Penser parallèle

Vincent Miele,
CNRS, Biométrie Biologie Evolutive

Ecole ≪ Découverte du Calcul ≫ 2013

Penser parallèle

Des calculs de très grande taille

- grande taille des données ou du problème étudié
- grands espaces de solutions (combinatoire...)
- algorithmes gourmands (MCMC...)
- (besoins en mémoire supérieurs à la capacité d'une seule machine)

La solution : penser parallèle !

Penser parallèle

Du parallélisme depuis les supercalculateurs...

Depuis les années 60-70, les supercalculateurs ont une structure parallèle.

En détails ▸ Une histoire de l'informatique ▸ Supercomputers
En France et en images d'archive ▸ Au CCRT
En infographie ▸ Supercomputing Information : 70 Years of Supercomputers
En visuel ▸ Supercomuting : a visual history
En dérision ▸ Les super-ordinateurs maléfiques
En anticipant ▸ Les radiateurs de calcul

Pour utiliser les supercalculateurs, penser parallèle !

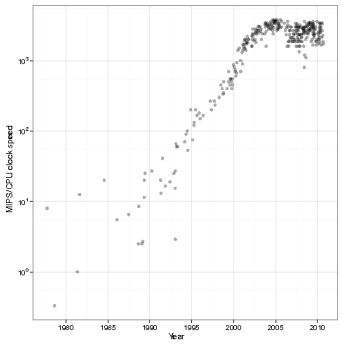
Penser parallèle

... jusqu'à votre simple machine

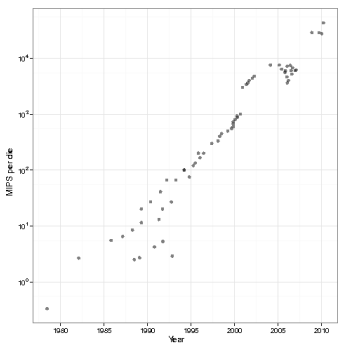
Depuis 2003, les processeurs courants possèdent plusieurs cœurs physiques qui travaillent en parallèle

Fréquence d'horloge d'une CPU*
usuellement inférieure à 3.5 Ghz
*(central processing unit

"celui(celle) qui fait le boulot" = étudiant)



Grâce aux processeurs et machines multi-coeurs (48 coeurs au Prabi par ex.), la puissance de calcul disponible continue à augmenter (Moore's law)



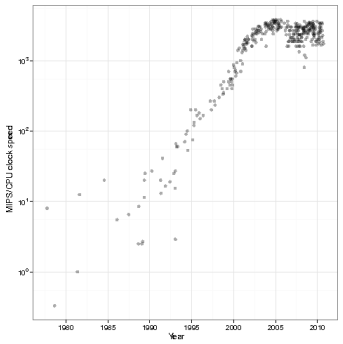
Penser parallèle

... jusqu'à votre simple machine

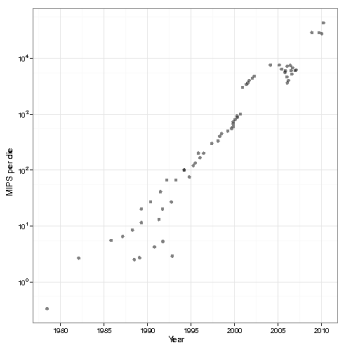
Depuis 2003, les processeurs courants possèdent plusieurs cœurs physiques qui travaillent en parallèle

Fréquence d'horloge d'une CPU*
usuellement inférieure à 3.5 Ghz
*(central processing unit

"celui(celle) qui fait le boulot" = étudiant)



Grâce aux processeurs et machines multi-coeurs (48 coeurs au Prabi par ex.), la puissance de calcul disponible continue à augmenter (Moore's law)



- Avant 2003 : attendre un nouvel ordinateur "un étudiant plus balèze"
  travailler plus pour gagner plus
- Après 2003 : penser parallèle ! "plusieurs étudiants"
  partager plus le travail pour gagner plus

Penser parallèle

Design et implémentation

Le **design** and l'**implementation** d'algorithmes parallèles est la clé pour tirer profit des architectures parallèles. Un mélange d'**algorithmique** et d'**ingénierie**

Penser parallèle

Design et implémentation

Le **design** and l'**implementation** d'algorithmes parallèles est la clé pour tirer profit des architectures parallèles. Un mélange d'**algorithmique** et d'**ingénierie**

Et si on faisait un puzzle à plusieurs ? ! ?

