Programmation parallèle sur mémoire partagée

Juvigny Xavier

December 2, 2020

Contents

1 Modèles de programmation sur ordinateurs parallèles à mémoire partagée

1.1 Notion de thread et mise en œuvre en C++ 2011

Pour comprendre ce qu'est un thread, il est nécessaire au préalable d'avoir une bonne notion de ce qu'est un processus en informatique.

En informatique, **un processus** est l'instance d'un programme informatique en cours d'exécution par un ou plusieurs threads d'un ordinateur. Concrètement, cela signifie qu'un processus permet l'exécution de diverses instructions par le microprocesseur, en fonction du programme en cours de fonctionnement. Un processus contient à la fois le code du programme mais également son activité. Il demande lors de sa création une importante ressource CPU. Les processus contiennent des informations sur les ressources du programme et l'état d'exécution du programme, à savoir :

- Les identificateurs du processus, du groupe du processus, de l'utilisateur, et du groupe de l'utilisateur;
- L'environnement;
- Le répertoire de travail;
- Les instructions du programme;
- les registres;
- le tas;
- la pile;
- les descripteurs de fichiers;
- les signaux d'action;
- les bibliothèques partagées;
- Les outils d'intercommunication entre processus : pipes, sémaphores, mémoire partagée,...

Le thread, ou *fil d'exécution* en français, est un processus et il permet d'exécuter des instructions de langage machine au sein du processeur. La spécificité du thread est qu'il laisse la possibilité à deux instances en train d'interpréter le même programme de s'exécuter en simultané au sein du même processeur. Il se singularise également par le fait qu'il partage la mémoire virtuelle d'un processus ainsi que diverses autres ressources. Par contre, chaque thread possède sa propre pile d'exécution.

Une définition d'un thread peut donc être la suivante :

Définition : Un thread est un exécutable léger, créé par un processus et dont il partage le même espace mémoire. Le thread ne maintient que les informations suivantes :

- Le pointeur de tas;
- Les registres;
- La propriété d'exécution (politique d'exécution ou priorité);
- Un ensemble de signaux bloqués ou en veilles;
- Les données spécifiques au thread.

En résumé, un thread (dans un environnement UNIX) :

- N'existe qu'au sein d'un processus et utilise les ressources du processus;
- Possède son propre flot d'instructions aussi longtemps que son processus parent existe et que le système d'exploitation le supporte;
- Ne duplique que les ressources nécessaires pour son exécution;
- Partage les ressources du processus père avec d'autres threads qui agissent également de façon indépendante;
- Meurt si le processus parent meurt;
- Est léger car la plupart des ressources ont déjà été créées lors de la création du processus père.

Du fait que les threads partagent les mêmes ressources que le processus père :

- Un changement fait par un thread dans les ressources communes du processus père (comme fermer un fichier par exemple) sera vu par tous les autres threads;
- Deux pointeurs sur des threads différents auront la même donnée si ils pointent sur la même adresse (virtuelle);
- Lire et écrire au même endroit mémoire par différents threads est possible mais

demande une synchronisation par le programmeur.

Avant C++ 2011, la création et la gestion des threads dépendaient fortement du système d'exploitation employé; chaque OS proposait sa propre API. Depuis la publication de la norme 2011 du C++, le langage propose une librairie standardisée permettant une même gestion des threads quelque soit le système d'exploitation sur lequel on compile les sources et exécute le programme. Cette bibliothèque standardisée se repose principalement sur la gestion des threads proposée par la norme posix d'unix (dont linux suit la norme).

La création d'un thread provoque son exécution. Il suffit simplement de donner lors de la création du thread la fonction qu'il doit exécuter en parallèle de l'exécution du processus. Ainsi, un programme Helloworld affichant simultanément un message du processus et d'un thread, s'écrira de la manière suivante :

```
#include <iostream>
#include <thread>

// Cette fonction sera appelée par un thread
 void call_from_thread() {
   std::cout << "Hello, World du thread" << std::flush << std::endl;
}

int main()
{
   // Exécution d'un thread
   std::thread t1(call_from_thread);

   std::cout << "Hello world du processus" << std::flush << std::endl;
   //Join the thread with the main thread
   t1.join();

    return 0;
}
```

La création du thread, au début de la fonction `main` appelle immédiatement après création du thread la fonction `call_from_thread` qui affichera un message pendant que le programme principal essaiera d'afficher un second message.

On peut mesurer la légèreté de création d'un thread en observant le tableau (1).

Si les threads ont la réputation d'être diaboliques de part la relative complexité qu'ils peuvent engendrer en programmation, c'est encore le meilleurs moyen de prendre avantage des cœurs multiples d'un ordinateur pour une application.

Par défaut, un thread est **attaché** à son processus, c'est à dire qu'il est possible de synchroniser le processus et le thread pour attendre que le thread ait fini son exécution afin de continuer l'exécution du programme par le processus. C'est la méthode `join` qui permet au processus d'attendre la fin de l'exécution d'un thread. Ainsi dans le programme précédent, le processus principal attend que le thread `t1` finisse d'exécuter sa fonction affichant un message avant de quitter le programme grâce à la ligne `t1.join();`.

Remarque 1 *Il faut toujours penser à synchroniser un processus avec chacun de ses threads afin de s'assurer que le thread termine bien son exécution. En effet, dans le cas contraire, si le programme principal se termine avant le thread, il détruit toutes ses variables et au final, le thread n'est plus capable d'accéder aux ressources que le processus possédait. On se retrouve alors devant un problème majeure (plantage du thread). C'est la raison pour laquelle on a besoin en général de se synchroniser avec la fin du thread pour s'assurer qu'il se termine bien avant le programme principal.*

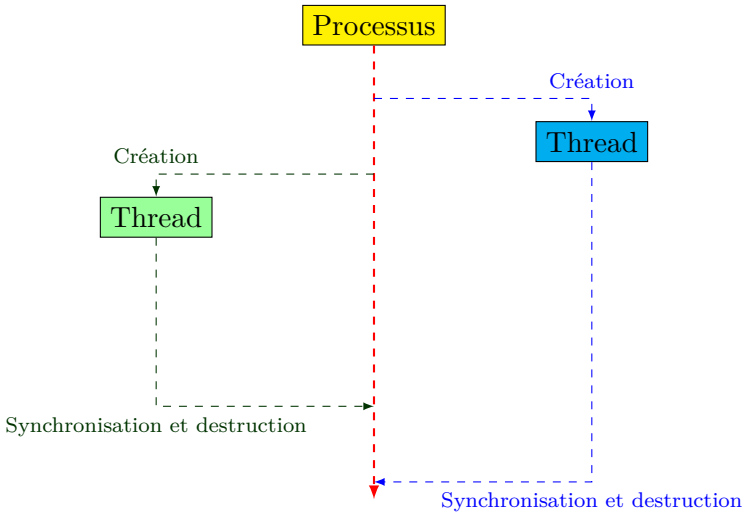

Figure 1: Exemple de création, synchronisation et terminaison de thread

Plateforme	fork()			std::thread		
	real	user	sys	real	user	sys
Intel 2.6Ghz Xeon E5-2670 (16 cœurs/nœuds)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8GHz Xeon 5660 (12 cœurs/nœuds)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.4GHz Opteron (8 cœurs/nœuds)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz Power 6 (8 cpus/nœuds)	9.5	0.6	8.8	1.6	0.1	0.4

Table 1: Comparaison création de 50 000 processus (fork()) contre création de 50 000 threads (std::thread)

Il est possible également de créer un thread `t2` à partir d'un thread `t1`. Là encore, une synchronisation du thread `t2` est nécessaire dans l'exécution du thread `t1` pour que `t1` attendent la fin de l'exécution de `t2` avant de finir lui-même son exécution, sous peine d'obtenir une levée d'exception de type `system_error`.

Cependant, il est possible dans ce cas de "détacher" le thread `t2` du thread `t1`, c'est à dire de faire en sorte que le thread `t2`, créé par `t1` continue à s'exécuter bien que `t1` ait été détruit après son exécution (le thread `t2` n'est pas détruit à la destruction de `t1`), cependant il est important de noter que le thread détaché s'arrête lorsqu'on quitte le programme principal comme cela est illustré et commenté dans le programme `example_detach_thread.cpp` donné dans les exemples.

Notons au passage qu'il est également facile (contrairement aux posix thread du C !) de passer des arguments à une fonction qui doit être exécutée par un thread :

```
void affichage_periodique(int id, int n)
{
  ...
}

int main()
{
    ...
    std::thread t1(affichage_periodique, 1,  5);
    ...
```

```
    t1.join();
    ...
```

Il suffit lors de la création du thread de faire suivre le nom de la fonction à exécuter par les arguments attendues par cette fonction. Attention cependant aux références, où il faut "repréciser" qu'on les passe en argument lors de l'appel de la fonction par le thread (cela est dû aux mécanismes sous-jacent du C++ et non aux threads proprement dit !). Voir par exemple l'extrait de code à la figure (2).

```
void compteur_periodique(const int& nb_max_sec, int& secondes)
{
  ...
}

int main()
{
    int max_sec = 20;
    int secondes;
    ...
    std::thread t(compteur_periodique, std::cref(nb_max_sec),
                  std::ref(secondes));
    ...
    t.join();
    ...
```

Figure 2: Extrait de l'exemple `ref_et_cref_pour_les_threads.cpp`

Une question importante qui vient avec la programmation multi-threads est : *Combien de threads est-il raisonnable de lancer simultanément ?*.

La réponse naïve serait de dire que le nombre de threads optimal est le nombre d'unités de calculs (cœurs) se trouvant sur la machine exécutant le code. Cependant, plusieurs autres paramètres devront être pris en compte pour espérer obtenir des accélérations significatives du code, à savoir :

1. La prise en compte de l'hyperthreading
2. Le goulot d'étranglement que représente l'accès à la mémoire vive

1.2 L'hyperthreading

Sur les CPUs modernes, les concepteurs ont mis deux fois plus de décodeurs d'instructions (ALU qui décode le langage machine décrivant le programme) que d'unités calculatoires (entiers ou réels qui exécute les instructions liées au langage machine). L'intérêt d'en mettre deux fois plus est de pouvoir ensuite exploiter sur une unité de calcul les différents circuits en parallèle : par exemple un thread pourra ainsi exécuter un calcul sur les entiers tandis qu'un deuxième thread, sur la même unité de calcul pourra exécuter un calcul sur des réels.

