Systèmes multicoeurs

Introduction et aspects logiciels

Alain MÉRIGOT

Université Paris Saclay

Position du problème

La loi de Moore indique un doublement du nombre de transistors tous les deux ans, et cela s'est traduit longtemps par une augmentation similaire de la puissance de calcul. « *the free lunch* »

L'amélioration des performances se faisait :

- en augmentant la fréquence des processeurs
- en améliorant les architectures des processeurs :
  - pipeline,
  - exécution dans le désordre, parallélisme d'instructions (superscalaires, VLIW),
  - exécution spéculative,
  - élargissement des chemins de données (8 bits, puis 16, 32, 64, 128 ?? bits),
  - parallélisme de données (SIMD),
  - *multithreading*,
  - ajout de caches L1, puis L2, et L3 dans les processeurs
  - etc

Position du problème (cont.)

Mais il s'avère que :

- l'augmentation de fréquence se heurte au problème de la consommation. Doubler la fréquence revient à quadrupler la consommation (à cause de l'augmentation de tension qui accroit la fuite des transistors).
  Il est deux fois plus économique en puissance d'avoir deux processeurs que de doubler la fréquence d'un processeur (*power wall*)
- il n'y a plus beaucoup de progrès architecturaux exploitables sur un processeur (*ILP wall*)[1]
- la différence entre performances des mémoires et des processeurs ne fait que s'accroitre (*memory wall*)

Passage à un concept multicoeurs

La continuité de la loi de Moore provient maintenant de l'utilisation de plusieurs processeurs sur un circuit.

[1] ILP : *Instruction level parallelism*

Position du problème (cont.)

Il existe actuellement de nombreux circuits intégrant des processeurs multicoeurs, à la fois dans les processeurs généralistes, le calcul haute performances et les systèmes embarqués.

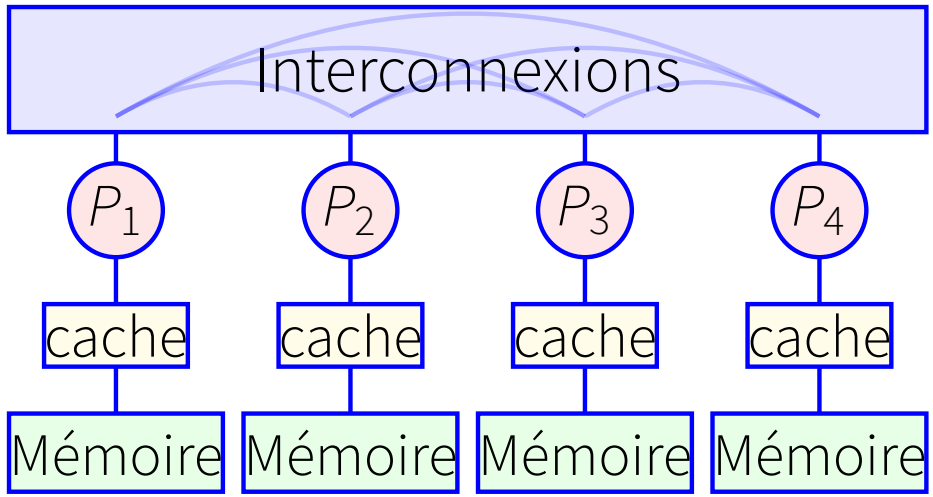
Le nombre de processeurs sur un même circuit peut actuellement atteindre quelques dizaines, voire quelques centaines, et la tendance est que le nombre de processeurs va s'accroitre.

Simultanément, on voit également une tendance à l'hétérogénéité des processeurs (intégration sur un même circuit de CPU, GPU, DSP, IP spécialisé, FPGA, etc).

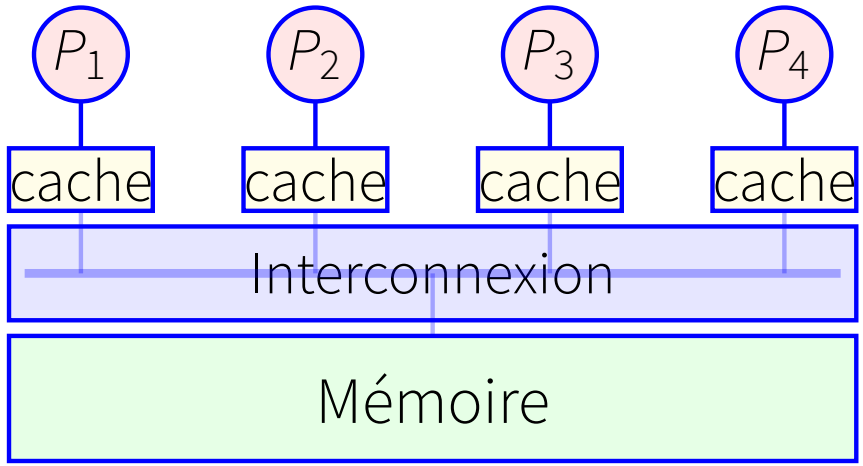
Différents types de multicoeurs

Deux grandes classes de multicoeurs:

Processeurs à mémoire distribuée
Chaque processeur a sa propre mémoire, mais les processeurs peuvent communiquer entre eux



Processeurs à mémoire partagée
Il existe une mémoire globale commune aux différents processeurs.