Le **design** and l'**implementation** d'algorithmes parallèles est la clé pour tirer profit des architectures parallèles. Un mélange d'**algorithmique** et d'**ingénierie**

Et si on faisait un puzzle à plusieurs ? ! ?

- "comment faire faire UNE tâche par plusieurs étudiants ?" **algorithmique**
- "si ils sont dans la même pièce ? dans des pièces différentes ?" **ingénierie**
- "Les faire communiquer ? Se coordonner ?" **ingénierie**
- "Qu'ils soient toujours occupés ?" **algorithmique**
- "Qui fait quoi quand où ?" **algorithmique**

THIS IS PARALLEL COMPUTING
papier+ordinateur
algorithmique+ingénierie

Des *unités CPU* ou *coeurs* sont disponibles sur un (super-)ordinateur

- mémoire partagée = plusieurs coeurs
- mémoire distribuée = plusieurs *noeuds*
- réseau d'interconnexion $\neq$ grille

On parlera ensuite d'unités de calcul abstraites (**process**) disponibles pour résoudre un problème de manière coopérative.

1. decomposer en *tâches* (unités indivisibles executées en parallèle) de tailles variées

1. decomposer en *tâches* (unités indivisibles executées en parallèle) de tailles variées
2. lister les dépendances entre les tâches (”charrue avant les boeufs”) et évaluer leur taille

1. decomposer en *tâches* (unités indivisibles executées en parallèle) de tailles variées
2. lister les dépendances entre les tâches (”charrue avant les boeufs”) et évaluer leur taille
3. assigner (mapper) les tâches à des processes : ordonnancer/synchronizer pour respecter les dépendances, éviter les temps d'attente et minimiser le temps total d'exécution

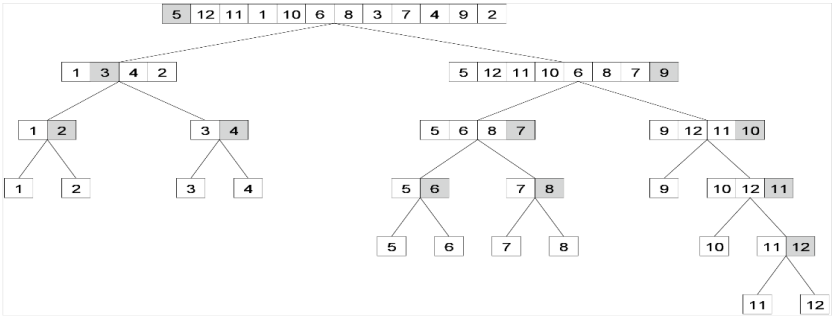
1. decomposer en *tâches* (unités indivisibles executées en parallèle) de tailles variées
2. lister les dépendances entre les tâches (”charrue avant les boeufs”) et évaluer leur taille
3. assigner (mapper) les tâches à des processes : ordonnancer/synchronizer pour respecter les dépendances, éviter les temps d'attente et minimiser le temps total d'exécution
4. distribuer les entrées/sorties

1. decomposer en *tâches* (unités indivisibles executées en parallèle) de tailles variées
2. lister les dépendances entre les tâches (”charrue avant les boeufs”) et évaluer leur taille
3. assigner (mapper) les tâches à des processes : ordonnancer/synchronizer pour respecter les dépendances, éviter les temps d'attente et minimiser le temps total d'exécution
4. distribuer les entrées/sorties
5. gérer l'accès à des données communes

1.1 Décomposition récursive ou *divide-and-conquer*

- un problème est divisé en un ensemble de sous-problèmes
- les résultats des sous-problèmes sont fusionnés
- les sous-problèmes sont récursivement décomposés jusqu'à une granularité désirée

Exemple : minimum d'un tableau, quicksort parallèle (une tâche correspond au partitionnement d'un tableau)



