Introduction au calcul parallèle

Grégory Mounié

3 juillet 2019

Univ. Grenoble Alpes, CNRS, Inria, Grenoble-INP (Ensimag, LIG (UMR 5217) - équipe-projet Inria Datamove) Grenoble-INP est l'institut d'ingénieurie de l'UGA, Grenoble, France

Résumé

Calcul informatique et parallélisme

l'architecture des ordinateurs est complexe

l'algorithmique parallèle $\neq$ l'algorithmique séquentielle

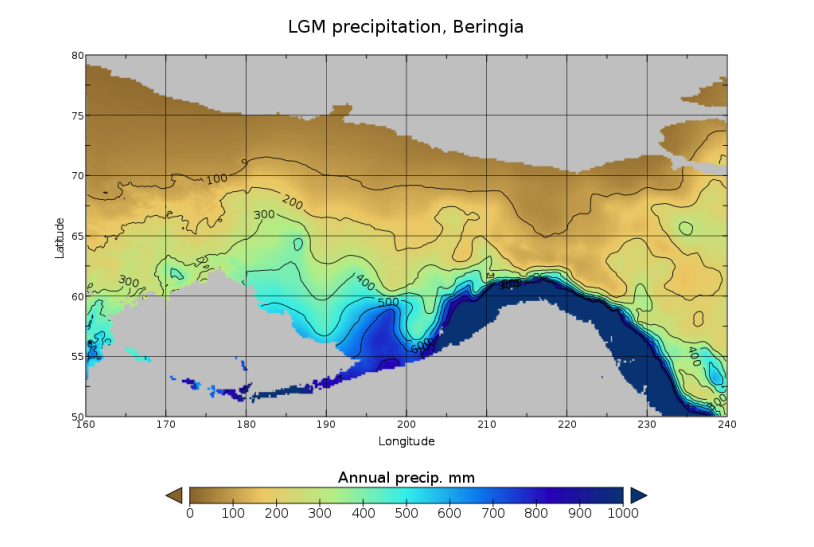
La programmation parallèle

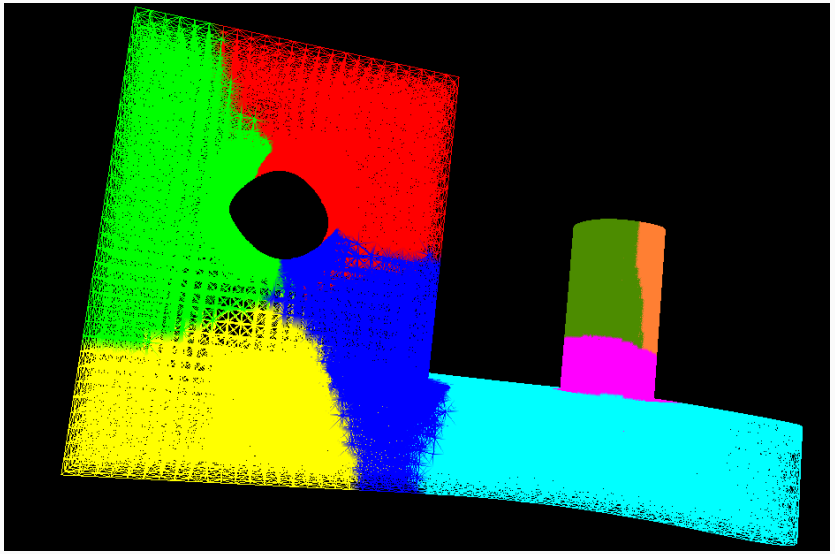
En résumé et conclusion

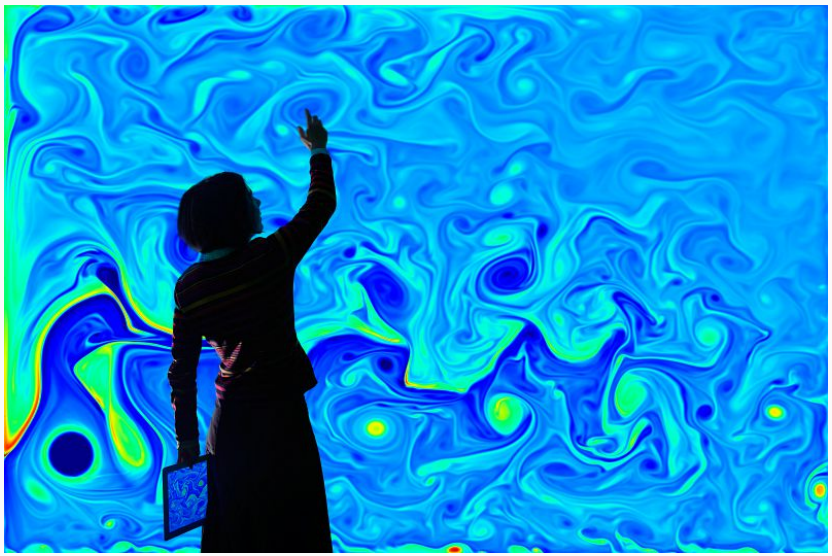
Annexes

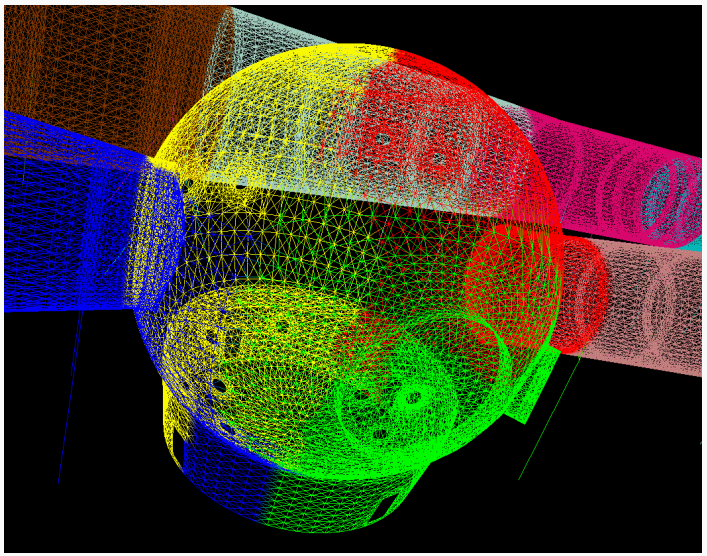
Calcul informatique et parallélisme

Le calcul scientifique, fondateur de l'informatique moderne, est maintenant parallèle.