Différents types de multicoeurs (cont.)

Processeurs à mémoire distribuée : ce sont les plus simples à concevoir. Chaque processeur a une mémoire privée, et un réseau de communication permet aux processeurs d'échanger des informations.

Problèmes :

- Il faut un réseau de communication efficace
- Les échanges d'informations sont réalisés dans le programme par des directives particulières.
  - Grosse lourdeur de programmation
  - Peu de réutilisation possible des programmes existants
  - Mise au point difficile

Principal mécanisme logiciel utilisé : `MPI` (Message Passing Interface)

Processeurs à mémoire partagée.

Tous les processeurs voient une grosse mémoire partagée par tous. Par contre chaque processeur va avoir son propre cache

Toutes les données de tous les processeurs sont disponibles, et la programmation est largement simplifiée.
Il existe de nombreux environnements permettant de programmer efficacement ces processeurs : *threads*, OpenMP, Cilk, TBB, Rust, Go, etc...

Principales difficultés.

- Cohérence des différents caches
- Réalisation d'un mécanisme rapide d'accès à la mémoire globale.

Différents types de multicoeurs (cont.)

...
Les premiers systèmes à mémoire partagée n'avaient qu'un faible parallélisme (quelques processeurs), et la mémoire était vraiment commune et globale.
On parle de **symmetric multiprocessors (SMP)**

Avec l'augmentation du nombre de processeurs, la tendance est de répartir la mémoire. Chaque processeur a sa propre mémoire, mais elle est accessible par tous. Par contre, l'accès à une donnée en mémoire locale est évidemment plus rapide que celui à une donnée dans la mémoire d'un autre processeur.
On parle de **processeurs NUMA (non-uniform memory access)**
Le placement optimal des données en mémoire est un des gros problèmes rencontrés.

Programmation des processeurs multicoeurs

Soit un algorithme à mettre en oeuvre sur un système paralléle.
A partir d'une description du problème, il doit être possible de décrire la répartition des traitements sur les différents processeurs.

Cette mise en oeuvre doit prendre en compte les différents niveaux de parallélisation du problème:

- parallélisme de tâches
- parallélisme de données

Tout en respectant des *dépendances* entre calculs.

Exemple : soit à calculer $\mathbf{S} = \mathbf{A} \times \mathbf{X} + \mathbf{B} \times \mathbf{Y}$ où $\mathbf{A}, \mathbf{B}, \mathbf{S}, \mathbf{X}, \mathbf{Y}$ sont des vecteurs de taille n et où $\times$ représente le produit point à point des éléments de deux vecteurs.

```
1  for(int i=0;i<n;i++)
2     S[i] = A[i] * X[i] + B[i] * Y[i] ;
```

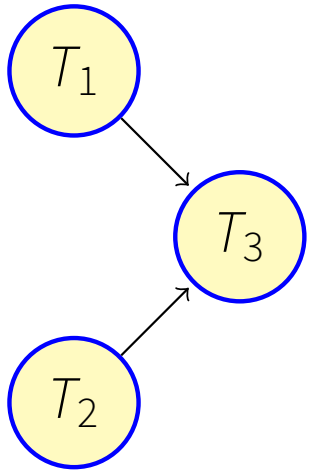
Une mise en oeuvre possible en trois tâches est la suivante, en utilisant des vecteurs de taille n `S1` et `S2`:

```
1  T1: for(int i=0;i<n;i++) S1[i] = A[i] * X[i] ;
2  T2: for(int i=0;i<n;i++) S2[i] = B[i] * Y[i] ;
3  T3: for(int i=0;i<n;i++) S[i] = S1[i] + S2[i] ;
```

Les trois tâches T_i peuvent s'exécuter sur différents processeurs ou opérateurs spécialisés.

— Les tâches T_1 et T_2 sont indépendantes et peuvent être réalisées en parallèle (*parallélisme de tâche*). T_3, par contre dépend de T_1 et T_2.

```
for(int i=0;i<n;i++)
  S1[i] = A[i] * X[i] ;
```




```
for(int i=0;i<n;i++)
  S[i] = S1[i] + S2[i] ;
```

```
for(int i=0;i<n;i++)
  S2[i] = B[i] * Y[i] ;
```

— Au sein de chaque tâche, les différents calculs sont indépendants. Les différents `A[i]*X[i]` peuvent être calculés indépendamment (*parallélisme de données*).

Programmation des processeurs multicoeurs (cont.)

- De plus, la dépendance entre les tâches T_1 et T_3 (ainsi qu'entre T_2 et T_3) n'est que partielle.

  On peut commencer à calculer `S[0]=S1[0]+S2[0]` dès que `S1[0]` et `S2[0]` sont disponibles (sans attendre la fin complète des tâches T_1 et T_2). (*Pipeline*)

Ces différents niveaux de parallélismes peuvent être exploités pour une mise en oeuvre parallèle efficace de l'algorithme, à condition de respecter les dépendances.

