Calcul Parallèle

Vincent Loechner
loechner@unistra.fr

UFR de Mathématique et d'Informatique
Université de Strasbourg

et

équipe ICPS
laboratoire ICube (UMR CNRS 7357) / INRIA

Plan

Introduction

Qu'est-ce que le parallélisme ?

→ exécuter plusieurs actions coordonnées en même temps

> *En informatique, le parallélisme consiste à mettre en œuvre des architectures [...] permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci. Ces techniques ont pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible.*
> (source : wikipedia)

Introduction

Qu'est-ce que le parallélisme ?

→ exécuter plusieurs actions coordonnées en même temps

> *En informatique, le parallélisme consiste à mettre en œuvre des architectures [...] permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci. Ces techniques ont pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible.*
> (source : wikipedia)

Les **architectures parallèles** sont les ordinateurs sur lesquels ce paradigme est utilisable ;
les **modèles de programmation parallèles** sont les techniques de programmation qui permettent de l'exploiter.

Temps d'exécution d'un programme

$$t_{\text{exéc}} = n_{\text{instructions}} * t_{\text{instruction}}$$

Introduction

Temps d'exécution d'un programme

$$t_{\text{exéc}} = n_{\text{instructions}} * t_{\text{instruction}}$$

Temps d'exécution d'un programme parallèle

$$t_{\text{exéc}} = \frac{n_{\text{instructions}} * t_{\text{instruction}}}{p}$$

Introduction : quelques *challenges*

Applications numériques

- → simulation physique : éléments finis, maillage de l'espace
- météo, modélisation globale et changements climatiques
- mécanique des fluides (aéronautique, moteurs, nucléaire, ...)
- simulation de matériaux (composites, catalyseurs, ...)
- → problèmes à N-corps
- astronomie, astrophysique
- modélisation moléculaire (médicaments, génôme, ...)

Applications informatiques

- temps réel, applications embarquées
- traitement d'image, visualisation, réalité virtuelle
- bases de données : systèmes d'information géographique, web, ...

Introduction : performance

1 flop/s = un calcul en virgule flotante par seconde
(*FLoating OPeration per Second*)

année	**vitesse**		**ordinateur**
1947	1 kflop/s	10^3	ENIAC
1984	50 kflop/s	10^4	8087 (co-processeur IBM PC)
1984	800 Mflop/s	10^9	**Cray X-MP/48**
1997	23 Mflop/s	10^7	Intel Pentium MMX (200MHz)
1997	1 Tflop/s	10^{12}	**Intel ASCI Red**
2007	1.5 Gflop/s	10^9	Core 2 Duo (2.4Ghz)
2007	500 Tflop/s	10^{14}	**IBM BlueGene/L**
2017	10 Tflop/s	10^{13}	Core i7 + GPU
2017	100 Pflop/s	10^{17}	**Sunway TaihuLight**

