Programmation parallèle sur mémoire partagée

Juvigny Xavier

February 12, 2018

Contents

1 Modèles de programmation sur ordinateurs parallèles à mémoire partagée

1.1 Notion de thread

Avant de comprendre ce qu'est un thread, il faut tout d'abord savoir ce qu'est un processus. Un processus est créé par le système d'exploitation et 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,...

On peut alors donner la définition d'un thread :

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.

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

Si les threads ont la réputation d'être diaboliques de part la relative complexité qu'ils peuvent engendre 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 pour le processus d'attendre que le thread ait fini pour continuer l'exécution du programme.

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*

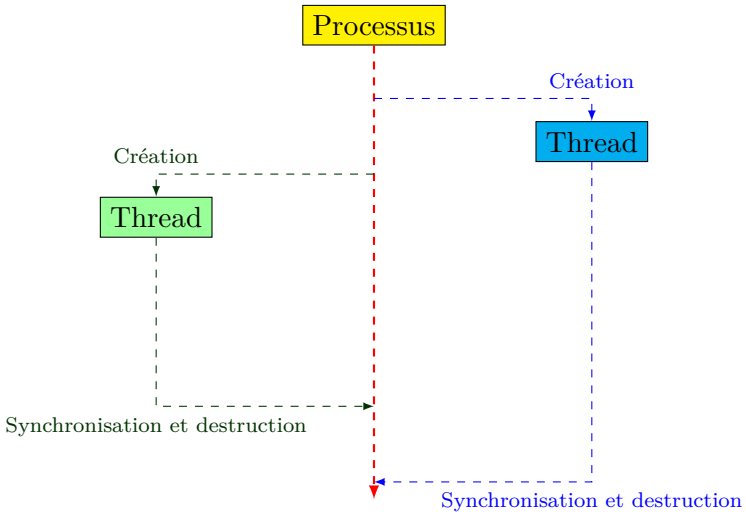

Figure 1: Exemple de création, synchronisation et terminaison de thread

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.

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)

Dans le cas d'un thread autonome, ne dépendant pas des resources du programme principal, il est possible de le "détacher" de sorte qu'il continue à s'exécuter même après la fin du programme principal (il devient alors en terme unix ce qu'on appelle un daemon).

1.2 hyperthreading

Sur les CPUs modernes, les concepteurs ont mis deux fois plus de décodeur d'instructions (ALU) que d'unités calculatoires (entiers ou réels). 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 au mieux un gain de trente pourcent 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).

Il faut cependant savoir que les systèmes d'exploitation actuels se basent sur le nombre de décodeurs d'instructions pour donner le nombre d'unités de calculs. Ainsi, si on tape sous Linux :

```
more /proc/cpuinfo
```

on aura par exemple obtenir la sortie suivante :

```
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz
stepping : 3
microcode : 0x200
cpu MHz : 2530.822
cache size : 6144 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi
bugs :
bogomips : 4988.84
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:

processor : 1
...

processor : 2
...

processor : 3
...

processor : 4
...

processor : 5
...

processor : 6
```

```
...

processor : 7
...
```

ou bien si on veut une information résumée :

```
[juvigny@Frickicare Ensta]$ lscpu
Architecture :        x86_64
Mode(s) opératoire(s) des processeurs : 32-bit, 64-bit
Boutisme :            Little Endian
Processeur(s) :       8
Liste de processeur(s) en ligne : 0-7
Thread(s) par cœur : 2
Cœur(s) par socket : 4
Socket(s) :           1
Nœud(s) NUMA :       1
Identifiant constructeur : GenuineIntel
Famille de processeur : 6
Modèle :             60
Nom de modèle :      Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz
Révision :           3
Vitesse du processeur en MHz : 3313.598
Vitesse maximale du processeur en MHz : 3500,0000
Vitesse minimale du processeur en MHz : 800,0000
BogoMIPS :            4988.11
Virtualisation :      VT-x
Cache L1d :           32K
Cache L1i :           32K
Cache L2 :            256K
Cache L3 :            6144K
Nœud NUMA 0 de processeur(s) : 0-7
```

Cependant, l'ordinateur sur lequel l'instruction a été exécuté ne contient que quatre unités de calculs !

