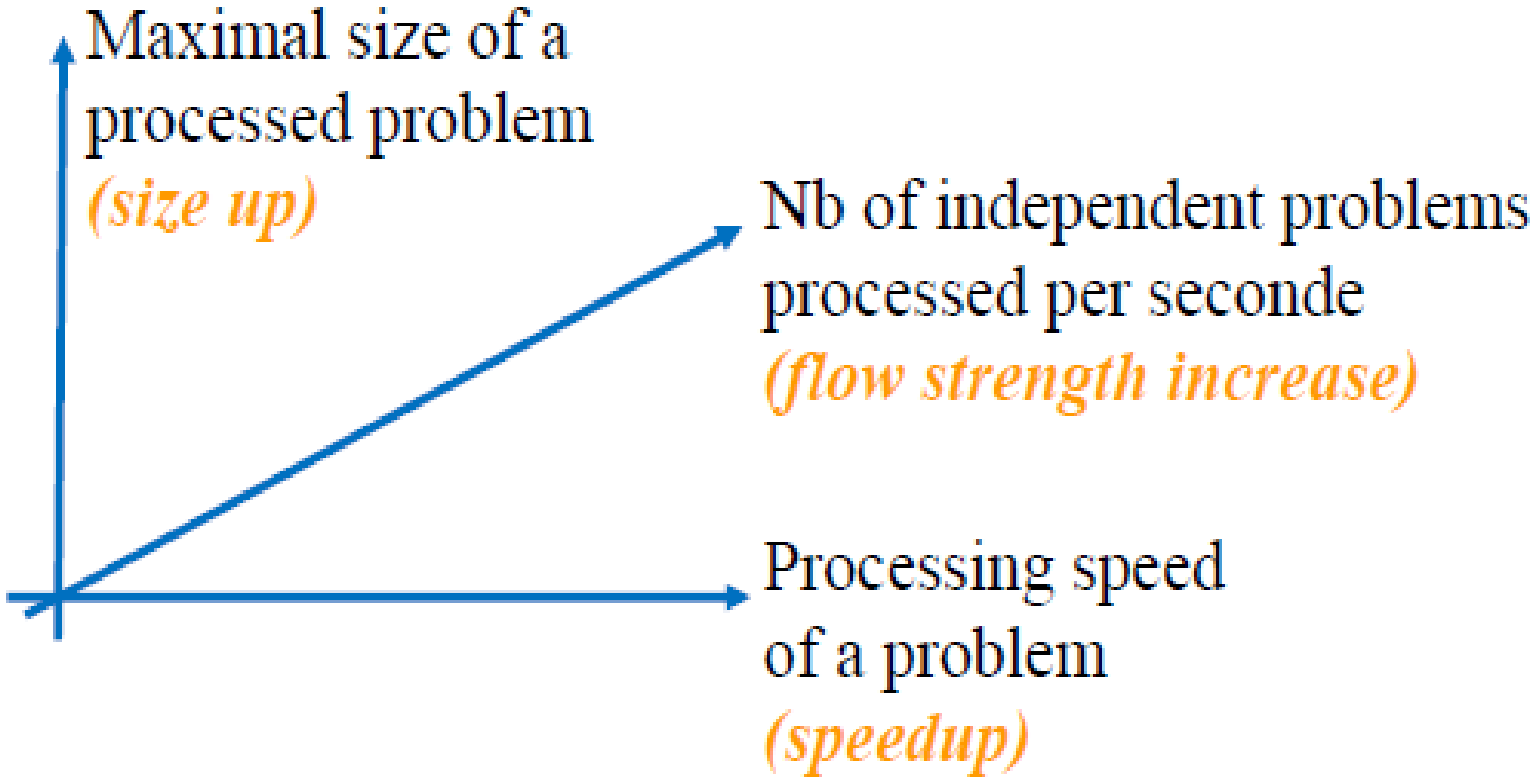
Objectifs du mapping

- **La recherche de la performance :** vitesse de calcul, vitesse d'accès en lecture et/ou écriture aux données, qui impose d'installer la tâche sur une ressource assez proche des données et assez puissante.
- **La mise en oeuvre d'une tolérance aux pannes :** car une ressource de calcul ou de données peut très bien tomber en panne au cours de l'exécution de la tâche qu'elle héberge.
- **La capacité de passage à l'échelle :** qui demande la conception d'une solution extensible tant au niveau de l'algorithme et de l'implantation de l'application.
- **La maîtrise du coût d'exécution :** quand on souhaite augmenter la quantité de ressources allouées et la durée des traitements.

Que faire avec plus de ressources informatiques ?

- Plus de ressources informatiques: plus d'unités de calculs (coeurs, processeurs), plus de mémoire, plus d'espace de stockage (disques)…
- Trois directions:
  - **Exécuter concurremment l'application** sur les différents coeurs d'une machine, ou sur différentes machines.
  - **Accélérer l'exécution d'une application** en la parallélisant sur plusieurs ressources de calcul (*speedup*).
