Calcul Haute Performance

Quentin Meunier - Adrien Cassagne

Multi-Thread et Synchronisation (2)

Quentin Meunier

Laboratoire d'Informatique de Paris 6
Équipe Alsoc
4 Place Jussieu, 75252 Paris, France

Polytech, EISE5, HPC

La synchronisation lock-free

Limites liées aux locks

- Les algorithmes à base de locks sont souvent peu scalables
- De plus, ils sont en général difficiles à concevoir, implémenter, débugger et maintenir si on veut une granularité fine (i.e. plus de parallélisme)
- $\Rightarrow$ Recherche d'autres moyens pour synchroniser des threads

Programmation lock-free

- Comprend les algorithmes sans sections critiques
- Basée sur la primitive Compare-and-Swap et ses variantes
- Basée sur les primitives LL/SC
- Sans primitive spécifique

Transactions

- Vous en êtes épargnés dans ce cours

Exemples de programmes lock-free

Exemple avec un compare-and-swap

- Comment incrémenter une variable partagée à l'aide de la primitive `compare_and_swap` ?

Exemples de programmes lock-free

Exemple avec un compare-and-swap

- Comment incrémenter une variable partagée à l'aide de la primitive `compare_and_swap` ?

Solution

```
do {
  local_var_old = shared_var;
  local_var_new = local_var_old + 1;
} while (!cas(&shared_var, local_var_old, local_var_new));
```

Exemples de programmes lock-free

Exemple avec un compare-and-swap

- Comment faire un min atomique d'une variable partagée avec une variable locale à l'aide de la primitive `compare_and_swap` ?

Exemples de programmes lock-free

Exemple avec un compare-and-swap

- Comment faire un min atomique d'une variable partagée avec une variable locale à l'aide de la primitive compare_and_swap ?

Solution

```
do {
  local_var_old = shared_var;
  local_var_new = min(local_min, local_var_old);
} while (!cas(&shared_var, local_var_old, local_var_new));
```

Exemples de programmes lock-free

Comment mettre à jour plusieurs champs simultanément avec un CAS ?

- Avoir un CAS sur plusieurs mots
  - Matériel spécifique
- Approche générale :
  - Allouer une structure
  - Copier tous les champs de la structure actuelle, en mettant à jour ceux que l'on veut
  - Réaliser un CAS sur les adresses des structures

```
typedef struct _s {
  int a;
  int b;
  int c;
  int d;
  ...
} s_t;

s_t * sh_pt;
int main() {
  ...
  sh_pt = ...;
  ...
}
```

```
void local_func() {
  s_t * lo_pt = malloc();
  s_t * sh_cp;
  do {
    sh_cp = sh_pt;
    memcpy(lo_pt, sh_cp, sizeof(s_t));
    if (sh_cp != sh_pt) {
      // Données copiées invalides
      continue;
    }
    // Données copiées valides
    lo_pt->a += 1;
    ...
  } while (!cas(&sh_pt, sh_cp, lo_pt));
  ...
}
```

Exemples de programmes lock-free

Problème de l'approche générale présentée

- À deux moments, on fait l'hypothèse que si l'adresse contenue dans la variable globale `sh_pt` est identique, le contenu associé – après déréférencement – est également identique
- Ce n'est pas forcément le cas
- Problème connu depuis longtemps sous le nom de problème ABA
- Ce problème apparait quand il y a une indirection des valeurs manipulées dans un CAS, donc typiquement quand on fait un CAS sur des pointeurs, ce qui est le cas ici

Quelle solution à ce problème ?

- La solution la plus simple est d'utiliser un numéro de *version*
- Requiert un CAS sur 2 mots (qui peuvent être contigüs) : en général, supporté par les architectures
- On considère qu'il faut au moins 32 bits pour le numéro...
- Il existe d'autres techniques...

Exemples de programmes lock-free

Exemple sans primitive spécifique

- Comment écrire une barrière entre 2 threads sans primitive spécifique ?

Exemples de programmes lock-free

Exemple sans primitive spécifique

- Comment écrire une barrière entre 2 threads sans primitive spécifique ?

Solution

```
volatile bool t0_ok = false;
volatile bool t1_ok = false;
```

```
// T0
t0_ok = true;
while (!t1_ok);
```

```
// T1
t1_ok = true;
while (!t0_ok);
```

Exemples de programmes lock-free

Autre exemple sans primitive spécifique

- Comment implémenter une section critique entre 2 threads sans primitive spécifique ?
- Algorithme de Peterson (intialement, `flag0 = flag1 = false;`)

```
// T0
flag0 = true;
turn = 1;
while (flag1 && turn == 1);
// Début de la section critique
...
// Fin de la section critique
flag0 = false;
```

```
// T1
flag1 = true;
turn = 0;
while (flag0 && turn == 0);
// Début de la section critique
...
// fin de la section critique
flag1 = false;
```

Exemple de programmes lock-free

Remarques sur l'algorithme de Peterson

- Code compliqué à comprendre
  - Si les codes à base de locks peuvent être compliqués à comprendre, les programmes sans sont souvent bien plus compliqués
- En pratique, on n'utilise pas cet algorithme pour réaliser des sections critiques
- Une des raisons : le nombre de threads doit être connu à l'avance (il existe des extensions pour N threads)
  - Et puis il existe toujours un minimum de support d'instructions atomiques sur les architectures
- L'algorithme fait des hypothèses fortes sur le modèle de consistance mémoire

Les Read-Write locks (RW-locks)

