Formation doctorale
Introduction au calcul intensif : de la programmation à l'utilisation d'un cluster de calcul

J. Bruchon - N. Moulin
École des Mines de Saint-Étienne
Pôle Modélisation et Calcul Numérique

30 novembre et 8 décembre 2016

Table des matières

1 Premiers programmes **6**
1.1 Hello World 6
1.2 Compilation et exécution 7
1.3 Échange de données 9

2 Marche aléatoire d'une particule **10**
2.1 Étape 1 : choix de la position de départ non occupée sur l'un des quatre bords de la grille 13
2.2 Étape 2 : marche aléatoire jusqu'à tomber sur une case occupée . . . 13
2.3 Étape 3 : envoi de la position trouvée, réception des positions, incrémentation du compteur n 13
2.4 Étape 4 : écriture au format VTK 14
2.5 Compilation, exécution et visualisation 14
2.6 Rapatrier des fichiers en local 16

3 Somme des n premiers entiers **16**
3.1 Partitionnement des données 17
3.2 Sommation locale 17
3.3 Méthode de communication 1. 18
3.4 Méthode de communication 2. 19
3.5 Méthode de communication 3. 19
3.6 Les fonctions MPI de réduction 20

4 Annexe : gestion de la mémoire **21**

5 Annexe : introduction au calcul intensif **28**

6 Annexe : présentation du cluster centaure **38**

Les commandes de base de LINUX

Les commandes de gestion des répertoires et des fichiers

pwd (affiche le chemin absolu du répertoire courant)
ls (list, affiche les répertoires et les fichiers du répertoire actif)
ls (affiche seulement les noms)
ls toto* (affiche les fichiers commençant par toto)
ls -l (affiche le format long : types + droits + Nbre de liens + ...)
cd (change directory)
cp chemin (vers le répertoire dont le chemin absolu est donné)
cd .. (répertoire parent)
cd ~ (répertoire de base)
cd - (répertoire précedent)
cd / (répertoire racine)
cp (copie)
cp rapport*.txt sauvegarde
cp * dossier (copie
mv (move, renomme et déplace un fichier)
mv source destination
mv * dossier (déplace tous les fichiers du répertoire actif vers le répertoire dossier)
mkdir (créer un répertoire)
mkdir répertoire
rmdir (effacer un répertoire)
rmdir dossier (supprime un répertoire vide)
rm (remove, éfface!!!)
rm -R (enlèvement récursif!!!)
rm fichier
rm -i fichier (interactivement, avec demande de confirmation)
rm -f fichier (avec force, sans demande de confirmation)
rm -r fichier (avec récursivité, avec les sous répertoires)
rm -rf dossier (supprime le répertoire et tou son contenu, sans confirmation)

Les commandes de recherche

grep (recherche les occurences de mots à l'intérieur de fichier)
grep motif fichier
grep -i motif fichier (sans tenir compte de la casse)
grep -c motif fichier (en comptant les occurences)
grep -v motif fichier (inverse la recherche, en excluant le "motif")
grep expression /répertoire/fichier
grep [aFm]in /répertoire/fichier
grep "\$" *.txt

Les commandes d'édition

more ("pager" qui affiche page par page sans retour en arrière, "h" affiche l'aide contextuelle)

more fichier
more fichier1 fichier2
more *.txt

cat (concatenate avec le code de fin de fichier eof=CTRL + D)
cat fichier-un fichier-deux > fichier-un-deux
cat -n fichier > fichier-numéroté (crée un fichier dont les lignes sont numérotés)
cat -nb fichier (affiche sur la sortie standard les lignes numéroté, sauf les lignes vides)

head (affiche les 10 premières lignes d'un fichier)
head -n22 fichier (affiche les 22 premières lignes)

vi (l'éditeur en mode texte universel)

emacs (l'éditeur GNU Emacs multi fonction pour l'édition, les mails, les news, la programmation, la gestion des fichiers,...)

xemacs (l'éditeur GNU Emacs sous X)

diff (différence entre deux fichiers, utiles pour chercher les modifications)

diff fishier1 fichier2

Les commandes d'impression et de conversion

lp (la commande d'impression sur les systèmes Unix Système V)

lpr (la commande d'impression sur les systèmes BSD et Linux)
lpr fichier
echo $PRINTER

lpc status (affiche l'état de la file d'attente)

lpq (affiche les travaux d'impression et leur numéro)

lprm (supprime un travail d'impression avec son numéro comme argument)

gv ("ghostview" permet de visualiser des fichiers POST SCRIPT)
gv fichier.ps

a2ps (convertit les fichiers ASCII en POST SCRIPT)
a2ps -4 fichier -P fichier-post-script

Les commandes de compilation et d'execution

f77 compile un programme en fortran 77

f77 program.f (la terminaison .f indique que le fichier program.f est écrit en f77

./a.out exécution

Les autres commandes

cal (calendar)
cal 2002

date (affiche la date, le mois, l'heure et l'année du jour. Les messages d'erreur et les e-mails sont toujours datés avec la date système)
date -s

wc ("word & count", affiche le nombre de lignes + mots + caractères)
who | wc -l (affiche uniquement le nombre de lignes)

Les trois TP présentés ici ont pour objectif d'introduire la notion de calcul parallèle dans un contexte scientifique, en répondant à ces questions : qu'est-ce qu'un programme parallèle ? Comment différents cœurs de calcul peuvent s'échanger des informations ? Comment se servir d'un cluster de calcul ? Nous avons choisi le langage C++, qui est un langage orienté objet. Cependant les notions de C++ que nous utilisons ici sont très restreintes. L'aspect parallèle de la programmation se fera à l'aide la librairie MPI (Message Passing Interface, ici déclinée dans sa version OpenMPI utilisant strictement les mêmes fonctions). Cette librairie n'est pas liée au C++ et peut être également utilisée avec d'autres langages comme le Fortran ou le C. Ses fonctions permettent de gérer la communication entre les cœurs de calcul, les cœurs pouvant être ou non sur le même processeur.

Nous allons travailler sur le cluster de calcul (240 cœurs) de l'École des Mines appelé *centaure*. Un point important de vocabulaire est le suivant. Le cluster est formé de plusieurs machines (ordinateurs) reliées entre elles par un réseau. Chaque machine est appelée **nœud** du cluster (ou nœud de calcul). Chaque nœud est multi-processeur, et chaque processeur (ou *socket*) est lui même multi-cœur (*multi-cores*). Il est alors possible d'exécuter sur chaque cœur une ou plusieurs tâches (appelées *thread*) (Figure 0.1). **Connectez** vous d'abord à un ordinateur de la salle J3-05 (choisir linux au démarrage, pour nous la distribution Linux Ubuntu).

Un fois connecté, ouvrez un terminal (cliquez avec la souris dans le coin supérieur gauche et tapez « terminal »), et tapez

```
ssh -X login@centaure.emse.fr
```

où `login` est votre nom d'utilisateur sur le cluster. Entrez votre mot de passe cluster (qui ne s'affiche pas lorsque vous le tapez, c'est normal) puis répondez « `yes` » à la question posée. Vous êtes connecté au cluster, et plus précisément à ce que l'on appelle le nœud maître ou *la frontale* du cluster qui est son point d'entrée.

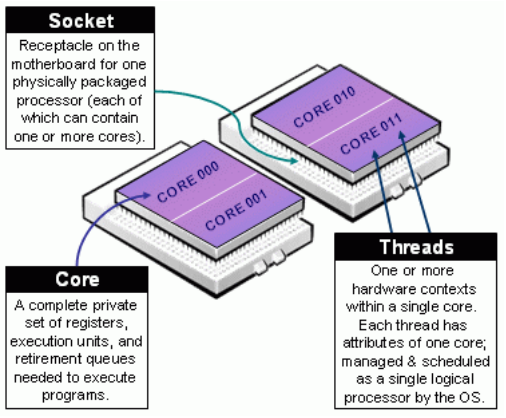


FIGURE 0.1 – Définition des *sockets*, *cores* et *threads*

Le système d'exploitation du cluster est linux. Les commandes de base sont : `ls` ou `ll` pour voir le contenu d'un répertoire, `cp` ou `cp -r` pour copier des fichiers ou

un répertoire, `mv` pour déplacer / renommer un fichier ou un répertoire, `pwd` pour voir le nom du répertoire où l'on est. Utilisez la commande `man` pour accéder à l'aide en ligne (exemple : `man cp`). Vous éditerez les fichiers avec l'éditeur de texte *gedit*. Par exemple :

```
gedit toto.cc &
```

Le « & » permet de lancer un programme en arrière plan, et ainsi de pouvoir continuer à utiliser la ligne de commande durant son exécution.

1 Premiers programmes

1.1 Hello World

Le premier programme que l'on considère doit être écrit dans le fichier `Hello_World.cc` placé dans le répertoire `Hello` de votre compte. Pour vous entraîner, complétez le fichier avec le programme écrit ci-dessous, en lisant également les commentaires indiqués ci-après. Pour cela, tapez d'abord

```
cd Hello
```

puis,

```
gedit Hello_World.cc &
```

Le programme à écrire :

```
#include <iomanip>
#include <mpi.h>
using namespace std;

int main(int argc, char ** argv)
{
        MPI_Init(&argc,&argv);
        int total, rank;

        MPI_Comm_size(MPI_COMM_WORLD,&total);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);

        if ( rank == 0 )
        {
                cout << "Programme s'exécutant sur " << total <<
                        " coeur(s) " << endl;
        }
        cout << "Process " << rank << " : Hello World" << endl;

        MPI_Finalize();

        return 0;
}
```

Remarques : les deux premières lignes indiquent au compilateur que le programme utilise des fonctions dont les en-têtes sont définies dans `iomanip.h` (input / output, comme la fonction `cout` qui permet d'afficher des chaînes de caractères ainsi que le contenu des variables) et dans `mpi.h` (fonctions de la librairie MPI assurant la communication entre les cœurs de calcul).

La troisième ligne indique que le programme utilise des fonctions « regroupées » sous la dénomination `std`. C'est le cas de `cout` et `endl`. L'appel à ces fonctions devrait se faire par `std::cout` et `std::endl`. La déclaration de l'utilisation du `namespace std` permet d'omettre `std::`.

Un programme C++ contient une et une seule fonction principale `main`, qui est la fonction « d'entrée » dans le programme. Cette fonction renvoie un entier (le type entier est `int` en C++, le type réel est `double`), d'où le `return 0` en fin de programme. Les deux arguments de cette fonction sont le nombre de paramètres passés en ligne de commande, et un tableau contenant les chaînes de caractères passées en ligne de commande. Il y a toujours au moins un paramètre passé, qui est le nom du fichier exécutable lancé. Le type caractère est `char`, et `char *` désigne le type pointeur vers une variable de type caractère.

L'expression `A == B` vaut `1` si l'expression `A` est égale à l'expression `B`, et zéro sinon. À ne pas confondre avec l'opérateur d'affectation `A = B`. Une expression est une suite d'opérateur et d'opérandes pouvant être évaluée comme vraie ou fausse.

L'opérateur & utilisé ici renvoie l'adresse mémoire de la variable sur laquelle il s'applique (ici `rank` et `total`). Le passage de l'adresse d'une variable en argument d'une fonction permet de modifier directement la valeur de cette variable dans cette fonction (voir aussi l'annexe à la section 4).

Concernant l'aspect parallèle, le `main` d'un programme utilisant des fonctions MPI doit commencer par `MPI_Init(...)` et finir par `MPI_Finalize()`. Deux autres fonctions MPI sont utilisées ici, `MPI_Comm_size(...)` et `MPI_Comm_rank(...)`. À vous de voir ce que font ces fonctions.

Notons enfin que `MPI_Comm_WORLD` est une variable prédéfinie de la librairie MPI, de type communicateur (`MPI_Comm`), spécifiant un ensemble de processeurs pouvant communiquer ensemble.

1.2 Compilation et exécution

Pour compiler votre programme, il faut charger l'environnement MPI, pour cela tapez `module load mpi/openmpi-x86_64` et ensuite tapez `make` dans le répertoire Hello. Cette commande fait appel au fichier `Makefile` contenant les instructions de compilation (vous pouvez éditer ce fichier). Notons que la compilation se fait en deux étapes. D'abord la phase de compilation elle-même, à l'issue de laquelle les fichiers en langage C++ sont convertis en fichiers en langage machine (fichiers .o). Puis, la phase d'édition de liens, permettant de lier les différentes parties du programme. Le compilateur utilisé ici est `g++`, avec des options définies par MPI, encapsulées dans `mpicxx`. À l'issue de cette compilation, le fichier exécutable `Hello` est créé.