  - **Traiter un problème plus gros (size up).** On cherche alors à distribuer l'application sur plusieurs machines pour (1) être capable de traiter le problème (si on manquait de mémoire), et (2) maintenir un temps d'exécution constant quelle que soit la taille du problème.

Que faire avec plus de ressources informatiques ?



« **Passage à l'échelle** » : size up + speedup + maîtrise des coûts

Accélération d'un traitement

Difficultés pour accélérer une application (sur plusieurs ressources de calcul) **:**

1. Les parties de son code les plus gourmandes en temps doivent être décomposables en sous-tâches pouvant s'exécuter en parallèle.

- Certains algorithmes sont intrinsèquement séquentiels d'où chercher une nouvelle modélisation du problème !!

2. L'algorithme parallèle doit avoir une complexité identique ou pas beaucoup plus grande que le meilleur algorithme séquentiel.

3. Le surcoût de gestion de la décomposition en sous-tâches ne doit pas être trop important.

- Coût de gestion du parallélisme : temps de synchronisation et

temps de communications.

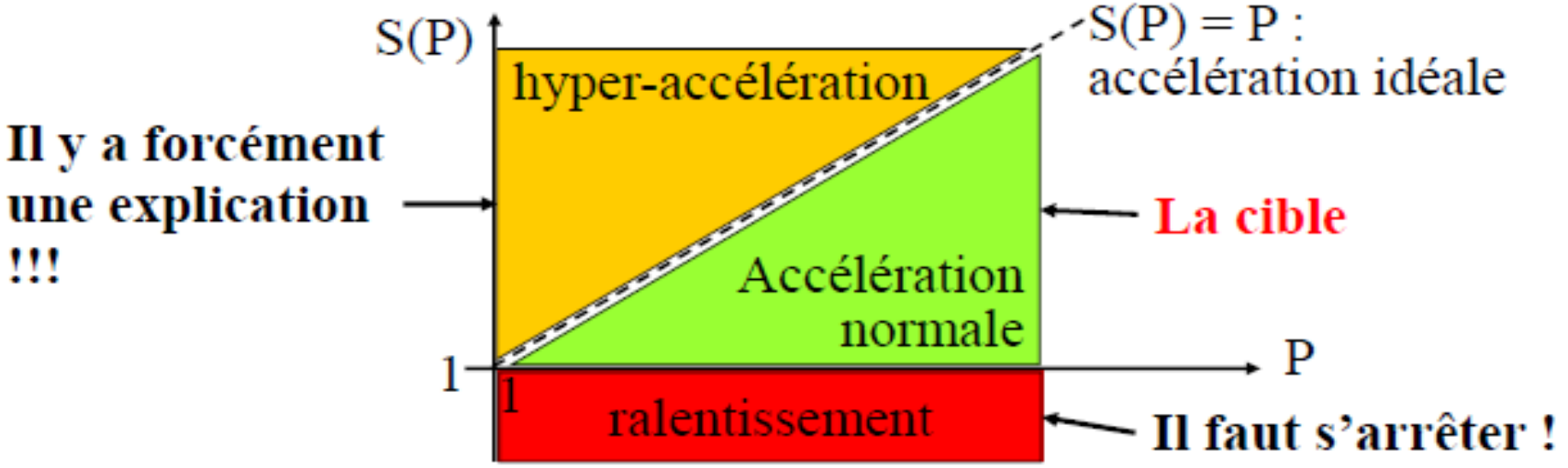
- *Le paradigme de programmation Map-Reduce est une tentative de réponse à ses problèmes pour le Big Data. Il propose un schéma de parallélisation à la fois adapté à de nombreux algorithmes d'analyse de données, et facilement exploitable par (presque) tous les développeurs.*