Dans les cas réels d'utilisation de l'hyperthreading, on peut espérer en moyenne un gain de trente pour cent sur le temps d'exécution du programme par rapport à une exécution en multithreading sans utilisation de l'hyperthreading (il y aura dans ce cas un thread par unité de calcul).

Cela rend l'analyse hardware d'un ordinateur un peu complexe à lire sous un système d'exploitation. Sous windows, en allant dans le gestionnaire de tâche (accessible via la combinaison de touche Ctrl + Alt + del), sous l'onglet performance, on voit en bas à droite que l'ordinateur possède (par exemple) :

- Un socket
- Quatre cœurs
- Huit processeurs logiques

De même, sous Linux, en tapant la commande

```
lscpu
```

on obtiendra, par exemple, la sortie suivante :

```
...
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
...
```

où

- `CPU(s)` correspond aux processeurs logiques de Windows, c'est à dire au nombre de décodeurs d'instruction
- `Core(s) per socket` correspond aux quatre cœurs de Windows, c'est à dire au nombre physique d'unité de calcul présent sur l'ordinateur
- `Socket(s)` correspond lui-même à une architecture NUMA où on a plusieurs processeurs (et leurs mémoires locales) reliés par une crossbar (voir le premier cours). Une seule socket (ce que vous devez logiquement tous avoir) correspond donc à un ordinateur sans architecture NUMA.

1.3 Communications efficaces pour l'échange des données

La motivation principale pour utiliser les threads dans un environnement parallèle haute performance est d'atteindre la performance maximale. En particulier, si une application utilise MPI pour communiquer point à point, il y a de forte chance que les performances seront améliorées en utilisant des threads à la place.

En effet, les bibliothèques MPI en mémoire partagée mettent en œuvre les communications point à point à l'aide de la mémoire partagée ce qui demande au moins une opération de copie mémoire (de processus à processus).

Pour les threads, il n'y a pas besoin de cette copie mémoire intermédiaire puisque les threads partagent le même espace mémoire que le processus père. Il n'y a donc pas de transfert mémoire.

Dans le pire des scénarii, les problèmes de communications par threads peuvent venir d'une atteinte à la limite de bande passante de la mémoire cache vers le CPU ou de la mémoire principale vers le CPU. Ces limitations sont bien moins contraignantes que celles imposées par les communications MPI en mémoire partagée.

Le tableau (2) montre sur différentes plateformes les différentes bandes passantes mesurées lors de l'échange de données pour MPI et les threads.

1.4 Concurrence d'accès aux données

Qui dit mémoire partagée entre les threads dit que plusieurs threads peuvent accéder en lecture ou en écriture à la même donnée !

Avec deux threads, il existe trois scénarii possibles :

Plateforme	Bande passante MPI en mémoire partagée	Plus mauvais cas de bande passante en thread de ram à CPU
Intel 2.6Ghz Xeon E5-2670	4.5	51.2
Intel 2.8GHz Xeon 5660	5.6	32
AMD 2.4GHz Opteron	1.2	5.3
IBM 4.0 GHz Power 6	4.1	16

Table 2: Comparaison bande passante MPI et bande passante par thread

- Les deux threads lisent la même donnée en même temps : pas de problème spécifique à cela ;
- Un thread lit une donnée qu'un autre thread modifie au même moment. Le résultat est aléatoire : soit le premier thread a eu la priorité sur le deuxième thread et a donc lu l'ancienne valeur ; soit le deuxième thread était prioritaire et dans ce cas, le premier thread a donc lu la nouvelle valeur. Par exemple, considérons le code suivant :

```
#include <thread>
#include <iostream>

void increment_counter(int& counter) {
    for ( int i = 0; i < 11; ++i )  {
        ++ counter;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void display_counter(const int& counter) {
    for (int i = 0; i < 10; ++i )  {
        std::cout <<  counter << " " << std::flush;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

int main()
{
    int counter = 0;
    std::thread t1(increment_counter, std::ref(counter));
    std::thread t2(display_counter, std::cref(counter));

    t2.detach();
    t1.join();

    return EXIT_SUCCESS;
}
```

Dans la majeure partie des cas, nous allons afficher

```
1 2 3 4 5 6 7 8 9 10
```

ce qui est ce qu'on voulait afficher avec ce programme, mais parfois (dans mes essais, c'était une fois sur dix environ), le programme retourne pour affichage :

```
1 2 3 3 4 5 6 7 8 9
```

Que s'est-il passé ? En fait, si les deux threads possèdent cent pour cent des ressources de leur unité de calcul respectif, l'affichage se déroule correctement, car le thread `t1` étant créé avant le thread `t2`, il incrémente le compteur juste avant l'affichage par le deuxième thread.
Mais parfois, il arrive que le premier thread prenne un peu de retard (ressource système ou autre lui prenant du temps de calcul), et dans ce cas, c'est le second thread qui se met à afficher le compteur avant qu'il soit incrémenté (d'où la répétition du trois dans le second affichage).
On voit donc que ce type de bogue peut être très dur à détecter, puisque dans l'exemple donné ci-dessus, le programme fonctionne la majorité du temps !

- Les deux threads écrivent en même temps dans la même zone mémoire : là encore, le résultat est aléatoire. Au mieux, la valeur mise à jour est une des valeurs écrites par l'un des threads, au pire, la valeur est incohérente et mène à un plantage de l'exécutable.
Considérons ainsi le programme suivant :

```cpp
#include <thread>
#include <iostream>

void incremente_counter(int& counter) {
    for (int i = 0; i < 5; ++i ) {
        counter = counter + 1;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}
void divide_by_two_if_even(int& counter)
{
    for (int i = 0; i < 5; ++i )  {
        if (counter %2 == 0) counter /= 2;
        else counter = counter + 1;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

int main()
{
    int counter = 0;
    std::thread t1(incremente_counter, std::ref(counter));
    std::thread t2(divide_by_two_if_even, std::ref(counter));

    t1.join(); t2.join();
    std::cout << "compteur final : " << counter << std::flush << std::endl;
    return EXIT_SUCCESS;
}
```

En exécutant plusieurs fois ce programme, on constate que l'affichage varie selon les exécutions :

```
> ./example_data_race_write_write.exe
compteur final : 4
> ./example_data_race_write_write.exe
compteur final : 8
> ./example_data_race_write_write.exe
compteur final : 7
...
```

On voit ici que l'accès à l'écriture sur la même variable tient de l'aléatoire puisqu'on

ne tombe jamais sur le même résultat !

Ces problèmes de détection des conflits mémoires lorsqu'on programme en multithreading sont des problèmes très difficiles à trouver dans un code conséquent pour plusieurs raisons :

- Le résultat est aléatoire à chaque exécution et le problème peut n'être détecté qu'à de rares exécutions. Par exemple il peut arriver à cause d'un conflit d'accès qu'un code marche neuf fois sur dix en moyenne;
- L'utilisation d'un déboggueur peut modifier la façon dont s'exécutent les threads, et il arrive souvent que le code marche avec un déboggueur alors qu'il se plante presque systématiquement sans le déboggueur;
- Les conflits peuvent occasionner des erreurs subtiles qui altèrent le résultat final sans pour autant faire planter l'exécutable.

Il existe cependant des outils permettant de détecter ces conflits d'accès (et les interblocages qu'on verra plus loin), dont certains sont gratuits, d'autres payant. Voici une liste non exhaustive de tels outils qui permettent de gagner un temps précieux pour la mise au point des codes :

- Intel inspector : Livré avec leur compilateur, payant et cher, mais efficace !
- Clang intègre un outil permettant de détecter les conflits d'accès, il suffit de rajouter l'option `-fsanitize=thread` pour analyser les éventuels conflits mémoires dûs au multithreading. Ainsi, sous Linux, on peut tester l'outil à l'aide des exemples donnés dans le répertoire `Example` du cours. Ainsi pour tester la détection sur le conflit lecture-écriture du premier exemple :

```
> clang++ -fsanitize=thread -g -std=c++11 -fPIC -O1 -march=native -Wall \
          -o example_data_race_read_write.exe example_data_race_read_write.cpp -lpthr

> ./example_data_race_read_write.exe
counter : LLVMSymbolizer: error reading file: No such file or directory
==================
WARNING: ThreadSanitizer: data race (pid=1732)
  Read of size 4 at 0x7ffd5244378c by thread T2:
    #0 display_counter(int const&) /home/juvigny/Home/Cours/Ensta/IN203_SUPPORT_COURS
    ...
    #6 <null> <null> (libstdc++.so.6+0xd6d83)

  Previous write of size 4 at 0x7ffd5244378c by thread T1:
    #0 increment_counter(int&) /home/juvigny/Home/Cours/Ensta/IN203_SUPPORT_COURS/Exa
    ...
    #6 <null> <null> (libstdc++.so.6+0xd6d83)

  Location is stack of main thread.

  Location is global '??' at 0x7ffd52424000 ([stack]+0x00000001f78c)

  Thread T2 (tid=1735, running) created by main thread at:
    #0 pthread_create <null> (example_data_race_read_write.exe+0x424b0b)
    ...

  Thread T1 (tid=1734, finished) created by main thread at:
    #0 pthread_create <null> (example_data_race_read_write.exe+0x424b0b)
```

```
    ...

SUMMARY: ThreadSanitizer: data race /home/juvigny/Home/Cours/Ensta/IN203_SUPPORT_COUR
==================
ThreadSanitizer: reported 1 warnings
```

et pour le deuxième exemple donnant un conflit écriture-écriture pour deux threads :

```
> ./example_data_race_write_write.exe
LLVMSymbolizer: error reading file: No such file or directory
==================
WARNING: ThreadSanitizer: data race (pid=1950)
  Read of size 4 at 0x7ffd4c14ec18 by thread T2:
    #0 divide_by_two_if_even(int&) ./example_data_race_write_write.cpp:16:13 ...
    ...
    #6 <null> <null> (libstdc++.so.6+0xd6d83)

  Previous write of size 4 at 0x7ffd4c14ec18 by thread T1:
    #0 incremente_counter(int&) example_data_race_write_write.cpp:8:17 ...
    ...
    #6 <null> <null> (libstdc++.so.6+0xd6d83)

  Location is stack of main thread.

  Location is global '??' at 0x7ffd4c130000 ([stack]+0x00000001ec18)

...

SUMMARY: ThreadSanitizer: data race example_data_race_write_write.cpp:16:13 in divide
==================
compteur final : 10
ThreadSanitizer: reported 1 warnings
```

L'outil marche donc plutôt bien sous Linux. Il est par contre dommage qu'il ne soit pas supporté par clang sous Windows...

- Sous Windows, il existe un utilitaire drace pour détecter les data races, mais difficile à installer (écher personnel !).