1.2 Décomposition des données

- chaque process traite une partie indépendante des données (la règle *owner-computes*)
- les données peuvent être des entrées, des sorties ou intermédiaires
- les processes effectuent les même opérations sur leurs données avec une parfaite indépendance

Example : produit de matrice par blocs (décomposition en tâches $\neq$ décomposition des données)

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1:	$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$	$C_{1,1} = A_{1,1}B_{1,1}$
Task 2:	$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$	$C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3:	$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$	$C_{1,2} = A_{1,1}B_{1,2}$
Task 4:	$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$	$C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$
Task 5:		$C_{2,1} = A_{2,1}B_{1,1}$
Task 6:		$C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$
Task 7:		$C_{2,2} = A_{2,1}B_{1,2}$
Task 8:		$C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

1.3 Décomposition exploratoire

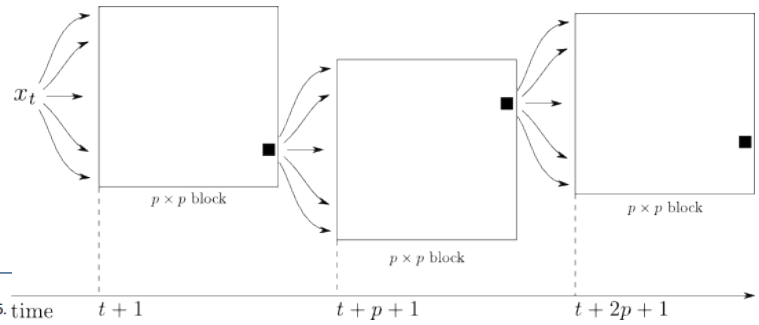
- l'espace de recherche peut être partitionné et correspondre à plusieurs tâches executable en parallèle
- l'espace de recherche peut être exploré à partir d'une solution courante dans plusieurs directions qui forment des tâches indépendantes

Example : MCMC parallèle

Journal of Computational and Graphical Statistics

Using Parallel Computation to Improve Independent Metropolis–Hastings Based Estimation

To cite this paper:
P. Jacob, C. P. Robert, M. H. Smith. Journal of Computational and Graphical Statistics. September 1, 2011, 20(3): 616-635.



1.4 (beurk) Décomposition fonctionnelle

- les différentes fonctions à appliquer aux données forment les tâches, i.e. *pipelines*
- ne passe pas bien à l'échelle, à cause du nombre limité de fonctions

- Sequential

- Parallel

1.5 Décomposition hybride
Example : ajout d'une étape de decomposition des tableaux ds le quicksort (partitionnement d'un tableau = plusieurs tâches)

Analyser le plat de spaghetti des tâches

2.1 Construire une *graphe de dépendance de tâches* pour exprimer l'odre relatif d'exécution : les sommets sont les tâches et les arètes indiquent que le résultat d'une tâche est nécessaire pour exécuter une autre tâche

Analyser le plat de spaghetti des tâches

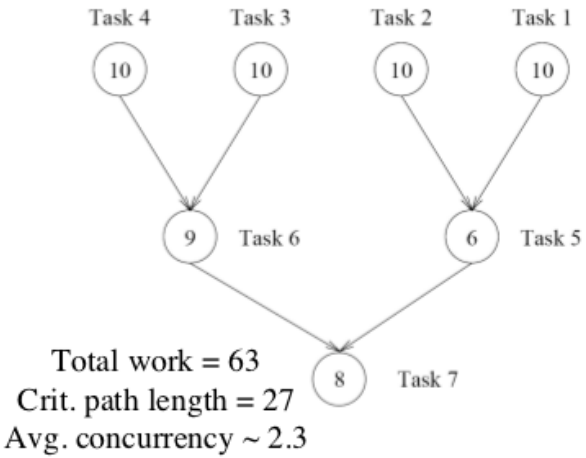
2.1 Construire une *graphe de dépendance de tâches* pour exprimer l'odre relatif d'exécution : les sommets sont les tâches et les arètes indiquent que le résultat d'une tâche est nécessaire pour exécuter une autre tâche