Calculer plus vite, plus gros, plus loin : les super-calculateurs

Summit (1er du `http://top500.org`), IBM, Oak Ridge (Tennessee), 2,4M cœurs dans 4096 nodes, contenant 2 Power 9 à 22C et 6 Nvidia Volta 96C, 512 Gio de RAM + 800 Gio NVRAM. Réseaux Mellanox Infiniband 100 Gib/s, 90 ns de latence hard, en *Fat Tree*. 148 PFLOPS, pour une consommation de 10 MW.

Les deux plus grosses machines et la majorité de la puissance sont au USA. 219 des 500 sont en Chine. Toutes les machines sont sous Linux.

Trois autres exemples de super-calculateurs

Numéro 11 du TOP 500 PANGEA III, 17 PFLOPS, 1er française, mini-summit à 291 000 cœurs, Total

Numéro 111 et 113 du TOP500 (60ème en 2018) Prolix2 et Beaufix2 (Météo France) : 72000 et 73440 cœurs, répartis en Intel Xeon E5-2698 20 cœurs, réseau Infiniband, pour une consommation de 850 KW chacun.

Numéro 500 du TOP 500 Internet Company N D2 (Lenovo) : 1PFLOPS, 60000 cœurs, Xeon 10C, avec Ethernet 10Gb/s pour une consommation probablement autour du MW

Les super-calculateurs actuels ressemblent à de (très gros) PC, sous Linux, reliés par un réseau (très) rapide. Cette tendance est apparue dans les années 1990, puis s'est démocratisé pour devenir maintenant la norme.

BOINC : le super-calcul planétaire

BOINC est logiciel et un service issu de SETI@HOME et géré par l'université de Berkeley. Des volontaires offrent leur puissance de calcul (SETI@home, Einstein@Home, Folding@Home, etc.).

Au 27 juin 2019 : 17.7 PFLOPS. Il serait 12 ème au top500 derrière PANGEA III, la troisième machine européenne (Pitz Daint, 6ème, Suisse, 21 PFLOPS)

BOINC exécute des applications embarrassingly parallel : peu de données à échanger, de gros et nombreux calculs paramétriques indépendants les uns des autres.

Ce dont je ne parlerai pas dans cet exposé

- Le parallélisme pour calculer plus gros
- le passage à l'ExaFLOPS : le point le plus dur est de ne pas construire une tranche de centrale nucléaire moderne (1 GW). Une grande difficulté technique est le réseau rapide qui compte pour la moitié de la consommation électrique (et du prix)
- L'informatique dans les nuages (Cloud)
- Les grands systèmes distribués (Linky)
- Le stockage et traitement massif de données à grande échelle (Facebook, Google)
- etc.

Codes et démonstrations

Les codes, et exécutions, de cet exposé sont volontairement écrits dans plusieurs langages (Python, Julia, C, C + OpenMP, C++) pour plusieurs raisons :

- concision
- précision (pouvoir rentrer dans les détails)
- comparaison des approches et des performances
- illustration des diverses notations et expressions.

Ils utilisent trois algorithmes :

- produit scalaire de 2 vecteurs de 300 millions de flottants
- calcul récursif de Fibonacci(42)
- fractal de Mandelbrot (code issu la documentation de Numpy, un peu enrichi)

1 calcul ici vaut mieux que 1000 calculs là-bas

Tout est loin du processeur, tout le reste est lent

La vitesse du courant électrique (75% de c) est lente.

75% de c / fréquence d'un processeur à 3.5 Ghz

$$0.75 \times \frac{299792458 m/s}{3.5 \times 10^9} = 6.4 cm$$

Le goulot d'étranglement est le transfert des données ! (Meilleures latences : RAM 10 ns ; Réseaux : 1 μ s ; SSD 30 μ s ; HD 4 ms)

Performance fortement liée aux caches, à tous les niveaux

Les calculs doivent être organisés pour favoriser la localité spatiale et temporelle. Pour un tableau `T[]`, `T[i+1]` doit être manipulé juste après `T[i]` par la même unité de calcul. C'est le contraire du parallélisme !

l'architecture des ordinateurs est complexe

La loi de Moore est encore vraie (doublement de la densité de transistors)

La fréquence des processeurs n'explique plus sa vitesse

1. Un Iphone 8 calcule à 54.4 GFLOPS avec 6C à 2.39 GHz.
2. En 1993, lors du premier Top50, il aurait été numéro 2 !
3. La première machine de 1993 (59 GFLOPS) était composées de 1024 SuperSparc à 32 Mhz.

$\frac{54}{59} \times 1024 \times 32 \sim 12 \times (2390)$

L'augmentation de la fréquence n'explique pas tout. Il manque un ordre de grandeur !

Le parallélisme interne dans le processeur

Chaque processeur est beaucoup devenu plus complexe. Il réalise maintenant plusieurs opérations simultanément.

Points majeurs

- Multi-cœurs
- Vectorisation
- Out-of-Order
- Prédiction de branchement
- Cache
- Des problèmes de sécurité posés par les 3 points précédents (Meltdown et Spectre)
- De nombreuses unités de calcul indépendantes pipelinées

Le calcul vectoriel programmable

Au siècle dernier, les super-calculateurs CRAY (CRAY 1, 1975, 160 MFLOPS) permettait de calculer très rapidement sur de grands vecteurs de nombres flottants.

Puis, sont apparues les cartes graphiques grand public pour les jeux en 3D.

Les GPGPU

Ils sont des accélérateurs vectoriels programmables et permettent d'obtenir d'excellentes performances avec peu de Watt, sur des calculs réguliers (Algèbre linéaire, *Deep Learning*).

La Nvidia TITAN V (dec 2017, 3700 €) a une puissance crête de 110 TeraFLOPS. Il reste difficile de dépasser les 25-50% d'efficacité réelle mais elle aurait été première du TOP 500 jusqu'en 2004.