L'exécution d'un programme en parallèle (ici le programme Hello) se fait par la

commande :

```
mpirun -np nbcoeurs -host centaure ./nom_fic
```

où `nbcoeurs` est le nombre de cœurs sur lequel le programme s'exécute, et `nom_fic` est le nom du fichier exécutable. `centaure` est le nom de la machine d'accès au cluster sur laquelle vous êtes connectée (on parle aussi de « nœud maître » ou de « frontale du cluster »). Ainsi, `-host centaure` signifie que votre calcul ne sera exécuté que sur les différents cœurs du nœud maître du cluster. Les autres nœuds du cluster sont appelés `compute-0-1`, `compute-0-2`, ... Pour que le calcul y ait accès il suffit de taper « `-host compute-0-1,compute-0-2,...` » . Cependant, lorsque l'on travaille sur un cluster de calcul, il est nécessaire d'utiliser un gestionnaire de queue de calculs. On soumet alors un « job » à ce gestionnaire de ressources, en lui précisant un certain nombre de paramètres (nombre de nœuds, nombre de cœurs par nœud, mémoire à réserver, temps maximal du calcul, etc.). Selon ces paramètres, le gestionnaire lancera le calcul lorsque les ressources demandées seront disponibles.

Le fichier déclarant ces ressources demandées s'appelle ici `Hello.job`. Vous devez l'éditer (`gedit Hello.job &`). Parmi les premières lignes du fichier, vous trouverez :

```
#SBATCH --job-name=Hello
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=05:00:00
```

Respectivement, le résultat de ces options est : le job s'appelle *Hello*. Quatre cœurs sur deux nœuds sont réservés (soit un total de 8 cœurs), et enfin le calcul ne durera pas plus de 5 heures.

Un peu plus loin, vous trouvez la commande

```
mpirun -np $SLURM_NTASKS -npernode $SLURM_NTASKS_PER_NODE -mca btl
   openib,self -host $COMPUTEHOSTLIST $SLURM_SUBMIT_DIR/./Hello
```

qui lancera l'application Hello sur 8 cœurs. L'option `-npernode` spécifie à la commande `mpirun` le nombre d'instances du programme exécutées sur chaque nœud de calcul, tandis que la variable `COMPUTEHOSTLIST` contient les noms des nœuds de calcul réservés par le gestionnaire et `SLURM_SUBMIT_DIR` le nom du répertoire de travail local. Les options `-mca btl openib,self` sont des options très spécifiques aux clusters de calcul et permettent de spécifier le type de couches réseaux utilisées pour les communications, ici un réseau de type *infiniband*.

Enfin, pour soumettre le calcul, tapez en ligne de commande

```
sbatch Hello.job
```

La commande `squeue` vous permet de voir l'ensemble des jobs actuels soumis ou s'exécutant sur le cluster. Si le votre n'apparaît pas, c'est qu'il est déjà fini ! Si c'est le cas, le fichier `slurm-xxx.out` doit avoir été créé dans votre répertoire de travail. Éditez ce fichier, examinez-le et cherchez l'affichage généré par votre programme.

1.3 Échange de données

Nous considérons à présent un deuxième programme, l'équivalent de celui présenté sur les transparents : lancé sur deux cœurs, il consiste en un échange de données entre le cœur 0 et le cœur 1. Le principe est le même que précédemment : nous vous laissons écrire le programme dans le ficher `Communication.cc` situé dans le répertoire `Communication`. Le programme est :

```
#include <iomanip>
#include <mpi.h>

#include <cstdlib>

using namespace std;

int main(int argc, char ** argv)
{
int nb_coeurs, rang;
int tag = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nb_coeurs);
MPI_Comm_rank(MPI_COMM_WORLD,&rang);

if ( nb_coeurs != 2 ) // Si le nombre de coeurs est différent de 2
{
    if ( rang == 0 )
    {
                cout << "Erreur : le programme s'exécute sur " << nb_coeurs <<
                " coeur(s) " << "au lieu de 2" << endl;
    }

    exit(1); // sortie du programme
}

srand(time(NULL)*rang);  // Initialisation du générateur de nombres aléatoires
int a = rand()%100; // % = reste de la division euclidienne
cout << "Coeur numéro " << rang << ", avant échange a = " << a << endl;

MPI_Send(&a,1,MPI_INT,1-rang,tag,MPI_COMM_WORLD);
// envoie d'un entier contenu à l'adresse mémoire &a au coeur 1-rang

MPI_Recv(&a,1,MPI_INT,1-rang,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
// Reception d'un entier mis à l'adresse &a et provenant du coeur 1-rang.

cout << "Coeur numéro " << rang << ", après échange a = " << a << endl;
```

```
MPI_Finalize();
}
```

Dans le programme précédent, la ligne

```
MPI_Send(&a,1,MPI_INT,1-rang,tag,MPI_COMM_WORLD);
```

signifie que le cœur sur lequel est exécutée cette instruction **envoie** 1 entier (`MPI_INT`) situé à l'adresse `&a`, et que cet envoi est effectué à destination du cœur `1-rang`. La variable `tag` est juste un "marqueur" et doit avoir la même valeur dans l'instruction de réception. Enfin, `MPI_COMM_WORLD` est une variable MPI de type `Communicator` regroupant l'ensemble des nœuds avec lesquels il est possible de dialoguer. On pourrait créer des communicateurs ne regroupant que certains nœuds. Notez que `a` est une variable entière, alors que `&a` contient l'adresse mémoire à laquelle se trouve la variable `a` (voir le poly sur la gestion de la mémoire en annexe).

À chaque instruction d'envoi de données doit correspondre une instruction de réception de ces données. Ici, ceci se fait par la commande

```
MPI_Recv(&a,1,MPI_INT,1-rang,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```

qui signifie que l'on reçoit un entier, placé à l'adresse `&a`, en provenance du nœud `1-rang`. Évaluez ce que vaut `1-rang` dans l'instruction d'envoi et dans celle de réception.

2 Marche aléatoire d'une particule

Ce premier exemple est représentatif de toute une classe de problèmes nécessitant d'exécuter un grand nombre de fois le même calcul mais avec différents jeux de données. C'est typiquement le cas d'algorithmes d'optimisation qui impliquent l'exécution de n simulations numériques différentes et indépendantes où seul va changer le paramètre dont on cherche la valeur minimisant une certaine fonction « coût » . Par exemple, ce paramètre peut être le module de Young d'un matériau, la fonction coût une fonction quantifiant l'endommagement du matériau, et la simulation numérique la simulation d'un certain type de sollicitation du matériau. Ou encore, le paramètre peut être la position à laquelle un certain fluide (résine polymère par exemple) est injecté dans un moule, la fonction coût le volume non rempli du moule (caractérisant le défaut de remplissage), et la simulation numérique celle de l'écoulement du fluide dans le moule. Enfin, dans des applications plus orientées génie industriel, la fonction coût peut être un temps d'attente, celui d'un patient aux urgences d'un hôpital par exemple.

Ici, nous ne sommes pas dans cette optique d'optimisation, puisque nous simulons la marche aléatoire de particules. On considère un ensemble de points formant une grille cartésienne comme indiqué sur la figure 2.1. Il y a `nx` points selon x et `ny` points selon y. La règle du jeu est que la particule ne peut se déplacer que de point en point. Lorsqu'elle est sur le point (i, j), elle a neuf possibilités : soit rester en (i, j), soit se déplacer sur l'un des huit voisins de ce point. Le mouvement est tiré aléatoirement, avec dans ce TP (mais ça peut être modifié) une équiprobabilité

d'effectuer chacune de ces possibilités. Ceci correspond à un transport par diffusion, tandis qu'une probabilité différente d'effectuer un mouvement plutôt qu'un autre introduit une vitesse de dérive, modélisant ainsi un phénomène de convection. Enfin, initialement, une structure sous forme de « germe » est introduite au centre de la grille : lorsque la particule rencontre ce germe, elle se fige à sa dernière position et s'agglomère à la structure qui ainsi s'étend. La simulation se déroule alors en deux étapes : la position initiale d'une particule est tirée au hasard sur l'un des quatre bords de la grille, puis le cheminement de la particule est calculé jusqu'à son figeage. Ce processus est recommencé autant de fois que l'on a de particules.

La parallélisation de ce processus consiste à effectuer simultanément et indépendamment la marche de n particules sur n cœurs de calcul. Chaque cœur possède la même grille cartésienne. Une fois que chacune de ces n particules figées, les cœurs s'envoient mutuellement les positions de figeage afin de mettre à jour l'état de leur grille. Si deux particules sont figées au même endroit, la trajectoire d'une particule aura été calculée pour rien, ce qui peut nuire à l'efficacité de la parallélisation.

Allez dans le répertoire `marche_alea`, et **éditez le fichier** `marche_alea.cc`. Les valeurs de `nx`, `ny`, ainsi que le nombre de particules `nb_particules` et la fréquence à laquelle sont écrits les fichiers de sortie sont lus dans un fichier (ici `data.dat`) passé en ligne de commande. La première étape du programme est de lire ce fichier, d'initialiser les variables correspondantes, puis d'écrire ces informations à l'écran.

Ensuite la grille cartésienne est allouée dynamiquement. Elle est représentée par la variable `grille` de type `bool*`, c'est-à-dire un pointeur de booléen (voir aussi la section 4 pour la notion de pointeur) :

```
bool * grille = new bool [nx*ny];
```

Ceci signifie que `grille` contient l'adresse mémoire de la première case d'une série de `nx*ny` cases contiguës en mémoire contenant chacune un booléen (codé sur 8 bits (soit 1 octet), bien que 1 bit soit suffisant pour représenter les deux valeurs 0 ou 1 prises par un booléen). L'accès à la case `i` se fait par `grille[i]`, avec `i` compris entre 0 et `nx*ny-1`. `grille[i]` vaut 0 si la case `i` est libre, et 1 si elle

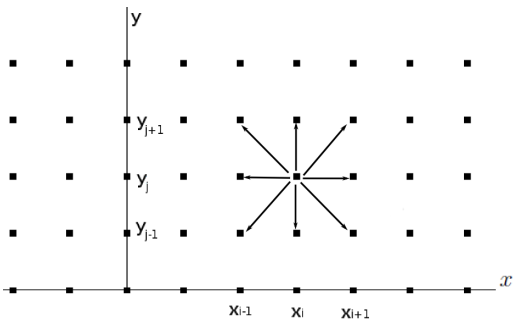


FIGURE 2.1 – Déplacement aléatoire d'une particule positionnée en (i, j) sur les cases voisines : 9 possibilités.

est occupée. Nous avons représenté une grille 2D par un tableau 1D, car cela nous laisse le choix de la disposition des éléments de la grille en mémoire (ligne par ligne, colonne par colonne, etc). Nous ferons plus tard le même choix pour représenter les matrices. Ici, la grille se lit de gauche à droite et de bas en haut, si bien que le point (i, j) correspond à la case $j \times nx + i$, et son état (occupé ou libre, 1 ou 0) est donné par `grille[j*nx+i]`. Enfin, la quantité de mémoire réservée pour le tableau `grille` ayant été allouée dynamiquement (*i.e.* durant l'exécution du programme et non fixée à la compilation), il est nécessaire de libérer cette mémoire une fois que `grille` n'est plus utilisée. Ceci est fait par `delete [] grille;`. Sans cette désallocation, la place mémoire est perdue et ne peut plus être réutilisée, conduisant ainsi à terme à la saturation de la mémoire vive de l'ordinateur (dans ce cas, on parle des fuites mémoire d'un programme).

Repérez dans le programme, sur quel(s) cœur(s) de calcul est alloué le tableau `grille`. Est-ce sur tous les cœurs ou uniquement certains ? **Repérez** de même sur quel(s) cœur(s) sont réalisées les opérations de lecture du fichier de données et d'écriture (affichage à l'écran et création des fichiers de sortie au format VTK en fin de programme, dont nous reparlerons).

La position courante (i, j) de la particule est stockée dans le tableau `position` (i est dans `position[0]`, j dans `position[1]`, tandis que `position_old` stocke la position immédiatement antérieure. La fonction `void srand(unsigned int seed)` initialise le générateur de nombres pseudo-aléatoires à partir d'un germe passé en argument (ici `void` est le type de la fonction et signifie qu'elle ne renvoie rien). Ce germe est habituellement pris comme la valeur renvoyée par la fonction `time(0)` (ou, de manière équivalente, `time(NULL)`), qui est le nombre de secondes écoulées depuis le 1er janvier 1970, minuit heure UTC. Si l'on exécute `srand(time(NULL))` simultanément sur tous les cœrs de calcul, alors les générateurs de nombres pseudo-aléatoires sont initialisés avec le même germe et vont induire la même suite de nombres. **Sachant cela, expliquez** l'argument passé à `srand`. Enfin, la fonction `rand()` renvoie le terme suivant de la suite pseudo-aléatoire initialisée, à savoir un entier compris entre 0 et `RAND_MAX` (une constante dépendant des distributions, mais garantie supérieure à 32767). **Expliquez alors** quelle est la valeur renvoyée par `rand() % n` où `n` est un entier, et `a % b` est le reste de la division entière de `a` par `b`.

Nous rencontrons ici une nouvelle fonction de la librairie MPI,