- Dans un bon nombre de cas, les accès aux structures partagées sont plus fréquemment des lectures
  - Introduction des notions de thread lecteur et de thread écrivain
- Les zones de lectures atomiques peuvent accéder à un grand nombre de variables
- Idée des RW-locks : permettre les lectures en parallèle
  - Pas d'écriture en parallèle avec les lectures, ni 2 écritures en parallèle
- Introduction d'une variable pour compter le nombre de threads lecteur en cours d'accès
  - Implique un lock pour accéder à cette variable (ne peut pas être fait avec un CAS – cf. suite)
- Un lock global est pris par le premier lecteur, et relâché par le dernier lecteur
- Ce même lock est également pris par un thread écrivain

Les Read-Write locks (RW-locks)

```
typedef struct _rwlock_t {
  lock_t rd_lock;
  lock_t wr_lock;
  int nb_readers;
} rw_lock_t;

void lock_write(rw_lock_t * l) {
  lock(&l->wr_lock);
}

void unlock_write(rw_lock_t * l) {
  unlock(&l->wr_lock);
}
```

```
void lock_read(rw_lock_t * l) {
  lock(&l->rd_lock);
  l->nb_readers += 1;
  if (l->nb_readers == 1) {
    lock(&l->wr_lock);
  }
  unlock(&l->rd_lock);
}

void unlock_read(rw_lock_t * l) {
  lock(&l->rd_lock);
  // Décrémentation impossible avec
  // un CAS car la lecture dans le
  // test ensuite doit être atomique
  l->nb_readers -= 1;
  if (l->nb_readers == 0) {
    unlock(&l->wr_lock);
  }
  unlock(&l->rd_lock);
}
```

Read-Write locks avec priorité aux écritures

- Dans l'algo précédent, un thread écrivain peut subir une famine si de nouveaux lecteurs arrivent continuellement
- Autre problème : nécessite que le lock puisse être pris par un thread et relâché par un autre thread (pas le cas des spinlock et mutex_lock de posix par exemple)
- Idées pour adresser ces deux points :
  - N'utiliser plus qu'un seul lock de type spinlock
  - Décrémenter `nb_readers` avec un CAS
  - L'écrivain attend que `nb_readers` repasse à 0 après avoir pris le lock
- Remarque : après le relâchement du lock par un écrivain, pas de priorité spécifique

Read-Write locks avec priorité aux écritures

```
typedef struct {
  int cpt; // init à 0
  // lock_t peut être par exemple
  // un pthread_spinlock_t
  lock_t lock;
} rw_lock_t;

void lock_read(rw_lock_t * l) {
  lock(&l->lock);
  l->cpt += 1;
  unlock(&l->lock);
}

void unlock_read(rw_lock_t * l) {
  int val;
  do {
    val = l->cpt;
  } while (!cas(&l->cpt, val, val - 1));
}
```

```
void lock_write(rw_lock_t * l) {
  lock(&l->lock);
  while (l->cpt != 0);
}

void unlock_write(rw_lock_t * l) {
  unlock(&l->lock);
}
```

Read-Write locks avec priorité aux écritures

- Idée d'amélioration de l'algorithme précédent : utiliser la même variable pour compter le nombre de lecteurs et servir de lock
- Principe :
  - Les lecteurs incrémentent (resp. décrémentent) atomiquement la variable à leur entrée (resp. sortie) de la section critique
  - On se sert d'un bit du mot servant au lock pour enregistrer (atomiquement) l'information qu'un écrivain a fait une demande (bit de poids fort $\Rightarrow$ Nombre négatif)
  - Les lecteurs ne prennent plus le lock tant qu'un écrivain a fait une demande, mais les lecteurs dans la section critique finissent normalement
  - Quand il n'y a plus de lecteurs dans la section critique, l'écrivain obtient le lock
- Remarque : après le relâchement du lock par un écrivain, pas de priorité spécifique

Read-Write locks avec priorité aux écritures : variante

```
#define WR_REQ 0x80000000

void lock_read(int * lock) {
  int val;
  bool ok = false;
  while (!ok) {
    val = *lock;
    if (val < 0) {
      // waiting for the writer to
      // release the lock or to take
      // the lock then release it
      continue;
    }
    ok = cas(lock, val, val + 1);
  }
}

void unlock_read(int * lock) {
  int val;
  do {
    val = *lock;
  } while (!cas(lock, val, val - 1));
}
```

```
// if single writer
void lock_write(int * lock) {
  int val;
  do {
    val = *lock;
    if (val >= 0) {
      cas(lock, val, val | WR_REQ);
    }
  } while (val != WR_REQ);
}
// if several writers
void lock_write(int * lock) {
  int val;
  bool ok = false;
  while (!ok) {
    val = *lock;
    if (val >= 0) {
      cas(lock, val, val | WR_REQ);
      continue;
    }
    ok = cas(lock, WR_REQ, -1);
  }
}

void unlock_write(int * lock) {
  *lock = 0;
}
```

Les seqlocks (Sequential Locks)

- Type de RW-lock dans lequel les threads lecteurs n'ont pas besoin de prendre de lock pour accéder aux données
  - $\Rightarrow$ Accélère en général le programme si peu de threads écrivains
- Besoin de vérifier à la fin de la section critique ques les valeurs lues sont consistantes
  - Fait avec un compteur, qui est incrémenté par les écrivains au début et à la fin de la section critique
  - Ce compteur permet aussi de vérifier qu'une écriture n'est pas en cours
- En cas d'échec, le thread lecteur doit recommencer
  - Problème : peut mener à un livelock (famine) s'il y a trop d'écrivains