Vectorisation automatique : produit scalaire

Un produit scalaire en Julia (compilé, syntaxe $\sim$ python)

```
function dotp(x,y)
    v = 0.0
    for i=1:length(x)
        @inbounds v += x[i] * y[i]
    end
    return v
end

n = 300000000
x = rand(Float32,n)
y = rand(Float32,n)

@time v1 = dotp(x,y)
```

Temps d'exécution du produit scalaire (1/2)

Produit scalaire de 2 vecteurs de 300 millions de flottants

Langage + détails	Temps
Python avec slice (ajoute une copie temporaire)	KILL
Python avec for et Range	250.9
Python fonctionnel	270.69
10 * 1/10 Python avec slice	27.84
10 * 1/10 Python avec for et Range	18.76
10 * 1/10 Python fonctionnel	14.98
Julia fonctionnel	1.15
Python + Numpy	0.44
Julia (exemple précédent)	0.31

Temps d'exécution du produit scalaire (2/2)

Produit scalaire de 2 vecteurs de 300 millions de flottants

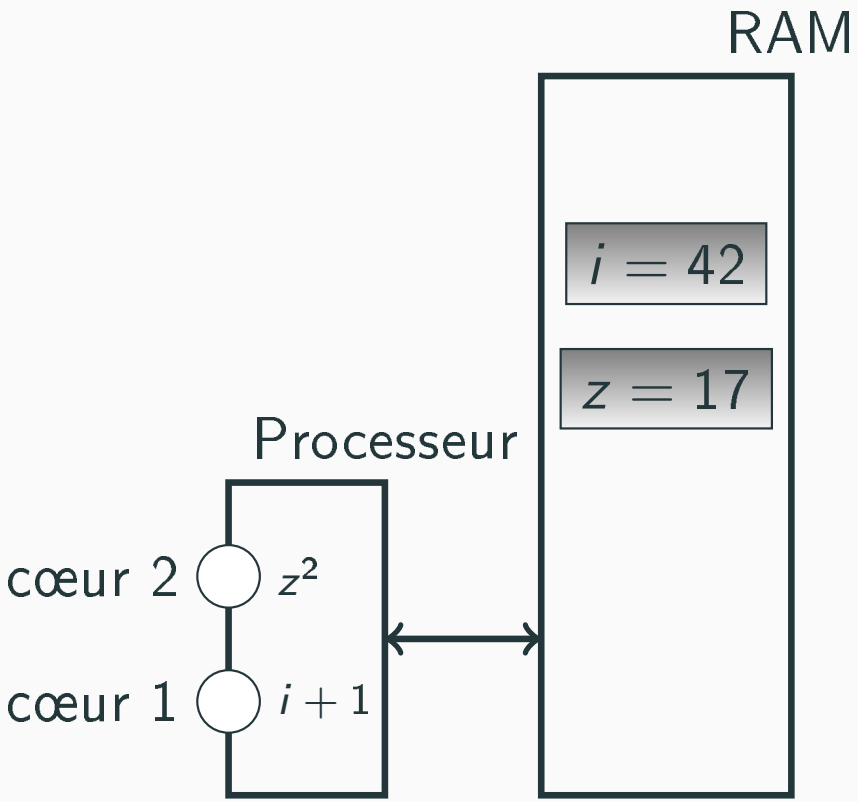
Langage + détails	Temps
Python + Numpy	0.44
Julia (celui du slide précédent) ; Java	0.31
C ; Go ; Rust ; D	0.33
C Vectorisé à la main	0.31
C vectorisé -ffastmath ; C++ Lib	0.26
C++ loop	0.24
C + OpenMP ; C++ TBB	0.23
C + MKL OpenMP ; Fortran ; C -ffastmath	0.22

La haute performance est aussi affaire de programmation

`-O3 -march=native` ; Vectorisation (automatique) ; "Bon" niveau d'abstraction ; Parallélisme ; Précision ; Bibliothèques ;

Le parallélisme en mémoire partagée

Un programme en exécution est un processus. L'entité qui exécute la séquence d'instruction d'un programme est un thread. Il est possible, dans un processus, d'utiliser plusieurs threads. Chacun pouvant être exécuté simultanément par plusieurs cœurs.




```
float f(float z) {
  return z*z; }
int g(int i) {
  return i+1; }
void main() {
  t1 = thrd_create(f,42);
  t2 = thrd_create(g,17);
  thrd_join(t1,0);
  thrd_join(t2,0);
}
```

Un programme multi-threadé est "facile" à écrire

Presque tous les langages "modernes" compilés ont un support pour faire tourner efficacement des threads : C, C++, Objective-C, D, Java, Go, Rust, Ada, OCaml, Haskell, Erlang, Fortran (avec OpenMP, en 2018 pour le langage de base)

À petite échelle (quelques cœurs)

- facile à mettre en œuvre car ils partagent déjà la même mémoire physique (RAM). Il *"suffit"* alors d'écrire un programme qui fasse plusieurs séquences simultanément (threads, processus léger)

Les threads sont difficiles à utiliser efficacement

À moyenne et grande échelle (grand nombre de cœurs)

Beaucoup plus difficile, car les choix de découpage et d'ordonnancement ont un poids important dans la performance.

Les facteurs à prendre en compte

Ils sont nombreux et chacun peut dégrader la performance de plusieurs ordres de grandeur. Les plus importants :

- bande passante mémoire ; caches
- verrouillage des threads et des données à organiser (NUMA, couleur, pollution, faux-partage),
- scrutation versus interruption et bande passante des entrées-sorties,
- topologie des périphériques d'entrées-sorties.

Les langages interprétés et la mémoire partagée

Pour les langages interprétés (ou compilé qui s'en rapproche comme Julia), ce n'est pas toujours facile d'utiliser efficacement des threads :

Les langages très dynamiques (Julia, Python, Ruby) ont bien la notion de threads, ou de tâches, mais l'interpréteur, ou le modèle mémoire pour Julia, ne fonctionne pas en parallèle (pour l'instant). Les threads ou les tâches s'exécutent l'un après l'autre. Cela reste utile pour les entrées/sorties bloquantes et les appels aux bibliothèques externes (ex. Numpy).

Contre-exemples de langages interprétés avec des threads efficaces

Perl 5 et 6, les langages interprétés fonctionnels comme LISP et ses enfants comme Scheme ou Clojure (JVM Java).

Le parallélisme en mémoire distribuée

Pour obtenir un très grand nombre cœurs, il faut mettre plusieurs PC dans une grande pièce réfrigérée et de les relier par un réseau rapide.

C'est le modèle dominant depuis la fin des années 1990. Il s'est accentué au fil du temps : uniformisation des architectures de processeurs (X86-64, quelques unités Power et SPARC), 1 système (Linux), 3 "cartes graphiques" (Nvidia, AMD, (Intel))

l'algorithmique parallèle $\neq$ l'algorithmique séquentielle

Les 2 buts de l'algorithmique

1. La transmission entre humain de la "recette" sans se noyer dans les détails nécessaires pour implanter l'algorithme
    - C'est la nuance entre algorithmique (entre humains) et programmation (humain vers machine)
2. La comparaison des algorithmes pour choisir lesquels implanter : en séquentiel, on compte les opérations élémentaires. C'est grossier mais suffisant.
    - pour des algorithmes avec la même complexité, c'est plus délicat : dépend d'autres facteurs (ex. localité)

Complexité d'un algorithme parallèle

Contrairement à l'algorithmique séquentiel, il n'y a pas de modèle universel de machine. Il y a de nombreux modèles différents !