```
double Wtime()
```

qui renvoie la même valeur que `time(NULL)`, mais évaluée sur chaque cœur de calcul (les cœurs sont synchronisés, mais la valeur dépend du moment de l'appel). La différence entre les valeurs évaluées en fin et en début de calcul donnera donc la durée du calcul sur chaque cœur. **Quelle peut être une mesure pertinente de la durée d'un calcul parallèle ?**

À présent, vous devez compléter les trous mis dans le programme (les lignes à compléter sont également commentées grâce aux caractères `//`), en vous appuyant sur l'aide donnée ci-dessous.

2.1 Étape 1 : choix de la position de départ non occupée sur l'un des quatre bords de la grille

Cette étape a pour but de choisir la position initiale de la particule. Cette position se situe sur un des quatre bords de la grille de calcul. On effectue donc un premier tirage aléatoire pour savoir sur quel bord on place la particule. Ceci fixe donc l'une des coordonnées i ou j de la particule. Puis on effectue un second tirage pour déterminer la seconde coordonnée. Enfin, on ne veut pas que le point (i, j) choisi soit déjà occupé par une particule figée. Ce processus est donc placé à l'intérieur d'une boucle qui se termine lorsque la case (i, j) du tableau `grille` contient la valeur 0.

Au niveau C++, on introduit ici **l'opérateur NON logique**, noté « `!` », qui inverse l'état d'une variable booléenne. Ainsi, si `a` vaut 0, `!a` vaut 1 et inversement. **Comprenez** pourquoi il est équivalent d'écrire `while( ok == 0)` et `while( !ok )` sachant que la boucle `while ( expression ) { ... }` est effectuée tant que `expression` est vraie, *i.e.* non nulle (l'opérateur de comparaison `==` renvoie 1 si l'égalité est vérifiée, 0 sinon).

2.2 Étape 2 : marche aléatoire jusqu'à tomber sur une case occupée

Tant que la particule se situe sur un point inoccupé, on réalise deux tirages aléatoires pour déterminer les composantes horizontales et verticales de son mouvement. Deux opérateurs spécifiques au C++ sont introduits, les **opérateurs d'incrémentation** `++` et `--` qui respectivement ajoutent ou soustraient 1 à la variable sur laquelle ils s'appliquent. Ainsi `a++` est équivalent à `a = a + 1` ou encore `a += 1` (on a aussi `a -= 1`).

Enfin, remarquez les conditions de périodicité de la grille : une particule sortant par un bord entre par le bord opposé.

2.3 Étape 3 : envoi de la position trouvée, réception des positions, incrémentation du compteur n

Cette étape est spécifique au calcul parallèle, puisque c'est l'étape de communication entre les cœurs de calcul. Premièrement, chaque cœur met à 1 la valeur de la grille correspondant à la position de figeage de la particule et incrémente d'une unité le nombre de particules utilisées. Deuxièmement, chaque cœur envoie à tous les autres cœurs cette position. Troisièmement, chaque cœur reçoit ces données, met à 1 les valeurs correspondantes de sa grille si elle n'avait pas déjà cette valeur, et incrémente en fonction le nombre de particules utilisées.

Cette communication est effectuée à l'aide des deux fonctions MPI suivantes.

```
MPI_Send(tab,n,type,i,1,MPI_COMM_WORLD)
```

Cette fonction permet d'envoyer les `n` données pointées par la variable `tab` (qui est donc une adresse), qui sont de type `type` (ici `MPI_INT`), au nœud `i`, avec le « tag » 1, et ce nœud `i` se trouve dans le groupe `MPI_COMM_WORLD`.

```
MPI_Recv(tab_aux,n,type,i,1,MPI_COMM_WORLD,&status)
```

Cette fonction permet de recevoir dans les cases mémoire pointées par `tab_aux`, `n` valeurs de type `type`, provenant du nœud i avec le « tag » 1. On ne se préoccupe pas ici de la variable `status` (qui doit qu'en même être déclarée au préalable par « `MPI_Status status;` »). Le tag peut servir à distinguer les envois d'un même cœur. Il est important de noter qu'un message envoyé avec un certain tag ne pourra être reçu qu'avec le même tag.

2.4 Étape 4 : écriture au format VTK

Un moyen simple de visualiser le résultat, *i.e.* l'état final de la grille, est de laisser cette étape à un logiciel de visualisation graphique. Pour ce faire, nous considérons les valeurs 0 ou 1 de `grille` comme les valeurs d'un champ défini en chaque nœud d'une grille cartésienne. Nous déclarons cette grille et écrivons ces valeurs dans un fichier au format VTK (Visualization ToolKit) qui est un format (et même une librairie) standard en visualisation graphique. Nous créons un champ « `Concentration` » qui prend les valeurs 0 / 1 contenues dans `grille`.

Pour finir ce programme, vous devez compléter le calcul de l'affichage de la durée du calcul.

2.5 Compilation, exécution et visualisation

Compilez en tapant `make` en ligne de commande, puis corrigez les erreurs qu'il ne manquera pas d'y avoir. Enfin, lorsque tout est opérationnel, un fichier exécutable `marche` est créé. Vérifiez les données contenues dans `data.dat`. En particulier, une grille 100×100 est suffisante au départ pour tester le programme, avec également 100 particules. Imposez une fréquence d'écriture des fichiers VTK qui permet d'obtenir une dizaine de fichiers, c'est suffisant et l'étape d'écriture prend du temps. Éditez le fichier `marche.job` et vérifiez les ressources, par exemple `#SBATCH --nodes=4`, `#SBATCH --ntasks-per-node=8`. Ici, 4 est le nombre de nœuds de calcul (= machines physiques) demandés, et 8 le nombre de cœurs de calcul demandés sur chaque nœuds. On réserve donc un total de 32 cœurs. La ligne qui exécute le programme est donc ici

```
mpirun -np $SLURM_NTASKS -npernode $SLURM_NTASKS_PER_NODE -mca btl
   openib,self -host $COMPUTEHOSTLIST $SLURM_SUBMIT_DIR/./marche
      ./data.dat
```

Pour commencer, lancez votre programme uniquement sur 1 nœud et 2 cœurs. On aura alors :

`#SBATCH --nodes=1` et `#SBATCH --ntasks-per-node=2`

Enfin, lancez le job via `sbatch marche.job`. La commande `squeue` permet de voir l'état du cluster, la file d'attente et les jobs en cours d'exécution. Une fois votre calcul démarré, vous pouvez suivre son déroulement en faisant `tail -f slurm-xxx.out`, où `xxx` correspond au numéro de votre job (`Ctrl+c` pour sortir).

Le calcul terminé, un ensemble de fichiers *vtk* se trouvent dans votre répertoire (avec `inc_` comme nom générique suivi du nombre de particules figées au moment

de l'écriture). Nous utiliserons le logiciel libre *Visit* pour la visualisation graphique. Pour ouvrir les fichiers, tapez `visit -o` suivi du nom d'un des fichiers (au préalable il faut charger l'environnement de *Visit* en tapant `module load visit/2.10.2`). Dans *Visit*, visualisez le champ `Concentration` en cliquant sur `Ajouter` → `Pseudo Couleur` → `Concentration` puis sur `Tracer`. L'affichage peut prendre quelques secondes. Vous pouvez superposer la grille cartésienne en cliquant sur `Ajouter` → `Maillage` → `mesh` puis sur `Tracer`. Les icônes `Supprimer` et `Afficher/Cacher` permettent d'effectuer les opérations mentionnées sur l'objet sélectionné. Il est possible de visualiser les différents fichiers grâce aux boutons de la barre de contrôle `Time`. Enfin, la molette de la souris permet de faire des zooms dans la fenêtre graphique.

Remarque : Si vous constatez des problèmes ou latences d'affichage, il est possible de lancer *Visit* en local sur la machine de TP et de « dire » à *Visit* d'aller chercher les fichiers .vtk sur le cluster. Pour cela, ouvrir un nouveau terminal sur la machine de TP et lancer *Visit* (commande : `visit &`). Ensuite dans *Visit*, vous cliquez sur `Options` → `Hôte distants`. Vous cliquez sur `Nouvel hôte`, vous saisissez comme `Nom: centaure`, `Nom de l'hôte distant: centaure.emse.fr`, `Login: username` et cocher la case `utiliser un tunnel ssh pour les données`. Ensuite, vous n'avez plus qu'à faire `Fichier` → `Ouvrir` et choisir *centaure* comme nom d'hôte dans la liste. *Visit* devrait vous demander votre mot de passe.

L'efficacité de la stratégie mise en place pour le calcul parallèle, ou encore la scalabilité du calcul, peut être mesurée par l'accélération (ou speed-up) $S(p)$, définie pour p cœurs par

$$S(p) = \frac{t(1)}{t(p)} \tag{2.1}$$

où $t(n)$ est le temps d'exécution du calcul sur n cœurs. La figure 2.2 montre les

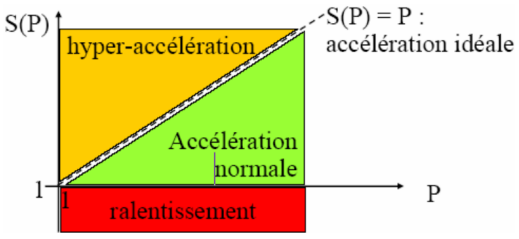


FIGURE 2.2 – Accélération d'un calcul parallèle.

différentes valeurs que peut prendre $S(p)$. La droite $S(p) = p$ représente le comportement parfait. $S(p) > p$ correspond à une hyper-accélération qui n'est pas magique et doit donc être expliquée : bugs (souvent), effets de cache, algorithme parallèle plus efficace. Enfin, la plupart du temps $1 < S(p) < p$, et même $S(p)$ peut diminuer au-delà d'une certaine valeur de p, car les temps de communication deviennent de plus en plus élevés. On définit également l'efficacité du calcul $E(p)$ comme

$$E(p) = \frac{S(p)}{p} \tag{2.2}$$

Lancez plusieurs calculs, par exemple avec une grille 500×500 et 1000 particules, en utilisant, disons 1, 2, 4 et 8 cœurs afin d'évaluer les performances du

programme. **Important** : lorsque vous lancez un nouveau calcul, n'oubliez pas de détruire préalablement les fichiers *vtk* se trouvant dans votre répertoire, en faisant `rm *.vtk`. Faites très attention avec la commande `rm *...` Un espace en trop, et tout est effacé !

2.6 Rapatrier des fichiers en local

Pour rapatrier depuis le cluster sur l'ordinateur local des fichiers ou des répertoires, on utilise la commande `scp`. Plus précisément, ouvrez un nouveau terminal depuis l'ordinateur local, puis, dans ce terminal, placez-vous dans le répertoire d'accueil souhaité (`cd nom_rep` pour aller dans le répertoire `nom_rep`, `mkdir nom_rep` pour créer le répertoire `nom_rep`). Tapez (à adapter selon votre groupe)

```
scp -r login@centaure.emse.fr:marche_alea/ .
```

pour recopier le répertoire `marche_alea` dans l'emplacement actuel (identifié grâce au point `.`). Le `:` indique que ce qui précède ce signe n'est pas un nom d'un fichier mais une adresse.

Pour copier par exemple uniquement les fichier d'extension `vtk` contenus dans le sous-répertoire `toto` de `marche_alea`, tapez

```
scp login@centaure.emse.fr:marche_alea/toto/*.vtk .
```

Enfin, vous pouvez accéder sur la machine locale à une interface graphique du système de gestion des fichiers en tapant `nautilus &`. Ceci peut être utile pour copier les fichier rapatriés sur une clef USB.

3 Somme des n premiers entiers

Nous allons à présent écrire un programme effectuant la somme des n premiers entiers en utilisant plusieurs méthodes de communication entre les cœurs de calcul. En réalité, vu la très faible taille des informations échangées ici, nous ne verrons pas de grandes différences entre ces méthodes. Ce qui est important, c'est que nous nous familiarisons avec le processus d'envoi et de réception d'information. De plus, la stratégie de calcul parallèle employée se déroule en trois étapes, qui sont les étapes que l'on retrouvera toujours par la suite :

1. Distribution des données (ou partition du domaine de calcul) : quelles sont les données que le cœur de rang i doit traiter ?
2. Traitement simultané des données par l'ensemble des cœurs du calcul, ici faire des sommes partielles.
3. Communication entre les cœurs afin de déterminer le résultat final, ici afin de sommer les sommes partielles.

Vous trouverez dans le répertoire `Somme`, les fichiers `Principal.cc`, `Somme.cc` et `Somme.h` que vous pouvez éditer en tapant