`http://top500.org`

Introduction : vitesse

Un produit de matrices

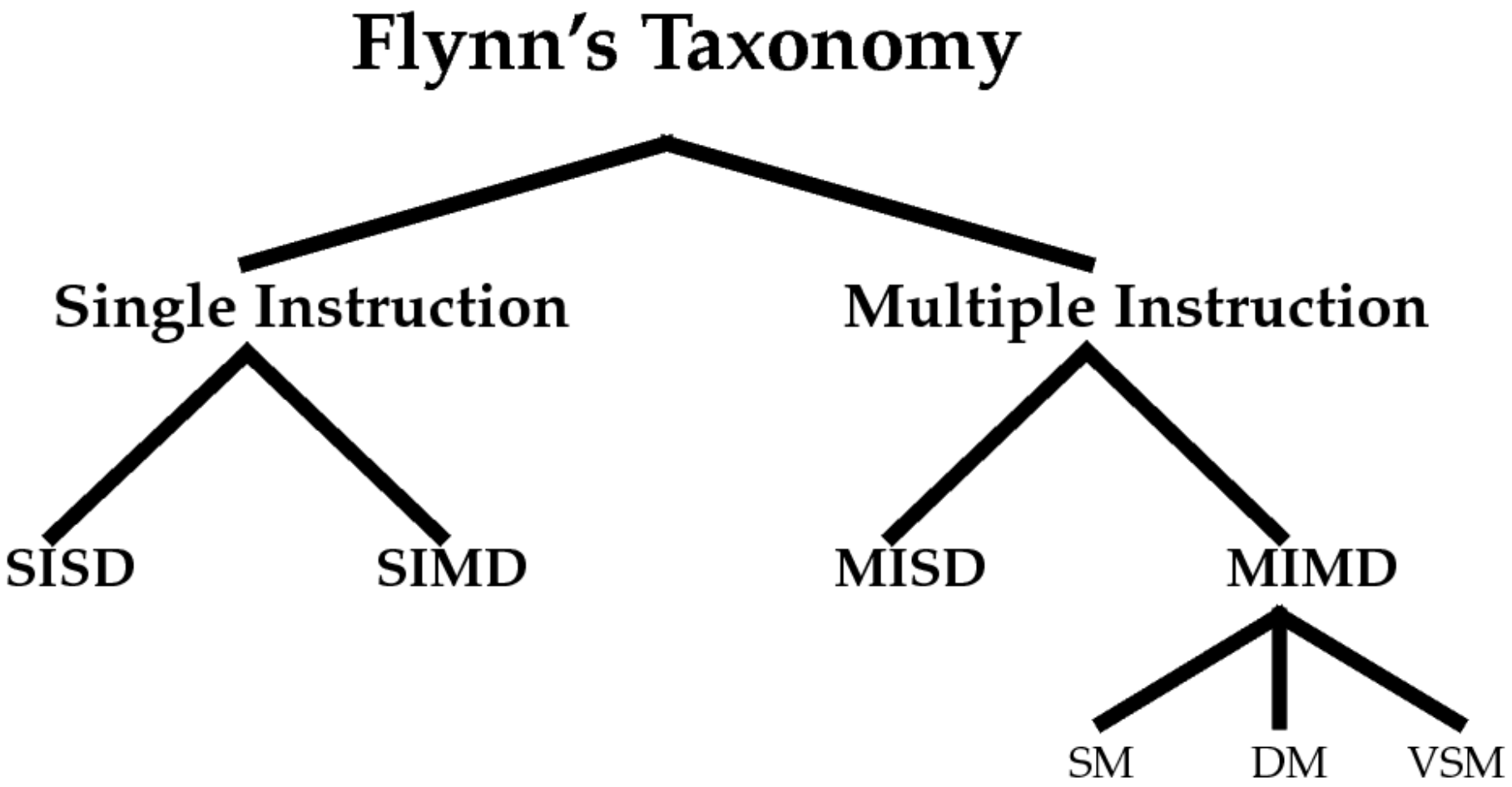
- produit de deux matrices 10000×10000 : $C = AB$
- calcul de 10000 fois $c_{i,j} = \sum a_{i,k} * b_{k,j}$
- nombre d'opérations flotantes = $2 * 10^4 * 10^4 * 10^4 = 2.10^{12}$

Temps d'exécution :

- ENIAC : 2.10^9s = 63ans
- PC de 1997 : 24 minutes
- ASCI Red (1997) : 2 s
- PC actuel : 0.2s $\rightarrow$ 5 par seconde
- TaihuLight : 0.02 ms $\rightarrow$ 50000 par seconde

Plan

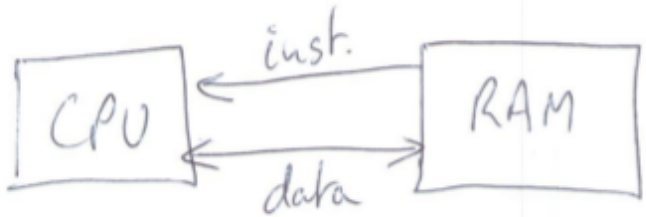
Architectures parallèles



Architectures parallèles

SISD

- une seule unité de calcul, séquentielle
- accédant à une seule donnée à la fois



Exemples : ENIAC, un PC de 1982 (Intel8088)