Deux grandes familles de modèles

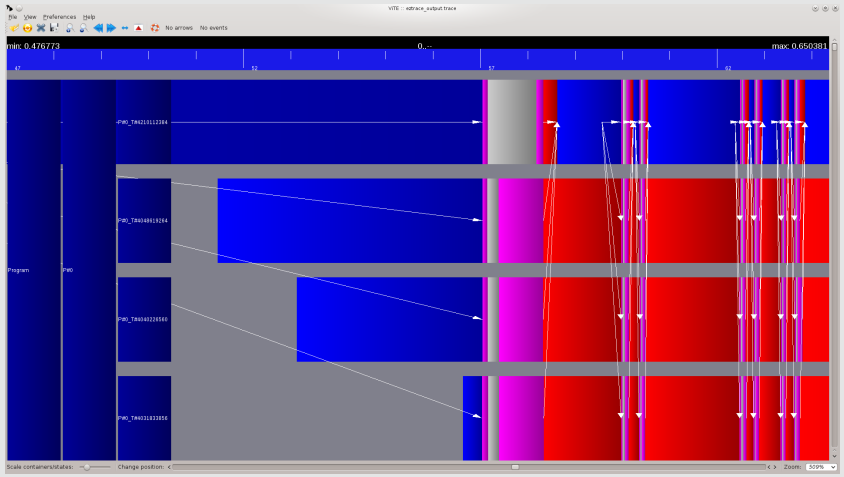
1. Les extensions du modèle RAM (séquentiel), comme PRAM, où l'on compte uniquement les calculs, sans, explicitement, de coûts de communication,
2. Les modèles prenant en compte finement les coûts de communications, où l'on compte surtout les messages et leurs coûts.

Comparer des algorithmes parallèles avec PRAM

Comment compter le coût des opérations

À chaque instant, certaines machines calculent tandis que d'autres attendent des communications :

- Le temps d'exécution parallèle est le temps entre le premier et le dernier calcul
- on compte le nombre maximum de processeurs utilisés
- on multiplie le tout : c'est le travail parallèle de l'exécution



Le chemin critique et le travail

Lors de l'écriture d'un programme parallèle, deux grandeurs guident la conception de l'algorithme :

le chemin critique (CP) la plus longue séquence d'opérations, communications comprises. Plus elle est courte par rapport aux calculs à faire, plus il sera facile d'aller plus vite en ajoutant des unités de calculs

le travail (W) c'est le nombre global d'opérations qui seront réparties.

Exemple (Loi de Amdhal, approx. du temps d'exécution parallèle)

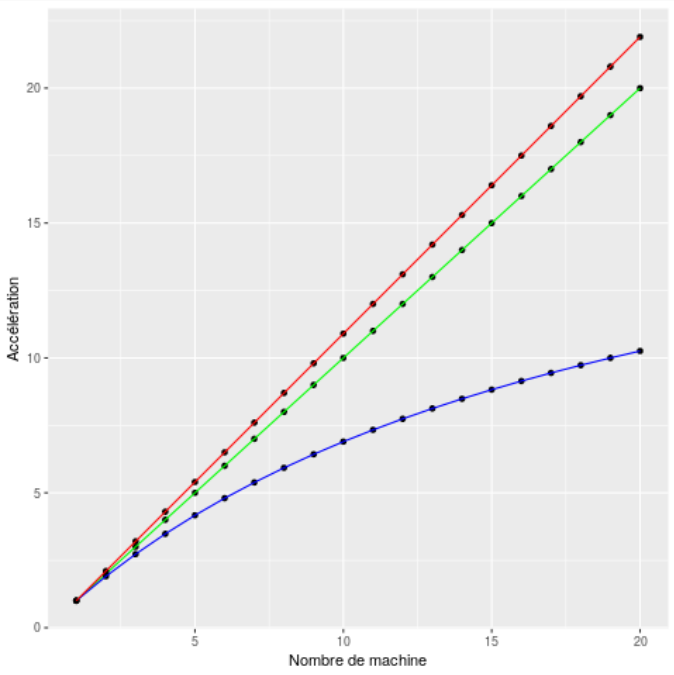
Le temps d'exécution de l'application parallèle sur m machines :

$$T_m = CP + \frac{W}{m}$$

L'accélération (Speedup)

C'est la métrique la plus utilisée. Elle consiste à calculer le rapport entre le temps du meilleur algorithme séquentiel et le temps parallèle obtenu en utilisant m machines.

$$S_m = \frac{T_1}{T_m}$$



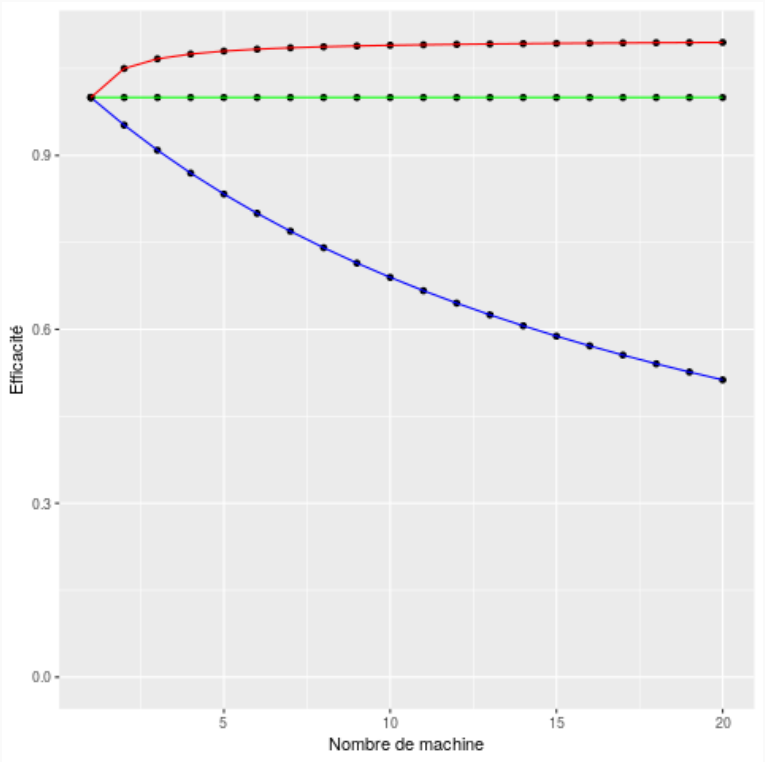
Idéalement, il devrait être proche de m. Des accélérations super-linéaires sont possibles (effets de caches, ou de changement dans l'ordre des opérations).