Lorsque l'algorithme exige que plusieurs threads écrivent dans la même zone mémoire (pour incrémenter un compteur par exemple), il faut utiliser un mécanisme qui va obliger les threads a n'exécuter que un par un la partie de l'algorithme où ils écrivent sur la même adresse mémoire. On dit alors que cette partie de l'algorithme est une partie **séquentielle** de l'algorithme.

Si cette partie d'algorithme est complexe, on protégera cette partie de l'algorithme par des **gardes** qui obligeront les threads à ne passer que un par un.

Si c'est une instruction élémentaire (une addition, soustraction, ...), on peut protéger uniquement cette instruction sans définir une zone, en la déclarant **atomique**.

Si la partie séquentielle de l'algorithme se réduit à une instruction simple, on peut bien sûr utiliser des gardes pour la protéger comme s'il s'agissait d'instructions complexes. Cependant, la mise en place de ces gardes est bien plus lourde que de simplement déclarer cette instruction atomique (on parle alors d'**atomicité**).

1.5 Notion d'affinité

Tout système d'exploitation moderne supporte l'affinité par thread. Une affinité signifie qu'au lieu de laisser un thread s'exécuter librement sur n'importe quelle unité de calcul, on demande au gestionnaire de tâche du système d'exploitation de n'exécuter un thread particulier que sur une unité de calcul ou un ensemble prédéfini d'unités de calcul.

Par défaut, un thread peut s'exécuter sur n'importe quel CPU logique (c'est à dire n'importe quelle unité de calcul), si bien que le système d'exploitation va exécuter un thread sur une unité de calcul selon des critères du gestionnaire de tâche. De plus, parfois, le système d'exploitation va migrer un thread d'une unité de calcul vers une autre, ce qui peut avoir un sens du point de vue du gestionnaire de tâche (bien qu'il évitera au maximum de le faire, du fait qu'on perd toutes les données qui étaient en cache dans le cœur d'où le thread provient).

Le programme suivant (ici dans sa version Linux) permet de suivre quatre threads dans une boucle infinie et afficher l'unité de calcul (le processeur logique) sur laquelle il s'exécute au fur et à mesure du temps (pour le code complet fonctionnant sur toutes les plateformes, voir le fichier `get_cpu_for_threads.cpp` dans le répertoire `Example`) :

```
int main(int argc, const char** argv) {
  constexpr unsigned num_threads = 4;
  // A mutex ensures orderly access to std::cout from multiple threads.
  std::mutex iomutex;
  std::vector<std::thread> threads(num_threads);
  for (unsigned i = 0; i < num_threads; ++i) {
    threads[i] = std::thread([&iomutex, i] {
      while (1) {
        {
          // Use a lexical scope and lock_guard to safely lock the mutex only
          // for the duration of std::cout usage.
          std::lock_guard<std::mutex> iolock(iomutex);
          std::cout << "Thread #" << i << ": on CPU " << sched_getcpu() << "\n";
        }

        // Simulate important work done by the tread by sleeping for a bit...
        std::this_thread::sleep_for(std::chrono::milliseconds(900));
      }
    });
  }

  for (auto& t : threads) {
    t.join();
  }
  return 0;
}
```

La connaissance du nœud sur lequel le thread s'exécute est obtenue grâce à la fonction système sched_getcpu (spécifique ici à linux ou les systèmes d'exploitation utilisant la glibc, sinon pour une fonction similaire sur les autres plateformes, voir le code source `get_cpu_for_threads.cpp`). Une exécution de ce programme pourra par exemple donner le résultat suivant :

```
$ ./get_cpu_for_threads
Thread #0: on CPU 5
Thread #1: on CPU 5
Thread #2: on CPU 2
Thread #3: on CPU 5
```

```
Thread #0: on CPU 2
Thread #1: on CPU 5
Thread #2: on CPU 3
Thread #3: on CPU 5
Thread #0: on CPU 3
Thread #2: on CPU 7
Thread #1: on CPU 5
Thread #3: on CPU 0
Thread #0: on CPU 3
Thread #2: on CPU 7
Thread #1: on CPU 5
Thread #3: on CPU 0
Thread #0: on CPU 3
Thread #2: on CPU 7
Thread #1: on CPU 5
Thread #3: on CPU 0
^C
```

Plusieurs observations : les threads sont quelquefois exécutés sur le même CPU (logique), parfois sur différents CPUs. On peut également observer un peu de migration de threads. Enfin, le gestionnaire de tâche arrive à placer les threads sur différentes unités de calcul et les garde sur ces unités. Différentes contraintes (dont la charge du système) peuvent bien sûr conduire à des résultats différents.

On peut restreindre l'exécution des threads sur des processeurs logiques spécifiques. Sous Linux, cela peut-être fait à l'aide de la commande taskset comme dans l'exemple ci–dessous où on restreint les threads à ne s'exécuter que sur les processeurs logiques cinq et six :

```
$ taskset -c 5,6 ./get_cpu_for_threads
Thread #0: on CPU 6
Thread #1: on CPU 5
Thread #3: on CPU 5
Thread #2: on CPU 6
Thread #0: on CPU 6
Thread #1: on CPU 5
Thread #3: on CPU 5
Thread #2: on CPU 5
Thread #0: on CPU 6
Thread #1: on CPU 5
Thread #3: on CPU 6
Thread #2: on CPU 5
Thread #0: on CPU 6
Thread #1: on CPU 5
Thread #3: on CPU 5
Thread #2: on CPU 5
Thread #0: on CPU 6
Thread #3: on CPU 6
Thread #1: on CPU 5
Thread #2: on CPU 5
Thread #1: on CPU 5
```

```
Thread #0: on CPU 6
Thread #3: on CPU 6
Thread #2: on CPU 6
...
```

Comme attendu, bien qu'on puisse encore observer des migrations de thread, les threads restent confinés aux cinquième et sixième CPU.

Il est bien sûr possible de restreindre chaque thread à un processeur logique particulier pour un programme donné à l'exécution. Cependant cela dépend fortement de l'API proposé par le système d'exploitation. On utilisera par exemple la commande `taskset` sous linux.

Il est bien sûr possible de restreindre chaque thread à un CPU dans un programme. Cependant, il n'existe pas de solutions communes à tous les systèmes d'exploitation, et la solution pour définir l'affinité de chaque thread dépendra donc du système d'exploitation sur lequel le programme sera mis en œuvre. Néanmoins, dans le répertoire `Examples`, vous trouverez une petite librairie `thread_extension.hpp` et `thread_extension.cpp` qui permettent sous Linux, Windows ou Mac de définir les processeurs logiques sur lesquels chaque thread s'exécutera. Un exemple de son utilisation est donné par le programme `exemple_set_affinity.cpp`.

1.6 Applications "memory bound" vs "cpu bound"

Nous avons vu dans le premier cours que la bande passante mémoire était un facteur limitant quant à la vitesse d'exécution d'un programme et que pour palier à la lenteur relative de la mémoire vive, on avait recours soit à de la mémoire vive interlacée, soit à de la mémoire cache, les deux pouvant bien sûr coexister sur une même machine.

Ce problème est d'autant plus crucial lorsqu'on programme en parallèle sur mémoire partagée, car la mémoire est bien plus sollicitée qu'avec un programme séquentiel.

Malheureusement, selon les problèmes traités et les algorithmes employés, il n'est pas toujours possible d'exploiter efficacement la mémoire cache ou la mémoire vive interlacée.

En ce qui concerne la mémoire cache : L'exploitation de la mémoire cache se base sur une exploitation locale en espace et en temps de variables qu'on pourra lire ou modifier:

- Locale en espace car lors du chargement d'une variable en mémoire cache, on charge de fait une ligne de cache contenant cette variable mais aussi les variables suivantes, contiguës en mémoire;
- Locale en temps car une variable ne restera pas très longtemps en mémoire cache du fait que lors du déroulement du programme, d'autres variables devront être aussi chargées en mémoire cache, et il est fort probable que la variable chargée préalablement sera déchargée de la mémoire cache.

Cela demande donc d'écrire des algorithmes pouvant exploiter plusieurs fois dans un bref délai les mêmes variables, à condition que votre algorithme le permette !

De fait, la mise en œuvre d'un algorithme pourra exploiter la mémoire cache si la complexité en accès mémoire (nombre d'accès **non redondants** à différentes variables) est inférieure à la complexité algorithmique, c'est à dire au nombre d'opérations effectuées.

En effet, si le nombre d'opérations est supérieur au nombre de données exploitées par l'algorithme, il est certain (on applique le principe des trous de pigeon qui dit que si il y a n trous et $p > n$ pigeons, alors il y a sûrement deux pigeons au moins dans un même trou) qu'on va exploiter plusieurs fois certaines variables. Il devient donc intéressant de tenter

de modifier notre algorithme pour exploiter localement dans le temps la même variable afin qu'elle soit après la première utilisation dans la mémoire cache.

En ce qui concerne la mémoire interlacée : L'exploitation efficace d'une mémoire interlacée se base sur une lecture contiguë des données en mémoire. Lorsque l'algorithme utilise une lecture aléatoire ou selon un tableau d'indice des données, ce type de mémoire ne se montrera pas plus efficace qu'une mémoire classique.

Définition 1 *On dira qu'une fonction est memory bound si sa vitesse d'exécution est limitée par la bande passante mémoire.*

Définition 2 *On dira qu'une fonction est cpu bound si sa vitesse d'exécution est limitée par la vitesse de traitement des instructions du ou des CPUs.*

Il est bien évident qu'un but majeur de l'optimisation d'un code est d'être cpu bound plutôt que memory bound !

Quelques exemples d'applications memory bound ou cpu bound

La suite itérative :

```
double u = 56547;
unsigned long iter = 0;
for ( unsigned int iter = 0; iter < 1023; ++iter ) {
  u = (u%2 == 0 ? u/2 : (3*u+1)/2);
}
```

Visiblement, cet algorithme a une complexité de 2 en accès mémoire et une complexité algorithmique bien supérieure ! Cet algorithme sera clairement adapté à une architecture mémoire contenant de la mémoire cache et exploite naturellement les variables localement en temps et espace. Le CPU pourra facilement calculer une itération de la suite à chaque cycle d'horloge.

Il en est de même pour une mémoire interlacée puisqu'il est fort probable que les deux seules variables utilisées seront stockées dans des registres du processeur !

Par contre, la récurrence empêche toute parallélisation de la boucle !