MISD

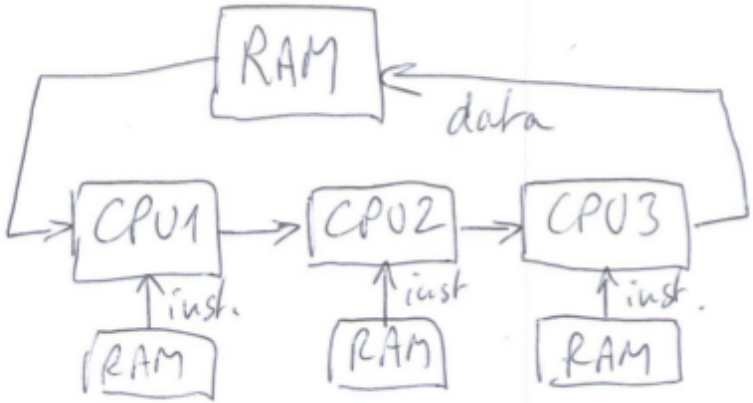
- plusieurs unités de calcul, qui exécutent des instructions différentes
- mais accédant à une seule donnée à la fois

Architectures parallèles

MISD

- plusieurs unités de calcul, qui exécutent des instructions différentes
- mais accédant à une seule donnée à la fois

→ fonctionnement de type **pipeline**



Exemples : pipeline d'instructions du coeur de processeur (RISC à 5 étages : 1988), pipeline graphique

SIMD

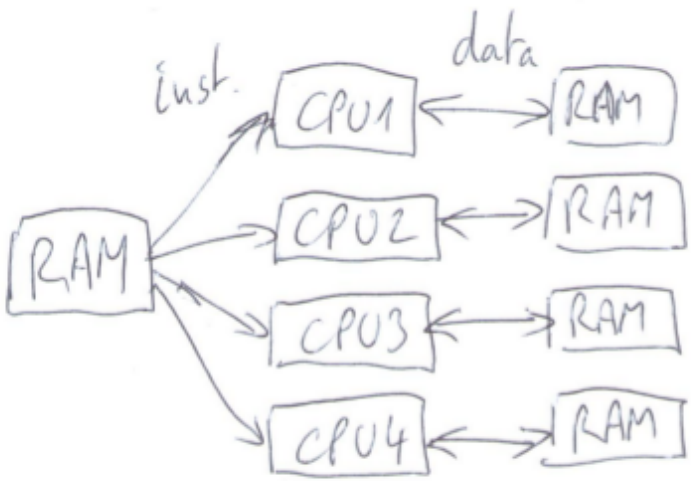
- plusieurs unités de calcul, exécutant toutes la même instruction (*SI*)
- sur des jeux de données différents

Architectures parallèles

SIMD

- plusieurs unités de calcul, exécutant toutes la même instruction (*SI*)
- sur des jeux de données différents

→ fonctionnement de type **vectoriel**

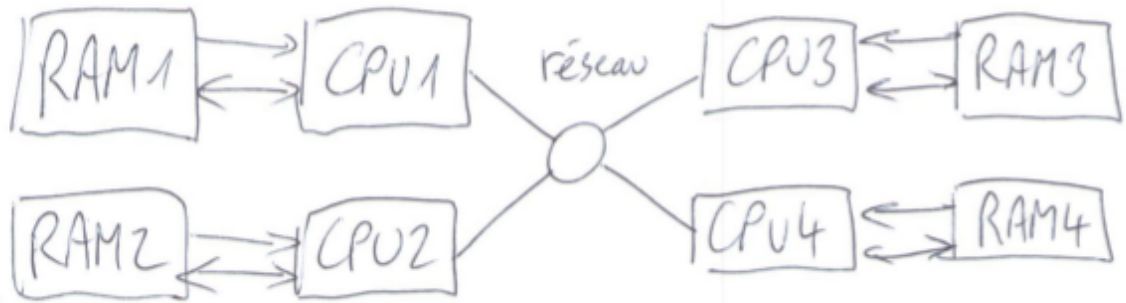


Exemples : MasPar (1990), instructions vectorielles (MMX, SSE, AVX), GPUs, ...