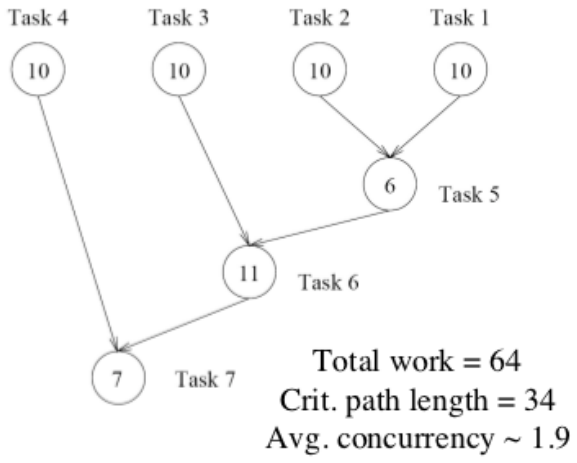
2.2 Evaluer la taille des tâches (theorique/benchmark...) and reprendre 2.1

2.1 Construire une *graphe de dépendance de tâches* pour exprimer l'odre relatif d'exécution : les sommets sont les tâches et les arètes indiquent que le résultat d'une tâche est nécessaire pour exécuter une autre tâche

2.2 Evaluer la taille des tâches (theorique/benchmark...) and reprendre 2.1

2.3 *Average degree of concurency* = nombre moyen de tâches qui peuvent être exécutées simultanement (calculé à partir du *chemin critique*)

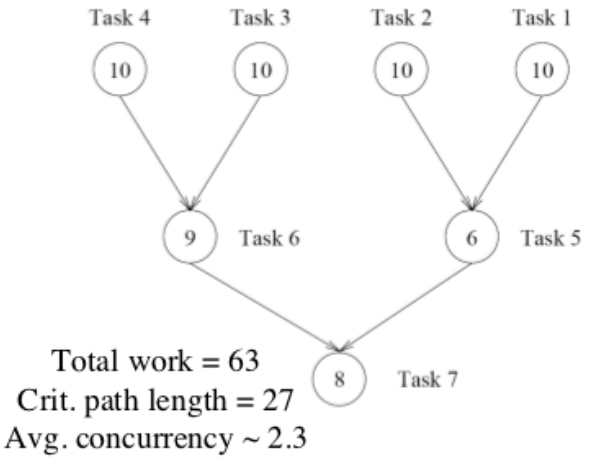


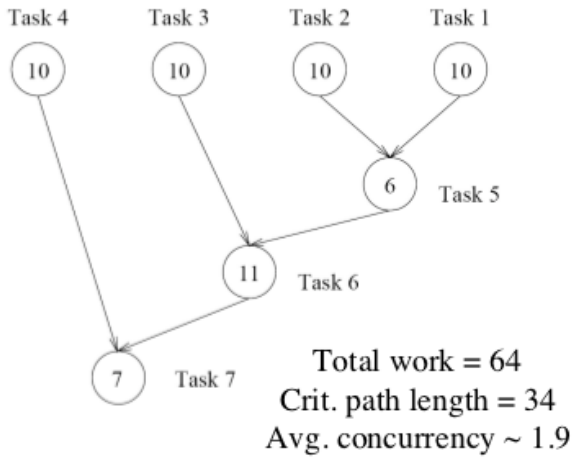


2.1 Construire une *graphe de dépendance de tâches* pour exprimer l'odre relatif d'exécution : les sommets sont les tâches et les arètes indiquent que le résultat d'une tâche est nécessaire pour exécuter une autre tâche

2.2 Evaluer la taille des tâches (theorique/benchmark...) and reprendre 2.1

2.3 *Average degree of concurency* = nombre moyen de tâches qui peuvent être exécutées simultanement (calculé à partir du *chemin critique*)