Un langage de programmation comme le C n'est pas adapté à faire apparaître les dépendances (notamment à cause des *effets de bord* des fonctions, et des alias mémoire).

La parallélisation automatique à partir d'une description séquentielle est encore un problème ouvert.

Existence de langages spécifiques permettant au programmeur d'exprimer le parallélisme.

Programmation des processeurs multicoeurs (cont.)

Pour la programmation des systèmes multicoeurs, les modèles de calculs sont liés aux caractéritiques des architectures.

Mémoire distribuée : MPI *message passing interface*

Mémoire partagée :

- *threads*
- openMP (open *multiprocessors*)

Message Passing Interface

Suppose un ensemble de process coopérants.

Chaque process a sa propre mémoire (et peut avoir plusieurs *threads*).

MPI permet de spécifier la communications entre ces process

- Echanges de données
- Synchronisation

MPI est une bibliothèque standardisée.

Très utilisé en calcul haute performance quand le nombre de coeurs est très important ou pour du calcul *en grille* (ensemble de calculateurs communicant par internet).

L'échange de données suppose :

– un des process envoie des informations

– un autre process reçoit les informations

Problèmes :

– Identifier les process

– Décrire les données

– Reconnaitre les messages

– Savoir si les opérations ont été effectuées

MPI a des notions de *groupe* et de *contexte* qui permettent d'identifier les messages.

Les échanges sont basés sur des types de données simples (`MPI_INT`, `MPI_DOUBLE`, ...).
Le protocole MPI réorganise les octets si le boutisme des processeurs est différent.

Le protocole MPI permet d'assurer qu'aucun message n'est perdu, mais l'ordre n'est pas garanti.
Un marqueur (*tag*) permet d'identifier et d'ordonner les messages.

Principales fonctions :

`MPI_Init()` A utiliser en début de programme
`MPI_Finalize()` En fin de programme
`MPI_Comm_size()` Indique le nombre de process communicants
`MPI_Comm_rank()` Permet de donner à chaque process un numéro unique

Communications point à point :
`MPI_Send()` Envoi un bloc de données à un autre process
`MPI_Recv()` Reçoit un bloc de données

Communications collectives :
`MPI_Scatter()` Envoi 1 vers N
`MPI_Gather()` Envoi N vers 1

Exemple de programme MPI
Le process 0 envoie un message à tous les autres et reçoit leur réponse.

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

int main(int argc, char *argv[])
{
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
     // definit le nombre de process
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    // chaque process a son id dans myid
  ...
```

Message Passing Interface (cont.)

```
if(myid == 0)   // code du processeur 0
{
  for(i=1;i<numprocs;i++)  // le proc. 0 envoie quelque chose aux autres
  {
    sprintf(buff, "Hello %d! ", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
  }
  for(i=1;i<numprocs;i++) // et reçoit leur réponse
  {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
    printf("%d: %s\n", myid, buff);
  }
}
else
 ...
```

```
  ...
  else  // code des autres processeurs
  {
    //On reçoit de 0
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    // la donnée est dans buff. On peut la modifier
    // ...
    // et renvoyer le résultat à 0
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

Message Passing Interface (cont.)

MPI est un standard de fait très utilisé en calcul haute performances.

Il permet la parallélisation sur des architectures distribuées de grande taille[2], en assurant une portabilité des programmes entre architectures très différentes.

Indispensable pour les machines à mémoire distribuée.

Permet également de mélanger mémoire distribuée et partagée depuis MPI-3.0 (et utilisable sur les architectures NUMA).

Par contre,

- programmation très lourde et verbeuse
- mise au point et *debug* très difficile

[2] Plus gros calculateur (2022) : Fugaku (Japon) 7.6M processeurs (Arm V8), 440 PFlops

Rappels : Processus et *threads*

Les **processus** correspondent à des activations différentes d'un ou plusieurs programme.

Chaque activation a

- son propre contexte processeur (état des registres, PC),
- et son propre contexte système (mémoire allouée, utilisateur, répertoire de travail, fichiers ouverts, etc)

Les différents processus sont entièrement indépendants.

La communication entre processus utilise des mécanismes particuliers mis en oeuvre par le système d'exploitation (signaux, *pipes*, *sockets*, mémoire partagée (*shm*), etc)

Les *threads* (ou *fils d'exécution*) sont lancés au sein d'un processus.

Tous les *threads* d'un processus partagent le même contexte système, et notamment la même mémoire.
Seul leur contexte processeur est différent.

Tous les *threads* voient les modifications de l'état de la mémoire effectuées par tous les *threads* d'un même processus.
Aucun mécanisme de communication particulier n'est donc nécessaire.

Rappels : Processus et *threads* (cont.)