Architectures parallèles

MIMD - mémoire distribuée

- plusieurs machines SISD, reliées entre elles par un réseau
- fonctionnement asynchrone, chaque machine a sa mémoire

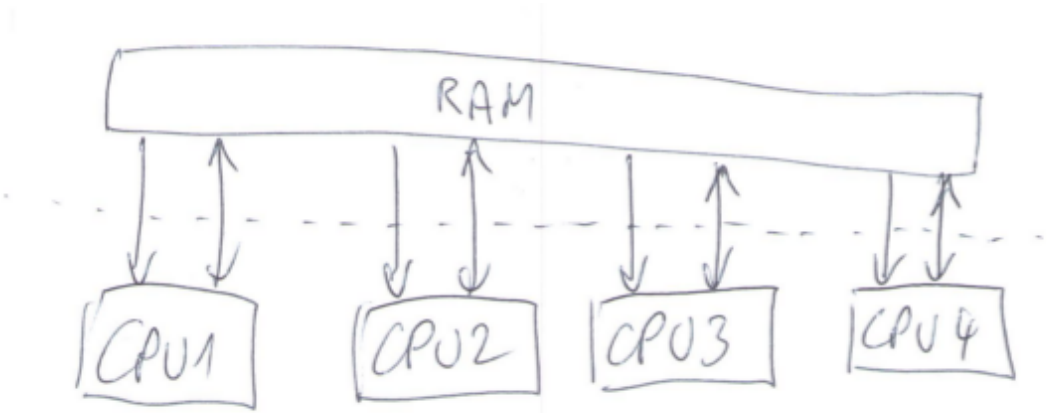


Exemples : IBM RS/6000 (1990), clusters de PCs

Architectures parallèles

MIMD - mémoire partagée

- plusieurs machines SISD,
- partageant une mémoire unique



Exemples : Cray X-MP (1983), SGI Challenge (1990), processeurs multi-coeurs

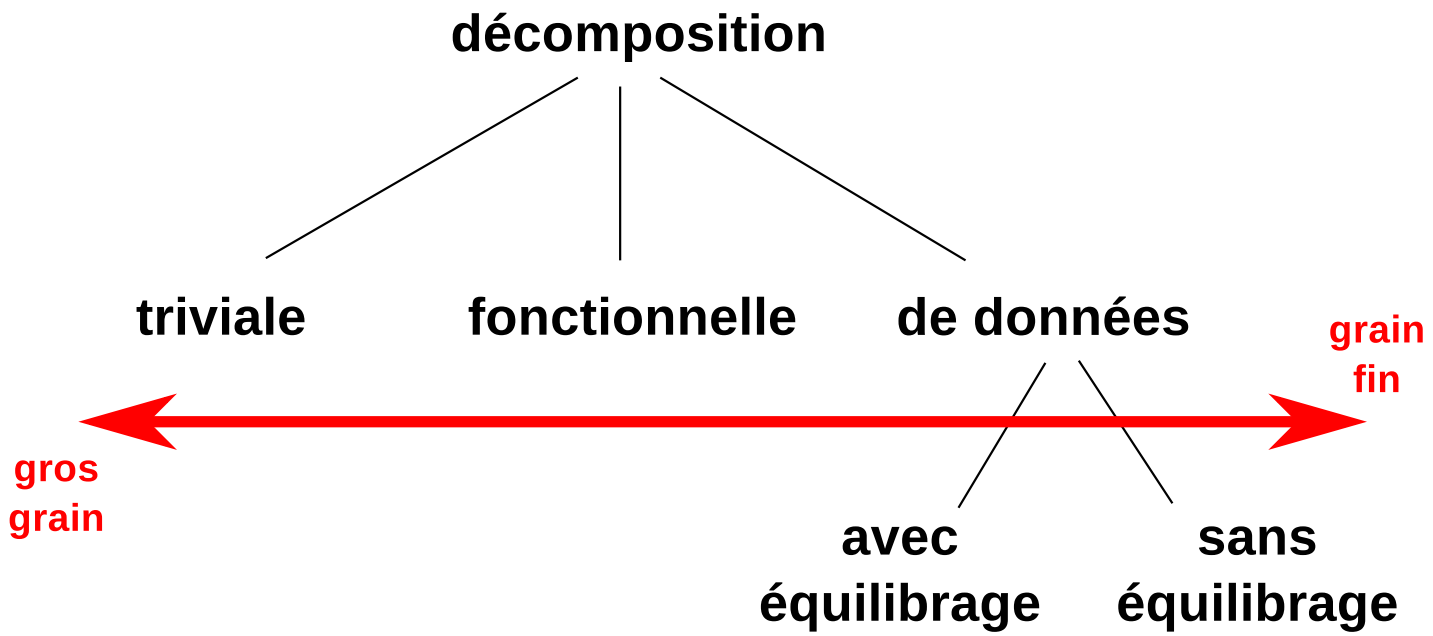
Architectures parallèles

MIMD-VSM (*Virtually Shared Memory*)
aussi appelée NUMA (*Non-Uniform Memory Access*)

- architecure physique MIMD à mémoire distribuée
- mais... l'ensemble de la machine possède une vision globale de la mémoire
- un processeur X peut accéder à la mémoire du processeur Y, sans l'interrompre, grâce à un réseau qui interconnecte les RAM !