En conclusion :

Mémoire cache Fonction CPU-Bound. Boucle non parallélisable du fait de la récurrence.
Mémoire interlacée Fonction CPU-Bound. Boucle non parallélisable du fait de la récurrence.

Opération vectorielle

```
unsigned long N = 1'000'000;
std::vector<double> u(N),v(N),w(N);
...
for ( int i = 0; i < N; ++i ) {
  u[i] = std::max(v[i],w[i]);
...
}
```

La complexité algorithmique de cet algorithme est de N tandis que la complexité en accès mémoire est de $3N$. Il est donc inutile de chercher à optimiser cette fonction afin d'exploiter la mémoire cache.

Si l'accès mémoire ne sera pas vraiment pénalisant en utilisant un seul cœur de calcul (l'accès mémoire est encore "assez rapide" pour cela), il le sera si on cherche à paralléliser

cette boucle sur une machine à mémoire partagée avec mémoire cache, d'autant plus que le nombre de threads utilisés augmente !

Par contre, la lecture des données se fait de manière contiguë, et donc cet algorithme est naturellement adapté à une machine avec de la mémoire interlacée (comme un GPGPU par exemple). Il sera donc intéressant de paralléliser cette boucle pour ce type de machine (ce qui se fait "naturellement" puisqu'il n'y a aucune dépendance entre les opérations d'une itération à l'autre) pour gagner en performance !

En conclusion :

Mémoire cache Memory-bound, impossible à exploiter le cache. Inutile de paralléliser le code, l'accès mémoire empêchera tout gain substantiel en temps de calcul.

Mémoire entrelacée CPU-Bound. Oui, il est intéressant de paralléliser si la boucle est suffisamment longue, on gagnera vraiment en performance !

Opération matricielle

Considérons le produit matrice-matrice suivant :

```
std::vector<double> A(N*N);
std::vector<double> B(N*N);
std::vector<double> C(N*N);
...
for ( int i = 0; i < N; ++i )
  for ( int j = 0; j < N; ++j )
    for ( int k = 0; k < N; ++k )
      C[i+j*N] += A[i+k*N]*B[k+j*N];
```

La complexité algorithmique du produit matrice-matrice est environ de $2n^3$ tandis que la complexité en accès mémoire est de $3N^2$. La complexité algorithmique étant supérieure à la complexité en accès mémoire, il est donc possible d'exploiter la mémoire cache dans cet algorithme.

Par contre, tel que le code est écrit, cet algorithme n'exploite que très partiellement la mémoire cache, puisque seule la variable `C[i+j*N]` est réutilisée dans la boucle la plus interne. De plus, des sauts mémoires sont effectués pour accéder aux cœfficients de `A` (on saute `N` valeurs en mémoire à chaque itération sur la boucle en `k`), ce qui pour certaines valeurs de N peut s'avérer très pénalisant, autant pour une architecture à mémoire interlacée qu'avec une mémoire cache !

En effet, il faut savoir que la mémoire cache, du moins sur les processeurs de type Intel (AMD y compris), la mémoire cache est dite "associative". Qu'est ce que cela signifie-t-il donc ? En fait, le terme associatif désigne la stratégie adoptée pour stocker les variables provenant de la mémoire principale dans une mémoire RAM. La mémoire cache étant bien plus petite que la mémoire vive (RAM), cette stratégie dite "associative" décide de ranger les variables issues de la RAM selon leurs adresses physiques en RAM par le calcul d'un simple modulo.

Les ingénieurs ayant conçu cette stratégie ont bien sûr pensé au cas où des sauts mémoires en RAM faisaient que le hardware essaierai de ranger plusieurs variables à la même adresse dans la mémoire cache. D'une façon très limitée, ils ont donc décidé qu'on pourrait stocker une, deux, quatre ou huit variables (de soixante quatre bits), selon la mémoire cache utilisée par le processeur (on parle alors respectivement de mémoire cache 1-way, 2-ways, 4-ways, 8-ways).

Ainsi, tant qu'on peut stocker une variable supplémentaire à une adresse spécifique du cache, il n'y a pas de soucis, mais lorsque cherche à stoker une nouvelle variable à cette adresse alors qu'il n'y a plus de place pour stocker cette variable, le processeur enlèvera la

variable la plus ancienne qu'il ira réécrire dans la mémoire vive avant de stocker la nouvelle variable.

Pour connaître grosso-modo l'adresse en cache où sera stockée une variable, on peut considérer la formule suivante (la taille étant exprimée en octet) :

$$\text{Adresse mémoire cache} = \text{Adresse mémoire vive} \mod \frac{\text{Taille mémoire cache}}{8.\text{Nombre de ways}}$$

En considérant qu'une mémoire cache L2 a une capacité en général multiple de 1024^2 octets, on voit qu'en prenant une dimension pour la matrice de 1024, nous allons faire pour `A` des sauts de $8 \times 1024 = 8192$ octets, nombre divisible par 1024^2, et que nous allons retomber souvent sur la même adresse cache dans notre boucle interne, ce qui provoquera alors beaucoup de transfert entre la mémoire cache et la mémoire vive, ralentissant drastiquement le calcul (ce qu'on pourra constater en TP).

Heureusement, on peut remarquer que l'ordre des trois boucles imbriquées n'a aucune importance pour le calcul, et qu'on peut permuter l'ordre des boucles sans modifier le résultat !

En permutant l'ordre des boucles de sorte que la boucle en `i` soit la plus interne, on voit qu'on effectuera deux accès linéaires (pour `A` et `C`) ce qui permet d'exploiter les lignes de cache ou profiter de la mémoire interlacée, et qu'on réutilise le même `B` dans la boucle la plus interne.

Ainsi, on modifie le code tel que :

```
std::vector<double> A(N*N);
std::vector<double> B(N*N);
std::vector<double> C(N*N);
...
for ( int k = 0; k < N; ++k )
  for ( int j = 0; j < N; ++j )
    for ( int i = 0; i < N; ++i )
      C[i+j*N] += A[i+k*N]*B[k+j*N];
```

Avec ce nouveau code, on pourra constater une accélération substantielle pour certaines dimensions sur une machine avec mémoire cache et une accélération également avec une machine à mémoire interlacée.

Si le code s'exécute bien plus rapidement, il reste néanmoins du travail à effectuer sur une machine possédant de la mémoire cache, car la lecture linéaire des cœfficients de `A` et `C` nous empêche d'exploiter vraiment la mémoire cache.

La solution pour exploiter la redondance des données pour la mémoire cache est ici de faire le produit matrice–matrice à l'aide d'une approche par bloc :

```
std::vector<double> A(N*N);
std::vector<double> B(N*N);
std::vector<double> C(N*N);
...
const int szBloc = 127;
// On saucissone les boucles en i et j en plusieurs morceaux
// de taille szBloc :
for ( int kb = 0; kb < N; kb += szBloc )
  for ( int jb = 0; jb < N; jb += szBloc )
    for ( int ib = 0; ib < N; ib += szBloc )
      for ( int k = kb; k < kb+szBloc; ++k )
        for ( int j = jb; j < jb+szBloc; ++j )
```

```
            for ( int i = ib; i < ib+szBloc; ++i )
              C[i+j*N] += A[i+k*N]*B[k+j*N];
```

Dans le code modifié par bloc, on réexploite bien dans un court intervalle une partie des données de `C` et de `A`. De plus, le nombre de cœfficients de C et de A stockés pour chaque boucle en `i` et `j` est suffisamment petit pour rester en mémoire cache, si bien que la boucle en `k` pourra au fur et à mesure des itérations réutiliser les données de `C` stockées en cache. De même, la boucle en `j` pourra réutiliser les données de `A` stockées en cache lors du premier passage dans la boucle en `i`.

Pour la parallélisation du produit matrice-matrice, il faut bien observer qu'on ne pourra pas paralléliser la boucle en `kb`. En effet, on cumule une somme sur une variable dont l'adresse mémoire ne dépend que de `i` et `j` (et donc de `ib` et `jb`). Paralléliser sur la boucle `kb` reviendrait donc à ce que plusieurs threads chercheraient en même temps à écrire sur la même adresse mémoire, ce qui nous engendrera un conflit mémoire et un résultat faux et aléatoire au final !

La parallélisation ne pourra donc se faire que par la boucle en `jb`. L'ordre des boucles sur les blocs étant elle-même indépendante, on peut donc très bien intervertir la boucle en `jb` et `kb` pour paralléliser la boucle la plus externe.

Ainsi, on cherchera donc à paralléliser plutôt le code suivant :

```
std::vector<double> A(N*N);
std::vector<double> B(N*N);
std::vector<double> C(N*N);
...
const int szBloc = 127;
// On saucissone les boucles en i,j et k en plusieurs morceaux
// de taille szBloc :
for ( int jb = 0; jb < N; jb += szBloc )
  for ( int kb = 0; kb < N; kb += szBloc )
    for ( int ib = 0; ib < N; ib += szBloc )
      for ( int k = kb; k < kb+szBloc; ++k )
        for ( int j = jb; j < jb+szBloc; ++j )
          for ( int i = ib; i < ib+szBloc; ++i )
            C[i+j*N] += A[i+k*N]*B[k+j*N];
```

On verra en TD comment paralléliser "facilement" ce code à l'aide des directives OPENMP (voir plus loin dans ce cours pour les directives OPENMP).

En ce qui concerne la mémoire entrelacée, en plus de la modification de l'ordre des boucles en mettant la boucle en i comme boucle la plus interne (voir la deuxième version du produit), il est possible d'optimiser l'accès aux données de B. En effet, supposons dans un premier temps que pour une mémoire entrelacée à quatre voix, on ait un produit avec une dimension $N = 4*d+1$ (où $d > 0$ est un entier). Dans ce cas, la valeur B[k+j*N] et la valeur B[k+(j+1)*N] se trouveront sur un banc mémoire différent (en effet, le nombre de valeurs entre B[k+j*N] et B[k+(j+1)*N] est de $N = 4*d+1$ si bien que si B[k+j*N] est sur le banc i, B[k+(j+1)*N] sera sur le banc $i+1$). On aura dans ce cas un accès optimal aux valeurs de B dans le produit matrice–matrice.

En revanche, si $N = 4*d$, alors la valeur B[k+j*N] et la valeur B[k+(j+1)*N] seront sur le même banc mémoire et l'accès à la mémoire entrelacée sera sous–optimal.

L'astuce dans ce cas pour s'assurer un accès optimal à la mémoire entrelacée est de rajouter zéro à trois éléments (selon la valeur de N) tous les N éléments de sorte que d'accéder à la $(j+1)^{\text{ème}}$ colonne à partir de la $j^{\text{ème}}$ colonne permet de changer de banc mémoire.