Un *thread* :

- Existe au sein d'un process et utilise ses ressources
- A son propre flot d'exécution
- Partage les ressources du process avec les autres *threads*
- N'existe que tant que le process existe

Les *threads* permettent :

- de simplifier la programmation de programmes devant réagir à des requêtes (interface utilisateur, client-serveur, etc)
- de masquer les temps d'attente d'un monoprocesseur (ex: défauts de cache), s'il y a un support multithreads dans le processeur (et plusieurs *threads* dans le programme).
- de fournir un modèle de parallélisation utilisable à la fois sur un seul processeur et sur différents coeurs à mémoire partagée

Les *pthreads*

Le mécanisme de *threads* le plus utilisé sont les *threads* Posix (`pthreads`) (Ansi/IEEE Posix 1003.1)
Une fois créés, le passage d'un *thread* à un autre peut :

- Être géré par le programme (*threads* coopératifs)
- Être géré par le système d'exploitation en fonction des E/S, d'un quota de temps, etc (*threads* préemptifs, actuellement supportés par la plupart des systèmes).

Permet la gestion des *threads* (création, suppression, etc), la synchronisation (`mutex`)

Gestion des *threads*

`pthread_create (thread_id,attr,start_routine,arg)`
Crée un *thread*. `thread_id` est un identificateur retourné par la fonction, `attr` permet de positionner des attributs spécifiques (taille de pile, priorité, etc), `start_routine` est la fonction à exécuter par le *thread*, `arg` est l'argument *unique* à passer à cette fonction.

`pthread_exit (status)` Termine le *thread* courant. `status` est la valeur renvoyée par le *thread*

`pthread_join (thread_id,status)` Attend la fin du *thread* `thread_id`

`pthread_cancel (thread_id)` Termine le *thread* `thread_id`

Problèmes liés à l'accès simultané à la mémoire

Les accès simultanés à la mémoire par des *threads* différents peuvent conduire à des résultats inprédictibles Exemple : 2 *threads* ajoutent une valeur à une variable globale (copiée dans un registre du processeur).

Thread A

```
locale=globale
locale += 20
globale = locale
```

Thread B

```
locale=globale
locale += 10
globale = locale
```

Exemple d'exécution

initialement `globale = 1000`

Thread A

```
locale=globale    // (=1000)
// on change de thread


locale += 20      // (=1020)
globale = locale  // (=1020)
```

Thread B

```


locale=globale    // (=1000)
locale += 10      // (=1010)
globale = locale  // (=1010
```

La mise à jour d'une variable globale peut conduire à un résultat incorrect.

Ce problème est résolu par des mécanismes d'**exclusion mutuelle**
On peut définir un **verrou** protégeant l'accès à une variable globale partagée.
A un moment donné, un seul *thread* peut fermer (posséder) le verrou.
L'acquisition du verrou par les autres *threads* sera interdit.
Une fois la modification de la variable partagée faite, le premier *thread* peut libérer le verrou.

`pthread_mutex`

`pthread_mutex_t mutex` Déclaration du verrou

`pthread_mutex_init (mutex,attr)` Créé le verrou mutex

`pthread_mutex_destroy (mutex)` Détruit le verrou mutex

`pthread_mutex_lock (mutex)` Attend que le verrou mutex soit libre (bloquant) et le verrouille

`pthread_mutex_trylock (mutex)` Teste l'état du verrou mutex (non bloquant) ; si verrou libre, acquisition et renvoie 0, sinon renvoie -1.

`pthread_mutex_unlock (mutex)` Déverrouille mutex

Exemple :

Thread A

```
pthread_mutex_t verrou ;
...
pthread_mutex_init(&verrou);
...
pthread_mutex_lock(&verrou);
locale=globale;    //(=1000)
// on change de thread



// on repasse au thread A
locale += 20       //(=1020)
globale = locale   //(=1020)
pthread_mutex_unlock(&verrou)
//déverrouille et débloque thread B
...
// suite du thread A
```

Thread B

```
pthread_mutex_lock(&verrou);
// le verrou est bloqué et
// le thread suspendu



// le thread B est débloqué
locale=globale;    //(=1020)
locale += 10       //(=1030)
globale = locale   //(=1030)
pthread_mutex_unlock(&verrou)
...
```

Le résultat final est correct.

Les verrous permettent de gérer correctement les modifications de variables globales et plus généralement la synchronisation des *threads*.

Mais leur utilisation incorrecte peut conduire à des problèmes.

Thread A

```
pthread_mutex_t verrou1,verrou2 ;
pthread_mutex_init(&verrou1);
pthread_mutex_init(&verrou2);
pthread_mutex_lock(&verrou1);
pthread_mutex_lock(&verrou2);
...
```

Thread B

```
pthread_mutex_lock(&verrou2);
pthread_mutex_lock(&verrou1);
...
```

Certaines exécutions vont conduire à une interblocage (*deadlock*)

Thread A

```
pthread_mutex_lock(verrou1);
// bascule vers thread B

pthread_mutex_lock(verrou2);
// attend la libération de verrou2
...
```

Thread B

```
pthread_mutex_lock(verrou2);
pthread_mutex_lock(verrou1);
// attend la libération de verrou1
...
...
```

Les *threads* peuvent être utilisés pour :

- du macroparallélisme.
  Chaque *thread* exécute une tache indépendante.
- du microparallélisme
  L'exécution d'une opération complexe (par exemple la manipulation d'un gros tableau de données) est alors partagée entre plusieurs *threads*.