Exemples : Cray T3D (1993), SGI Origin 2000 (1996), ...

Plan


décomposition
triviale
fonctionnelle
de données
grain fin
gros grain
avec équilibrage
sans équilibrage

Décomposition triviale

- un programme à lancer plusieurs fois
- sur des jeux de données (ex. : fichiers) différents...
- ... et *indépendants*

$\rightarrow$ il suffit de lancer le programme plusieurs fois

Décomposition fonctionnelle

- un programme est découpé en fonctions, qui réalisent une opération
  - sur des données en entrée
  - pour produire des données en sortie

→ graphe de dépendances

- les fonctions indépendantes peuvent être exécutées en parallèle :
  - les données en entrée sont des *jetons*
  - une fonction qui dispose de tous ses jetons peut être exécutée

Décomposition de données, avec équilibrage

- *ferme* de calcul
- un processus *maître* répartit les calculs à effectuer...
- ... par un ensemble de processus *esclaves*

→ le processus maître envoie des ordres aux processus esclaves, qui lui renvoient les résultats

Décomposition de données régulière

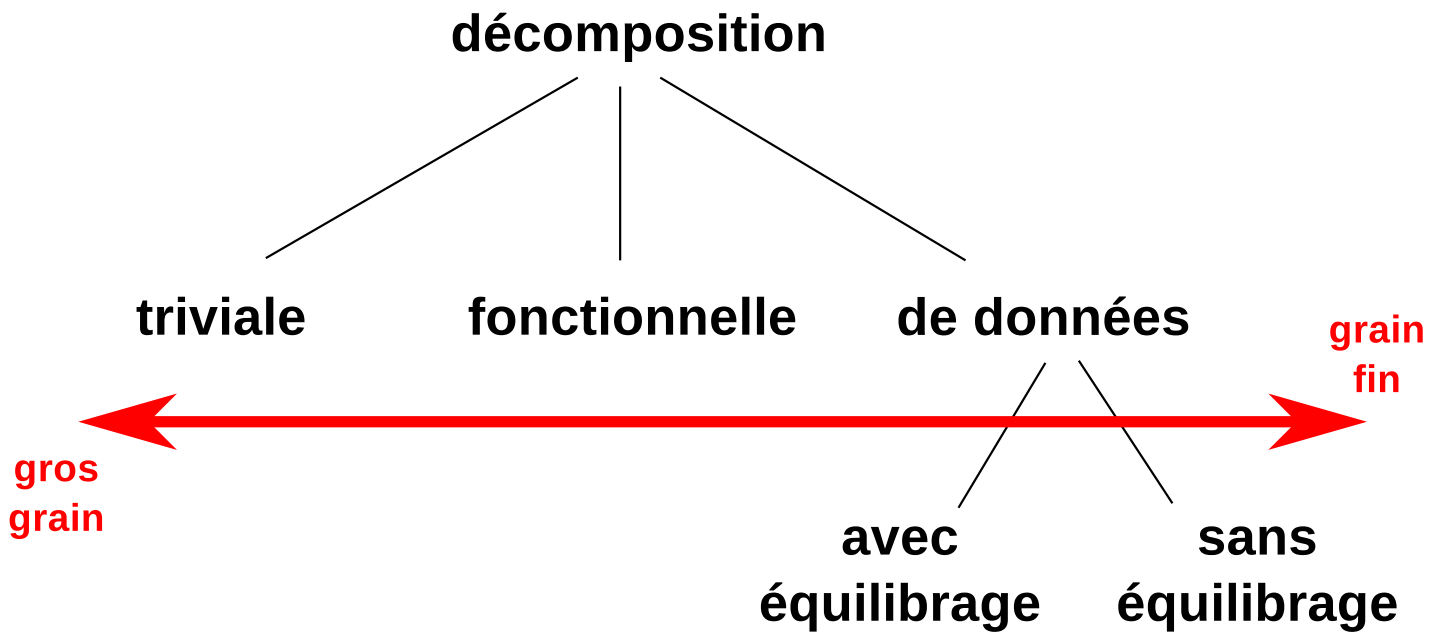
- les calculs et les données sont répartis de manière automatique et régulière entre plusieurs processus
- tous les processus participent au calcul d'un ensemble de données (exemple : une matrice)

Attention aux dépendances !