L'efficacité

L'efficacité est le rapport entre le travail effectué en séquentiel et en parallèle.

$$Eff_m = \frac{S_m}{m} = \frac{T_1}{mT_m}$$

.



Les algorithmes parallèles en pratique

Le découpage du travail est nécessaire pour pouvoir le répartir.
Mais si l'on découpe trop, les surcoûts liés au découpage peuvent dégrader les performances

Les coûts additionnels induits par le découpage

- L'algorithme de découpage lui-même
- le transfert ou l'arbitrage du découpage
  - ou la duplication de calcul de découpage pour éviter les transferts
- les différents transferts en cours de calculs nécessaires pour maintenir la cohérence des résultats
- le transfert des résultats
- recoller les résultats

Exemple de Fibonacci avec des tâches parallèles

LE B.A.BA : séparer l'expression du calcul de son exécution

Écrire le programme sous la forme d'un ensemble de tâches à faire. Il faut exprimer les dépendances entre les tâches.

Les tâches ne dépendent pas du nombre de processeurs qui feront le calcul. Mais l'idéal est de pouvoir adapter ce nombre à l'exécution pour minimiser les surcoûts.

Exemple (Fibonacci en C + OpenMP)

Calcul récursif très parallèle mais à très faible coût unitaire : il faut maîtriser les surcoûts pour aller plus vite que le séquentiel.

Les algorithmes supplémentaires impliqués dans l'exécution parallèle

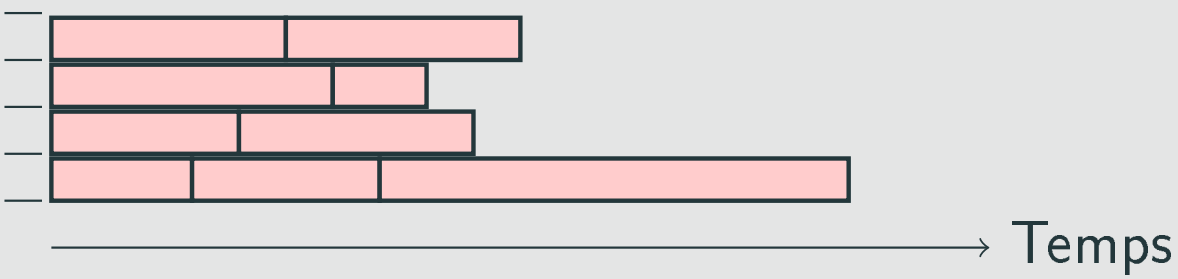
Paralléliser un algorithme demande au moins deux choses :

1. Découper le travail en morceaux à répartir ; si besoin, échanger les conditions aux bords de découpe pendant les calculs ; recoller les morceaux à la fin.
2. Répartir le travail : choisir qui fera quoi et quand : c'est l'ordonnancement.

L'ordonnancement : les algorithmes de listes

Dès qu'un processeur est libre, commencer dessus, immédiatement, une des tâches restantes

Avec plusieurs processeurs identiques et un coût de communication négligeable, il est facile d'obtenir un bon ordonnancement avec un algorithme de liste (au pire à un facteur 2 du meilleur possible).



Dans tous les autres cas, c'est plus difficile

L'algorithme de décision peut se tromper lourdement. De l'ordre du nombre de machines, ou de la granularité calcul communication.

L'ordonnancement : l'art de répartir le travail à faire

Le problème de l'ordonnancement devient très difficile à approcher et les heuristiques utilisées peuvent se tromper (beaucoup) si, par exemple :

- les processeurs ont des vitesses très différentes et le temps de tâches est inconnu.
- le temps d'exécution d'une tâche sur chaque processeur est connu mais dépend fortement de son processeur (processeurs non uniformes) et elles arrivent au fil de l'eau
- le coût des communications est grand par rapport aux calculs
- le placement des données influe fortement sur le temps d'exécution d'une tâche
- si les tâches sont multi-processeurs et communiquent

Les heuristiques utilisées pour l'ordonnancement

3 grandes familles d'heuristiques

statiques le programme fixe le lieu et la date des calculs (GPGPU)

dynamique centralisées schéma maître-esclave, un maître sert le travail à des esclaves (BOINC). Idéal pour les calculs indépendants avec peu de données. L'esclave demande du travail et renvoie la réponse quand il a fini puis il redemande du travail.

dynamique distribuées par vol de travail. Une liste de tâches par cœur. Lorsque sa liste est vide, un cœur choisit au hasard une victime et va lui prendre une tâche (Threads en Linux et Windows)

La programmation parallèle

Le modèle séquentiel : l'automate à état (mémoire)

Le modèle séquentiel consiste à avoir une seule séquence d'instructions, avec ses `if`, `for`, ses fonctions, ses calculs, ses lectures et écritures de variables scalaires et de ses tableaux.

La mémoire comme médium (temporel) de communication

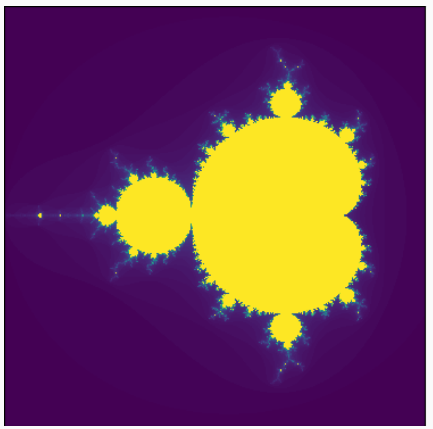
La communication entre des deux parties du programme passe par ces lectures et écritures de la mémoire. Par construction de la séquence, la lecture d'une donnée est postérieure à son écriture. C'est l'ordre d'exécution des instructions qui garantie la cohérence du résultat final.

Exécution simultanée de plusieurs séquences d'instructions (facile)

Si les séquences ne lisent et modifient pas la même chose, il n'y a aucun problème de cohérence.

Exemple d'applications typiques :

- simulations paramétriques (code de simulation identique mais avec des centaines de milliers de valeurs de paramètres différents)
- simulations stochastiques : marche aléatoire, Monté Carlo
- rendu d'image avec des pixels indépendants : fractal de Mandelbrot, lancé de rayon.