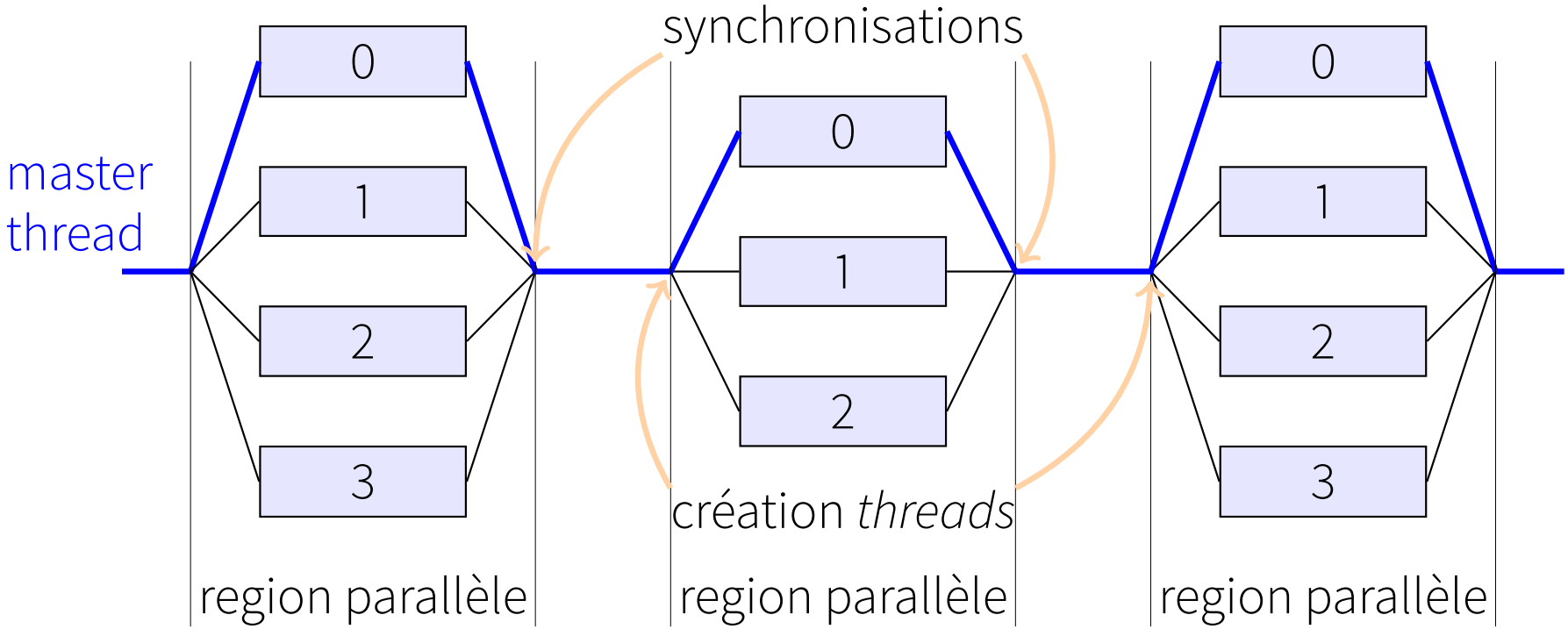
Le deuxième mode de parallélisme est potentiellement plus efficace, mais sa progammation au moyen de *threads* est particulièrement lourde.
OpenMP vise à résoudre ce problème.

openMP

Standard de programmation multithread (1997)
Initialement orienté vers du parallélisme de données régulier (vecteurs/matrices).
La version 3 introduit la notion de *task* facilitant la mise en oeuvre de parallélisme de données général.
La version 4 étend la parallélisation des traitements SIMD et à des accélérateurs (GPU).

Basé sur la bibliothèque `pthread`
Utilise des `pragma`s donnant des directives au compilateur
Un compilateur ne comprenant pas les pragmas openMP générera du code séquentiel.
Un compilateur openMP générera du code multithreads en répartissant la tache à effectuer entre plusieurs *threads* avec les verrous adaptés.
Supporté par divers compilateur dont gcc, depuis la version 4.2.
Open MP est particulièrement intéressant pour du microparallélisme.

Le modèle de calcul openMP suppose un fil d'exécution principal (*master thread*).
Il s'exécute en séquentiel, jusqu'à ce qu'il rencontre des directives pour lancer des *threads* en parallèle.
On attend la fin d'exécution des ces *threads* (synchronisation), et on revient au *thread* principal.



L'ensemble des *threads* d'une région parallèle est appelé une *team*.

Directive de répartition des tâches

`parallel` exécute le bloc suivant en parallèle sur tous les *threads*.

```
#pragma omp parallel
{ printf("Hello, world!\n"); }
```

A la fin d'un bloc `parallel`, il y a une ***barrière de synchronisation***.

Une *barrière de synchronisation* met en attente un *thread* tant de *tous* les *threads* n'ont pas atteint la barrière.