- certains calculs dépendent du résultat de calculs précédents, et certains réutilisent les variables que d'autres calculs utilisent
- il faut s'assurer qu'une instruction n'est pas exécutée avant que toutes les instructions dont il dépend ne soient terminées

Conditions de Bernstein

- deux instructions S_1 et S_2 sont indépendantes si (condition suffisante) :
  - $W(S_1) \cap R(S_2) = \emptyset$ $\rightarrow$ (*true dependence*)
  - $R(S_1) \cap W(S_2) = \emptyset$ $\rightarrow$ (*anti dependence*)
  - $W(S_1) \cap W(S_2) = \emptyset$ $\rightarrow$ (*output dependence*)
- sinon, il faut attendre la fin de S_1 avant le début de S_2


décomposition
triviale
fonctionnelle
de données
grain fin
gros grain
avec équilibrage
sans équilibrage

Plan

décomposition triviale : facile !

il suffit de lancer le programme plusieurs fois (éventuellement sur des ordinateurs différents)

décomposition triviale : facile !

il suffit de lancer le programme plusieurs fois (éventuellement sur des ordinateurs différents)

décomposition fonctionnelle et maître/esclave

gérer les communications entre des *processus* lancés sur des unités de calcul différents

décomposition triviale : facile !

il suffit de lancer le programme plusieurs fois (éventuellement sur des ordinateurs différents)

décomposition fonctionnelle et maître/esclave

gérer les communications entre des *processus* lancés sur des unités de calcul différents

décomposition régulière :

idem, ou gérer les synchronisations entre des *threads* (à mémoire partagée)

Gérer des communications entre processus

- tubes/files de messages/signaux (POSIX) ou fichiers sur un disque partagé entre les processus
- modèle de composants (des composants inter-dépendants sont connectés)
  - Corba, XPCOM (mozilla), DCOM (M$), SOAP (http/XML), ...
- passage de messages
  - langages de programmation concurrente : Ada, Erlang, Go (google), ...
  - langages à base de *streams* ou files d'attentes : OpenStream, *multiprocessing* de Python
  - bibliothèques de passage de messages explicite : sockets TCP/IP, PVM, MPI, ...

Synchroniser des threads

contrairement aux processus, les threads *partagent* leur vision de la mémoire (comme dans le modèle d'architecture MIMD-SM)

- threads et synchronisations (mutex, variables condition) fournis par le système d'exploitation (POSIX)
- certaines bibliothèques/extensions de langages :
  threading de Python, threads java, C11 *native threads*, Intel TBB, Cilk Plus, ...
- langages : OpenMP, HPF, UPC, OpenACC, ...
- langages hybrides (avec gestion de la mémoire) : CUDA, OpenCL

Soit T_i le temps d'exécution d'un programme parallèle sur i unités de calculs.

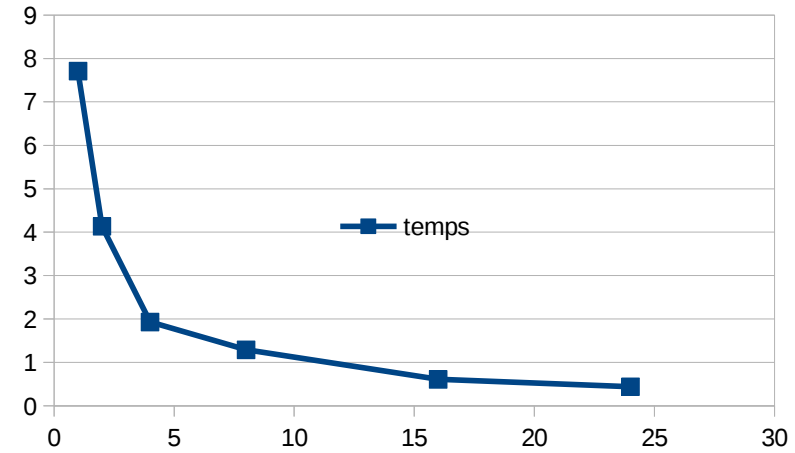
- accélération (*speedup*) :