2.4 Evaluer les interactions entre tâches (données partagées ou échangées)

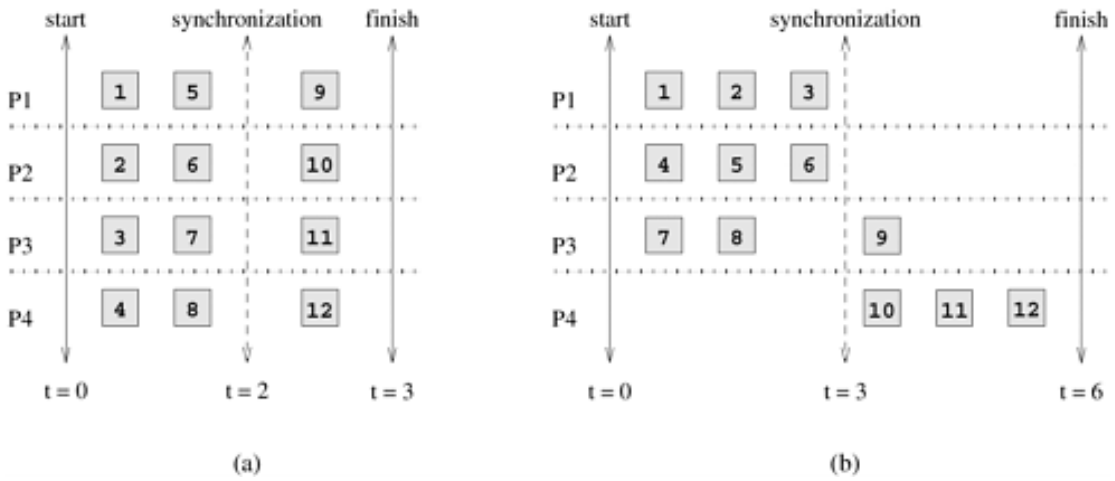
Ordonnancer les tâches

Assigner les tâches à des processes pour minimiser le temps total d'exécution.

- réduire le temps d'interaction entre processes
  Exemple : trop d'échanges de données
- éviter qu'un process soit en attente et inactif ("chômage technique")

Assigner les tâches à des processes pour minimiser le temps total d'exécution.

- réduire le temps d'interaction entre processes
  Exemple : trop d'échanges de données
- éviter qu'un process soit en attente et inactif ("chômage technique")
  Exemples :
  1) goulot d'étranglement si des multiples tâches dépendent d'un tâche très longue...
  2) mauvaise synchronisation entre processes qui s'attendent mutuellement...



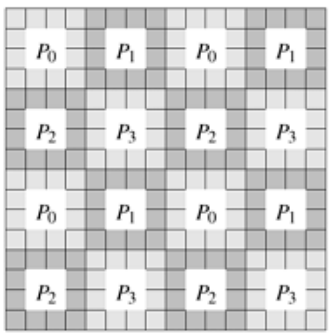
3.1 Assignation (*mapping*) statique

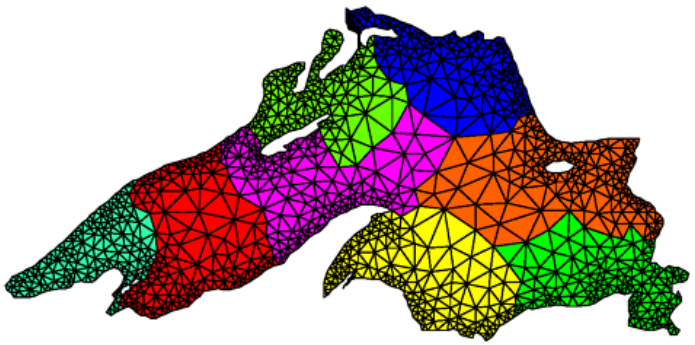
- distribution a priori avant l'exécution
- nécessite la connaissance de la taille des tâches (pas simple !)
- en général la recherche du mapping optimal est NP-complet !

Exemple : 3 processes et 20 tâches de taille $\{20, \ldots, 1\}$: quel mapping ?

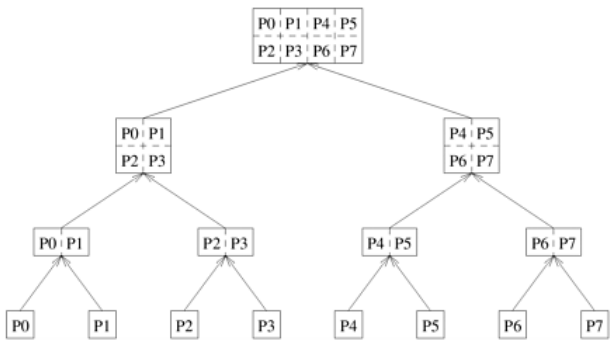
```
P1:   [1, 4, 7, 10, 13, 16, 19] -> 77
P2:   [2, 5, 8, 11, 14, 17, 20] -> 70
P3:   [3, 6, 9, 12, 15, 18]     -> 63
```