Si on regarde plus en détail les caractéristiques de chacun des "processeurs" détectés par Linux (dans la sortie donnée en exemple, on a mis des points de suspension pour les sorties des processeurs autres que zéro car le texte y est similaire), on oberve une ligne marquée `cpu cores : 4` indiquant bien que le nombre d'unités de calcul est de quatre !

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 particuliers, 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 plus hautes que celles imposées par les communications MPI en mémoire partagée.

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

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 :

- 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;
- Les deux threads écrivent en même temps dans la même zone mémoire : Là encore, le résultat est aléatoire. Au mieux, la valeur mis à jour est une des valeurs écrite par l'un des threads, au pire, la valeur est incohérente.

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 **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 quel unité de calcul, on demande au gestionnaire de tâche du système d'exploitation de n'exécuter un thread particuliers 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 considérations du gestionnaire de tâche. De plus, parfois, le système d'exploitation va faire 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 permet de suivre quatre threads dans une boucle infinie et afficher l'unité de calcul sur laquelle il s'exécute au fur et à mesure du temps :

```
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 on aura une fonction similaire). Une exécution de ce programme pourra par exemple donner le résultat suivant :

```
$ ./launch-threads-report-cpu
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.

Sous Linux, exécutons de nouveau ce programme en utilisant l'utilitaire taskset qui permet de restreindre l'affinité d'un processus à un ou quelques CPUs. Ici, restreignons l'affinité du processus à seulement deux CPUs : le cinquième et le sixième :

```
$ ./launch-threads-report-cpu
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
```

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 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.

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.

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, contigües 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 décharger de la mémoire cache.

Cela demande donc d'écrire des algorithmes pouvant exploiter plusieurs fois dans un bref délai les mêmes variables.

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 supérieure à la complexité algorithmique, c'est à dire au nombre d'opérations effectuées.

En ce qui concerne la mémoire interlacée : L'exploitation efficace d'une mémoire interlacée se base sur une lecture contigüe 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.

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

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 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 supérieure à 1024 ! Cet algorithme sera clairement adapté à une architecture mémoire contenant de la mémoire cache. Le CPU pourra calculer une itération de la suite à chaque cycle d'horloge. Ce sera donc une fonction cpu bound. Il en est de même pour

une mémoire interlacée puisqu'il est fort probable que ces deux variables seront stockées dans des registres du processeur !

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]);
...
}
```

Pour cet algorithme, il n'est pas possible d'utiliser la mémoire cache vu que nous faisons $3N$ accès mémoire pour N opérations de comparaison.

Dans le cas d'une machine possédant une mémoire vive non interlacée, cette fonction sera donc "memory bound".

Par contre, la lecture des données se fait de manière contigüe, et il est probable que sur une machine à mémoire interlacée, le CPU pourra à chacun de ses cycles d'horloge traiter une boucle de l'itération. On aura alors une fonction cpu bound.

Opération matricielle

```
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];
```

En l'état des choses, cet algorithme tel qu'il est écrit n'exploite ni la mémoire cache ni une possible mémoire interlacée. Cependant, on remarque que sa complexité en accès mémoire est de $3N^2$ tandis que sa complexité algorithmique est de $2N^3$. Donc pour $N > 1$, sa complexité algorithmique est supérieure à sa complexité en accès mémoire. De plus, en reordonnant les boucles, tel que la boucle en i soit interne (ce qui ne changera en aucun cas le résultat final), on fera deux accès linéaires contre un seul accès avec des sauts mémoires, ce qui permet de mieux exploiter les lignes de cache ainsi qu'une éventuelle mémoire entrelacée :