```
gedit Principal.cc Somme.* &
```

Nous utiliserons une classe `Somme`. Une classe est un ensemble de données, comme une structure en C, plus un ensemble de méthodes (fonctions) opérant sur ces données. Une fonction prend éventuellement des arguments en entrée et renvoie un résultat. Elle possède donc un type (`int`, `double`, ... ou `void` si il n'y a aucun renvoi explicitement défini par le programmeur, et sauf exceptions). Une classe est définie dans un fichier `.h` et est implémentée dans un fichier `.cc`. Éditez le fichier `Somme.h` et comprenez comment la classe `Somme` est définie.

Un objet est une instance particulière d'une classe. Dans `Principal.cc`, vous trouverez l'objet `UneSomme`, de type `Somme`. Les lignes placées entre `/*` et `*/` sont en commentaires. Vous les décommenterez au fur et à mesure de votre progression.
Remarquez que l'accès à une méthode de la classe se fait par : `UneSomme.NomMethode(...)`. Ne sont accessibles que les méthodes et données déclarées comme `public` dans le fichier en-tête de la classe (le `.h`). Ce qui est `protected` n'est accessible que par les autres fonctions de la classe. On doit alors définir des « fonctions d'accès » (souvent appelées `void set...(...)` pour l'affectation d'une valeur et `UnType get...()` pour le renvoi d'une valeur) qui permettent d'accéder de façon contrôlée à ces paramètres. C'est le principe d'encapsulation (notion propre à la programmation orientée objet), qui, lorsqu'il est appliqué, interdit la modification « brutale » de n'importe quelle variable, de façon à ne pas nuire à l'intégrité du programme.
Pour construire un objet (l'initialiser), il faut un constructeur, qui est une méthode sans type de la classe, possédant le même nom que celle-ci. Ici, le constructeur de `Somme` est `Somme(int taille)`. Regardez ce que cette fonction fait. Elle est appelée dans `Principal.cc` lorsque l'on fait `Somme UneSomme(taille)` (on crée la variable `UneSomme` de type `Somme`, qui est un objet, en appelant la fonction (le constructeur) `Somme(...)` avec la variable `taille` comme paramètre).
Vous retrouvez ensuite les différentes étapes décrites ci-dessus, appelées dans `Principal.cc` et implémentées dans `Somme.cc`. **Vous devrez compléter les trous mis dans le programme (« ... »)**.

3.1 Partitionnement des données

Le partitionnement des données est effectué dans la méthode `Partitionnement` : à l'issue de cette étape, chaque cœur de calcul connaît les données qu'il a à traiter. Ici, il n'y a pas de trou à compléter. Comprenez sur un exemple à quoi correspondent les variables `debut_local` et `fin_locale`.

3.2 Sommation locale

L'adjectif « local » se réfère au fait que l'opération de sommation est effectuée sur chaque cœur en fonction des données disponibles sur ce cœur. Complétez les « ... » de la méthode `Sommation`. L'instruction `for (i=n, i<m; i++)` fait une boucle de `i = n` à `i = m-1`, en incrémentant la variable entière `i` de 1 à la fin de chaque incrément (`i++` équivaut à `i = i+1`, ou encore à `i += 1`).

3.3 Méthode de communication 1.

Une fois les sommes partielles calculées sur chaque cœur, la somme des n premiers entiers est obtenue en sommant ces sommes partielles. Les cœurs de calcul doivent donc communiquer entre eux afin d'effectuer cette sommation globale, puis d'en connaître chacun le résultat. Trois manières de procéder sont envisagées ici (par ordre croissant de performance). Dans la première méthode de communication envisagée, décrite sur la figure 3.1, chaque cœur envoie à tous les autres sa somme

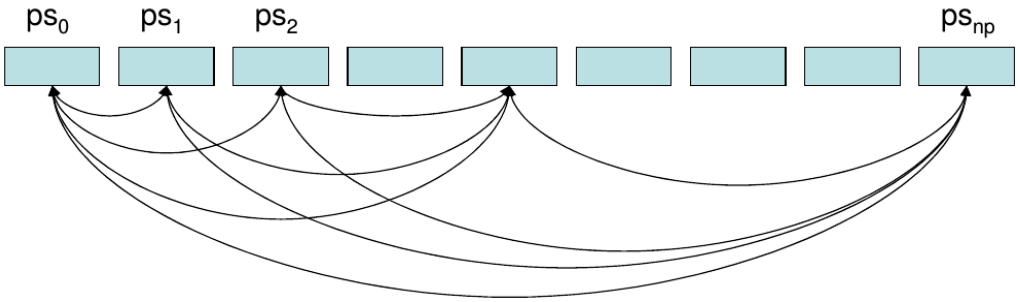


FIGURE 3.1 – Méthode de communication « la pire ».

partielle puis effectue la sommation totale. Pour un calcul sur p cœurs, cette approche nécessite $p(p-1)$ envois de messages, $p(p-1)$ additions, et a une profondeur (nombre maximal de messages que peut recevoir un cœur) de $p-1$. Elle est implémentée dans la méthode `Communication_pire()`. On rappelle que la somme partielle est stockée dans la variable entière `sum_p`, et que les fonctions MPI `MPI_Send(...)` et `MPI_Recv(...)` dont la nomenclature est rappelée ci-dessous, sont utilisées pour son envoi et sa réception. **Complétez les « ... ».** Notez que l'expression `( i != j )` vaut 1 si `i` est différent de `j` et 0 sinon.

```
MPI_Send(&sum_p,1,MPI_INT,i,1,MPI_COMM_WORLD)
```

Cette fonction permet d'envoyer `1` valeur entière contenue dans la variable `sum_p`, au nœud `i`, avec le « tag » `1`, et ce nœud `i` se trouve dans le groupe `MPI_COMM_WORLD`. Notons que le `&` devant la variable `sum_p` signifie que l'on passe en argument l'adresse de `sum_p` et non sa valeur (ce qui revient, comme dans le TP précédent, à passer un tableau, ici ne contenant qu'un élément).

```
MPI_Recv(&sum_aux,1,MPI_INT,i,1,MPI_COMM_WORLD,&status)
```

Cette fonction permet de recevoir dans la variable `sum_aux` (passage par adresse), `1` valeur entière, envoyée par le nœud `i` avec le « tag » `1`. Vous n'avez pas à vous occuper de la variable `status` (qui doit cependant être déclarée au préalable par `MPI_Status status;`).

Vous remarquerez dans `Principal.cc`, que les appels aux méthodes de communication sont répétés N fois (N étant fixé en dur). Ceci est fait pour avoir des temps de communication relativement longs, et donc plus représentatifs de la méthode.

3.4 Méthode de communication 2.

La deuxième stratégie de communication envisagée est celle où tous les cœurs envoient leur somme partielle au coeur 0, qui fait l'addition totale, puis envoie le résultat à tous les cœurs (voir figure 3.2). Cette approche est plus efficace que la précédente puisque le nombre de communications est de $2(p-1)$, le nombre d'additions de $p-1$, et la profondeur est toujours égale à $p-1$. Elles est implémentée dans la méthode `Communication_intermediaire()` où **vous devez complétez les « ... »**.

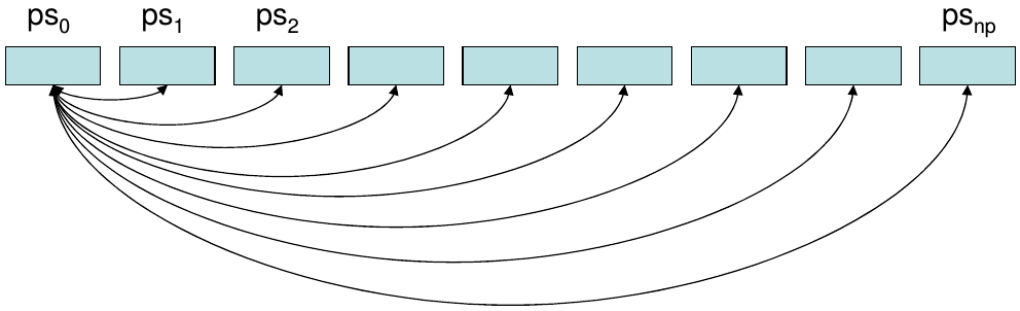


FIGURE 3.2 – Méthode de communication « intermédiaire ».

3.5 Méthode de communication 3.

Enfin cette troisième approche est l'approche optimale, puisqu'elle a le même nombre de communications et d'additions effectuées que la méthode 2, mais avec une profondeur de $\log_2 p$. Chaque cœur de numéro impair envoie sa somme partielle au cœur pair immédiatement inférieur qui l'ajoute à sa propre somme partielle. Les cœurs impairs se retirent du cycle, et après renumérotation le processus est recommencé jusqu'à ce que le cœur 0 ait la somme totale. Le processus est alors inversé pour redistribuer la somme totale à tous les cœurs. Si le cœur de numéro maximal est pair, alors celui-ci est retiré de la liste et sa somme est directement envoyée au cœur 0. Cet algorithme, illustré sur la figure 3.3, n'est pas à programmer cette année (cependant les personnes intéressées peuvent compléter la méthode `Communication_optimale()`).

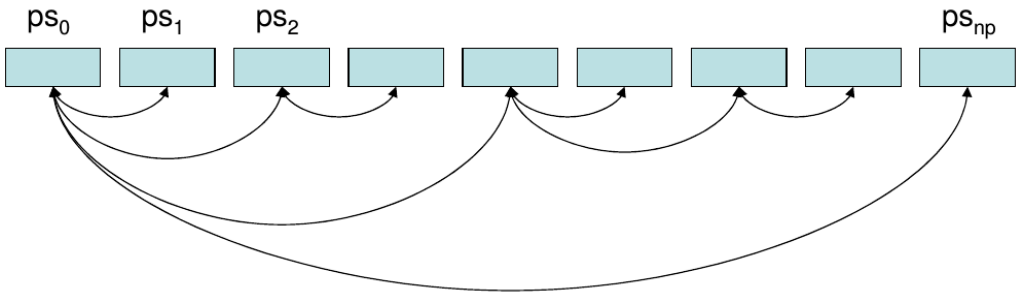


FIGURE 3.3 – Méthode de communication optimale.

3.6 Les fonctions MPI de réduction

L'opération que l'on vient de programmer de plusieurs manières s'appelle réduction (des données locales en une donnée globale). Il existe deux fonctions MPI qui opèrent cette réduction (de manière optimisée),

```
MPI_Reduce(tab_p,tab,n,type,MPI_SUM,root,MPI_COMM_WORLD)
```

et

```
MPI_Allreduce(tab_p,tab,n,type,MPI_SUM,MPI_COMM_WORLD),
```

la différence étant que `MPI_Allreduce` redistribue la valeur finale sur tous les cœurs, contrairement à `MPI_Reduce`. Ici, `tab` est le « tableau » (ou buffer) contenant les données à envoyer. C'est donc un pointeur. Le paramètre `n` est le nombre de valeurs contenues dans ce tableau, `type` le type de ces données (`MPI_INT`, `MPI_DOUBLE`, ...), et `root` est le numéro du cœur qui recevra la valeur finale (`root` vaut 0 en général). Enfin, la somme n'est pas la seule manière de combiner différentes données. Il existe d'autres opérateurs que `MPI_SUM`, tels que `MPI_MIN`, `MPI_MAX` ... La fonction `MPI_Allreduce(...)` est appelée dans la méthode `Communication_Reduce()`.

A chaque fin de programmation d'une des méthodes de communication, exécutez le programme suivant le protocole vu pour le précédent TP. Observez que dans `Somme.job`, un entier est mis à la suite du nom de l'exécutable. En regardant le fichier `Principal.cc`, trouvez à quoi correspond cet entier.

Mémoire et allocation dynamique - C / C++

Toolbox Calcul Haute Performance

J. Bruchon

École Nationale Supérieure des Mines de Saint-Étienne

Sommaire

1 La mémoire

2 Allocation dynamique

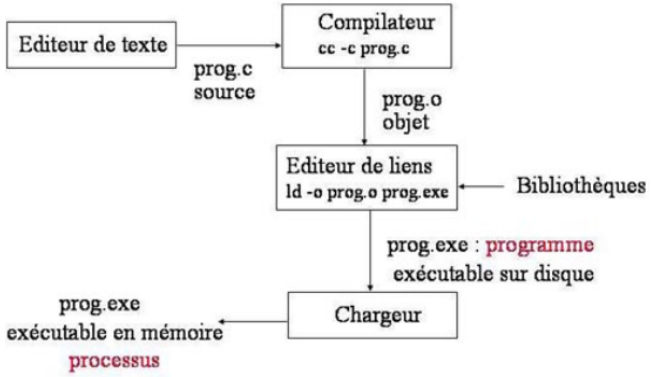
Sommaire

1 La mémoire

2 Allocation dynamique

Exécution d'un programme

Lors de l'exécution d'un programme informatique, le fichier exécutable est d'abord chargé dans la mémoire centrale de l'ordinateur.



Cette mémoire est une succession d'octets (8 bits), organisés les uns à la suite des autres et directement accessibles par une adresse.

La pile et le tas

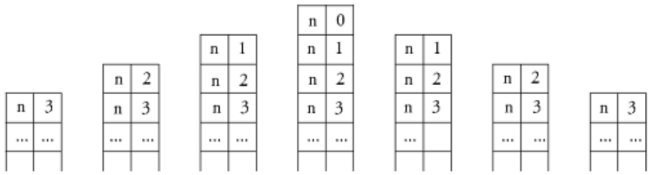
La mémoire centrale se divise en deux segments :

- La pile (stack)
- Le tas (heap)

Dans la plupart des langages de programmation compilés, la pile est l'endroit où sont stockés les paramètres d'appel et les variables locales des fonctions.

La pile (stack)

- La pile est un espace mémoire réservé au stockage des variables désallouées automatiquement.
- Sa taille est limitée mais on peut la régler.
- La pile est bâtie sur le modèle LIFO (Last In First Out ou Dernier Entré Premier Sorti).



- Lorsqu'une valeur est dépilée elle est effacée de la mémoire.

Le tas (heap)

- Le tas est l'autre segment de mémoire utilisé lors de l'allocation dynamique de mémoire durant l'exécution d'un programme informatique.
- Les fonctions `malloc` et `free`, ainsi que les opérateurs du langage C++ `new` et `delete` permettent, respectivement, d'allouer et désallouer la mémoire sur le tas.
- La mémoire allouée dans le tas doit être désallouée explicitement.

Commande Linux

Visualisation de la quantité de mémoire qu'un processus peut allouer sur le tas ou la pile