Souvent, mapping statique basé sur un partitionnement des données





ou basé sur un partitionnement du graphe de dépendance

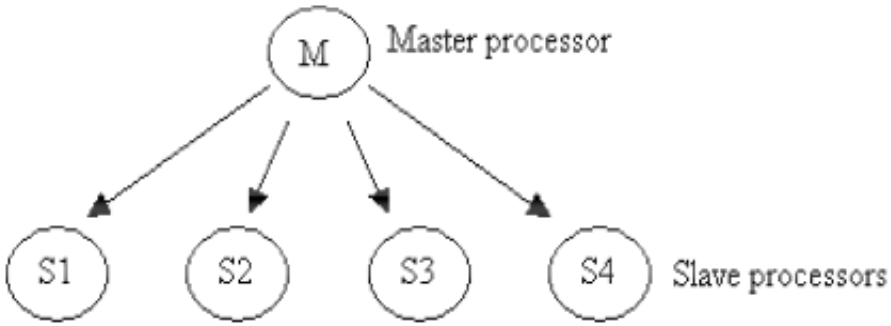


3.2 mapping dynamique

- distribution des tâches durant l'execution
- si la taille des tâche est inconnue ou dépendante du contenu des données (et non de la taille)

suivant une schéma centralisé

ou



NB : attention aux mouvements de données !

Le design d'un algorithme parallèle est fait...

implémentation pour un (super-)ordinateur à mémoire partagée ou distribuée ?

si partagée, organiser la synchronisation et éviter la bagarre sur les données communes

if distribuée, faire communiquer les processes

Les communications sont implicites
car toute la mémoire est accessible par tous les processes

Les paradigmes de programmation se concentrent sur :

- le lancement automatique de tâches en parallèle (moyennant un peu d'aide du développeur)
- le partage de données ("c'est à moi ! non c'est à moi ! non c'est à moi !")
- synchronisation ("on fait quoi dans quel ordre")

Interfaces bas (threads), medium (openMP, Intel TBB) ou haut niveau (Python multiprocessing, R parallel)

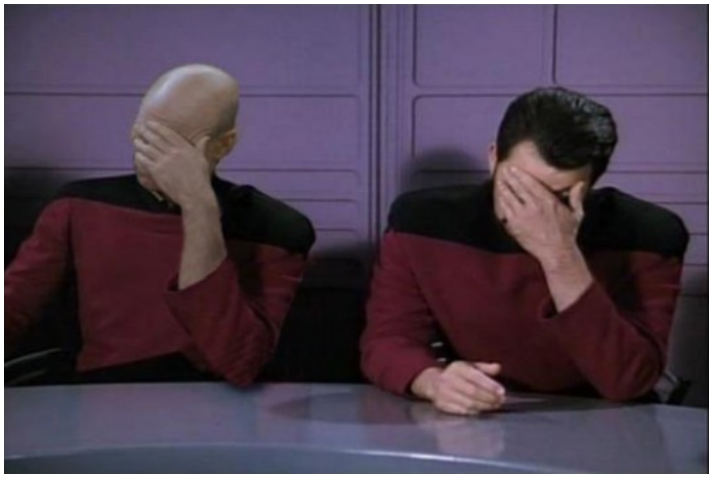
Le programmeur est responsable de la réécriture du code
et doit ajouter des communications explicites..