```
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];
```

La solution pour exploiter la redondance des données 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 reexploite bien dans un court intervalle une partie des données de B. On obtient dans ce cas une fonction cpu-bound.

De même, 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) tout 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.

2 Posix threads (C++ 2011)

2.1 Exécution et terminaison d'un thread en C++ 2011

Créer et terminer un thread en C++ 2011 est bien plus simple qu'avec la bibliothèque pthread proposée par UNIX.

Voyons comment créer un simple programme HelloWorld multi–threadé :

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

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

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

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

  return 0;
}
```

Dans le monde réel, la fonction call_from_thread fera un travail indépendant de la fonction principale. Dans le code donné ci–dessus, la fontion principale créée un thread et attend que le thread ait finit en appelant t1.join(). Si vous oubliez d'attendre qu'un thread

ait fini son travail, il est possible que la fonction principale termine avant le thread et le programme sortira en tuant le thread, qu'il est fini son travail ou non.

Remarquons qu'il est en C++ 2011 possible de passer comme fonction à exécuter par le thread une lamda fonction. Ainsi, l'exemple donné plus haut aurait pu s'écrire :

```
#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. Dans l'exemple suivant, la fonction principale créée un groupe de dix threads qui feront plusieur 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(call_from_thread);
  }

  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.

Comment faire pour passer des paramètres dans un thread en C++ 2011 ? En fait, C++ 11 nous laisse rajouter autant de paramètres que l'on veut dans l'appel du thread. On peut par exemple donner un entier à chaque thread dans le code ci–dessus :

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

static const int num_threads = 10;

//Cette fonction sera appelée d'un thread
void call_from_thread(int tid) {
  std::cout << "Launched by thread " << tid << std::endl;
```

```
}

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

  // Exécuter un groupe de threads :
  for (int i = 0; i < num_threads; ++i) {
    t[i] = std::thread(call_from_thread, i);
  }

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

  // Joindre les threads avec le thread principal :
  for (int i = 0; i < num_threads; ++i) {
    t[i].join();
  }

  return 0;
}
```

2.2 Lancer un thread par CPU

Le C++ 2011 nous fournit une fonction utilitaire permettant de savoir combien d'unités de calcul sont possédés 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(int argc, const char** argv) {
  unsigned num_cpus = std::thread::hardware_concurrency();
  std::cout << "Créer " << num_cpus << " threads\n";

  // Un mutex permet de s'assurer un accès ordonnées au std::cout pour
  // de multiples threads.
  std::mutex iomutex;
  std::vector<std::thread> threads(num_cpus);
  for (unsigned i = 0; i < num_cpus; ++i) {
    // On lance un thread exécutant une lambda fonction :
    threads[i] = std::thread([&iomutex, i] {
      {
        // std::lock_guard permet de bloquer un mutex pendant la durée d'un bloc
        // d'instruction. Ici cela permet de bloquer le mutex uniquement durant
        // l'affichage à l'iade de std::cout
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << "Thread #" << i << " is running\n";
      }

      // On simule un travail important en mettant le thread un peu en pause
      std::this_thread::sleep_for(std::chrono::milliseconds(200));

    });
  }

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

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 exécuté en parallèle et où on doit s'assurer que les threads ne peuvent passer que un par un.

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 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 de la 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 une map :

```cpp
#include <iostream>
#include <chrono>
#include <thread>
#include <mutex>
#include <map>
#include <string>

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

void save_page(const std::string &url)
{
    // simulate a long page fetch
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::string result = "fake content";

    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();

    g_pages_mutex.lock();
    for (const auto &pair : g_pages) {
        std::cout << pair.first << " => " << pair.second << '\n';
    }
    g_pages_mutex.unlock();
}
```

Résultat :

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

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

Si la donnée partagée par les threads est une donnée de type primitif (entier, réel, char, ...) et si l'opération effectuée dessus est basique (incrément, addition, soustraction, etc.), on peut, plutôt que d'utiliser des mutex, déclarer cette variable atomique. On sera ainsi garantit 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, si il possible d'avoir des deadlocks avec les mutex comme dans le programme suivant :

```
if ( pid == 0) {
  lock(S);
  lock(Q);
  ...
  unlock(Q);
  unlock(S);
} else if ( pid == 1 ) {
  lock(Q);
  lock(S);
  ...
  unlock(S);
  unlock(Q);
}
```

il est impossible d'avoir un deadlock à l'aide d'opérations atomiques.