Les seqlocks : structure du programme

```
typedef struct _lock_t {
  int seqnumber;
  pthread_spinlock_t lock;
} lock_t;

void lock_write(lock_t * l) {
  pthread_spin_lock(&l->lock);
  l->seqnumber += 1;
}

void unlock_write(lock_t * l) {
  l->seqnumber += 1;
  pthread_spin_unlock(&l->lock);
}

int lock_read(lock_t * l) {
  return l->seqnumber;
}
```

```
bool unlock_read(lock_t * l, int n_ini) {
  int n_end = l->seqnumber;
  if (n_end != n_ini) {
    return false;
  }
  return true;
}


  ...
  // in the reader thread
  bool ok = false;
  while (!ok) {
    int n_ini = lock_read(&the_lock);
    if (n_ini % 2 == 1) {
      continue;
    }
    // Read shared structure here
    ...
    ok = unlock_read(&the_lock, n_ini);
  }
  ...
```

Les seqlocks (Sequential Locks)

Remarque

- Les seqlocks ne fonctionnent pas s'il y a des indirections dans les valeurs lues (i.e. les valeurs lues sont des pointeurs), car il est possible qu'un thread écrivain invalide (par exemple, rende NULL) un pointeur qui a déjà été lu par un thread lecteur.
- Exemple

```
// thread lecteur
if shared_struct.next != NULL) {
  // Ici le thread écrivain modifie le champ next à NULL
  mavar = shared_struct.next->champ;
  // provoque une segmentation fault car déréférencement de NULL
  ...
}
```

Types de programmes parallèles

- D'une manière générale, paralléliser un programme est un problème difficile
  - Il n'existe pas d'approche systématique
- Parallélisation sur les données quand celles-ci sont facilement découpables
- Parallélisation sur les données de manière centralisée
- Parallélisation de l'application en tâches
- Parallélisation avec tâches créées dynamiquement

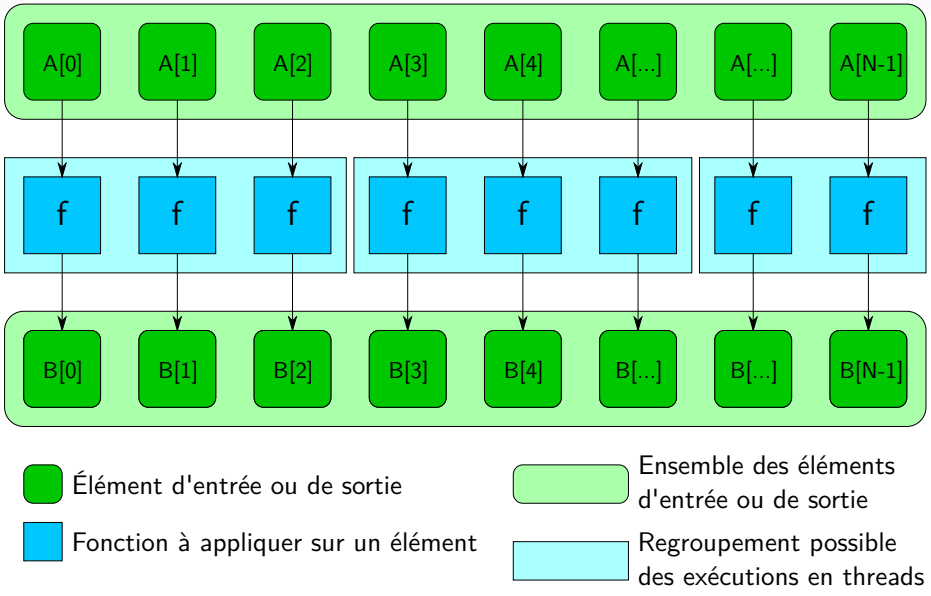
Parallélisation sur les données

- Chaque thread exécute la même fonction
- Chaque thread a en paramètre une description des données qu'il doit traiter
  - Le plus souvent, il s'agit de simples index
- Peut parfois nécessiter des synchronisations en plus (cas pour Kmeans)

Parallélisation sur les données de manière centralisée

- Les threads ne possèdent plus les informations du travail à effectuer au début de leur exécution
- Ils vont chercher régulièrement du travail dans une structure centralisée
  - Nécessite une synchronisation pour y accéder
- Efficace quand il y a des variations dans la durée de traitement d'une "unité" (et que l'on ne peut pas connaitre ces durées à priori)
- Peut devenir un point de contention si la granularité de l'extraction est mal choisie
- Variante : les données sont initialement réparties entre threads mais possibilité d'aller en voler à un autre thread s'il n'y en a plus localement
  - Intérêt : l'extraction locale peut être très rapide (peu de contention) et si c'est bien fait, les vols devraient rester rares

Exemple de parallélisation sur les données : Map



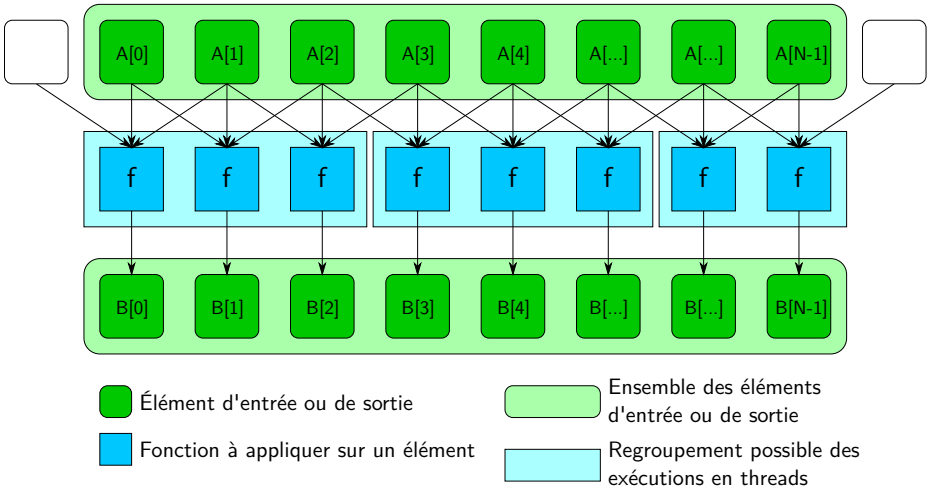
Examples

- Mandelbrot
- Raytracing
- Seuillage

- Map : réplication d'une fonction sur chaque élement d'un ensemble indexé
- Différentes granularités possibles (un pixel, une image, etc.)
- Le découpage en tâches peut être arbitraire ($\Rightarrow$ facile d'utiliser un framework)
- Conceptuellement :

```
for (int i = 0; i < N; i += 1) {
  B[i] = f(A[i]);
}
```