```
$ ulimit -a
...
data seg size           (kbytes, -d) unlimited
...
stack size              (kbytes, -s) 8192
```

La taille limitée de la pile (ici 8 Mo), peut provoquer des écrasements de variables et surtout des « **erreurs de segmentation** »en cas de dépassement. C'est ce qui se passe dans un appel récursif mal contrôlé, ou lors de l'allocation statique de variables trop volumineuses.

Sommaire

1 La mémoire

2 Allocation dynamique

Allocations statique et dynamique

- L'**allocation statique** de la mémoire se fait sur la **pile** au moment où le programme est chargé en mémoire centrale. Ceci signifie que la taille de l'espace à allouer est fixée dans le fichier exécutable du programme.
- L'**allocation dynamique** de la mémoire se fait sur le **tas** durant l'exécution du programme. Elle permet, par exemple, d'allouer des tableaux dont la taille est fixée par l'utilisateur.

Allocations statique et dynamique

- L'**allocation statique** de la mémoire se fait sur la **pile** au moment où le programme est chargé en mémoire centrale. Ceci signifie que la taille de l'espace à allouer est fixée dans le fichier exécutable du programme.
- L'**allocation dynamique** de la mémoire se fait sur le **tas** durant l'exécution du programme. Elle permet, par exemple, d'allouer des tableaux dont la taille est fixée par l'utilisateur.

Adresse d'une variable

L'emplacement d'une variable en mémoire est repéré par l'**adresse** de la variable. Ainsi, les lignes

```
int i = 5;
printf("La valeur de la variable i est %d \n",i);
printf("L'adresse de la variable i est %p \n",&i);
```

donnent l'affichage suivant

```
La valeur de la variable i est 5
L'adresse de la variable i est 0023FF74
```

où 0023FF74 est l'adresse de `i` dans la pile. C'est un nombre en hexadécimal. **L'opérateur « & »renvoie l'adresse de la variable à laquelle il s'applique**.

Pointeurs

Un **pointeur** est une variable contenant l'adresse d'une variable.

```
int i = 5;
int* pti = &i;   // pointeur sur un_Type : un_Type* pt;
printf("La valeur de la variable pti est %p \n",pti);
```

donne

```
La valeur de la variable pti est 0023FF74
```

Tandis que

```
printf("La valeur de la variable pointée par pti
                                est %d \n",*pti);
```

renvoie

```
La valeur de la variable pointée par pti est 5
```

L'opérateur « * »renvoie la valeur de la variable pointée par le pointeur auquel il s'applique.

Pointeurs passage par adresse

Passage par valeur

```
void ma_fonction(int a)
{
   a = 10;
}

int main()
{
  int b = 0;
  ma_fonction(b);
  printf("Valeur de b : %d \n",b);

  return 0;
}
```

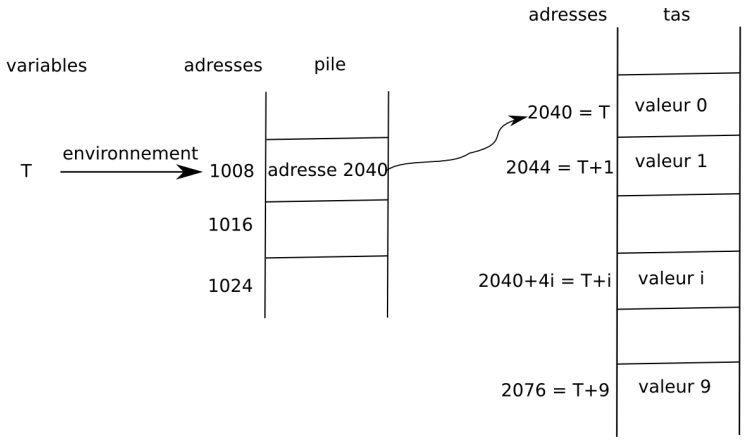
Affichage : `Valeur de b : 0`

Passage par adresse

```
void ma_fonction(int* a)
{
   *a = 10;
}

int main()
{
  int b = 0;
  ma_fonction(&b);
  printf("Valeur de b : %d \n",b);

  return 0;
}
```

Affichage : `Valeur de b : 10`

Pointeurs et allocation dynamique



Les données allouées dynamiquement sont référencées par des pointeurs de la pile, mais sont créées ailleurs que sur la pile, dans la zone mémoire appelée le tas.

Allocation dynamique en C

```
int *p; // pointeur sur un entier
int *T; // pointeur sur un entier

// allocation dynamique d'un entier
p = (int *)malloc(sizeof(int)); // alloue 4 octets (= int) en mémoire
*p = 1; // ecrit 1 dans la zone mémoire allouée

// allocation dynamique d'un tableau de 10 int
T = (int *)malloc(sizeof(int) * 10); // alloue 4 * 10 octets en mémoire

// initialise le tableau avec des 0
for(int i=0; i<10; i++)
{
  *(T+i) = 0; // les 2 écritures sont possibles
  T[i] = 0;   // identique à la ligne précèdente
}

// désallocation nécessaire
free(T);
free(p);
```

malloc et free

- Allocation dynamique sur le tas : fonction `malloc : void* malloc(size_t taille)` ; où `taille` est la taille en octets de la zone mémoire à allouer.
- Valeur retournée : pointeur `void *` sur la zone mémoire allouée, ou `NULL` (ou 0) en cas d'échec de l'allocation.
- Lorsque la mémoire allouée dynamique n'est plus utilisée, elle doit **impérativement** être libérée en appelant la fonction `void free(void *pointeur);`
- Une fois la zone mémoire libérée, on ne doit plus chercher à y accéder ou à la libérer une seconde fois !

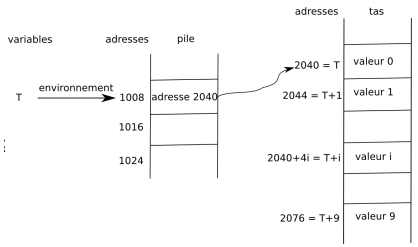
Quelques règles

- À tout `malloc` correspond un `free`
- Il est conseillé de désallouer les variables dans l'ordre inverse de leur allocation
- Lorsqu'on déclare un pointeur, on le met à `NULL` : `int* T = 0;`
- Avant l'appel à `malloc`, on vérifie que le pointeur est `NULL`. Si ce n'est pas le cas, on libère préalablement la mémoire
- Après un `malloc` on vérifie que le pointeur n'est pas `NULL`
- Après un `free` on met le pointeur à `NULL`.

Arithmétique des pointeurs

Soit,

```
int *T;
T = (int *)malloc(sizeof(int) * 10);
```



alors,

- `T` contient l'adresse de la première des dix cases mémoire allouées.
- `*T`, ou encore `T[0]` donne accès à la valeur contenue dans la première des dix cases allouées.
- `T+i` est une opération sur un pointeur : renvoie l'adresse de la ième case des dix cases allouées (décalage ici de $4i$ octets en mémoire). Si $i > 9$: problème grave !
- `*(T+i)` ou `T[i]` donne accès à la valeur contenue dans la i ième des dix cases allouées.

Allocation dynamique en C++

En C++, les opérateurs `new` et `delete` équivalent aux fonctions `malloc` et `free` précédentes.

```
#include <iostream>
#include <new>
using namespace std;
int main () {
   int * p = new int;
   int * T = new int[10];

   *p = 1; // ou bien p[0] = 1;
   for (int i=0; i<10; i++)
   { T[i] = 0; }

   // affiche le contenu de la zone mémoire allouée, ici 1 :
   cout << "Valeur pointée par la variable p : " << *p << endl;
   // affiche l'adresse contenue dans T :
   cout << "Valeur de la variable T : " << T << endl;
   cout << "Valeur contenu à l'adresse " << T+1 << " : " << T[1] << endl;

   delete p; // libère la zone mémoire allouée
   delete [] T; // delete T; marche aussi
   return 0;
}
```

Tableaux à plusieurs dimensions

Il est possible de créer statiquement ou dynamiquement des « tableaux de tableaux », c-à-d, des tableaux à plusieurs dimensions. On gère alors des pointeurs de pointeurs (de pointeurs de ...). Pourtant, il est plus simple (cf TP), de ne créer que des tableaux à 1 dimension. Ainsi, **une matrice $n \times m$ de réels est déclarée** :

```
int n = 100, m = 1000;
double * matrice = new double[n*m];
```

Pour accéder à la case (i, j) de la matrice, il suffit de faire

```
matrice[i*m+j] = 1;
```

par exemple, **si l'on a stocké la matrice ligne par ligne**. La désallocation se fait par `delete [] matrice;`

C++ : passage par adresse et par référence

Passage par adresse

```
void ma_fonction(int* a)
{
   *a = 10;
}

int main()
{
  int b = 0;
  ma_fonction(&b);
  printf("Valeur de b : %d \n",b);

  return 0;
}
```

Affichage : `Valeur de b : 10`

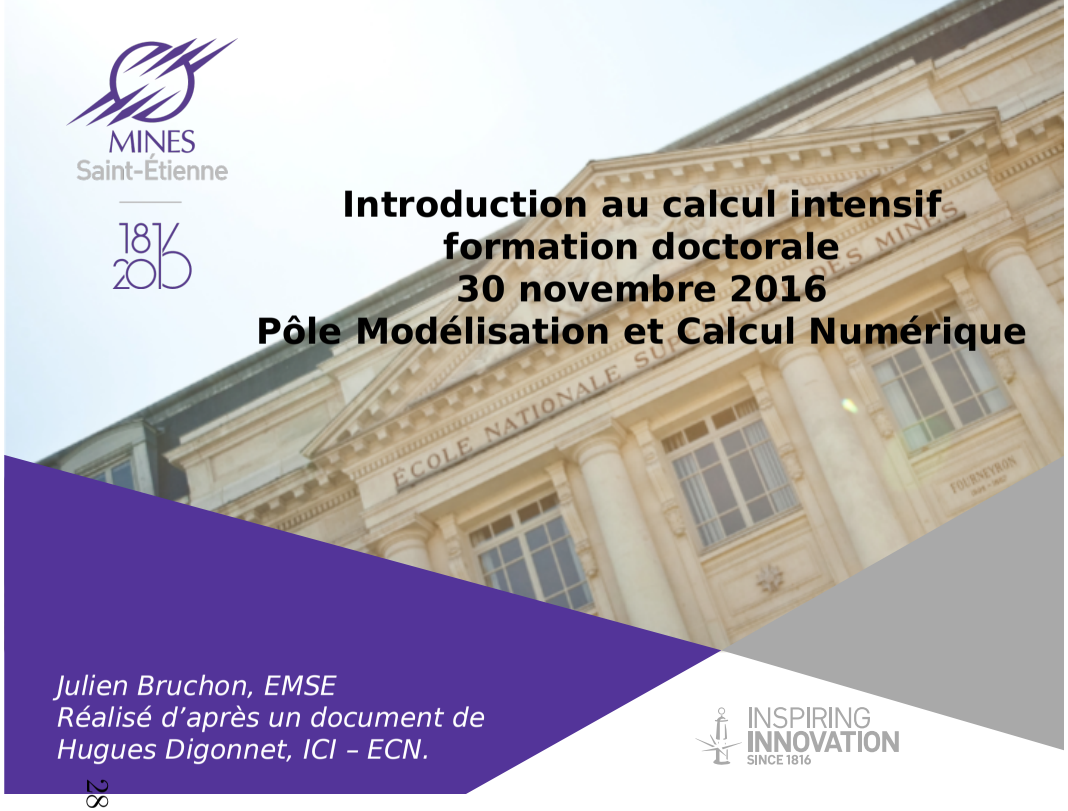
Passage par référence