3 OpenMP

3.1 Principe d'OpenMP

OpenMP est un outil intégré dans la plupart des compilateurs actuels (gnu c++, MS VC++, Intel, etc.). 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. Si cette option n'est pas donnée, le compilateur ignorera les directives OpenMP données par le programme.

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

On peut contrôler le nombre de threads générés par l'application à l'aide de la variable `OMP_NUM_THREADS`.

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 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 <cstdlib>
# include <omp.h>

int main()
{
```

```
  float a;
  int p;

  a = 92290; p = 0;
# pragma omp parallel
  {// Début de la région parallèle
    p = omp_in_parallel();
    std::cout << "a = " << a << " et p = " << p << std::endl;
  }// Fin de la région parallèle
  return EXIT_SUCCESS;
}
```

La sortie de ce programme donnera :

```
a = 92290, p = 1
a = 92290, p = 1
```

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.

On peut modifier le status d'une variable entre partagée ou privée à l'aide d'une *clause*.

Les variables privées sont créées au commencement du bloc parallèle pour chaque thread et ne sont pas définies à l'entrée de ce bloc.

Exemple de région parallèle avec clause private :

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

int main()
{
  float a = 92000;

# pragma omp parallel default(none) private(a)
  {// Début de la région parallèle
    a = a + 290;
    std::cout << "a = " << a << " et p = " << p << std::endl;
  }// Fin de la section parallèle
  return EXIT_SUCCESS;
}
```

La sortie de ce programme donnera :

```
a = 290
a = 290
```

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

Exemple de région parallèle avec clause firstprivate :

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

int main()
{
  float a = 92000;

# pragma omp parallel default(none) firstprivate(a)
  {// Début de la région parallèle
    a = a + 290;
```

```
    std::cout << "a = " << a << " et p = " << p << std::endl;
  }// Fin de la région parallèle
  return EXIT_SUCCESS;
}
```

La sortie de ce programme donnera :

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

3.5 Fonction appelée dans une région parallèle

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 paramêtre passé à la fonction par pointeur ou par référence prendra le même status 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 local 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 dans une région parallèle

Réduction : 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).

```
int main() {
    int s = 0;
#   pragma omp parallel default(none) private(p) reduction(+:s)
    {
        s = omp_get_thread_num()+1;
    }
    std::cout << "s = " << s << std::endl;
    return EXIT_SUCCESS;
}
```

Cette application affichera `s = 3`.

3.8 Division du travail

Les fonctionnalités vues ci–dessus sont suffisantes pour paralléliser un programme. 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.

En outre, OpenMP fournit des clauses permettant d'exclure toutes les tâches sauf une d'une zone critique définie dans la région parallèle ou bien définir une opération atomique.

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 parallélisé 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 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();
      nb_taches = omp_get_num_threads();
      i_min = n; i_max = 0;
#     pragma omp for schedule(static, n/nb_taches) 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 finie ses itérations, un nouveau paquet d'itérations lui est attribué.
- 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.

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 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 exécute 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. 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 devenu avec le temps une bibliothèque à part entière sous license 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 parallèlisme inhérant 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 exécutée).

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 exécutée leur tâche.

D'autres utilitaires sont proposées 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 (Draft)

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.

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 particuliers 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 particuliers pour faire des réductions ou des scans.

Le draft concernant le standard de la parallélisation de la STL peut être trouvé sur le lien suivant : `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3960.pdf`

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