Le problème de cohérence des données (difficile)

Si les séquences modifient et lisent les mêmes données, il peut y avoir des problèmes de cohérence.

```
#include <stdio.h>
#include <omp.h>
int compteur=0;
#define NB 100000

void add() { compteur++; }

void boucle( void(*f)() ) {
  for(int i=0; i < NB; i++)
    f();
}
```

Le problème de cohérence des données (difficile)

```
int main(int argc, char **argv) {
  int nbth=1;
#pragma omp parallel shared(nbth)
  {
#pragma omp single
    nbth = omp_get_num_threads();

    boucle(add);
  }
  printf("%d / %d (%d threads)\n",
         compteur, NB * nbth, nbth);
}
```

Résultat de l'incrémentation d'un compteur par plusieurs threads

```
> ./addition_omp
217688 / 400000 (4 threads)
```

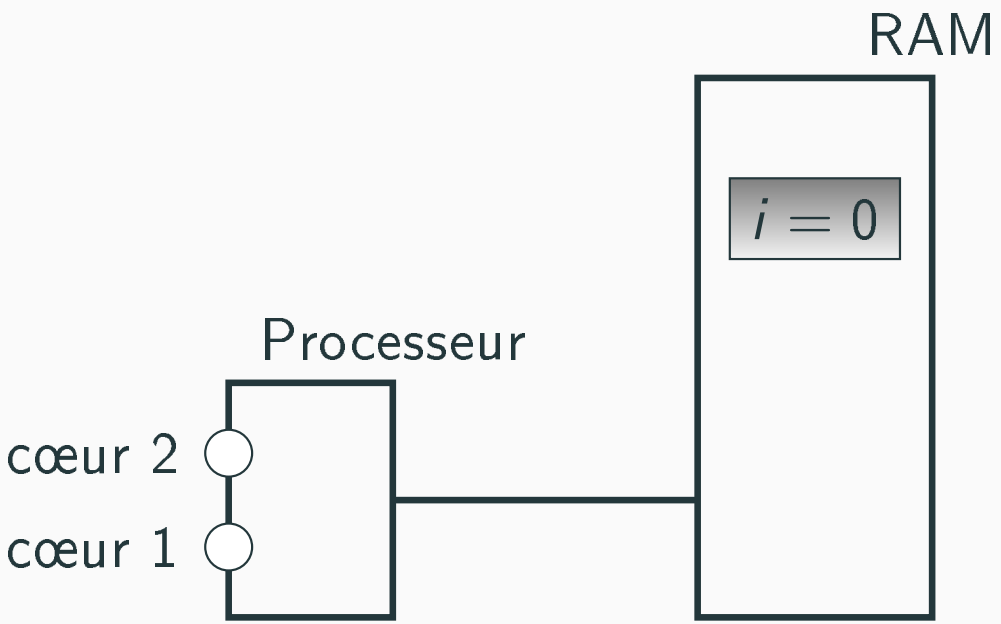
Le résultat de 4 threads faisant chacun 100000 incrémentations simultanément n'est pas 400000 !

Le problème est accentué pour des données plus complexes (tableaux, listes, tables de hachage, base de données)

Le problème de cohérence des données

Pour faire i++, il faut réaliser 3 opérations

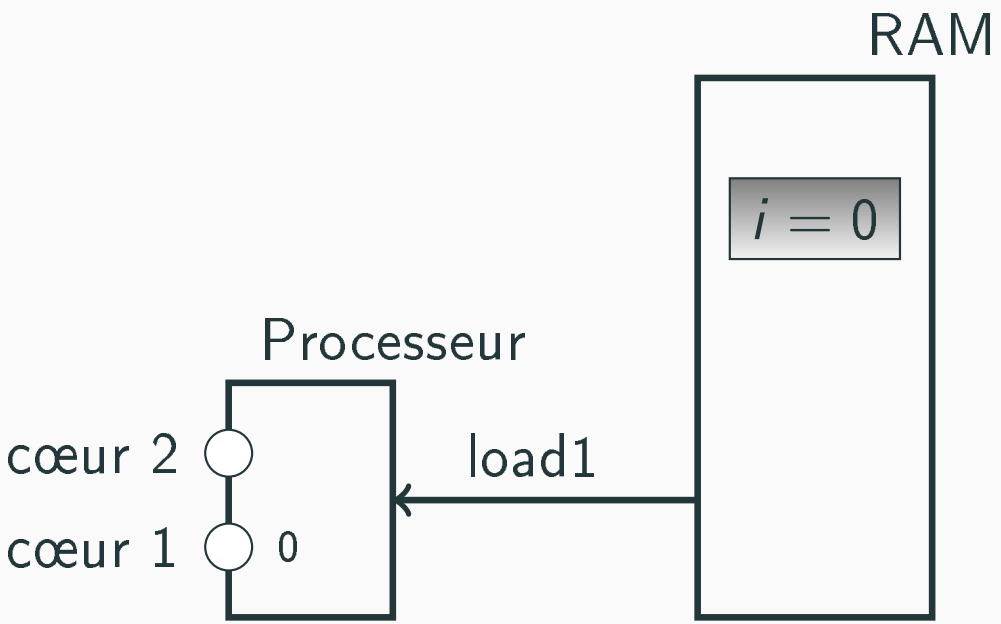
1. `load %i, R` charger la valeur de i depuis la mémoire dans un registre d'un cœur
2. `inc R`, incrémenter le registre
3. `store %i, R` écrire le contenu du registre dans la mémoire