```
void ma_fonction(int & a)
{
   a = 10;
}

int main()
{
  int b = 0;
  ma_fonction(b);
  printf("Valeur de b : %d \n",b);

  return 0;
}
```

Affichage : `Valeur de b : 10`. Ici `int & a` désigne une référence vers un entier.

Une référence n'est pas une nouvelle variable, il n'y a pas d'allocation mémoire. Elle doit donc être déclarée et affectée en même temps :

```
double c;
double & d = c;
d = 10; // équivaut à c = 10;
```



MINES Saint-Étienne

1816 2016

Introduction au calcul intensif
formation doctorale
30 novembre 2016
Pôle Modélisation et Calcul Numérique

Julien Bruchon, EMSE
Réalisé d'après un document de
Hugues Digonnet, ICI – ECN.

INSPIRING INNOVATION SINCE 1816

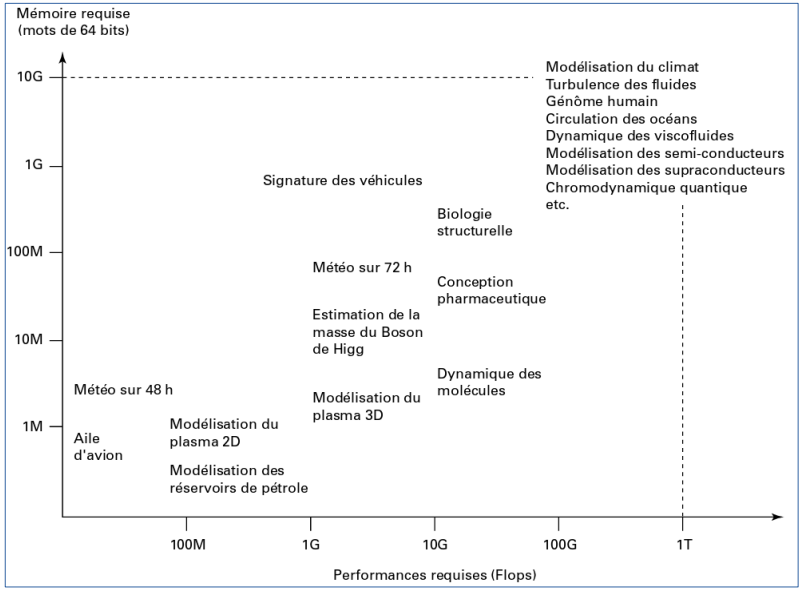
Introduction au calcul intensif

- Des généralités sur le parallélisme

- Mesures et analyse de performances

- Programmation en parallèle

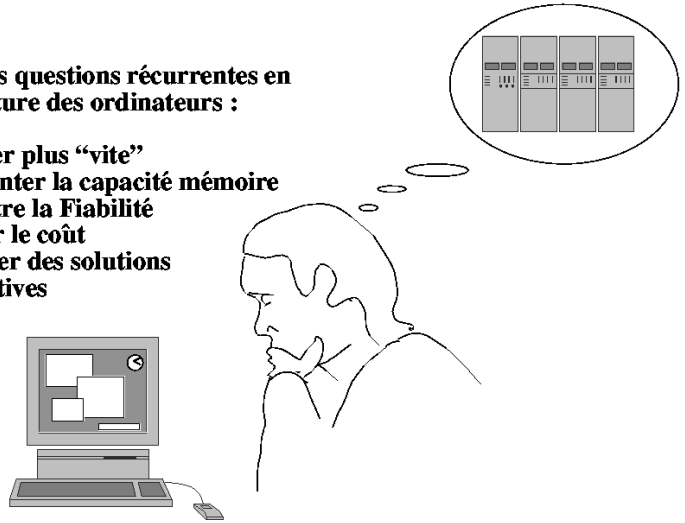
Motivation pour le prallélisme



Motivation pour le parallélisme

Quelques questions récurrentes en architecture des ordinateurs :

- **Calculer plus "vite"**
- **Augmenter la capacité mémoire**
- **Accroître la Fiabilité**
- **Limiter le coût**
- **Proposer des solutions alternatives**

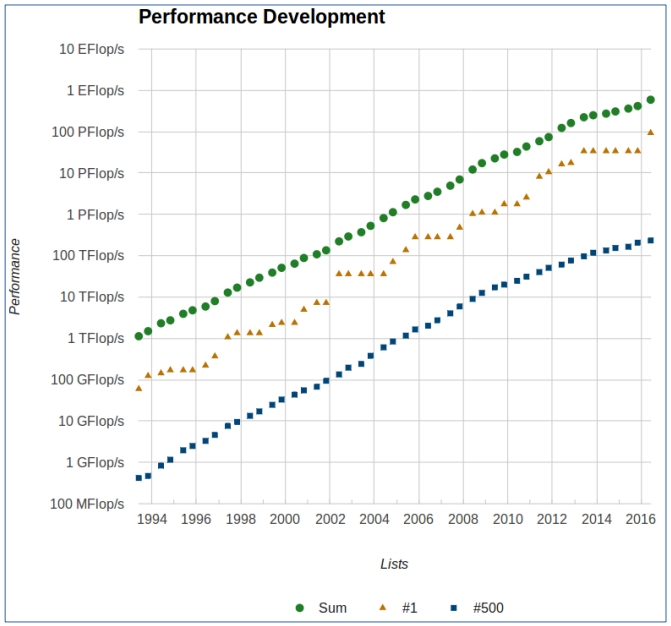


Motivation pour le parallélisme

https://www.top500.org/
(Novembre 2016)

Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	
Site:	National Supercomputing Center in Wuxi
Manufacturer:	NRCPC
Cores:	10,649,600
Linpack Performance (Rmax)	93,014.6 TFlop/s
Theoretical Peak (Rpeak)	125,436 TFlop/s
Nmax	12,288,000
Power:	15,371.00 kW
Memory:	1,310,720 GB
Processor:	Sunway SW26010 260C 1.45GHz
Interconnect:	Sunway
Operating System:	Sunway RaiseOS 2.0.5

Evolution des performances



Parallélisme dans les applications

Soit le programme séquentiel suivant :