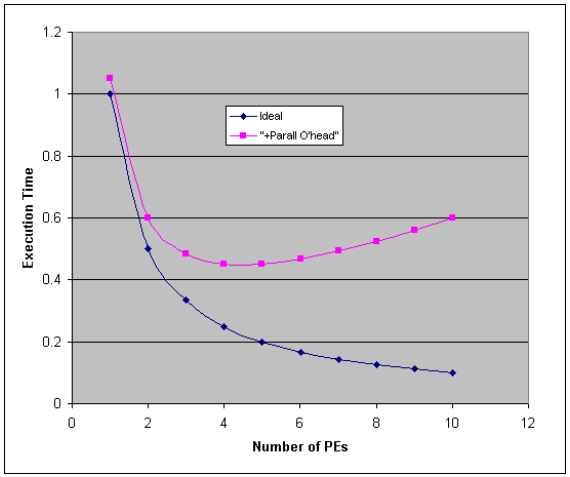
Le paradigme d'échange de message :

- suppose un partitionnement de l'espace mémoire ("chacun son jardin") - les processes ont une mémoire exclusive (même virtuellement)
- impose de coder la parallelisation ("faut tout leur dire")
- toutes les interactions nécessite la coopération process-à-process ("l'un a la donnée, l'autre veut la donnée").

Beware of a snake in the grass

De multiples sources d'*overhead* qui limite le passage à l'échelle (*scalabilité*)





- temps de communication, accès à des données communes
- attente : mauvaise répartition de charges
- excès de calculs

ET la loi d'Amdawl : le speedup théorique maximum est donné par $\frac{1}{\alpha+(1-\alpha)/c}$, où α est la fraction du temps total concernant la partie séquentielle and c le nombre de processes

Example : linear algebra

ScaLAPACK is a library of high-performance linear algebra routines for parallel (distributed memory) machines

`http://www.netlib.org/scalapack/`

- using explicit message passing

- block cyclic data distribution
- block-partitioned algorithms (in order to minimize the frequency of data movement between different levels of the memory hierarchy)
- source code the same as in the sequential case

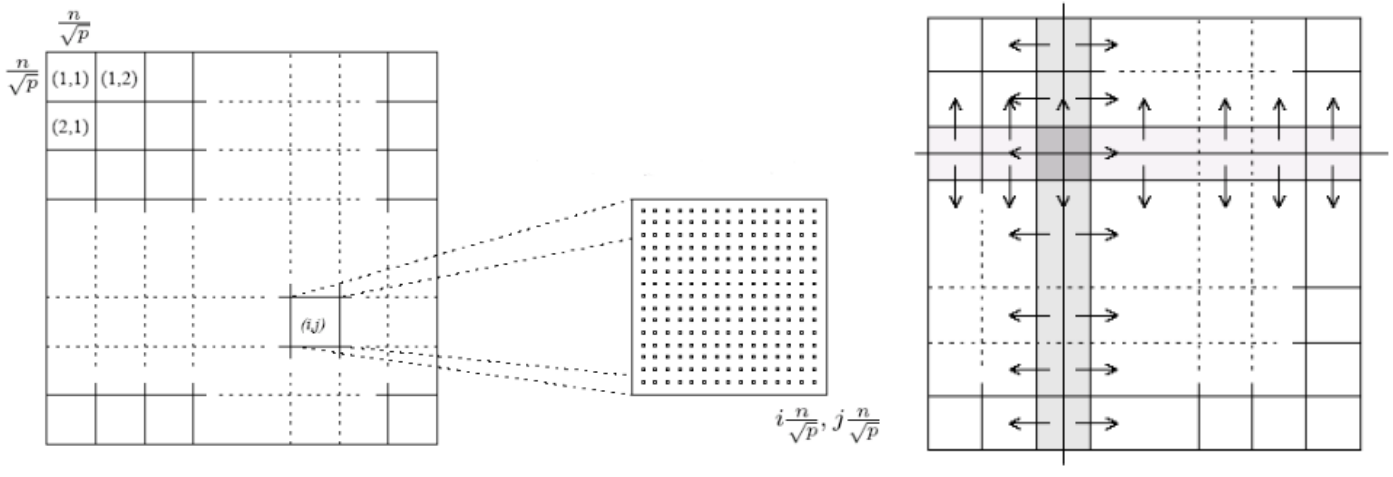
Example : dynamic programming

All shortest paths in a graph with n nodes (shared/distributed memory)

S1
- shared or duplicate matrix for the p processes (footprint !)
- each process executes the sequential single source shortest path for n/p vertices.
- no communication but at most n processes

All shortest paths in a graph with n nodes (shared/distributed memory)