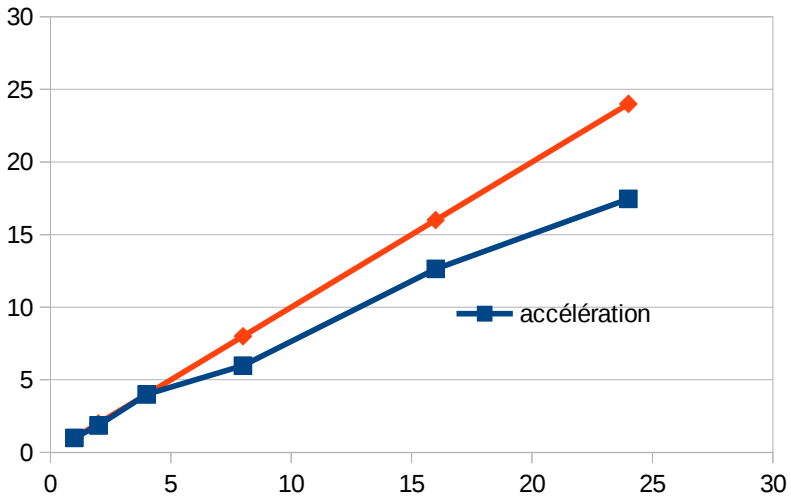
$$S_p = \frac{T_1}{T_p}$$

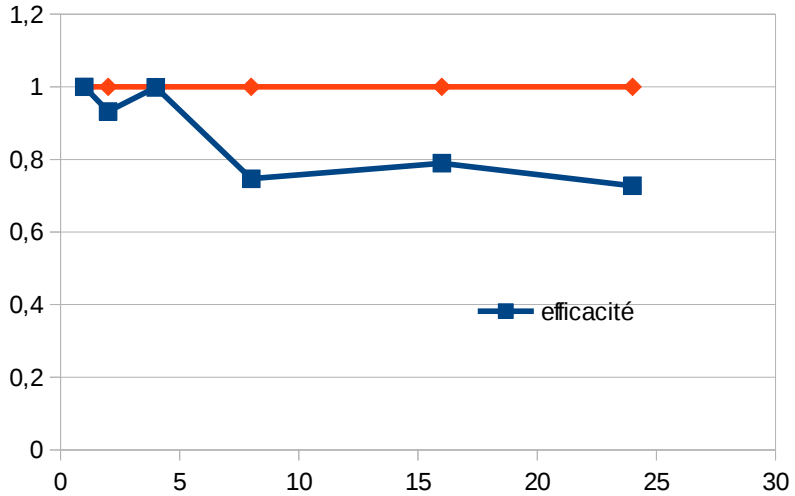
- efficacité :

$$E_p = \frac{S_p}{p} = \frac{T_1}{p.T_p}$$

#proc	temps	accélération	efficacité
1	7,708953	1	1
2	4,134969	1,86433151	0,932165755
4	1,930356	3,993539534	0,998384883
8	1,2905	5,973617203	0,74670215
16	0,610198	12,63352715	0,789595447
24	0,441649	17,4549314	0,727288809







Loi d'Amdahl

Soient :

- z la proportion du temps d'exécution de la partie de programme pouvant être parallélisée
- p le nombre d'unités de calcul (= accélération de la partie parallélisée)

alors :

$$S_p = \frac{1}{(1-z) + z/p}$$

Loi d'Amdahl

Soient :

- z la proportion du temps d'exécution de la partie de programme pouvant être parallélisée
- p le nombre d'unités de calcul (= accélération de la partie parallélisée)

alors :

$$S_p = \frac{1}{(1-z) + z/p}$$

Exemples

Soit un programme dont on peut paralléliser 90% de son temps d'exécution.

- Quelle est l'accélération théorique lorsque la partie parallélisée est 9 fois plus rapide ?
- Quelle est l'accélération maximale théorique ($\lim_{p\to\infty} S_p$) ?

Plan

OpenMP

- OpenMP est un standard défini par le *OpenMP Architecture Review Board* en 1998, définissant une interface de programmation (*API*) parallèle, pour systèmes multi-threads à mémoire partagée, disponible dans les langages C, C++ et Fortran, basé sur l'ajout de *directives* dans le code source.