La directive `for` parallélise une boucle *for* entre plusieurs *threads*.

Permet du **microparallélisme**

```
int main()
{
  int tab[1000];
# pragma omp parallel for
  for(int i=0;i<1000;i++)
    tab[i]=i;
}
```

OpenMP divise la boucle entre les différents *threads*.
Par exemple, avec 10 *threads* et 1000 itérations, le *thread* 0 traitera les itérations 0–99, les *thread* 1, les itérations 100–199, etc.

`sections` indique que le bloc suivant va être réparti en *threads* indépendantes.
`section` indique un *thread* à exécuter en parallèle.
Permet du **macroparallélisme**

```
int main()
{
# pragma omp parallel // Crée une région parallèle
  {
    calcul1(); //exécuté par tous les threads
#   pragma omp sections // partage du travail en sections
    {
      { calcul2();} // exécuté par le master
#     pragma omp section
      { calcul3(); }  // exéuté par un thread
#     pragma omp section
      {  calcul4();    //Ces fonctions sont exécutées
         calcul5(); } //séquentiellement par un thread
#     pragma omp section
      { calcul6(); }  // exéuté par un thread
    } // fin de la zone de sections et resynchronisation par barrière
    calcul7(); // exécuté par tous les threads
  }// fin de la zone parallèle et barrière de synchronisation
}
```

`single` le bloc n'est exécuté que par un seul *thread*.
A la fin, une barrière resynchronise.

`master` identique à `single`, mais le code est exécuté par le *thread* principal.
Plus rapide, et *aucune* resynchronisation n'est nécessaire à la fin.

La synchronisation des *threads* utilise des directives particulières.
Normalement, il y a une barrière de synchronisation implicite à la fin d'une zone `for` ou `sections`.

On peut supprimer cette barrière implicite avec la clause **nowait**.
On peut ajouter une barrière explicite avec **barrier**.

`barrier` Permet de rajouter une barrière à tout endroit.
On attend que *tous* les *threads* atteigne la barrière avant de continuer.
Permet notamment d'assurer que tous les calculs précédents sont finis avant de passer à la suite (*cohérence des données*).

```
#pragma omp parallel
{
  calcul1(); // exécuté par tous les threads
# pragma omp barrier    // attente que tous les threads
                        // aient fini calcul1()
  calcul2(); // exécuté par tous les threads
} // barrière implicite à la fin d'une zone parallèle
```

`nowait` Toutes les barrières implicitement générées par le compilateur sont supprimées.

```
int A[100], B[100], C[100];
#pragma omp parallel
{
# pragma omp for                    // barrière implicite après le for
  for(int i=0;i<100;i++) A[i]=i;                    // Calcul1;
# pragma omp for                    // barrière implicite après le for
  for(int i=0;i<50;i++) B[i]=B[i+1]=A[2*i];         // Calcul2;
# pragma omp for nowait             // boucle for *sans* barrière
  for(int i=0;i<100;i++) A[i]=3*B[i];               // Calcul3;
# pragma omp for                    // barrière implicite après le for
  for(int i=0;i<99;i++) C[i]=2*B[i+1];             // Calcul4;
} // barrière implicite à la fin d'une zone parallèle
```

Une clause `nowait` permet d'accélérer l'exécution.
MAIS il faut que les calculs n'aient pas de dépendance.

Calcul2 a besoin du résultat de *Calcul1* (`A[i]`) et *Calcul3* a besoin de *Calcul2* (`B[i]`). Une barrière est nécessaire après les deux premiers `for`.
Par contre, *Calcul4* peut commencer même si tous les *threads* n'ont pas terminé *Calcul3* (pas de dépendance).

Depuis openMP 3.0, une clause `task` permet de paralléliser de manière plus souple. Chaque *thread* d'un région parallèle exécute une des `task`s.
Exemple : parcours de liste. Un *thread* parcourt la liste et crée une `task` pour chaque noeud rencontré.

```
struct node { int data; node* next; };
extern void process(node* );
void list_traversal(node* head)
{
# pragma omp parallel
  {
#   pragma omp single
    {
      for(node* p = head; p; p = p->next)
      {
#       pragma omp task firstprivate(p)
        process(p);
      }
    }
  }
}
```

L'ordre d'exécution des *threads* dans une clause `for` est non spécifié (dépend du compilateur et de l'exécution).
Il peut être défini à la compilation (par défaut), ou de manière dynamique.
Dans ce dernier cas, chaque *thread* demande un numéro d'itération et exécute quelques itérations. Utile si la fonction à exécuter peut prendre une durée variable.

```
#pragma omp parallel for schedule(dynamic,4)
for(int i=0; i<1000; i++) {
  // chaque thread exécute 4 itérations et redemande du travail
  une_fonction_avec_duree_variable(i);
}
```

Si les traitements dans chaque itération sont de longueur comparable, la répartition dynamique est moins efficace (à cause des synchronisations et du travail de répartition des itérations).

On peut forcer une exécution comme si le programme était séquentiel par la clause `ordered`. Pas de parallélisation.