Exemple de parallélisation sur les données : Stencil



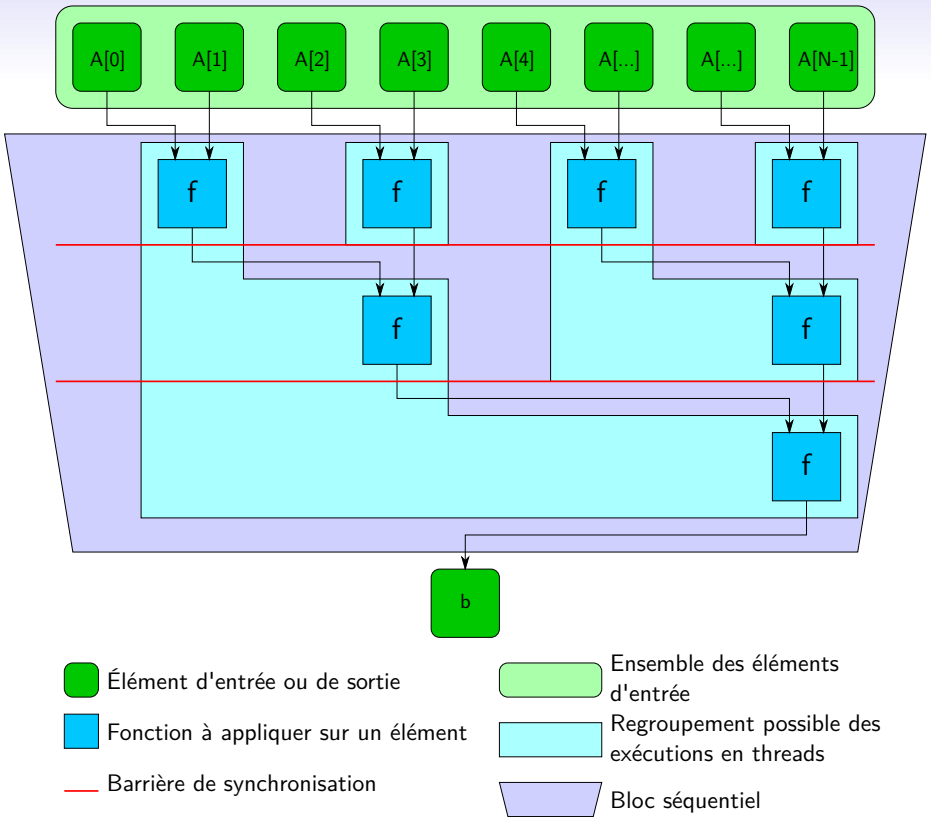
- Stencil : application d'une fonction à un voisinage
- En général, granularité fine (pixel)
- Là aussi, le découpage en tâches peut être arbitraire ($\Rightarrow$ facile d'utiliser un framework)
- Problème de la gestion des bords
- Conceptuellement :

Example

- Filtrage d'image

```
for (int i = 0; i < N; i += 1) {
  B[i] = f(A[i - 1], A[i], A[i + 1]);
}
```

Exemple de parallélisation sur les données avec synchro : Réduction



Examples

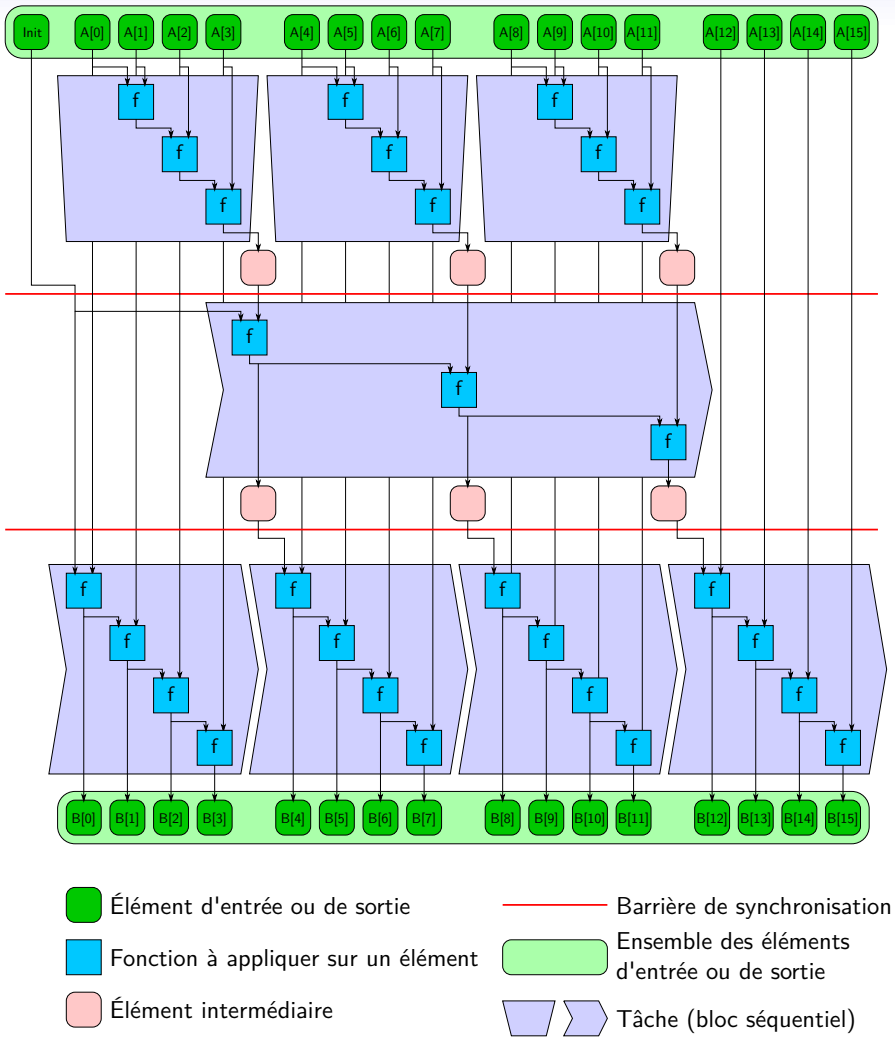
- Métriques d'image
- Opérations matricielles