Le code devriendra donc :

```
// nb_ways est le nombre de voies de la mémoire interlacée
const int NbRows = N+ (nb_ways - 1 - N%nb_ways);
std::vector<double> A(N*NbRows);
std::vector<double> B(N*NbRows);
std::vector<double> C(N*NbRows);
...
// Remarque : les matrices sont toujours de dimension N,
//            même si on a rajouté des lignes aux tableaux !
for ( int k = 0; k < N; ++k )
  for ( int j = 0; j < N; ++j )
    for ( int i = 0; i < N; ++i )
      C[i+j*NbRows] += A[i+k*NbRows]*B[k+j*NbRows];
```

2 Pour aller un peu plus loin avec les Posix threads (C++ 2011)

2.1 Exécution d'un thread en C++ 2011 (compléments)

C++ 2011 a apporté beaucoup de nouveautés au langage C++. En particulier, C++ 2011 a introduit en C++ la notion de lambda fonctions (déjà présentes dans d'autres langages comme Python).

Il est parfaitement possible d'exécuter une lambda fonction par un thread. Ainsi, pour reprendre le programme `Helloworld_multitreade.cpp`, on peut le modifier pour appeler une lambda fonction au lieu d'une fonction, ce qui simplifie et clarifie (en partie) le code :

```
#include <iostream>
#include <thread>

int main() {
  // Exécution d'un thread
  std::thread t1([](){std::cout << "Hello world" << std::endl; });

  //Join the thread with the main thread
  t1.join();

  return 0;
}
```

En général, on souhaite lancer plus qu'un thread et faire plusieurs tâches en parallèle. Pour faire cela, il faut créer un tableau de thread pour pouvoir les gérer. Dans l'exemple suivant, la fonction principale crée un groupe de dix threads qui feront plusieurs tâches puis elle attendra que les dix threads aient fini :

```
static const int num_threads = 10;

int main() {
  std::thread t[num_threads];

  // Exécution d'un groupe de threads
  //Launch a group of threads
  for (int i = 0; i < num_threads; ++i) {
    t[i] = std::thread([](int i){ std::cout << "Hello from thread "
                                            << i << std::endl; }, i);
  }
```

```
  std::cout << "Launched from the main\n";

  //Join the threads with the main thread
  for (int i = 0; i < num_threads; ++i) {
    t[i].join();
  }

  return 0;
}
```

Souvenez vous que le programme principal est lui-même un thread, nommé le thread principal, si bien que le code ci–dessus exécute onze threads en tout. Cela nous permet de faire d'autres tâches dans la fonction principale après que l'on ait lancé les dix threads et avant de se synchroniser avec eux.

2.2 Lancer un thread par processeur logique

Le C++ 2011 nous fournit une fonction utilitaire permettant de savoir combien d'unités de calcul logiques sont possédées par la machine sur laquelle s'exécute le programme, afin que nous puissions ensuite calibrer notre stratégie de parallélisation. La fonction est appelée hardware_concurrency et l'exemple ci–dessous montre un exemple qui l'utilise pour exécuter un nombre approprié de threads :

```
int main()
{
    // num_cpus contiendra le nombre de processeurs logiques !
    auto num_cpus = std::thread::hardware_concurrency();
    std::vector<std::thread> threads;
    threads.reserve(num_cpus);

    for ( decltype(num_cpus) i = 0; i < num_cpus; ++i )
        threads.emplace_back([](int id){std::cout << "Hello from "
                                                  << id << " !" << std::endl; },i);

    std::cout << "Hello from main !" << std::endl;

    for ( auto& t : threads )
        t.join();

    return EXIT_SUCCESS;
}
```

2.3 Section séquentielle

On a vu dans les généralités qu'il est possible d'avoir une partie d'un code qui ne peut être exécutée en parallèle et où on doit s'assurer que les threads ne peuvent passer que un par un.

2.3.1 Les mutex (Mutual Exclusion)

On utilise dans ce cas une instance de la classe mutex qui est une primitive de synchronisation qui peut être utilisée pour protéger des données partagées simultanément par plusieurs threads.

Le mutex propose une sémantique de propriété exclusif et non-récursif :

- Un thread appelant possède un mutex quand il réussit l'appel à lock ou try_lock et cela jusqu'à ce qu'il appelle unlock.
- Quand un thread possède un mutex, tous les autres threads bloqueront (pour les appels à lock) ou recevront une valeur de retour false (pour try_lock) s'ils tentent de revendiquer la propriété du mutex.
- Un thread appelant ne doit pas posséder un mutex avant d'appeler lock ou try_lock.

Le comportement d'un programme n'est pas défini si un mutex est détruit alors qu'il est toujours détenu par un autre thread. La classe mutex n'est ni copiable ni déplaçable.

Cet exemple montre comment un mutex peut être utilisé pour protéger un dictionnaire lorsqu'on lui rajoute des éléments :

```
std::map<std::string, std::string> g_pages;
std::mutex g_pages_mutex;

// Fonction censée sauver une page html accessible par une adresse internet.
void save_page(const std::string &url)
{
    // On simule un temps long pour l'obtention d'une page internet
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::string result = "fake content";

    // On a rajouté un mutex pour empêcher les threads de rajouter en même
    // temps un élément au dictionnaire.
    g_pages_mutex.lock();
    g_pages[url] = result;
    g_pages_mutex.unlock();
}

int main()
{
    std::thread t1(save_page, "http://foo");
    std::thread t2(save_page, "http://bar");
    t1.join();
    t2.join();

    // Pas besoin de protéger ici, puisque les threads ont terminé
    // leur travail.
    for (const auto &pair : g_pages) {
        std::cout << pair.first << " => " << pair.second << '\n';
    }
}
```

Résultat :

```
http://bar => fake content
http://foo => fake content
```

On peut vérifier avec clang (mais pas sous Windows directement) qu'il n'y a pas de conflit avec ce code, mais qu'en enlevant les lignes `g_pages_mutex.lock();` et `g_pages_mutex.unlock();`, clang détecte bien des conflits !

Si les mutex permettent de résoudre les problèmes de conflits mémoires éventuels, ils coûtent chers en ressource CPU et peuvent amener à des inter-blocages. Considérons le programme suivant :

```
#include <thread>
```

```cpp
#include <mutex>
#include <iostream>

std::mutex lock_value1;
std::mutex lock_value2;

int value1, value2;

void increment_value1_and_decrement_value2() {
    std::cout << "Start increment value 1 decrement value 2"
              << std::flush << std::endl;
    for ( int i = 0; i < 100; ++i ) {
        lock_value2.lock();
        lock_value1.lock();
        value1 ++; value2 --;
        lock_value1.unlock();
        lock_value2.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    std::cout << "End increment value 1 decrement value 2"
              << std::flush << std::endl;
}

void increment_value2() {
    std::cout << "Start increment value 2"
              << std::flush << std::endl;
    for ( int i = 0; i < 50; ++i ) {
        lock_value2.lock();
        value2 ++;
        lock_value2.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
    }
    std::cout << "End increment value 2" << std::flush << std::endl;
}

void increment_both_values() {
    std::cout << "Start increment values 1 and 2"
              << std::flush << std::endl;
    for ( int i = 0; i < 100; ++i ) {
        lock_value1.lock();
        lock_value2.lock();
        value1 ++; value2 ++;
        lock_value2.unlock();
        lock_value1.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    std::cout << "End increment values 1 and 2"
              << std::flush << std::endl;
}

int main() {
    std::thread t1(increment_value1_and_decrement_value2);
    std::thread t2(increment_value2);
    std::thread t3(increment_both_values);

    t1.detach(); t2.detach(); t3.detach();
    std::cout << "Processus principal : j'ai detache mes trois processus"
              << std::flush << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(10));
    std::cout << "J'ai attendu dix secondes. Bye bye !" << std::flush << std::endl;
    return EXIT_SUCCESS;
```

```
}
```

Si vous le compilez et l'exécutez plusieurs fois dans le répertoire `Examples`, vous verrez que parfois les threads se terminent (en affichant "End increment..."), parfois ne se terminent pas et sont tués lorsque le programme principal se termine au bout de dix secondes. Dans le programme donné dans `Examples`, on affiche en plus la valeur incrémentée ou décrémentée de `value1` et `value2`. On constate que lorsque les threads semblent se geler à des itérations différentes à chaque exécution. L'explication est que nous sommes selon l'ordre d'exécution de chacun des threads dans une situation d'interblocage ou non.

En effet, imaginons le scénario suivant :

- Le processus `t1` verrouille `lock_value1`;
- Pendant ce temps, le processus `t2` verrouille `lock_value2`;
- Quand le processus `t2` déverrouille `lock_value2`, c'est `t3` qui verrouille `lock_value2`
- `t1` attend que `t3` déverrouille `lock_value2`;
- `t3` attend que `t1` déverrouille `lock_value1`;
- `t2` attend que `t3` déverrouille `lock_value2`;

Les threads `t1` et `t3` présentent bien une situation d'interblocage qui au passage va aussi bloquer `t2` !

Par contre, dans ce second scénario :

- Le processus `t1` verrouille `lock_value1`;
- Pendant ce temps, le processus `t2` verrouille `lock_value2`;
- Quand le processus `t2` déverrouille `lock_value2`, `t1` verrouille `lock_value2`
- `t1` déverrouille `lock_value2`
- `t3` verrouille `lock_value2`
- `t1` déverrouille `lock_value1`
- `t3` verrouille `lock_value1`
- `t3` déverrouille `lock_value1`
- `t3` déverrouille `lock_value2`

Il n'y a pas d'interblocage et chacun des processus peut passer à l'itération suivante !

En multithreading, l'interblocage n'est pas systématique (comme cela peut arriver en MPI) et présente toujours une difficulté pour déceler ces interblocages qui peuvent arriver rarement dans l'exécution d'un code.