Accélération d'un traitement

Métrique d'accélération : *Speedup* sur P ressources

$$S(P) = \frac{T(1)}{T(P)}$$

- S(P) < 1 : on ralentit ! mauvaise parallélisation
- 1 < S(P) < P : "normal"
- P < S(P) : hyper-accélération analyser & justifier



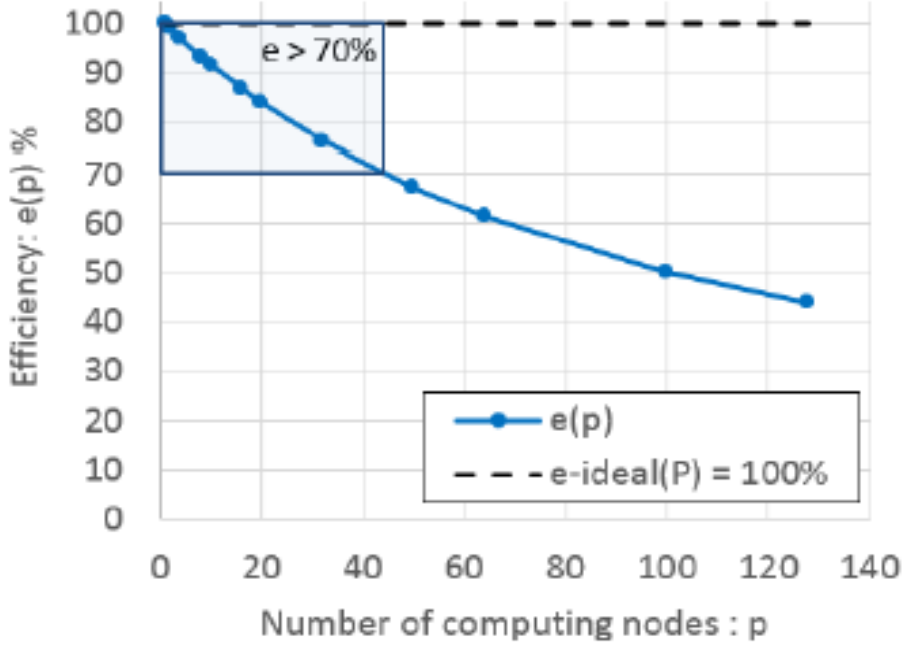
Accélération d'un traitement

Efficacité :

$$e(P) = \frac{S(P)}{P}$$

Taux d'utilisation des ressources, ou fraction obtenue de l'accélération idéale

- $e(P) \in [0;1], \in [0\%;100\%]$
- $e(P) > 100\% \Leftrightarrow$ hyper – accélération