```
Pour i de 1 à n
        faire A[ i ] ← B[ i ] + C[ i ]
Finpour
```

Les instructions : A[x] ← B[x] + C[x] , sont indépendantes pour 1≤ x ≤ n

Plus généralement deux instructions consécutives sont indépendantes si il n'y a

ni dépendance de flot

z ← x + y
w ← z + r

ni antidépendance

w ← z + y
z ← x + r

ni dépendance de sortie

z ← x + y
z ← w + r

Pas de dépendances de sortie ←

A[i - 1] ←	B[i - 1] + C[i - 1]
A[i] ←	B[i] + C[i]
A[i + 1] ←	B[i + 1] + C[i + 1]

→ **Pas de A[i] dans le membre droit**

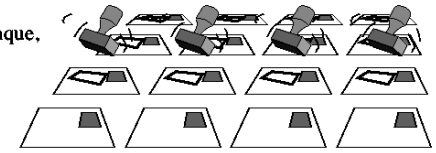
Augmenter les performances

Si les instructions : A[x] ← B[x] + C[x] , sont indépendantes pour 1≤ x ≤ n

Elles peuvent être exécutées dans un ordre quelconque,

Elles peuvent être exécutées simultanément !

Principe des architectures parallèles :



Exécuter simultanément des instructions indépendantes sur des ressources différentes :

A[i-1] ← B[i-1] + C[i-1]

B[i-1] C[i-1]
CPU1
A[i-1]

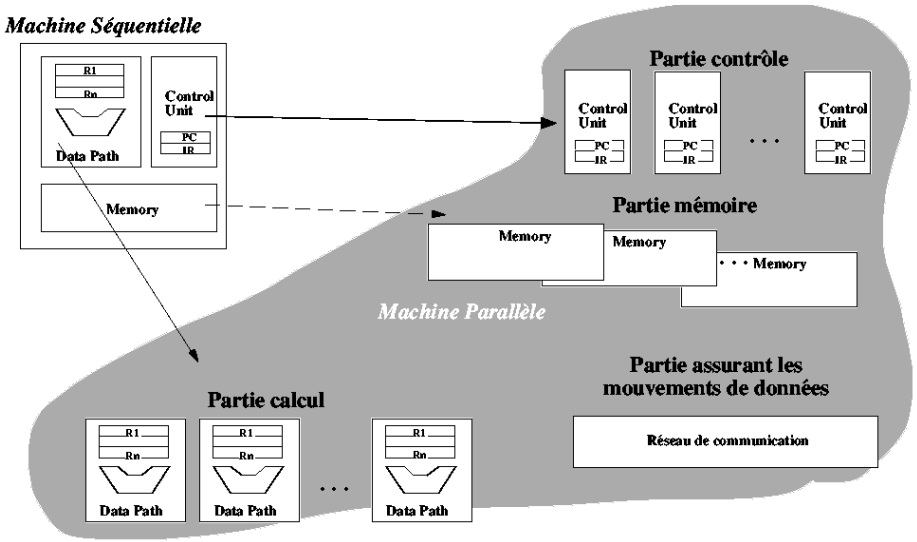
A[i] ← B[i] + C[i]

B[i] C[i]
CPU2
A[i]

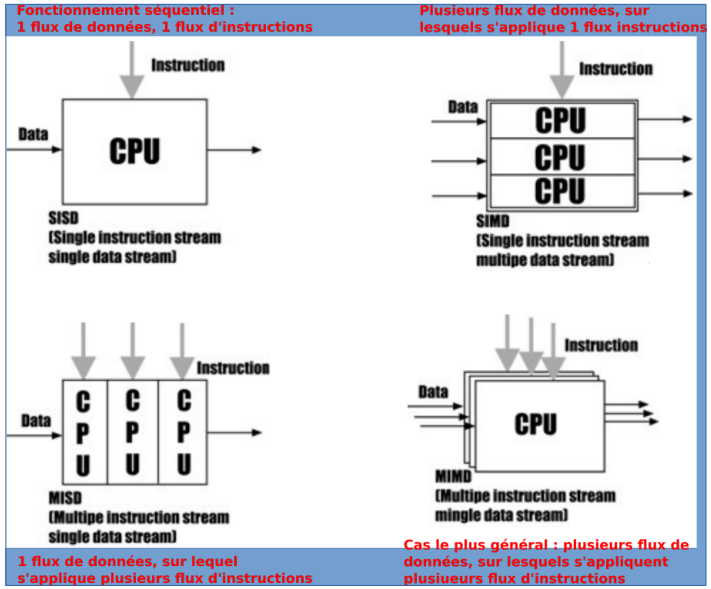
A[i+1] ← B[i+1] + C[i+1]

B[i+1] C[i+1]
CPU3
A[i+1]

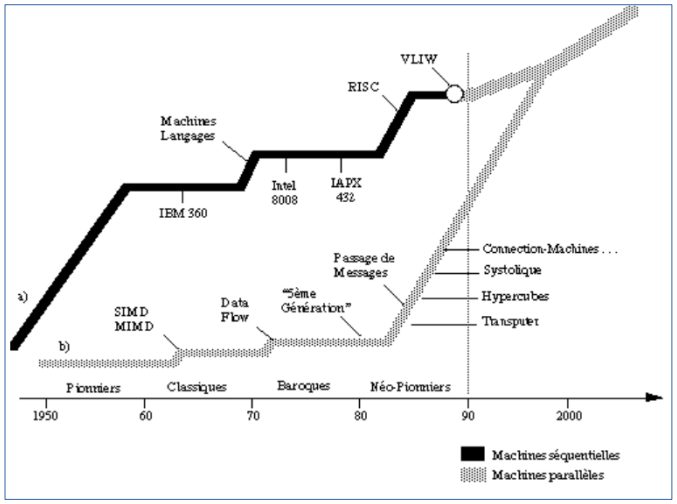
Ordinateurs parallèles



Classification de Flynn

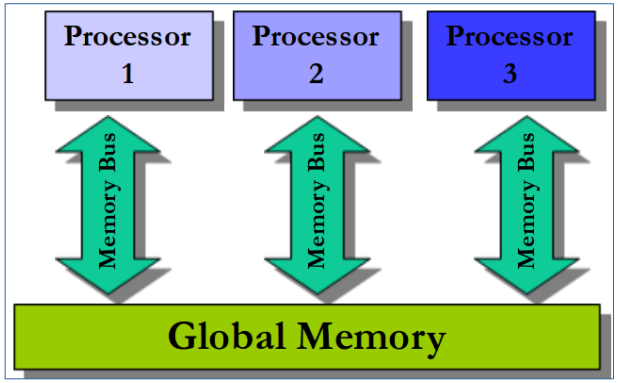


Histoire des architectures



Représentation (subjectives) des progrès réalisés dans les architectures a) séquentielles et b) parallèles. La différence entre les deux courbes reflète l'activité et les moyens de recherche mis en œuvre dans chaque domaine.

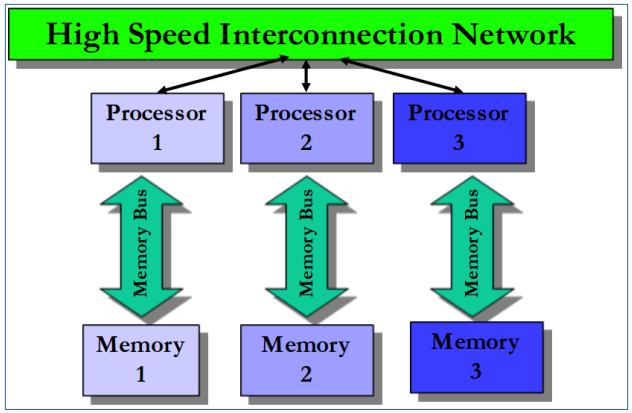
MIMD : Architecture à mémoire partagée



Chaque processeur accède à la même mémoire.

- Facile à construire
- Inconvénients : fiabilité & scalabilité : une panne d'un composant mémoire ou de l'un des processeurs affecte tout le système.
- Augmentation du nombre de processeurs → limitée par la taille mémoire.

MIMD : Architecture à mémoire distribuée



Chaque processeur accède à sa propre mémoire.

- Communication entre les procs par un réseau à haut débit
- Ce réseau peut être configuré selon différentes topologies : arbre, cube, ...
- Contrairement à MIMD à mémoire partagée :
  - Facilement extensible
  - Fiabilité accrue.

Réseaux dans les architectures MIMD

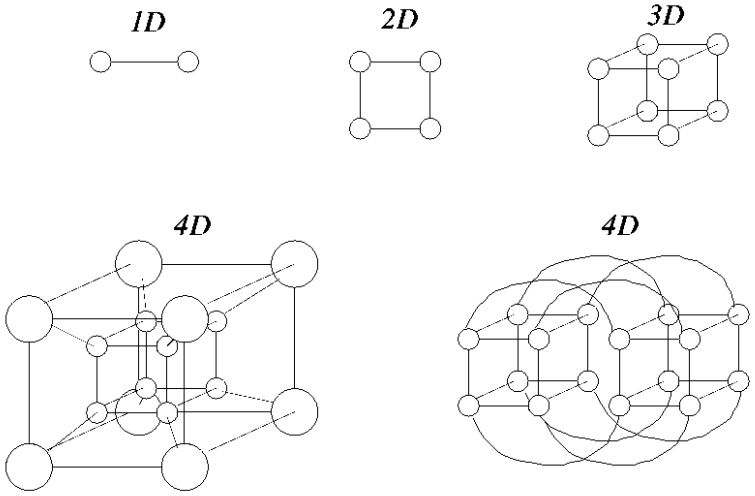
Tout échange entre les processeurs nécessite un transfert de données via le réseau d'interconnexions des processeurs.

Le coût d 'un tel transfert :

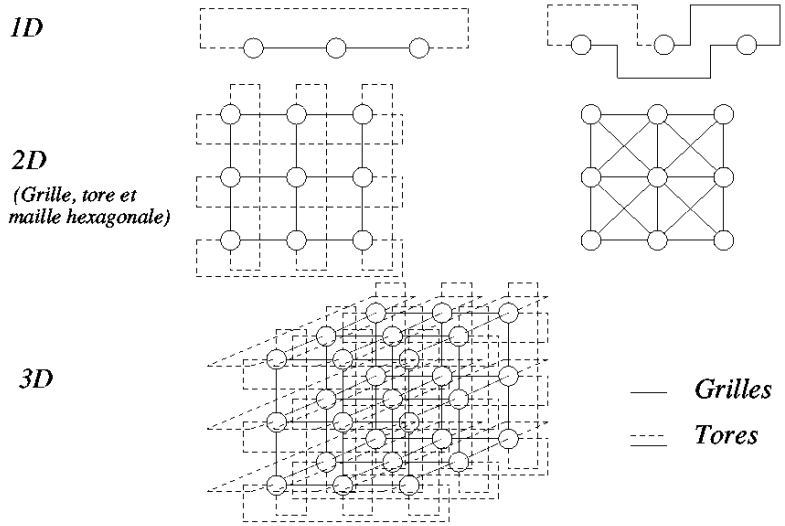
$$t(N,i,j) = t_{latence} + t_{pack} + \frac{N}{v_{transfert}} + (d-1)t_{transit}$$

- où d est le nombre de connections directes nécessaires pour passer du processeur i au processeur j.:

Topologie maillées : hypercubes



Topologie maillées : grilles et tores



Granularités des applications

C'est le rapport entre le temps passé par l'application à effectuer des opérations de calcul, et le temps passé dans les communications entre les processeurs :

$$\frac{t_{calcul}}{t_{communication}}$$

Si ce rapport est élevé l'application se prête bien au parallèle, au contraire s'il est faible, l'application sera difficilement parallélisable.

Temps d'exécution

Différents temps mesurables :

- En séquentiel :
  - temps cpu
  - temps IO (entrée-sortie)
  - temps appel - système
  - temps calculs
  - temps wall-clock
- En parallèle :
  - idem mais un Temps (Ti) par processeur d'où
  - moyenne(Ti)
  - min(Ti)
  - max(Ti) => le plus réaliste

Outils de mesure

Mesures externes :

> time mpirun –np 4 a.out
> /usr/bin/time mpirun –np 4 a.out
> times mpirun –np 4 a.out
> timex mpirun –np 4 a.out

Remarque : Nom et fonctionnement des variables selon le système utilisé !

12.002u user
0.128s system
12.150 total
Fréquemment : total > user + system !!

Les +
Simple à utiliser
Pas de modifications des codes sources
Les –
Peu précis: ± 0.5s

Outils de mesure

Mesures internes :

time()
clock() Compte les clicks d'horloge
gettimeofday() Appel système le plus précis

Remarques :
- Toutes ces routines ne sont pas toujours disponibles !
- "gettimeofday" est en général une bonne solution.
- Parfois, il existe des outils plus précis pour mesurer de petites durées.

Les +
Plus précis que les mesures externes
Les –
Besoin de modifier le code source
Pas toujours portable

Expression des performances : Speed-up

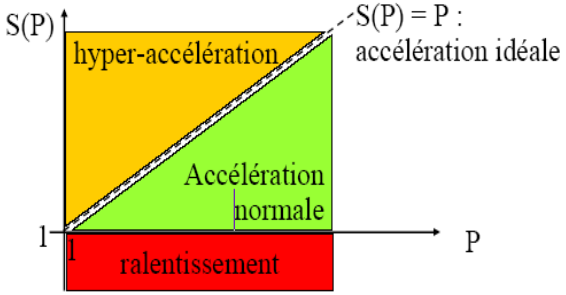
Définition : Speed-Up ou accélération

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

- S(P) < 1 : on ralentit ; très mauvaise parallélisation

- 1 < S(P) < P : accélération “normal”. Plus on est près de P, meilleure est la parallélisation

- P < S(P) : hyper accélération. Analyser & justifier



Expression des performances : Speed-up

Hyper accélération :

Ce n’est pas *magique*, et ce n’est pas normal :
- On doit analyser le phénomène et l’expliquer
- Corriger une erreur ou exploiter une optimisation

Exemples d’explications :

- on ne fait plus les bonnes opérations (résultat faux)
- les données tiennent dans le cache total des P processeurs
- on a modifié l’algorithme de départ et on converge plus vite

Expression des performances : Speed-up

Hyper accélération :

Ce n’est pas *magique*, et ce n’est pas normal :
- On doit analyser le phénomène et l’expliquer
- Corriger une erreur ou exploiter une optimisation

Exemples d’explications :

- on ne fait plus les bonnes opérations (résultat faux)
- les données tiennent dans le cache total des P processeurs
- on a modifié l’algorithme de départ et on converge plus vite

Expression des performances : Efficacité

Définition :

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

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

- $E(P) \in [0;1]$ ou [0%:100%]
- E(P) > 100% ⇔ hyper accélération

Remarques :

- L’utilisateur s’intéresse surtout à l’accélération obtenue
- L’acheteur de la machine s’intéresse beaucoup à l’efficacité
- Le développeur s’intéresse aux deux

Quelle référence séquentielle ?

Choix de la référence séquentielle :

- A quel programme et exécution séquentielle se comparer ?

- Même programme lancé sur un seul processeur ?
- Même algorithme implanté en séquentiel ?
- Meilleur algorithme séquentiel connu ?

- Compilation séquentielle avec le même compilateur ?
- Compilation avec le meilleur compilateur séquentiel ?

- Optimisations séquentielles autorisées par la parallélisation ?
- Optimisations séquentielles maximales ?

- Exécution sur un seul processeur de la machine parallèle ?
- Exécution sur la meilleure machine séquentielle ?

Quelle référence séquentielle ?

Tous les choix sont possibles,

Chaque choix de référence séquentielle correspond à :
- un point de vue différent,
- une préoccupation différente,
- un objectif d'analyse différent

L'important est de :
- Faire le choix correspondant à sa problématique
- Énoncer clairement ce choix

Exemple de choix :

- Utilisateur final : SON pgm séq. sur SA machine séq.
- Paralléliseur : même algo sur un proc de la machine parallèle

Source des pertes de performances

- Sous optimisation séquentielle → Aspects séquentiels

- Fraction séquentielle
- Surcoût des opérations de gestion du parallélisme → Algorithmique et programmation parallèle
- Déséquilibre de charge

- IO séquentielles/séquentiallisées
- Sous optimisation des outils et langages parallèles → Environnement de développement

Source des pertes de performances

La fraction séquentielle d'un programme :

la part d'un programme parallèle
ne s'exécutant que sur un seul processeur
ou identiquement sur tous.

Deux lois tentent de prédire les accélérations maximales de tels programmes :

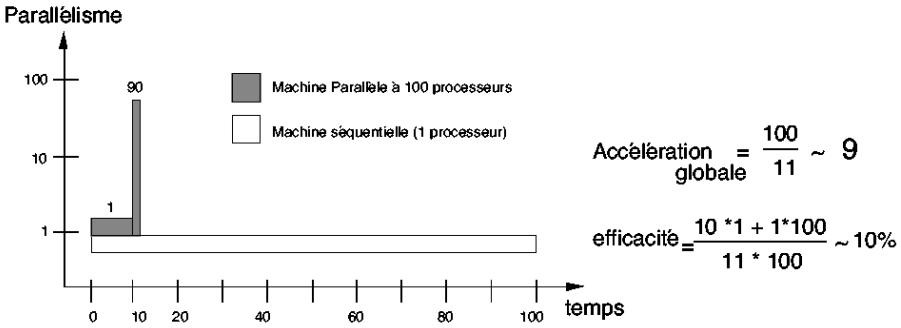
- la **loi d'Amdahl**

- la **loi de Gustafson**

Loi d'Amdahl

Limite de l'accélération d'une architecture parallèle : la borne est fixée par la partie séquentielle

$$\text{Accélération globale} = \frac{\text{Temps d'exécution}_{\text{ancien}}}{\text{Temps d'exécution}_{\text{nouveau}}} = \frac{1}{\left(1 - \text{fraction}_{\text{améliorée}}\right) + \frac{\text{fraction}_{\text{améliorée}}}{\text{facteur d'accélération}}}$$



$$\text{Accélération globale} = \frac{100}{11} \sim 9$$

$$\text{efficacité} = \frac{10 * 1 + 1 * 100}{11 * 100} \sim 10\%$$

Loi de Gustafson

La loi d'Amdahll suppose que la taille du problème est constante

Mais l'utilisateur veut traiter le plus gros problème possible

-> La partie parallèle augmente avec la taille du problème
La partie séquentielle reste fixe

Programme = partie séquentielle + partie parallèle = s + q

Avec P processeurs , Le temps de calcul selon la loi d'Amdahl : t = s+ q/p

En supposant que la partie parallèle augmente linéairement avec le nombre de processeur : q = α p

d'où, t = s+ α

et l'accélération $A = \frac{s + \alpha p}{s + \alpha}$, $A \rightarrow p$

Programmation parallèle

Par des données parallèles [HPF] [Open MP] :

- Introduction de nouveaux types de données (vecteur parallèle) et de nouvelles instructions (FORALL). Principalement utilisé sur des machines à mémoire partagée et à adressage unique.
- Synchronisation importante après chaque instruction scalaire ou parallèle mais implémentation relativement simple.

Par échanges de messages [pvm] [mpi] :

- Utilisation de librairies contenant des instructions d 'envois, de réceptions de messages plus des instructions de synchronisation.
- Le programmeur doit gérer le parallélisme, mais les synchronisations sont moins fréquentes.

OpenMP, HPF (GPU : Cuda + OpenCL)

Un modèle de programmation simple

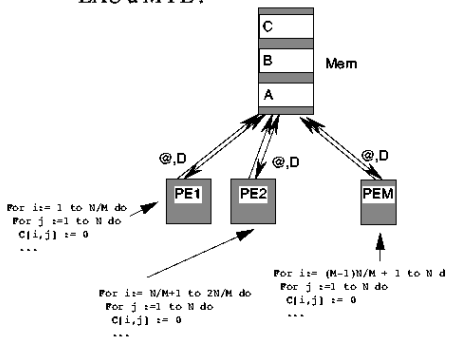
- spécialement lorsque les schémas de communication sont complexes et varient dynamiquement
- qui simplifie la répartition du calcul et l'équilibrage de charge.

Exemple : C = A * B ou A, B et C sont des matrices N*N

Une version parallèle :