Le problème de cohérence des données

Pour faire i++, il faut réaliser 3 opérations

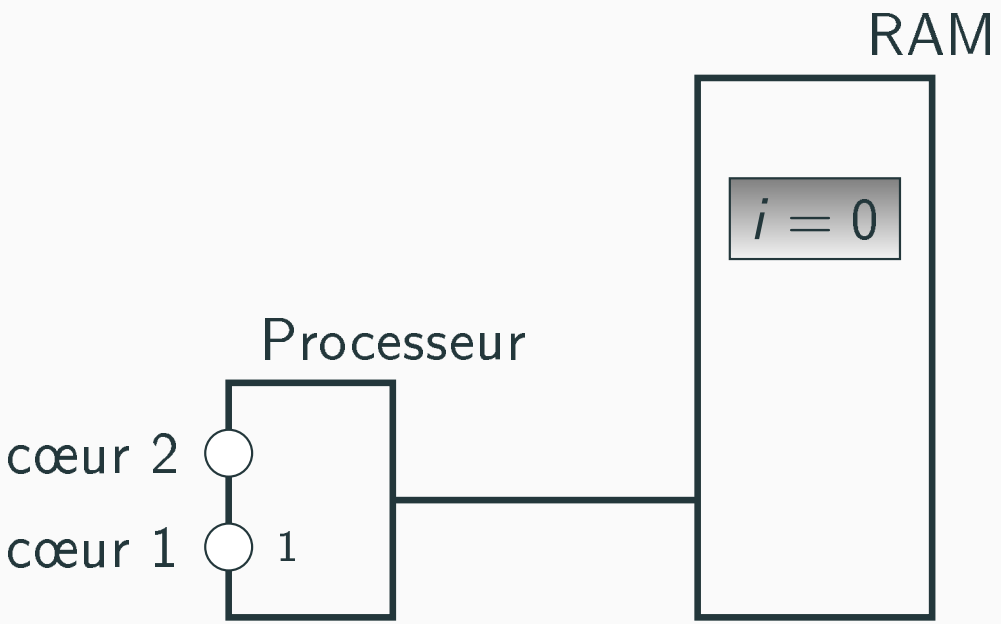
1. `load %i, R` charger la valeur de i depuis la mémoire dans un registre d'un cœur
2. `inc R`, incrémenter le registre
3. `store %i, R` écrire le contenu du registre dans la mémoire



Le problème de cohérence des données

Pour faire i++, il faut réaliser 3 opérations

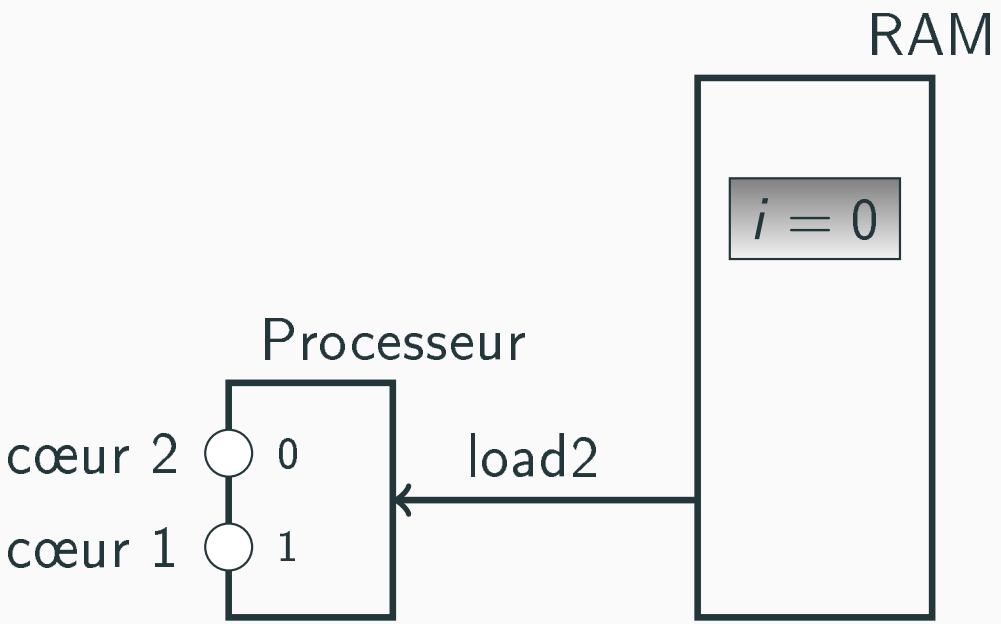
1. `load %i, R` charger la valeur de i depuis la mémoire dans un registre d'un cœur
2. `inc R`, incrémenter le registre
3. `store %i, R` écrire le contenu du registre dans la mémoire



Le problème de cohérence des données

Pour faire i++, il faut réaliser 3 opérations

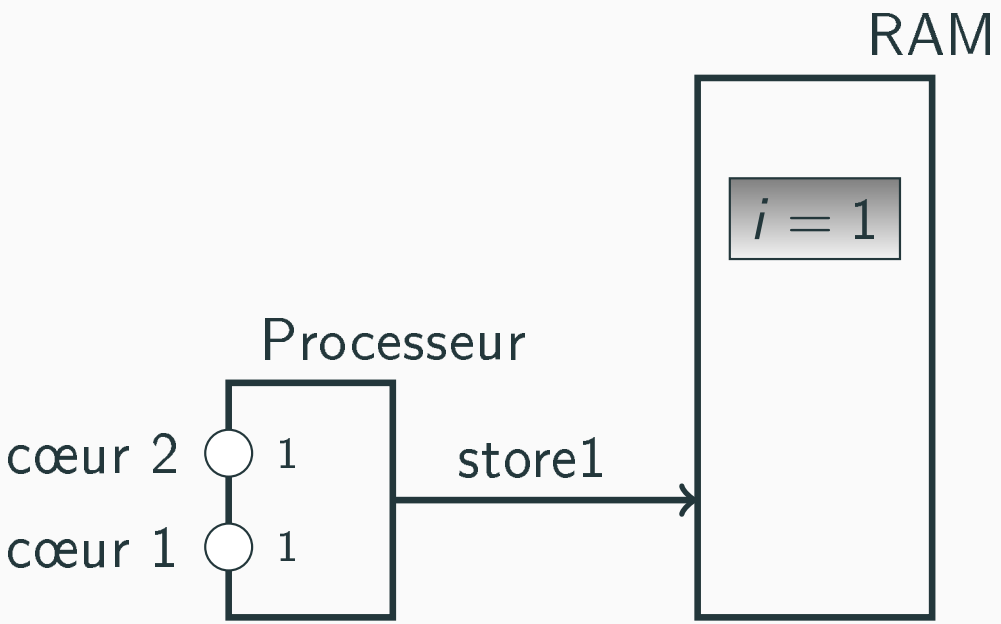
1. `load %i, R` charger la valeur de i depuis la mémoire dans un registre d'un cœur
2. `inc R`, incrémenter le registre
3. `store %i, R` écrire le contenu du registre dans la mémoire