- Réduction : combinaison de tous les éléments d'un ensemble dans un élement unique, en utilisant un opérateur associatif
- Conceptuellement :

```
int b = 0;
for (int i = 0; i < N; i += 1) {
  b = b + f(A[i]);
}
```

- L'associativité est requise pour permettre le parallélisme
- Gain maximal typique : $O(n) \rightarrow O(\log(n))$, mais le cout des synchronisations peut être très élevé
- $\Rightarrow$ En général, découpage en "blocs" pour avoir une quantité de travail par tâche conséquente comparé au cout de la synchronisation (ex : une tâche = le bloc complet sur la figure)

Exemple de parallélisation sur les données avec synchro : Scan



- Scan : calcul de toutes les réductions partielles
- Conceptuellement :

```
B[0] = A[0] + init;
for (i = 1; i < N; i += 1) {
  B[i] = A[i] + B[i - 1];
}
```

- Encore une fois, l'associativité est requise pour permettre le parallélisme
- L'exemple montre une implémentation parallèle possible en 3 phases (2 barrières)
- Existence de compromis entre le parallélisme et le cout de la synchronisation
- Exemple : génération de nombres aléatoires

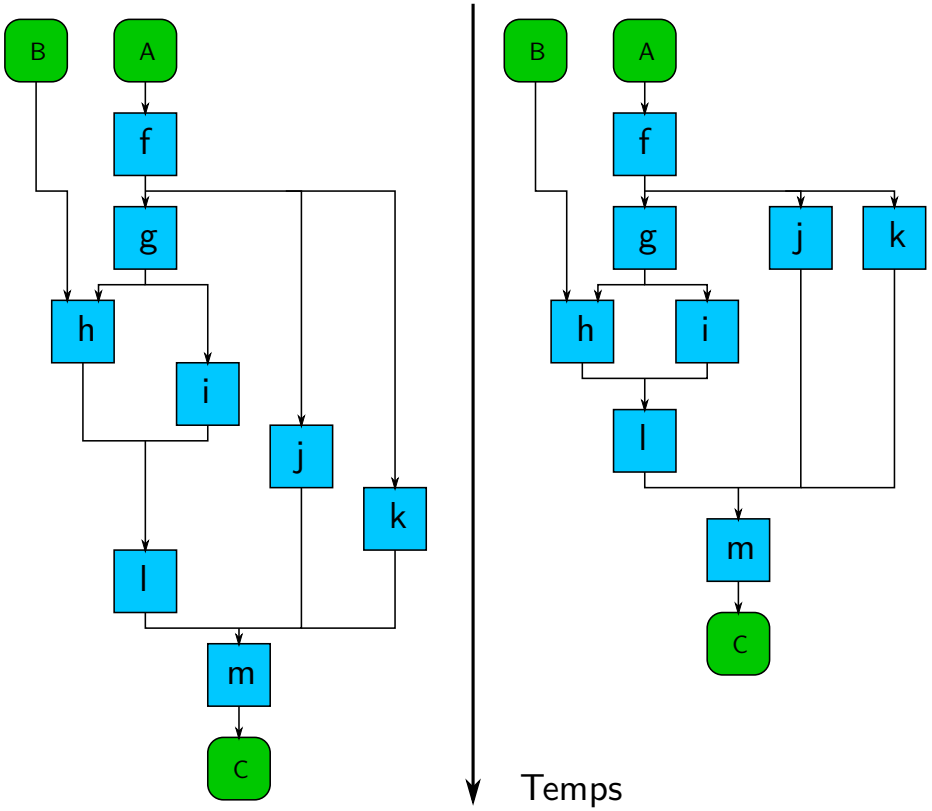
Parallélisation sur les données

- Pas toujours facile, car certaines dépendances ne sont pas aussi triviales
- En particulier, quand besoin du résultat de traitement des élements précédents pour l'élément courant
- Exemple :

```
for (int i = 1; i < N; i += 1) {
  for (int j = 1; j < M; j += 1) {
    A[i][j] = f(A[i - 1][j],
                A[i][j - 1],
                A[i - 1][j - 1],
                B[i][j]);
  }
}
```

Parallélisation d'une application en tâches

- L'application est représentée comme un graphe de tâches uniques avec des dépendances entre les tâches
- Exemple :