Parallélisation de boucles imbriquées par `collapse(n)`

```
#pragma omp parallel for
for(int i=0; i<N; i++) {
  for(int j=0; j<M; j++) {
    f(i,j);
  }
}
```

Seule la boucle extérieure est parallélisée. Peut être inefficace si **N** est petit.

```
for(int i=0; i<N; i++) {
# pragma omp parallel for
  for(int j=0; j<M; j++) {
    f(i,j);
  }
}
```

La boucle extérieure est séquentielle et la boucle intérieure parallélisée.

```
#pragma omp parallel for collapse(2) // parallélise 2 boucles inbriquées
for(int i=0; i<N; i++) {
  for(int j=0; j<M; j++) {
    f(i,j);
  }
}
```

Boucles déroulées et parallélisées. NB: Nécessite des bornes d'itérations fixes.

Visibilité des variables

Dans une boucle parallèle, les variables de boucles (index) sont privées à chaque *thread*, ainsi que les variables définies dans la section parallèle.
Par défaut toutes les autres variables sont partagées.

`shared(list)` la variable est visible par toutes les *threads* (défaut)

`private(list)` chaque *thread* a une version locale de la variable (non initialisée).

`firstprivate(list)` identique à `private`, mais la valeur initiale est copiée du *thread* principal.

`lastprivate(list)` identique à `private`, mais la valeur de la variable lors la dernière itération ou de la dernière section est recopiée dans la variable globale.

`threadprivate(list)` utilisé juste après la déclaration d'une variable globale, rendra la variable *private* dans tous les *threads*.

`default(none)` Force l'utilisateur à préciser le mode de partage de toutes ses variables (conseillé).

L'exclusion mutuelle (par exemple pour l'accès à une variable globale) peut se faire avec les clauses **atomic** et **critical**.
atomic permet une mise à jour atomique d'une variable[3].

```
float global;
somme_globale(float t[N]) {
# pragma omp parallel for
  for(int i=0;i<N;i++){
#   pragma omp atomic
    global+=t[i];
  }
}
```

Différents type d'opérations atomiques :

update (mise à jour) Par défaut. Exemple : `global += local ;`

read Exemple : `local = global;`

write Exemple : `global = local;`

capture Lecture et mise à jour. Exemple : `local = global++;`

[3] Un accès est dit *atomique* s'il ne peut être coupé. Permet d'assurer qu'aucun autre *thread* ne pourra accéder à la variable pendant sa mise à jour.

La quasi totalité des architectures multicoeur utilisent un cache privé et une mémoire globale.

Les caches généralement ont des fonctionnalités permettant d'assurer une cohérence entre le contenu des caches et la mémoire globale (*cohérence mémoire*).

Parfois cette cohérence n'est que partielle ou nécessite une action du programmeur.

La clause `flush(list)` permet de forcer la mise à jour en mémoire de une ou plusieurs données partagées présentes dans le cache du processeur. Sans argument, toutes les données partagées sont recopiées en mémoire.

L'accès atomique est nécessaire pour un traitement correct des variables globales, mais **très** couteux (plusieurs centaines de cycles).
L'exécution parallèle de la fonction précédente (`somme_globale()`) serait **très** sensiblement plus longue qu'une version séquentielle.

Pour optimiser, primitive particulière de mise à jour d'une variable globale par plusieurs *threads* : `reduction`.

Une `reduction` permet d'appliquer un opérateur (par exemple addition) à un ensemble de données et stocke le résultat dans une variable globale.
Opérations autorisées dans une `reduction` : +, &, |, ^, &&, ||, *, min, max

```
unsigned long long fact(int n)  // calcul parallèle de n!
{
  unsigned long long f = 1;     // f est global aux différents threads
# pragma omp parallel for reduction(*:f)
  for(int i=2; i<n; i++)
    f *= i;
  return f;
}
```

Dans une `reduction`, `open-mp` optimise le temps de calcul tout en assurant l'exclusion mutuelle lors de la mise à jour de la variable globale.
Les `reduction`s utilisent une variable privée initialisée à l'élément neutre de l'opérateur.
A la fin du `for`, application de l'opérateur de réduction entre ces variables privées et la variable globale (de manière atomique).
La `reduction` précédente est strictement équivalente à :

```
unsigned long long fact(int n)
{
  unsigned long long f = 1;    // f est global aux différents threads
  unsigned long long tmp = 1; // pour le calcul dans chaque thread
# pragma omp parallel firstprivate(tmp) // tmp est rendu privé
  {
#   pragma omp for nowait // synchronisation apres le for inutile
    for(int i=2; i<n; i++)
      tmp *= i;              // calcul avec variable privée tmp
#   pragma omp atomic       // accumulation atomique des res partiels
    f *= tmp ;
  } // fin de la zone parallèle et synchronisation
  return f;
}
```

L'exclusion mutuelle peut également être réalisée par une *section critique*[4] avec la clause `critical`

`critical` un seul *thread* sera dans cette section à un moment donné.

Un accès atomique peut être réalisé avec une clause `critical`, mais les performances seront moins bonnes.