En C++ 2011, il existe néanmoins des solutions pour éviter de tels interblocages (ici dû au fait qu'on ne bloque pas dans le même ordre les mutex, mais ce n'est pas toujours possible de les bloquer dans l'ordre)

Par exemple, pour notre cas, ci-dessus, il suffit d'appeler une fonction du C++ 11 qui adopte une autre stratégie pour le verrouillage des mutex et qui permet d'éviter l'interblocage :

```
#include <thread>
#include <mutex>
#include <iostream>

std::mutex lock_value1;
std::mutex lock_value2;

int value1, value2;

void increment_value1_and_decrement_value2() {
    std::cout << "Start increment value 1 decrement value 2"
              << std::flush << std::endl;
    for ( int i = 0; i < 100; ++i ) {
```

```
        std::lock(lock_value2, lock_value1);
        value1 ++; value2 --;
        lock_value1.unlock();
        lock_value2.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    std::cout << "End increment value 1 decrement value 2"
              << std::flush << std::endl;
}

void increment_value2() {
    std::cout << "Start increment value 2"
              << std::flush << std::endl;
    for ( int i = 0; i < 50; ++i ) {
        lock_value2.lock();
        value2 ++;
        lock_value2.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(20));
    }
    std::cout << "End increment value 2"
              << std::flush << std::endl;
}

void increment_both_values() {
    std::cout << "Start increment values 1 and 2" << std::flush << std::endl;
    for ( int i = 0; i < 100; ++i ) {
        std::lock(lock_value1, lock_value2);
        value1 ++; value2 ++;
        lock_value2.unlock();
        lock_value1.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    std::cout << "End increment values 1 and 2" << std::flush << std::endl;
}

int main() {
    std::thread t1(increment_value1_and_decrement_value2);
    std::thread t2(increment_value2);
    std::thread t3(increment_both_values);

    t1.detach(); t2.detach(); t3.detach();
    std::cout << "Processus principal : j'ai detache mes trois processus" << std::flush << std:
    std::this_thread::sleep_for(std::chrono::seconds(10));
    std::cout << "J'ai attendu dix secondes. Bye bye !" << std::flush << std::endl;
    return EXIT_SUCCESS;
}
```

Si vous exécutez le code `exemple_interblocage_correction.exe`, vous constaterez que les interblocages aléatoires ont disparus.

La gestion propre et rigoureuse des mutex est un sujet en soi. Il faut savoir que C++ 2011 (et 2017 qui en rajoute d'autres) propose une vaste panoplie de stratégies pour les mutex (en particulier pour les fonctions récursives), et que seul ce cours ne suffirait pas à les répertorier.

2.3.2 Les atomics

Si la donnée partagée par les threads est une donnée de type primitif (entier, réel, char, ...) et si l'opération effectuée est basique (incrément, addition, soustraction, etc.), on peut, plutôt que d'utiliser des mutex, déclarer cette variable atomique. On sera ainsi garanti

qu'il n'y aura pas de conflit lors de l'exécution en parallèle des threads.

Imaginons par exemple que nous avons un compteur que l'on doit incrémenter de un lorsque chaque thread finit une ligne de l'ensemble de mandelbrot et doit calculer la prochaine ligne de l'image encore non calculée :

```
# include <atomic>
// Calcul dans l'espace image la ième ligne de l'ensemble de mandelbrot :
void compLineMandelbrot( int i, std::vector<unsigned>& img  )
{
...
}

// Version multi-threadé de mandelbrot :
void compMandelbrot( int W, int H )
{
  std::atomic<int> num_line = -1;
  std::vector<unsigned> img(W*H);
  unsigned num_cpus = std::thread::hardware_concurrency();
  std::vector<std::thread> threads;
  for ( int c = 0; c < num_cpus-1; ++c ) {
    threads.push_back(std::thread([H,num_line,img] ()
          { while (num_line<H) {
              num_line ++;
              compLineMandelbrot( num_line, img );
            } } ) );
  }
  while ( num_line < H ) {
    num_line ++;
    compLineMandelbrot( num_line, img );
  }
  for ( auto& t : threads ) t.join();
}
```

Comme expliqué précédemment, l'atomicité est un procédé bien plus léger qu'un mutex et doit être utilisé si possible. De plus, il est impossible d'avoir un deadlock à l'aide d'opérations atomiques.

Un autre exemple peut-être trouvé dans le répertoire `Examples`. Il affiche à la fin toujours la même valeur et clang montre bien qu'il n'y a pas de conflit mémoire pouvant donner des valeurs aléatoires.

3 OpenMP

3.1 Principe d'OpenMP

OpenMP est un outil intégré dans la plupart des compilateurs actuels (gnu c++, MS VC++, Intel, etc.) en C, C++ et en Fortran. Son principe est de pouvoir exécuter des threads en définissant des régions parallèles à l'aide de directives de compilation (#pragma en C et C++).

Ces régions parallèles permettant aux threads soit d'exécuter la même section de code sur des données différentes soit des sections de code différentes. On peut également y définir les variables qui seront partagées entre les threads et celles qui seront privées, c'est à dire locales à chaque thread.

Enfin, on peut alterner entre des régions du code parallèles et des régions séquentielles.

Les directives de compilation liées à OpenMP commencent toutes par #pragma omp ....

3.2 Région parallèle

Les régions parallèles sous OpenMP peuvent être écrites sous diverses formes :

- Une boucle parallèle : On découpe la boucle en plusieurs parties de façon statique ou dynamique (au choix du programmeur);
- Exécuter plusieurs parties du code en parallèle, une partie par tâche;
- Exécuter la même section du code sur plusieurs tâches.

Il faut parfois synchroniser les tâches entre elles : par exemple, pour faire des opérations de réduction.

3.3 Production et exécution d'un programme OpenMP

Le compilateur fournit des options de compilation (par exemple pour gnu c/c++ : `-fopenmp`) qui permettent de prendre en compte les directives de compilation. Attention : si cette option n'est pas donnée, le compilateur ignorera les directives OpenMP données par le programme.

Lorsque l'option de compilation est mise, le compilateur intégrera automatiquement la bibliothèque OpenMP contenant les utilitaires pour OpenMP.

On peut contrôler dynamiquement le nombre de threads générés par l'application à l'aide de la variable d'environnement `OMP_NUM_THREADS` mais aussi à l'aide de fonctions fournies par la bibliothèque d'OpenMP.

Les prototypes des fonctions associées avec OpenMP sont définis dans le fichier d'entête omp.h.

3.4 Construire une région parallèle

Par défaut, les variables définies à l'extérieur de la section parallèle sont partagées et tous threads exécutent le même code.

Une barrière implicite de synchronisation se fait automatiquement par défaut à la fin de la section parallèle.

Il est interdit d'exécuter des instructions de saut dans une région parallèle (goto, break, etc.)

Exemple de région parallèle :

```
#include <iostream>
#include <iomanip>
#include <omp.h>

int main()
{
    int p;
    // Teste si on est dans une région parallèle
    p = omp_in_parallel();
    std::cout << "Dans une region parallele ? " << std::boolalpha << (p!=0) << std::flush << st
#pragma omp parallel
{
    // Teste si on est dans une région parallèle
    p = omp_in_parallel();
    std::cout << "Dans une region parallele ? " << std::boolalpha << (p!=0) << std::flush << st
}
    return EXIT_SUCCESS;
}
```

La sortie de ce programme donnera :

```
$ export OMP_NUM_THREADS=2; ./exemple_omp_parallel_region.exe
Dans une region parallele ? false
Dans une region parallele ? true
Dans une region parallele ? true
```

ou encore

```
$ export OMP_NUM_THREADS=4; ./exemple_omp_parallel_region.exe
Dans une region parallele ? false
Dans une region parallele ? true
Dans une region parallele ? true
Dans une region parallele ? true
Dans une region parallele ? true
```

(En fait, avec quatre processus, les messages ont tendance à s'entremêler, et n'est pas aussi clair que cela !)

Remarquons que le début et la fin de la région parallèle sont définis par la déclaration d'un début et fin de blocs d'instruction.

Enfin, toute fonction appelée dans une région parallèle sera elle-même considérée comme étant dans une région parallèle.

3.5 Variables privées, variables partagées

Commençons par donner la définition d'une variable privée et d'une variable partagée :

Définition 3 *Une variable privée est une variable gérée localement par un seul thread.*

Et a contrario

Définition 4 *Une variable partagée est une variable vue et gérée par l'ensemble des threads.*

Par défaut, une variable déclarée **hors d'une région parallèle** est partagée. Une variable déclarée **dans une région parallèle** est privée.

Par exemple :

```
#include <iostream>
#include <omp.h>

int main()
{
    int a = 314;
#   pragma omp parallel
    {
        // Lis un numéro unique pour le thread
        int id = omp_get_thread_num();
        std::cout << "Le thread n." << id << " peut lire la variable a = " << a
                  << std::flush << std::endl;
        a = 413;
    }
    std::cout << "A la sortie de la région parallele , a vaut " << a << std::flush << std::endl;
    return EXIT_SUCCESS;
}
```

Une sortie possible de ce code :

```
Le thread n.0 peut lire la variable a = 314
Le thread n.1 peut lire la variable a = 413
A la sortie de la région parallele, a vaut 413
```

Dans ce code, a est une variable qui sera donc partagée par tous les threads dans la zone parallèle tandis que `id` sera une variable privée, c'est à dire que chaque thread contiendra sa propre variable `id` (qui pourra donc être différente selon les threads).

Remarquons que ce code contient un conflit mémoire (data race) à l'image de ceux vu pour les pthreads, puisque `a` est modifié par les threads se trouvant dans la région parallèle (ce qui est d'ailleurs détecté par clang si on lui rajoute l'option `-fsanitize=thread`).

On peut modifier le statut d'une variable déclarée hors d'une région parallèle pour être soit partagée soit privée à l'aide d'une clause *private* ou *shared*.

Remarque : Attention, si une variable déclarée hors de la zone parallèle est déclarée comme privée lorsqu'on rentre dans la zone, on créera une nouvelle variable pour chaque thread qui seront ensuite perdues à la sortie de la zone parallèle

Ainsi, le code suivant :

```
#include <iostream>
#include <omp.h>

int main()
{
    int a = 314;
#   pragma omp parallel private(a)
    {
        // Lis un numéro unique pour le thread
        int id = omp_get_thread_num();
        std::cout << "Le thread n." << id << " peut lire la variable a = " << a
                  << std::flush << std::endl;
        a = 413;
    }
    std::cout << "A la sortie de la région parallele , a vaut " << a << std::flush << std::endl;
    return EXIT_SUCCESS;
}
```

très semblable au premier code déclare `a` comme privé.

On a pour sortie :

```
Le thread n.0 peut lire la variable a = 0
Le thread n.1 peut lire la variable a = 0
A la sortie de la région parallele, a vaut 314
```

Déclarer `a` en privée a donc pour conséquence que :

- Le compilateur vous prévient que la variable `a` dans la zone parallèle n'est pas initialisée lors de l'affichage;
- La valeur de a qu'affiche chaque thread sera différente de celle donnée dans le processus principal
- La valeur affichée en sortie de la zone parallèle est celle donnée avant l'entrée dans la zone parallèle, ne prenant pas en compte la modification du `a` fait dans la région parallèle (ce qui est normal, puisque le `a` dans la région parallèle n'est pas la même variable que celle dans le processus).