```
Par_For i:= 1 to N do
 For j :=1 to N do
  C[i,j] := 0
  For k := 1 to N do
   C[i,j] := C[i,j]
             + A[i,k] * B[k,j]
```

Une exécution sur un multiprocesseur EAU à M PE :



MPI (PVM)

Proc 0	Proc 1
#include <mpi.h >	#include <mpi.h>
MPI_Init(...)	MPI_Init(...)
int a=6;	int a=12;
MPI_Send(&a,1,MPI_INT,1,...)	MPI_Send(&a,1,MPI_INT,0,...)
MPI_Recv(&a,1,MPI_INT,1,...)	MPI_Recv(&a,1,MPI_INT,0,...)
cout << a << endl;	cout << a << endl;
MPI_Finalize()	MPI_Finalize()
12	**6**

MPI (PVM)

MPI_Init(...)

MPI_Finalize()

MPI_Comm_rank(...)

MPI_Comm_size(...)

double MPI_Wtime()

MPI_Send(…)

MPI_Recv(…)

MPI_COMM_WORLD

MPI_Allreduce(...)

MPI_Bcast(..)

MPI_Isend(…)

MPI_Irecv(...)

MPI_Waitall(..)

MPI_Barrier(...)

MINES
Saint-Étienne

1816
2016

Cluster de calcul CENTAURE

N. Moulin, Th. Louvancourt, J. Mancuso

INSPIRING
INNOVATION
SINCE 1816

Un cluster ?

Définition

- Le terme de cluster (ou grappe, en français) désigne un ensemble d'ordinateurs indépendants, appelés nœuds, tous interconnectés par un réseau dédié.
- On dispose ainsi d'une machine capable de traiter des problèmes de très grande taille, en utilisant la puissance cumulée de ses nœuds.
- Liste des plus « gros » clusters : https://www.top500.org/

Objectifs

- Effectuer de très gros calculs comportant typiquement plusieurs millions (milliards) d'inconnues.
- Effectuer de nombreux calculs séquentiels simultanés.

Clusters et EMSE

Petit historique...

- 1973 : La rotonde...*un centre de calcul*

Clusters et EMSE

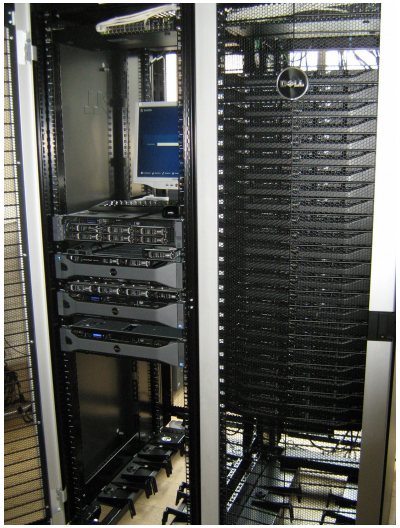
Petit historique...

- 1973 : La rotonde...*un centre de calcul*
- 2005 : Ferme de PC

Clusters et EMSE

Petit historique...

- 1973 : La rotonde...*un centre de calcul*
- 2005 : Ferme de PC
- 2009 : Cluster Pegase (centre SMS) 30k€



Clusters et EMSE

Petit historique...

- 2016 : Cluster Centaure (investissement EMSE : 80k€)

Descriptif technique

Architecture globale

- 1 nœud maître ou frontale,
- 26 nœuds de calcul (384 cœurs de calcul),
- 1 réseau Ethernet Gigabit (administration)
- 1 réseau Infiniband (calcul)
- 250 Go de disque pour le système (RAID1)
- 10 To de disque dur pour les données (NFS),
- 200 Go ~ 1 To de disque (scratch) sur les nœuds.

Descriptif technique

Configuration des nœuds (rack1 – rack2 - rack3)

- Bi-processeurs Intel Xeon E5-2660 v3 (2,6GHz, 10C/20T, 25Mo de mémoire cache, Turbo)
- 64Go de Ram, 700 Go de scratch
- 1 réseau Infiniband / 1 réseau Ethernet Gigabit
- Bi-processeurs Intel Xeon X-5650 (2,6GHz, 6C, 12Mo de mémoire cache)
- 24Go de Ram, 250 Go de scratch
- 2 réseaux Ethernet Gigabit
- Bi-processeurs Intel Xeon E-5530 (2,4GHz, 4C, 8Mo de mémoire cache)
- 32Go de Ram, 150 Go de scratch
- 2 réseaux Ethernet Gigabit

→ http://services-numeriques.emse.fr/pole-modelisation-et-calcul-numerique/cluster-centaure

Descriptif technique

Architecture logicielle

- Distribution CentOS 7 64bits (équivalent RedHat, durée des dépôts...),
- Logiciels de monitoring du cluster : ganglia (http://centaure/ganglia/),
- Compilateurs C/C++/fortran (GNU et Intel),
- Debugger (gdb),
- Python,
- Librairies spécifiques pour le calcul parallèle : openmpi, PETSC, BLAS, Lapack...
- Logiciels de gestion de queue de calcul : SLURM

Descriptif technique

Logiciels scientifiques

- Zset/ZeBuLoN : calcul parallèle par éléments finis (Mines-ParisTech – CdM, Onera, Safran, Mines-St-Etienne),
- CimLib : calcul parallèle par éléments finis (Mines-ParisTech – Cemef),
- Comsol / Abaqus /Ansys : codes commerciaux de calcul par éléments finis,
- GMSH : logiciels de maillage,
- Matlab : calcul scientifique,
- CPLEX : optimisation,
- Outils de visualisation : Paraview, Visit...
- ...

  → Base de données : référencement des méthodes numériques / logiciels / codes de calculs https://portailmetier.emse.fr/ApplisWeb/modelisation/index.php

Utilisation du Cluster

Comment accéder à Centaure

→ http://services-numeriques.emse.fr/pole-modelisation-et-calcul-numerique/cluster-centaure/acces-au-cluster-centaure

- Rappel : fonctionne sous **linux**.
- Nom de la machine sur le réseau : **centaure**
- Création de compte via **admin-centaure.emse.fr**
  (N. Moulin, Th. Louvancourt, J. Mancuso)
- Protocole de communication distant : **ssh** (pour Windows, utiliser putty ou Xming)
  ssh -X login centaure.emse.fr
- Transfert des données via les commandes **scp** ou rsync ou en mode interactif avec les logiciels gftp, filezilla ou **winscp**.
- Chaque utilisateur dispose d'un espace personnel (/export/home/login) mais **le cluster n'est pas un espace de stockage de données**

Utilisation du Cluster

Modules et environnements

→ http://services-numeriques.emse.fr/pole-modelisation-et-calcul-numerique/cluster-centaure/modules-et-environnements

- La commande module permet de lister et d'utiliser simplement les logiciels, librairies ou utilitaires installés sur le cluster et ainsi configurer l'environnement des utilisateurs. Pour lister l'ensemble des modules existant, il faut utiliser la commande :
  `module avail`
- Si un module vous semble manquant, n'hésitez pas à nous le faire savoir (admin-centaure@emse.fr).
- Pour charger un module, la commande est :
  `module load abaqus/6-14.1`
  Cette commande charge l'ensemble de l'environnement nécessaire à l'exécution du code Abaqus en version 6.14 .
- Pour lister les modules chargés dans votre environnement :
  `module list`
- Pour décharger un module chargés dans votre terminal ou un script :
  `module unload abaqus/6-14.1`
- Cette commande va décharger l'ensemble des modules chargés par le module abaqus/6-14.1.
- Pour décharger tous les modules :
  `module purge`

Gestionnaire de travaux et soumission

→ http://services-numeriques.emse.fr/pole-modelisation-et-calcul-numerique/cluster-centaure/gestionnaire-de-travaux-et-soumission

- **Les calculs sur le Cluster s'effectuent par l'intermédiaire d'un gestionnaire de travaux** qui s'occupe de gérer la file d'attente et de lancer les calculs lorsque les ressources demandées sont disponibles.
- Le gestionnaire de travaux du Cluster est **SLURM** (Simple Linux Utility for Resource Management). SLURM est open source et très utilisé parmi les calculateurs du top 500.

Soumission des travaux

- La soumission d'un job se fait avec la commande
  `sbatch slurm.job`
  où `slurm.job` est un fichier de script dans lequel sont contenues des instructions pour SLURM ainsi que des instructions pour le lancement de votre programme.
- Cette commande retourne un numéro de job (JOBID)

Exemple de script SLURM

```
#!/bin/bash
#SBATCH --job-name=job-slurm-mpi
#SBATCH --mail-user=you@emse.fr
#SBATCH --mail-type=ALL
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --time=01:00:00

module load mpi/openmpi-x86_64

echo ----------------------------------------------------------
echo SLURM_NNODES: $SLURM_NNODES
echo SLURM_JOB_NODELIST: $SLURM_JOB_NODELIST
echo SLURM_SUBMIT_DIR: $SLURM_SUBMIT_DIR
echo SLURM_SUBMIT_HOST: $SLURM_SUBMIT_HOST
echo SLURM_JOB_ID: $SLURM_JOB_ID
echo SLURM_JOB_NAME: $SLURM_JOB_NAME
echo SLURM_JOB_PARTITION: $SLURM_JOB_PARTITION
echo SLURM_NTASKS: $SLURM_NTASKS
echo SLURM_TASKS_PER_NODE: $SLURM_TASKS_PER_NODE
echo SLURM_NTASKS_PER_NODE: $SLURM_NTASKS_PER_NODE
echo ----------------------------------------------------------

echo Generating hostname list...
COMPUTEHOSTLIST=$( scontrol show hostnames $SLURM_JOB_NODELIST |paste -d, -s )
echo ----------------------------------------------------------

echo Creating SCRATCH directories on nodes $SLURM_JOB_NODELIST...
SCRATCH=/scratch/$USER-$SLURM_JOB_ID
srun -n$SLURM_NNODES mkdir -m 770 -p $SCRATCH  || exit $?
echo ----------------------------------------------------------
echo Transferring files from frontend to compute nodes $SLURM_JOB_NODELIST
srun -n$SLURM_NNODES cp -rvf $SLURM_SUBMIT_DIR/* $SCRATCH  || exit $?
echo ----------------------------------------------------------

echo Run -mpi program...
mpirun -np 4 -npernode 2 --mca btl openib,self -host $COMPUTEHOSTLIST
$SLURM_SUBMIT_DIR/mpi_hello_world-host
echo ----------------------------------------------------------

echo Transferring result files from compute nodes to frontend
srun -n$SLURM_NNODES cp -rvf $SCRATCH  $SLURM_SUBMIT_DIR   || exit $?
echo ----------------------------------------------------------
echo Deleting scratch...
srun -n$SLURM_NNODES rm -rvf $SCRATCH  || exit 0
echo ----------------------------------------------------------
```


pour simplifier...

- Un script spécifique a été développé pour générer automatiquement différents fichiers SLURM. Pour utiliser ce script, il faut charger d'abord le module correspondant :
  `module load tools/cluster-bin`
- Le fichier .job est créé en exécutant la commande :
  `cluster-create-slurm-script-01.sh`
  suivie d'une option permettant de spécifier le modèle de fichier .job que vous voulez créer (fichier pour lancer Abaqus, Zset, ...).
- La commande **`cluster-create-slurm-script-01.sh -h`** permet de connaître les différents modèles disponibles.

Gestion des travaux

- La commande pour voir l'état des jobs est :
  `squeue`
  Cette commande ne montre que vos propres jobs !
- La commande pour arrêter un job est :
  `scancel JOBID`
  avec JOBID le numéro du job.
- La commande pour vérifier l'état des nœuds et des partitions est :
  `sinfo`
- La version graphique :
  `sview`

Conclusions

Quelques recommandations..

- Ne pas hésiter à utiliser le cluster même pour des calculs modestes, cela décharge vos machines personnelles
- ⚠ à vos données : ce n'est pas espace de stockage !
- ⚠ Utilisation raisonnée (jetons de licences, walltime...)!

Merci de votre attention