A l'inverse, une section critique peut concerner toute une zone de code, avec éventuellement mise à jour de plusieurs variables.

L'exclusion mutuelle est globale au programme (un seul *thread* dans une section critique).

On peut aussi donner un nom aux sections critiques. L'exclusion devient « un seul *thread* dans des sections critiques de même nom ».

[4] Une *section critique* est à la base un concept lié aux systèmes d'exploitation. C'est une zone de code qui ne peut être exécutée que par un seul process à la fois.

A l'exécution, des fonctions utilisables par openMP sont déclarées dans `<omp.h>`

`omp_set_num_threads(int n)` définit le nombre de *threads* d'une région parallèle[5].

`int omp_get_num_threads()` renvoie le nombre de *threads* d'une région parallèle.

`int omp_get_max_threads()` nombre maximum de *threads* possibles

`int omp_get_num_procs()` nombre de processeurs

`int omp_get_thread_num()` numéro du `thread` courant (numéroté de 0 (*master thread*) à `(omp_get_num_threads() -1)`)

Il existe également des fonctions pour créer des verrous, définir ou lire divers paramètres, mesurer des temps, etc.

L'appel à ces fonctions peut être protégé par un `#ifdef _OPENMP` pour avoir un programme qui reste fonctionnel dans le cas séquentiel.

[5] Définit par défaut ou dans la variable d'environment `OMP_NUM_THREADS`

```
#include <stdlib.h>
#include <omp.h>
int main() { // Calcul de Pi. On tire un point aléatoire dans le carré
             // (0,0)-(1,1) et la probabilité qu'il tombe à l'intérieur
             // du quart de cercle de rayon 1 est de Pi/4.
  unsigned short x[3]; // pour la fonction aléatoire erand48()
  int pas=1000; // nombre de tirages élatoires
  int points=0; // nombre de points dans le 1er quart de cercle
  double a,b, pi;
# pragma omp parallel private(x)
  {             // début de la zone parallèle
    x[0]=1; x[1]=1;
#   ifdef _OPENMP
    x[2]=omp_get_thread_num(); // initialisation différente par thread
#   else
    x[2]=1;     // cas séquentiel
#   endif
#   pragma omp for private(a,b) reduction(+:points)
    for (int i=0;i<pas;i++) {
      // Tirer 2 valeurs aléatoires a et b dans l'intervalle [0.0,1.0[
      a=erand48(x); // NB : erand48() peut être appelée par plusieurs
      b=erand48(x); //      threads (posix)
      if(a*a+b*b < 1.0) points ++;
    }
  }             // fin de la zone parallèle
  pi = 4.0 * points / pas;
}
```

```
  // mise en oeuvre d'une barrière de synchronisation
  // A l'arrivée, un thread incrémente un compteur
  // Puis il attends que ce compteur soit égal au nombre de threads total
#include <omp.h>
int * compte_barriere=0;
void barriere() {
  // suppose que l'on est dans une zone parallèle
  // la globale compte_barrière est mise à zero par le master au début
# pragma omp atomic update
  *compte_barriere ++;  // incrementer variable de barrière à l'arrivée
  int thread_total=omp_get_num_threads();
  while(1) {     // spinlock pour attendre les autres threads
#   pragma omp atomic read
    int threads_arrives=*compte_barriere;
    if (thread_total == threads_arrives) break ;
  }
  // Tout le monde est arrivé. On passe la barrière
  // Faire un flush pour assurer la consistence mémoire
# pragma omp flush
  return;
}
```

Tout n'est pas parallélisable (simplement).
Possibilités de dépendances dans une boucle difficiles à contourner.

```
for(int i=1; i<N; i++)
  A[i]=A[i]+A[i-1]; // Distribution de + sur un vecteur (scan)
```

La parallélisation n'est pas toujours intéressante et repose sur des primitives couteuses.
Création de *thread* $\approx$ qq 10μs (10^4–10^5 cycles).
Opération atomique $\approx 10^3$ cycles
Synchronisation de *threads* 10^3–10^4 cycles.
Dans certains cas, la parallélisation peut *ralentir* les traitements.

Attention! Toutes les fonctions de bibliothèques ne sont pas « *thread safe* ».
Certaines manipulent des données globales et peuvent conduire à un résultat incorrect en cas d'appel depuis plusieurs *threads*.

Conclusion

OpenMP permet de simplifier largement la programmation multicoeurs.
Standard en évolution régulière
Depuis openMP 4 :

- possibilité de programmer des *accélérateurs* (type GPU) et de gérer les transferts entre mémoire CPU et mémoire de l'accélérateur.
  Clause `target`.
- possibilité d'utiliser des fonctionnalités SIMD
- contrôle du placement des *threads* (*thread affinity*)
- *reductions* sur une fonction définie par l'utilisateur

Depuis OpenMP 5 :

- gestion des dépendances entre itérations/tâches (clause *depend*)
- `atomic compare` pour comparaisons/modifications atomiques
- réductions sur des tâches
- clause `auto` en C++
- clause `loop` pour boucles plus générales que `for`
- transformations de boucles `tiles`, `unroll`