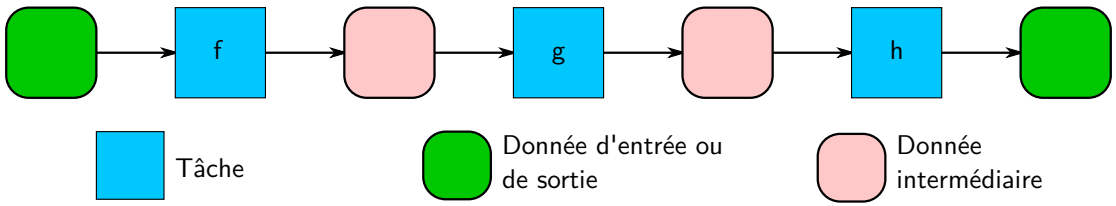



```
F = f(A);
G = g(F);
H = h(B, G);
I = i(G);
J = j(F);
K = k(F);
L = l(H, I);
C = m(L, J, K);
```

- Le parallélisme est possible entre plusieurs tâches n'ayant pas de dépendances entre elles
- Facilement réalisable dans un framework
- Malheureusement, il est rare d'avoir des graphes très "larges", en général beaucoup de dépendances séquentielles

Parallélisation pipeline d'une application en tâches

- Si les tâches sont cycliques (ex : traitement d'un flux d'images), il est possible de réaliser un pipeline composé des tâches
- Chaque thread est associé à une unique tâche (fonction) et l'exécute en boucle
- À un moment donné, chaque tâche est en cours d'exécution, mais traite des données différentes (par exemple, des images différentes ou des macro-blocs différents)
- Il faut que les tâches avancent "à la même vitesse" pour que le pipeline soit efficace
- + Besoin d'un moyen de synchronisation entre les tâches
- Exemple :

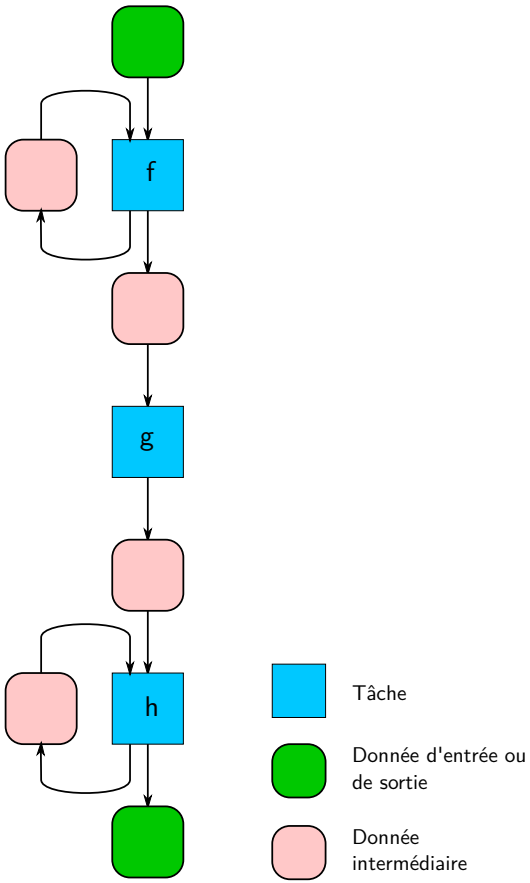


- Problèmes :
  - Parallélisation souvent complexe
  - Le nombre de threads est limité par le nombre de tâches
- Mais dans certains cas, il est possible d'instancier plusieurs fois la même tâche, ou d'avoir plusieurs pipelines en parallèle

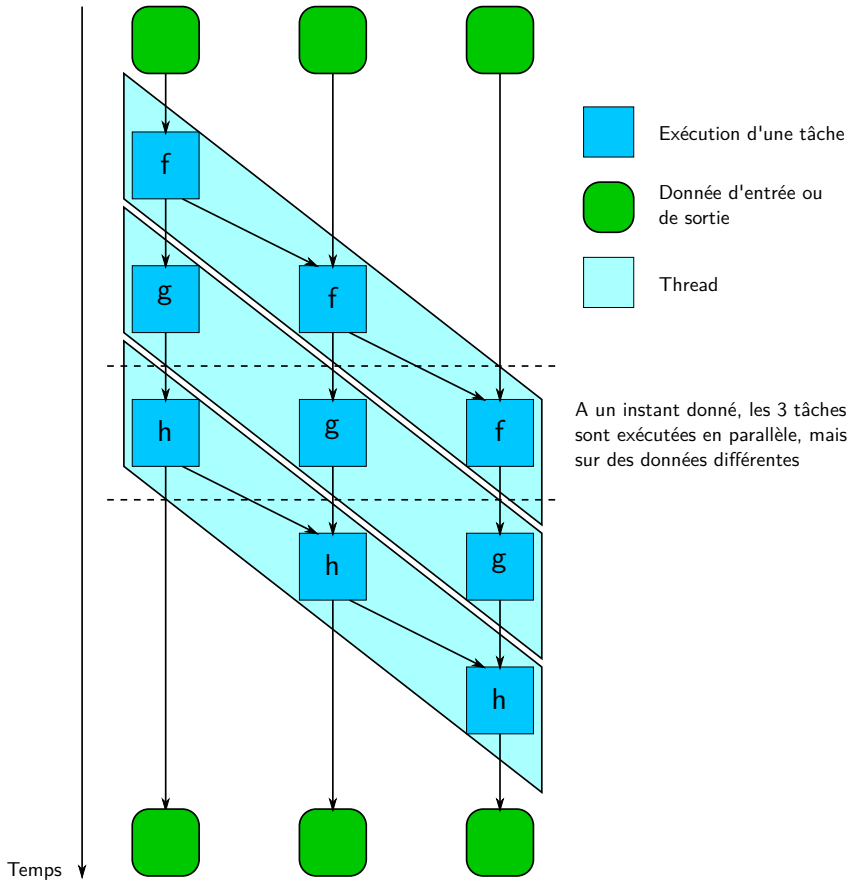
Parallélisation pipeline d'une application en tâches