Si on désire qu'une variable soit privée mais prenne la valeur qu'elle avait dans la région séquentielle, il faut remplacer la clause private par la clause firstprivate .

```
#include <iostream>
#include <omp.h>

int main()
{
    int a = 314;
#   pragma omp parallel firstprivate(a)
    {
        // Lis un numéro unique pour le thread
        int id = omp_get_thread_num();
        std::cout << "Le thread n." << id << " peut lire la variable a = " << a
                  << std::flush << std::endl;
        a = 413;
    }
    std::cout << "A la sortie de la région parallele , a vaut " << a << std::flush << std::endl;
    return EXIT_SUCCESS;
}
```

Sortie de la fonction :

```
Le thread n.0 peut lire la variable a = 314
Le thread n.1 peut lire la variable a = 314
A la sortie de la région parallele, a vaut 314
```

Si une fonction est appelée dans la région parallèle, elle sera elle–même considée comme faisant partie de la région parallèle. Par contre, les variables définies dans une telle fonction seront considérés comme privées :

```
# include <iostream>
# include <cstdlib>
# include <omp.h>

void function() {
  double a = 92290.;
  a += omp_get_thread_num();
  std::cout << "a = " << a << std::endl;
}

int main() {
#  pragma omp parallel
{
    function();
}
    return EXIT_SUCCESS;
}
```

Cette application affichera :

```
a = 92290
a = 92291
```

Tout paramètre passé à la fonction par pointeur ou par référence prendra le même statut que la variable passée en paramètre.

```
void function(double& a, double& b) {
    b = a + omp_get_thread_num();
    std::cout << "b = " << b << std::endl;
```

```
}

int main()
{
  double a = 92290, b;
# pragma omp parallel shared(a) private(b)
  {
      function(a,b);
  }
  return EXIT_SUCCESS;
}
```

Cette application affichera :

```
b = 92290
b = 92291
```

3.6 Allocation dynamique et région parallèle

Il est parfaitement possible d'allouer au sein d'une région parallèle.

Si le pointeur est privé, l'allocation sera locale au thread, sinon, le pointeur sera partagé et il faut s'assurer qu'un seul thread allouera la mémoire (souvent le maître qui sera numéroté par OpenMP comme le thread numéro zéro).

```
int main() {
    int nbTaches, i, deb, fin, rang, n = 1024;
    double* a;
#   pragma omp parallel
    { nbTaches = omp_get_num_threads(); }

    a = new double[n*nbTaches];

#   pragma omp parallel default(none) private(deb,fin,rang,i) \
                        shared(a,n)
    {
        rang = omp_get_thread_num();
        deb  = rang*n; fin = (rang+1)*n-1;
        for ( i = deb; i <= fin; i++ ) a[i] = 92290. + double(i);
        std::cout << "Rang : " << rang << " A[" << deb << "] = " << a[deb]
                  << ", A[" << fin << "] = " << a[fin] << std::endl;
    }

    delete [] a;
    return EXIT_SUCCESS;
}
```

3.7 Autres clauses possibles pour une région parallèle

reduction : Permet d'effectuer une opération de réduction avec une synchronisation implicite des threads.

num_threads : Spécifie le nombre de threads voulues pour une région parallèle donnée (même fonctionnalité que omp_set_num_threads).

Ainsi le programme suivant lancera deux threads qui additionneront à s leur numéro de thread + 1 (le numéro du premier thread est zéro) en appliquant une réduction pour avoir la somme globale à la sortie de la zone parallèle.

```
int main() {
    int s = 0;
#   pragma omp parallel reduction(+:s) num_threads(2)
    {
        s = omp_get_thread_num()+1;
    }
    std::cout << "s = " << s << std::endl;
    return EXIT_SUCCESS;
}
```

Cette application affichera bien `s = 3`.

3.8 Division du travail

Les fonctionnalités vues ci–dessus sont suffisantes pour paralléliser un programme (mis à part les exclusions mutuelles ou les opérations atomiques vues plus loin).

Cependant OpenMP possède des fonctionnalités permettant de partager le travail automatiquement. Dans les autres cas, ce sera la responsabilité du programmeur de répartir de façon équilibrée le travail au travers des tâches.

Division des boucles en parallèle OpenMP permet de diviser une boucle en plusieurs parties parallèles à l'aide d'une clause `for`.

À noter que les boucles infinies du style do ... while ne sont pas parallélisables en OpenMP.

La répartition statique ou dynamique des morceaux de boucle est spécifiée au travers de la clause `schedule`.

Le choix de ce contrôle de la répartition assure un équilibrage des charges pour les différents threads.

L'indice de la boucle parallélisée est privé sur chaque thread.

Par défaut, une synchronisation globale par thread est faite après la boucle sauf si la clause `nowait` est spécifiée.

Il est possible d'introduire plusieurs clauses `for` (une par une) dans la région parallèle.

La clause `schedule` divise la boucle selon plusieurs modes au choix :

- Selon un mode statique (pris par défaut par Open MP) où la boucle est sectionnée en itérations de tailles fixes distribuées de façon cyclique sur les threads;

```
  const int n = 4096;
  double a[n];
  int i, i_min, i_max, rang, nb_taches;
# pragma omp parallel private(rang, nb_taches, i_min, i_max)
  {
      rang = omp_get_thread_num();
      i_min = n; i_max = 0;
#     pragma omp for nowait
      for ( i = 0; i < n; ++i) {
          a[i] = 92290 + double(i);
          i_min = std::min(i,i_min); i_max = std::max(i,i_max);
      }
      std::cout << "rang : " << rang << " : i_min = " << i_min
                << ", i_max = " << i_max << std::endl;
  }
```

Cette application affichera par exemple sur deux threads :

```
rang 0 : i_min = 0, i_max = 2047
rang 1 : i_min = 2048, i_max = 4095
```

- Selon un mode dynamique : la boucle est découpée en plusieurs petits nombres d'itérations. Quand une tâche a fini ses itérations, un nouveau paquet d'itérations lui est attribué. On peut spécifier (comme pour la répartition statique d'ailleurs) la taille des paquets d'itérations. Par exemple :

```
#include <iostream>
#include <cstdint>
#include <cmath>
#include <vector>
#include <omp.h>

std::uint64_t syracuse(std::uint64_t u0)
{
    std::uint64_t un = u0;
    std::uint64_t cpteur = 0;
    do
    {
        un = (un%2 == 0 ? un / 2 : (3*un+1)/2);
        cpteur ++;
    } while (un != 1);
    return cpteur;
}

int main()
{
    const std::uint64_t N = 100000000;
    std::vector<std::uint64_t> lgth_fly(N);
    double t1 = omp_get_wtime();
#   pragma omp parallel for
    for ( std::uint64_t i = 3; i < N; ++i )
    {
        lgth_fly[i-3] = syracuse(i);
    }
    double t2 = omp_get_wtime();
    std::cout << "Temps parallélisation statique : " << t2-t1 << " secondes."
              << std::endl << std::flush;

    t1 = omp_get_wtime();
    // On choisit de faire beaucoup d'itérations pour
    // que le calcul soit quand même conséquent (granularité)
#   pragma omp parallel for schedule(dynamic,10000)
    for ( std::uint64_t i = 3; i < N; ++i )
    {
        lgth_fly[i-3] = syracuse(i);
    }
    t2 = omp_get_wtime();
    std::cout << "Temps parallélisation dynamique : " << t2-t1 << " secondes."
              << std::endl << std::flush;

    return EXIT_SUCCESS;
}
```

Attendre que la suite u_n atteigne le nombre un prend de plus en plus de temps au fur et à mesure qu'on choisit un u_0 grand et la probabilité de trouver un nombre d'itérations conséquent augmente.
Ainsi, la répartition statique de la boucle est très mal équilibrée. Pour le dynamique,

on choisit de faire de grande sections de boucles afin que la gestion dynamique de répartition des blocs de boucles soit invisible par rapport au calcul effectué dans chaque bloc.
Sur cet exemple, avec six threads (et six coeurs), on trouve que la répartition statique prend douze secondes contre six secondes pour la répartition dynamique !

- Selon un mode guidé : Les itérations sont découpées en des tailles exponentielles décroissantes en taille de paquets d'itérations. Tous les paquets ont une taille plus grande qu'une taille donnée. Les paquets sont attribués dynamiquement comme dans le mode précédent. Cette clause est utile pour des cas vraiment très spécifiques et donc rarement utilisée.

Si la clause `schedule` n'est pas spécifiée, on peut contrôler le mode de division de la boucle à l'aide de la variable d'environnement `OMP_SCHEDULE`.

Enfin, on peut rajouter une clause de réduction appliquée à une variable partagée. Les opérations supportées sont les opérations arithmétiques et logiques.

Chaque tâche calcule un résultat intermédiaire puis se synchronise avec les autres threads pour produire le résultat final :

```
  const int n = 5;
  int i, s = 0, p = 1, r = 1;

# pragma omp parallel for reduction(+:s) reduction(*:p,r)
  for ( i = 1; i <= n; ++i ) {
      s += 1;
      p *= 2;
      r *= 3;
  }
  std::cout << "s = " << s << ", p = " << p << ", r = " << r << std::endl;
```

ce qui affichera à l'exécution :

```
s = 5, p = 32, r = 243;
```

Sections parallèles Une section est une partie de code indépendant exécuté par un thread. On regroupe plusieurs sections indépendantes qui seront chacune exécutée par un thread.

La clause `sections` définit un regroupement de sections parallèles. La clause `section` définit dans un bloc d'instruction une de ces sections. Ces sections doivent être les plus indépendantes les unes des autres possible afin de limiter les conflits mémoires.

Les threads se synchronisent à la fin de la clause `sections` sauf si on spécifie la clause `nowait`.

```
# pragma omp parallel private(i)
{
#  pragma omp sections nowait
   {
#     pragma omp section
      for (i = 0; i < 10; ++i)
  std::cout << "Thread one ! ---> " << i << std::endl;
#     pragma omp section
      for (i = -10; i < 0; i += 2)
  std::cout << "Thread two ! ---> " << i << std::endl;
    }
}
```

3.9 Synchronisation et partie protégées

Exécution exclusive Quelque fois, à l'intérieur d'une région parallèle, on aimerait exécuter une portion du code sur un thread seulement.

Il existe pour cela deux directives OpenMP : `single` et `master`.

Le but est à peu près le même mais le comportement est différent.