Le problème de cohérence des données

Pour faire i++, il faut réaliser 3 opérations

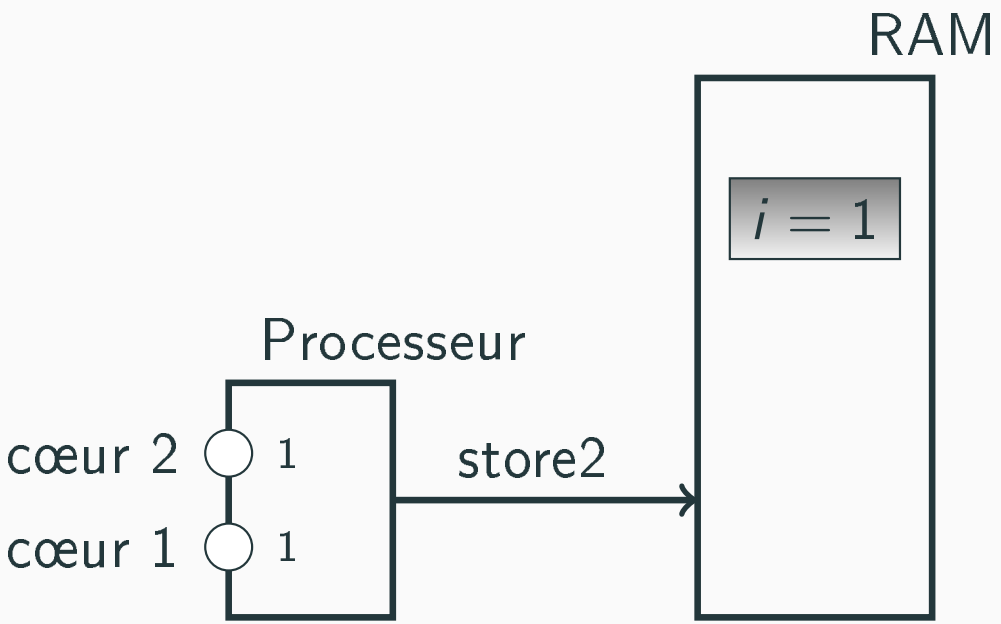
1. `load %i, R` charger la valeur de i depuis la mémoire dans un registre d'un cœur
2. `inc R`, incrémenter le registre
3. `store %i, R` écrire le contenu du registre dans la mémoire



Le problème de cohérence des données

Pour faire i++, il faut réaliser 3 opérations

1. `load %i, R` charger la valeur de i depuis la mémoire dans un registre d'un cœur
2. `inc R`, incrémenter le registre
3. `store %i, R` écrire le contenu du registre dans la mémoire



Le problème de cohérence des données

Exemple (Code d'addition avec plusieurs threads en C-OpenMP)

Le code mesure le temps des trois versions :

- fausse
- avec un verrou (mutex, lock)
- avec une opération atomique

3 familles de solutions (1/2)

Synchronisation

On "séquentialise" les parties de code qui modifient à plusieurs des portions commune de l'état global : blocage temporaire de certaines séquences pour n'avoir qu'une séquence à la fois modifiant les données communes.

Inconvénient il est très facile de provoquer des comportements indésirables : famine, interblocage, inversion de priorité, etc.

Les opérations atomiques (complexes à utiliser)

Le processeur possède des instructions spéciales pour manipuler 1 entier en mémoire de manière atomiques.

3 familles de solutions (2/2)

Échange de messages

Les parties modifiées par chaque séquence sont strictement séparées, mais on échange des messages (Send et Recv) pour communiquer entre les séquences

Avantage les messages peuvent circuler sur un réseau

Avantage programmation proche d'un programme séquentiel

Avantage la description des communications complexes permet leurs optimisations

Inconvénient chaque communication ajoute un coût supplémentaire

Inconvénient lors gros échanges de données, il faut les synchroniser (la mémoire est bornée) : famine, interblocage, etc.

Exemple (Fibonacci avec des échanges de messages)

Ordonnancement statique calculé par chaque processus. Envoie du résultat au processus 0. Code en C++ + Boost-MPI.

Fractal de Mandelbrot en python avec des processus

Modèle : MAP

Les données sont rangées sous la forme d'un vecteur. Une fonction est appliquée sur chaque élément du vecteur découpé en plusieurs morceaux. Un pool de processus se partage les morceaux.

Le modèle d'exécution sous-jacent est un modèle d'échange de message mais il est automatisé.

Exemple (Mandelbrot en Numpy + Multiprocessing)

En partant de la version de la documentation de Numpy : partage du vecteur des points et communications entre les processus gérés par Python

En résumé et conclusion

Le parallélisme : omniprésent mais restant difficile

Les architectures des processeurs sont devenues parallèles. Cela pousse à mettre du parallélisme partout :

- inter-processeurs : out-of-order, vectorisation, accélérateurs
- en mémoire partagée, en utilisant des threads
- en mémoire distribués entre des machines reliés par un réseau, en échangeant des messages

Le parallélisme impose de traiter des problèmes supplémentaires comme l'ordonnancement.

Les langages incluent de plus en plus des opérateurs de haut-niveaux pour faciliter l'écriture de programmes parallèles, mais sans vraiment diminuer le niveau d'expertise en parallélisme nécessaire pour écrire un programme qui calcule plus rapidement.

Crédits images et photos

Slide d'Introduction

- Simulation de climat : CC Wikimédia
- Clémentine Prieur (Courants océaniques) : 50 ans de Inria
- Maillage : image venant du site web de Scotch, le logiciel de partition de Graph

Sumit

- Image prise sur le site web du centre de Oak Ridge

BOINC, Seti

- Logo pris sur les sites concernés

Codes

Les codes de démonstrations sont disponibles dans l'entrepôt git de cette présentation.

`https://gricad-gitlab.univ-grenoble-alpes.fr/mounieg/IntroParallelisme.git`

Annexes

2018 : Calculer plus vite, plus gros, plus loin

Le numéro 1, en mai 2018 du `http://top500.org` des super-calculateurs, Sunway TaihuLight, 10 000 000 cœurs, répartis en processeurs à 260 cœurs, 93 PétaFLOPS, pour une consommation de 15 MW

Crédit TaihuLight

- Image prise sur le site web du centre de Wuxi

2018 : Trois autres exemples de super-calculateurs

Numéro 61 et 62 du TOP500 Prolix2 et Beaufix2 (Météo France) : 72000 et 73440 cœurs, répartis en Intel Xeon E5-2698 20 cœurs, réseau Infiniband, pour une consommation de 850 KW chacun.

Numéro 500 du TOP 500 NASA Discover SCU11 : 17000 cœurs, réparti en Xeon 14 cœurs, pour une consommation de 1.2 MW

Les super-calculateurs actuels ressemblent à de (très gros) PC, sous Linux, reliés par un réseau (très) rapide. Cette tendance est apparue dans les années 1990, puis s'est démocratisé pour devenir maintenant la norme.

-