- Les tâches peuvent aussi avoir des dépendances avec elles-mêmes
- Exemple d'un graphe de tâche et d'une vue de son exécution, dans laquelle chaque tâche est mappée sur un thread

- Graphe de tâche :

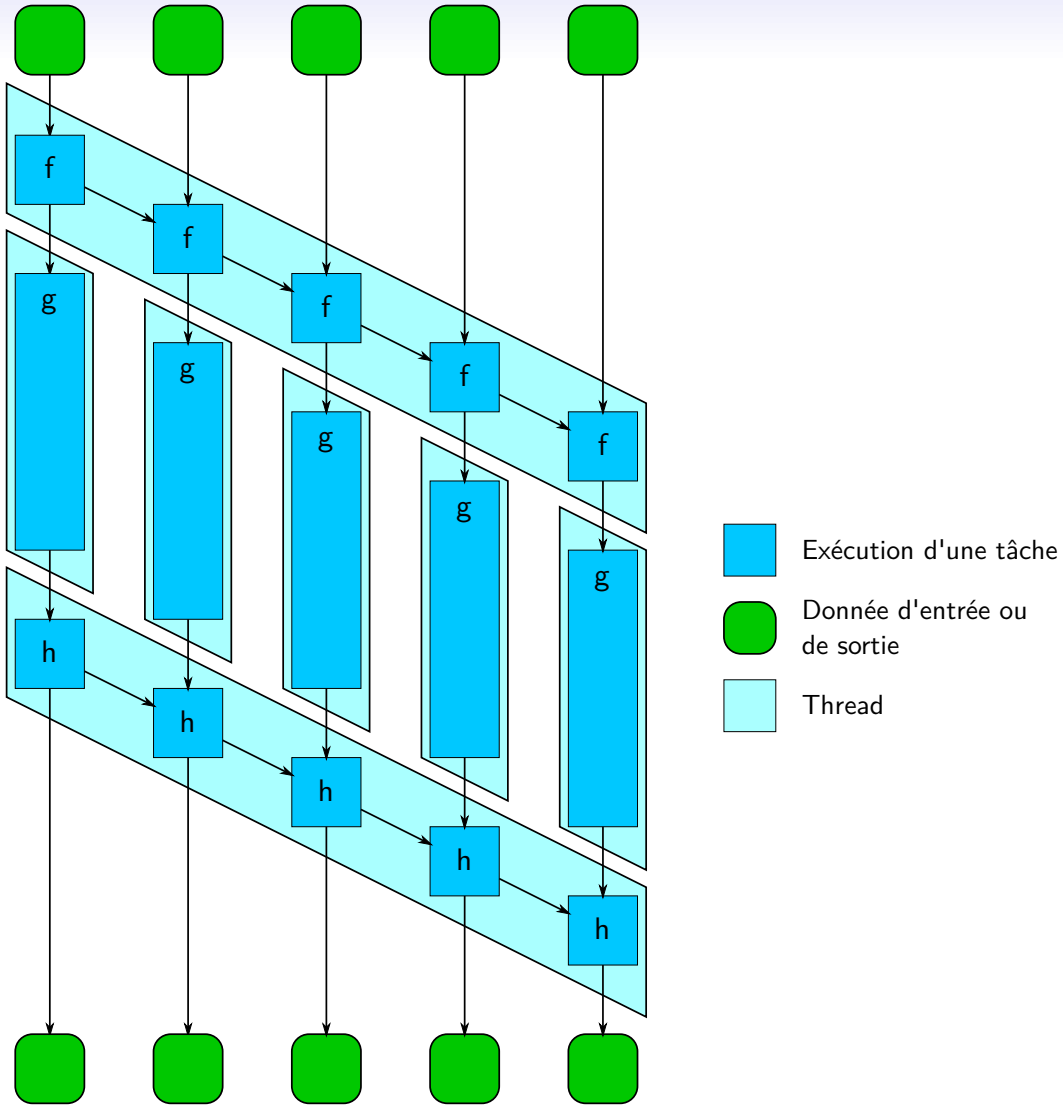


- Vue temporelle :



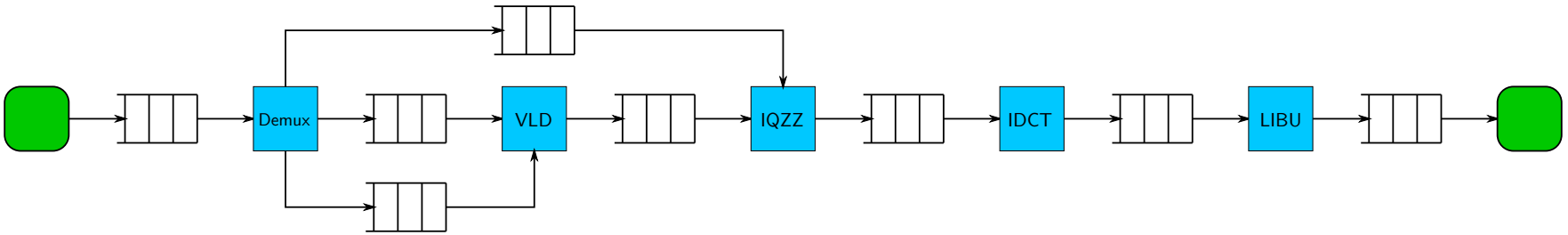
Parallélisation pipeline d'une application en tâches

- Si une tâche n'a pas d'état (i.e. de dépendance avec elle-même), possibilité d'en lancer plusieurs instances en parallèle
- Le tâches ayant des dépendances avec elles-mêmes ne peuvent avoir qu'une instance
- Utile dans l'exemple précédent si la tâche g est beaucoup plus longue que `f` et `h`
- Les sorties des différentes instances doivent être lues dans le bon ordre
- Jusqu'à 6 tâches exécutées en parallèle dans cet exemple
- Chacun des 5 threads de g représenté boucle sur une exécution de g (le 5e peut être le premier)