La construction `single` exécute une portion du code par un et un seul thread, sans qu'on spécifie le thread. En général, ce thread sera le premier arrivé sur cette portion de code mais la norme ne le spécifie pas.

Tous les threads qui n'exécutent pas cette portion de code attendent la fin de l'exécution de cette portion de code avant de continuer sauf si une clause `nowait` a été rajoutée.

```
#pragma omp parallel default(none) private(a,rang)
{
  a = 92290.;
# pragma omp single
  {
      a = -92290.;
  }
  rang = omp_get_thread_num();
  std::cout << "rang " << rang << " : a = " << a << std::endl;
}
```

Par exemple, le code ci–dessus, exécuté sur deux threads, pourra afficher :

```
rang 0 : a = 92290.
rang 1 : a = -92290.
```

La construction `master` exécute quant à elle sa portion de code que par le thread numéroté zéro par OpenMP. Aucune clause ou synchronisation n'est possible.

```
#pragma omp parallel default(none) private(a,rang)
{
  a = 92290.;
# pragma omp master
  {
      a = -92290.;
  }
  rang = omp_get_thread_num();
  std::cout << "rang " << rang << " : a = " << a << std::endl;
}
```

Le code ci–dessus affichera par exemple en prenant deux threads :

```
rang 0 : a = -92290.
rang 1 : a = 92290.
```

Synchronisation La synchronisation est nécessaire dans trois cas :

1. Pour être sûr que tous les threads concurrents exécutent en même temps la même ligne de code (barrière globale);
2. Ordonner l'exécution de tous les threads concurrents quand ils doivent exécuter la même portion de code qui modifie une ou plusieurs variables partagées et qu'on doit garantir la cohérence de la mémoire (lecture ou écriture en exclusion mutuelle);
3. Synchroniser deux ou plusieurs threads sans affecter les autres threads.

La directive `barrier` permet de synchroniser tous les threads dans une région parallèle : chaque thread attend que tous les autres threads aient atteint le point d'appel de la barrière avant de continuer l'exécution du programme.

```
  double *a, *b;
  int i, n=5;
# pragma omp parallel
  {
# pragma omp single
    { a = new double[n]; b = new double[n]; }
# pragma omp master
    { for (i = 0; i < n; i++)
        a[i] = (i+1)/2.; }
# pragma omp barrier
# pragma for schedule(static)
    for (i=0; i < n; i++) b[i] = 2.*a[i];
# pragma omp single nowait
    { delete [] a; }
  }
  printf("B equal : ");
  for (i = 0; i < n; i++) printf("%7.5lg\t",b[i]);
  printf("\n");
```

La directive `atomic` garantit qu'une variable n'est lue ou écrite que par un thread à la fois. Son effet est local à l'instruction qui suit juste après la directive.

```
  int counter, rank;
  counter = 100;
# pragma omp parallel private(rank)
  {
    rank = omp_get_thread_num();
#   pragma omp atomic
    counter += 1;

    printf("Rank : %d, counter = %d\n",rank,counter);
  }
  printf("Final counter : %d\n",counter);
```

Display :

```
Rank : 1, counter = 102
Rank : 0, counter = 101
Counter final : 102
```

On peut également protéger une zone de code plus complexe qu'une simple instruction à l'aide d'une section critique (`critical`). Dans ce cas, les threads exécutent cette région un par un dans un ordre quelconque. Du point de vue performance, cette instruction est moins performante que `atomic` mais il est parfois impossible d'employer une instruction d'atomicité.

```
  int s,p;

  s = 0; p = 1;
# pragma omp parallel
  {
#   pragma omp critical
    {
      s += 1;
```

```
      p *= 2;
    }
  }
  printf ("s = %d, p = %d\n", s, p);
```

Display : `s = 2, p = 4`

3.10 Gestion de la cohérence de cache

Il est possible en OpenMP de mettre à jour une variable globale dans la mémoire partagée. Cela permet de s'assurer de la cohérence des données entre la mémoire vive et les mémoires caches.

Cette instruction est utile dans certains mécanismes de synchronisation.

```
  int rank, number_of_tasks, synch = 0;
# pragma omp parallel private(rank, number_of_tasks)
  { rank=omp_get_thread_num();number_of_tasks=omp_get_num_threads();
    if (rank == 0) {
      while(synch != number_of_tasks-1) {
#       pragma omp flush(synch)
      }
    } else {
      while (synch != rank-1) {
#       pragma omp flush(synch)
      }
    }
    printf("rank = %d, synch = %d\n",rank, synch);
    synch = rank;
#   pragma omp flush(synch)
  }
```

```
 rank = 1, synch = 0
rank = 0, synch = 1
```

4 Autres outils existants

Il existe d'autres bibliothèques permettant une gestion simplifiée des threads, plus ou moins performante qu'OpenMP.

4.1 Threading Building Blocks (TBB)

C'est une bibliothèque de template C++ initialement proposée avec le compilateur d'Intel (payant) mais qui est devenue avec le temps une bibliothèque à part entière sous licence libre Apache. Elle permet de gérer des tâches en parallèle à l'aide des threads sous une forme plus légère et performante qu'OpenMP.

On peut la télécharger gratuitement sur `https://www.threadingbuildingblocks.org/`

TBB propose des boucles parallèles, des algorithmes de réduction, gestion de tâches en parallèle et gestion de la concurrence entre threads.

Voici un exemple de boucle en parallèle appelant à chaque itération une fonction `Foo`.

```
# include "tbb/tbb.h"
using namespace tbb;

void ParallelApplyFor( float a[], size_t n )
```

```
{
  tbb::parallel_for ( size_t(0), n, [&] ( size_t i ) {
                                                Foo(a[i]));
                                                    } );
}
```

TBB propose également des conteneurs permettant une gestion sûre dans un contexte parallèle.

Par exemple, au lieu du conteneur std::queue de la STL dont les méthodes push et pop ne sont pas protégées, on utilisera le conteneur concurrent_queue proposé par TBB :

```
extern tbb::concurrent_queue<T> MyQueue;
T item;
if( MyQueue.try_pop(item) ) {
    ...process item...
}
```

Il est également possible de définir un graphe de tâche, où les nœuds représentent des tâches à exécuter et les arêtes les dépendances entre ces tâches (par exemple une tâche T_2 ne pourra pas s'exécuter avant une tâche T_1 si la tâche T_2 prend en entrée ce que la tâche T_1 retourne en sortie). TBB exploitera alors le parallélisme inhérent à la topologie du graphe de tâche (quand il peut exécuter plusieurs tâches qui ne sont pas reliées par une arête et qui ne dépendent pas d'autres tâches encore non exécutées).

Voici un petit exemple d'Hello world écrit à l'aide d'un graphe de tâche :

```
#include "tbb/flow_graph.h"
#include <iostream>

using namespace std;
using namespace tbb::flow;

int main() {
    graph g;
    continue_node< continue_msg> hello( g,
      []( const continue_msg &) {
          cout << "Hello";
      }
    );
    continue_node< continue_msg> world( g,
      []( const continue_msg &) {
          cout << "␣World\n";
      }
    );
    make_edge(hello, world);
    hello.try_put(continue_msg());
    g.wait_for_all();
    return 0;
}
```

Dans le code ci–dessus, l'appel à hello.try_put(continue_msg()) envoie un message au nœud hello qui lui fera exécuter sa tâche puis envoyer un message au nœud World qui exécutera ensuite sa tâche. La fonction `g.wait_for_all` attend que le parcours du graphe soit complet et que tous les nœuds aient exécuté leur tâche.

D'autres utilitaires sont proposés par TBB comme des allocateurs permettant pour l'un d'allouer simultanément plusieurs zones mémoires et pour l'autre de s'assurer que deux objets gérés par deux threads différents n'appartiennent pas à la même ligne de cache. En effet, dans le cas contraire, afin d'assurer la cohérence des caches, lorsque deux unités de

calcul essaient d'accéder à deux objets appartenant à la même ligne de cache, le hardware doit déplacer la ligne de cache d'un processeur à l'autre (même si il n'y a en fait aucun conflit entre les deux unités de calcul) ce qui peut engendrer une perte inutile de plusieurs centaines de cycles d'horloge.

4.2 C++ 2017

La norme C++ 2017 (encore en cours de négociation) propose des politiques sur les boucles et les algorithmes de la STL qui permettront de les paralléliser sous forme de threads légers.

L'avantage ici est qu'il ne sera pas nécessaire d'utiliser de bibliothèques externes ce qui permet un déploiement plus facile d'un logiciel sur d'autres plateformes.

Cette parallélisation se fera sous forme de *politique d'exécution*, c'est à dire sous forme d'un objet qui exprimera la façon (séquentielle, parallèle, vectorielle, etc.) dont s'exécutera la fonction STL.

Notons que les versions parallèles de la STL supportées par g++ et clang++ utilisent la bibliothèque TBB d'Intel !

Par exemple :

```
std::vector<int> v = ...
// standard sequential sort
std::sort(vec.begin(), vec.end());
using namespace std::experimental::parallel;
// explicitly sequential sort
sort(seq, v.begin(), v.end());
// permitting parallel execution
sort(par, v.begin(), v.end());
// permitting vectorization as well
sort(vec, v.begin(), v.end());
// sort with dynamically-selected execution
size_t threshold = ...
execution_policy exec = seq;
if(v.size() > threshold)
{
exec = par;
}
sort(exec, v.begin(), v.end());
```

Lorsqu'une erreur est rencontrée lors d'une exécution parallèle, la fonction STL retourne une liste d'exceptions levées durant l'exécution parallèle de l'algorithme.

Il est par contre de la responsabilité du programmeur de s'assurer de l'emploi correct de la politique d'exécution. Il doit en particulier s'assurer que l'emploi de sa politique d'exécution ne conduira pas à des conflits d'accès aux données ou à de l'interblocage.

De nouveaux algorithmes sont également proposés, en particulier pour faire des réductions ou des scans.

Peu de compilateurs supportent encore la norme C++ 2017 en ce qui concerne la parallèlisation de la STL. Néanmoins, certains compilateurs "payants" ainsi que les dernières versions du compilateur gnu (à partir de la version 9.1) utilisant la bibliothèque TBB d'INTEL (qui est gratuite) permettent d'utiliser les algorithmes parallèles proposées par la STL.

Pour plus d'information sur la librairie standard parallèle, on peut se référer à la page suivante proposée par Intel : `https://software.intel.com/en-us/articles/get-started-with-parall`

On peut s'attendre à ce que la version définitive de C++ 2017 soit adoptée au milieu de cette année (2017).