- Implémenté dans les principaux compilateurs du marché (gcc, llvm, icc, visual studio, ...) en ajoutant une option de compilation habituellement

OpenMP : directives

```
// CONSTRUCTEURS :
  // Région parallèle
  #pragma omp parallel
  // Partage de travail (itérations d'une boucle)
  #pragma omp for
  // Partage de travail (blocs d'instructions)
  #pragma omp sections
  // Exécution par un seul thread
  #pragma omp single

// CONTRÔLE ET SYNCHRONISATIONS :
  #pragma omp critical
  #pragma omp atomic
  #pragma omp barrier
  #pragma omp ordered
```

OpenMP : exemple

```
#include <omp.h>
int main()
{
  int i;
  double A[N];

  #pragma omp parallel for
  for( i=0 ; i<N ; i++ )
  {
    A[i] = f();
  }

  afficher( A );
  return( 0 );
}
```

OpenMP : produit de matrices

```
int i,j,k;

// initialisation a, b, c...

#pragma omp parallel for private(j,k)
for (i=0;i<N;i++)
  for (k=0;k<N;k++)
    for (j=0;j<N;j++)
      c[i][j] += a[i][k] * b[k][j];

// ...
```

OpenMP : sections

```
#pragma omp parallel
{
  printf( "Every thread prints this\n" );
  #pragma omp sections
  {
  #pragma omp section
  {
    printf( "1st section. I am %d\n", omp_get_thread_num())
  }

  #pragma omp section
  {
    printf( "2nd section. I am %d\n", omp_get_thread_num())
  }
  } // end sections
} // end parallel region
```

OpenMP : ferme

```
int tid;
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();

  if( tid == 0 )
  {
    // master
    // ...
  }
  else
  {
    // slave(s)
    // ...
  }
} // end parallel region
```

Plan

MPI

MPI est :

- un standard défini par le *MPI forum* en 1994,
  contenant une bibliothèque de fonctions et un environnement d'exécution,
  utilisable avec les langages C, C++, Fortran (inclus dans la norme), et Python, java, ...
  permettant d'exécuter des programmes parallèles avec passage de messages dans un environnement distribué.
- disponible pour la plupart des systèmes d'exploitations (linux, MacOS, Windows, ...)
- implémentations : mpich, OpenMPI, ...

MPI : concepts

Une application MPI exécute un ensemble de **processus** indépendants, à mémoire distribuée (ayant des **variables locales**), et s'échangeant des **messages**.

La bibliothèque MPI permet de :

- définir des **communicateurs** :
  - ensemble de processus participant à une application distribuée,
  - qui s'échangent des messages contenant des données typées.
- réaliser des **communications point-à-point** :
  - envoi d'un message à un processus destination,
  - réception d'un message depuis un autre processus.
- réaliser des **communications collectives** :
  - tous les processus participent à une communication globale
  - ex. : *barrière*, *broadcast*, *distribution*, *réduction*, etc...

MPI : exemple

```
#include <mpi.h>
int main(int argc, char **argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  printf("I am nb. %d out of %d\n", rank, size);

  MPI_Finalize();
  return( 0 );
}
```

MPI : ferme

```
if( rank == 0 )
{
   // master
   // ...
}
else
{
   // slave(s)
   // ...
}
```

MPI : produit de matrices

```
int i,j,k;

// initialisation a, b, c...
// envoi a,b à tous les processus

for (i=rank*N/size; i<(rank+1)*N/size; i++)
  for (k=0;k<N;k++)
    for (j=0;j<N;j++)
      c[i][j] += a[i][k] * b[k][j];

// récupération de c sur le processus numéro 0
if( rank==0 )
{
  // ...
}
```

Plan

Conclusion

- le parallélisme est présent dans tous les équipements informatiques
- certaines de ses formes sont exploitées automatiquement par les compilateurs
  (pipeline processeur, vectorisation, ...)
- d'autres formes doivent être exploitées par le programmeur
  - mémoire partagée et threads : OpenMP
  - mémoire distribuée et processus : MPI
  - accélérateurs matériels (GPU, FPGA) : langages spécifiques (CUDA, OpenCL, OpenACC, ...)