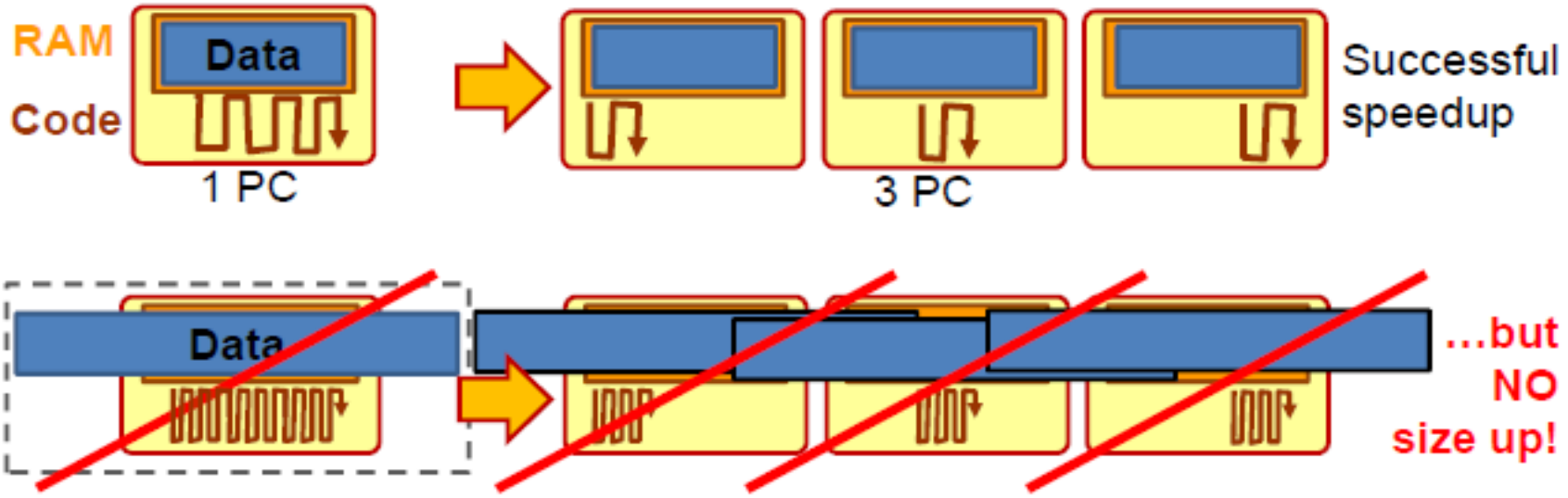
L'utilisateur s'intéresse à l'accélération obtenue

L'acheteur de la machine s'intéresse à l'efficacité des applications exécutées

Le **développeur** s'intéresse aux **deux**

Traitement de plus gros problèmes

Difficultés à traiter de plus gros problèmes : Un code qui **réplique** la plupart de ses données sur toutes les machines sera toujours limité par la taille mémoire d'une machine… …et ne sera **pas apte au *size up***



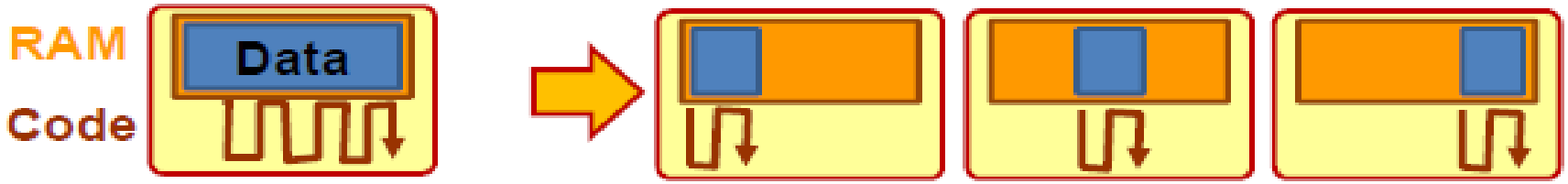
Traitement de plus gros problèmes

Difficultés à traiter de plus gros problèmes: Un code qui **répartit** la plupart de ses données sur toutes les machines pourra stocker plus de données sur plus de machines… et sera **apte au *size up.***