S2
- decompose matrix in p blocks of size $(n/\sqrt{p})X(n/\sqrt{p})$
- each process solve the all shortest path locally in sub-matrix (graph)
- and broadcast to the neigbours



BIOINFORMATICS APPLICATIONS NOTE Vol. 26 no. 15 2010, pages 1899–1900
doi:10.1093/bioinformatics/btq224

Sequence analysis Advance Access publication April 28, 2010

Parallelization of the MAFFT multiple sequence alignment program

(shared memory)

step 1 all-to-all comparison
multiple processes treat different pairwise alignments simultaneously and independently + sequential algorithm for the tree

BIOINFORMATICS APPLICATIONS NOTE Vol. 26 no. 15 2010, pages 1899–1900
doi:10.1093/bioinformatics/btq224

Sequence analysis Advance Access publication April 28, 2010

Parallelization of the MAFFT multiple sequence alignment program

(shared memory)

step 1 all-to-all comparison
multiple processes treat different pairwise alignments simultaneously and independently + sequential algorithm for the tree

step 2 progessive alignment along the guide tree
(group-to-group alignments from leaves to the root)
$\rightarrow$ tree-based parallelization not efficient since alignment at a node requires the alignments in its child nodes

Example : multiple alignment

BIOINFORMATICS APPLICATIONS NOTE Vol. 26 no. 15 2010, pages 1899–1900
doi:10.1093/bioinformatics/btq224

Sequence analysis Advance Access publication April 28, 2010

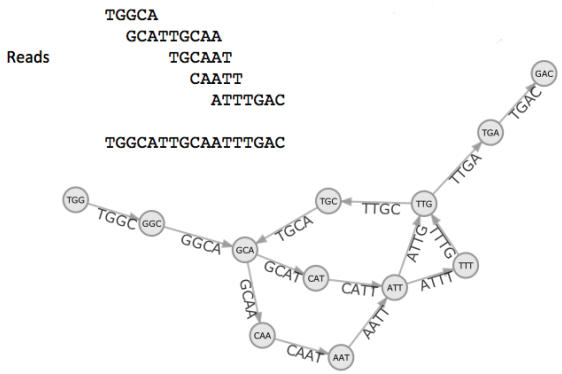
Parallelization of the MAFFT multiple sequence alignment program

(shared memory)

step 1 all-to-all comparison
multiple processes treat different pairwise alignments simultaneously and independently + sequential algorithm for the tree

step 2 progessive alignment along the guide tree
(group-to-group alignments from leaves to the root)
$\rightarrow$ tree-based parallelization not efficient since alignment at a node requires the alignments in its child nodes

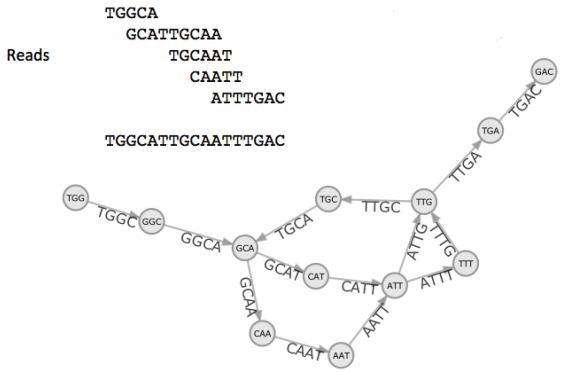
step 3 iterative refinement
(split alignments into 2 sub-alignments to realign)
$\rightarrow$ hill-climbing approach = multiple realignments performed in parallel at random and the best improvement is conserved



RNA-seq : Trinity (shared memory)

- partitions the sequence data into many individual de Bruijn graphs, each representing a given gene
- in parallel, Trinity computes the full-length transcript

Example : NGS assembler



ABySS: A parallel assembler for short read sequence data

Jared T. Simpson, Kim Wong, Shaun D. Jackman, et al.

Assembler : Abyss (shared/distributed memory)

Building a distributed de Bruijn graph

- each sequence broken into k-mers
- each k-mer is assigned to a process in a hash table

- for each k-mer a message is sent to its eight possible neighbors
- if the neighbor exists, the process to which it is assigned is recorded (ajacency information)

Then parallel path finding in the distributed graph...