Traitement de plus gros problèmes

Difficultés à traiter de plus gros problèmes: Un algorithme distribué avec répartition des données est souvent complexe :

- • ***Besoin de faire circuler les données initiales entre les nœuds de calcul*** : pour qu'un noeud puisse poursuivre ses calculs sur d'autres données que les siennes.
- • ***Besoin de faire circuler les résultats intermédiaires entre les noeuds*** : pour qu'un noeud puisse poursuivre les calculs d'un autre, avec ses données.
- Conception d'une répartition initiale des données, et d'un schéma de communication (à volume minimal…) : **nécessite un expert** !



Traitement de plus gros problèmes

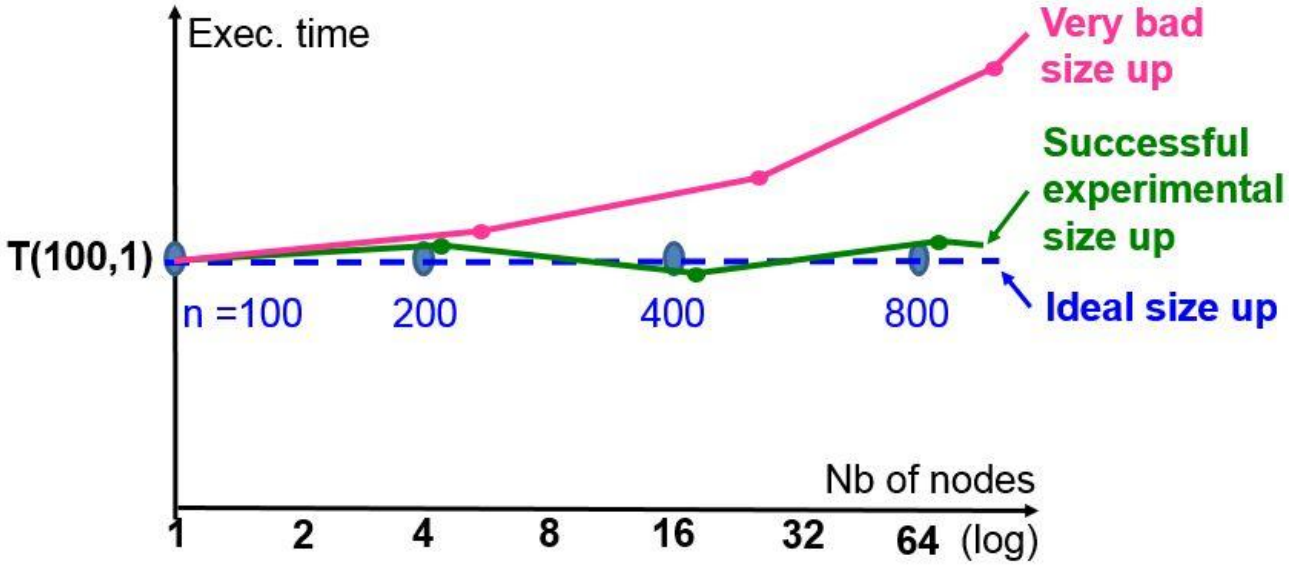
Métrique de *size up* (extensibilité) : courbe de temps

1 - Maintenir constant le temps d'exécution

$T(1\times n_1,p_1) = T(2\times n_1,p_2) = T(k\times n_1,p_k) = Cte$

avec : T(taille pb, nb rsrc)

- **Objectif pas toujours atteignable !**



Traitement de plus gros problèmes

Métrique de *size up* (extensibilité) : courbe de temps

2 - Maintenir constant le temps d'exécution MAIS avec le bon nombre de ressources

$T(1\times n_1,p_1) = T(2\times n_1,p_2) = T(k\times n_1,p_k) = Cte$

avec : T(taille pb, nb rsrc) **Avec $O(p_k) = O(calcul(n))$**

- **Objectif pas toujours atteignable !**