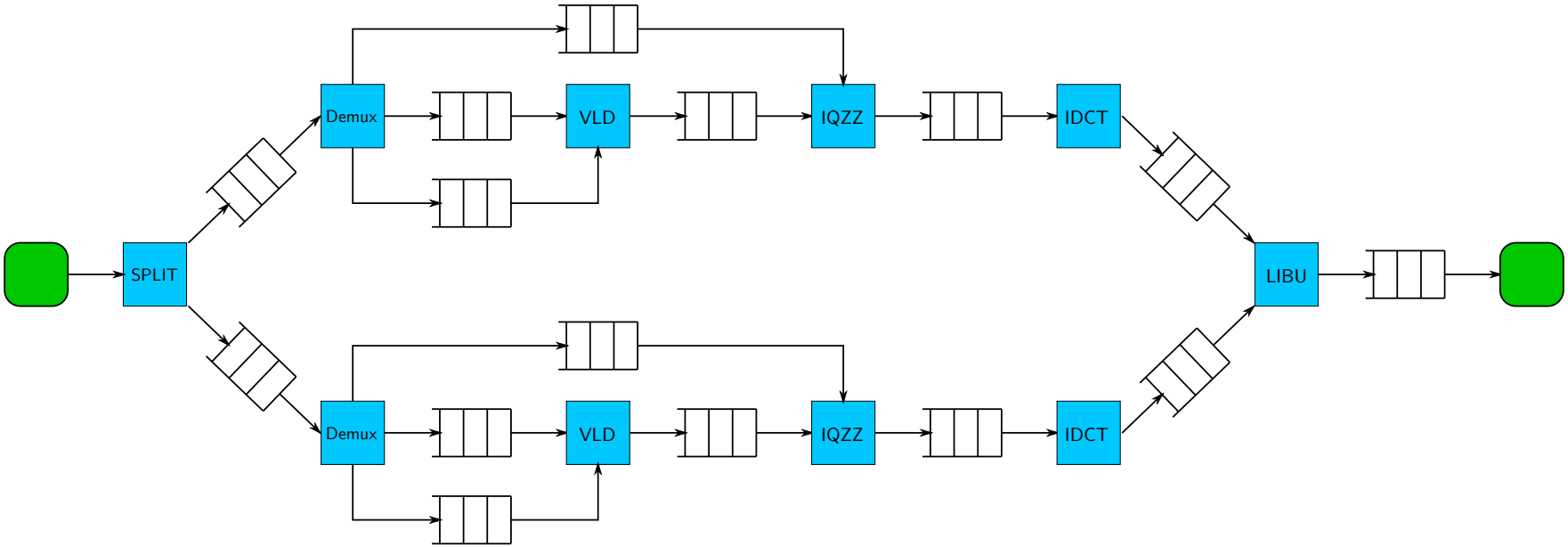
Parallélisation pipeline d'une application en tâches : Exemple de MJPEG

- Les exemples vus avant restent conceptuels et la réalité peut être plus compliquée
- Caractéristiques de l'application MJPEG :
  - Certaines tâches produisent des données vers plusieurs tâches différentes
  - Les "durées de vie" de ces données sont différentes (un bloc de 8 $\times$ 8 pixel ou une image complète)
  - Pour la gestion des synchronisations entre tâches, mise en place de canaux de communication (de type fifo), qui peuvent absorber les variations de vitesse entre tâche liées à la granularité de production/consommation/traitement (en général approche adoptée pour toutes les applications pipeline)
  - $\Rightarrow$ Les tâches sont désynchronisées, avec une limite quand les canaux sont pleins
- Graphe de tâche de l'application :



Parallélisation pipeline d'une application en tâches : Exemple de MJPEG

- Possibilité d'avoir plusieurs pipelines en parallèle, avec une granularité image
- Nécessite l'ajout d'une tâche qui dispatche alternativement les images sur les 2 chaines
- Nécessite aussi de modifier la tâche LIBU pour lire alternativement une image de chaque chaine
- Impose des contraintes sur la taille des canaux si on veut tirer parti du parallélisme...
- Graphe de tâche de l'application avec 2 pipelines :



Frameworks de tâches

Généralités

- Un “framework” est un environnement de description des tâches qui réalise une partie de la parallélisation de manière automatique, en général le mapping des tâches sur les threads, et peut fournir des primitives de synchronisation entre tâches de plus haut niveau que la mémoire partagée (échange de messages, communication par canaux de types fifo)
- Chaque framework suppose un cadre plus ou moins précis sur ce qu'il est possible de décrire (plus il est restrictif, plus l'automatisation est possible) ; par exemple, uniquement des applications de type pipeline communiquant par canaux
- Le code (C) des tâches doit toujours être fourni
- Le nombre de threads peut être fixé au début ou varier de manière dynamique
- Le placement des threads sur les coeurs peut-être spécifié par l'utilisateur ou délégué à l'OS
- Exemples de frameworks (plus ou moins connus) : OpenMP, MPI, Cilk, Kaapi, openstream, AWS, DSX, ...
- Jusqu'à OpenMP 4, tout le monde refaisait un peu son framework...

Framework de tâches : exemples de runtimes

Application pipeline

- Pour chaque modèle de tâche, description des modèles de tâches et avec ses entrées/sorties
- Description de l'instanciation des tâches (plusieurs instances d'un même modèle possible) et de ses communications avec les autres instances
- Mapping sur les coeurs automatique ou à la main
- Primitives pour communiquer par canaux fifo entre tâches
- Le runtime comprend un scheduler, qui deschedule une tâche lorsqu'elle ne peut plus lire/écrire dans ses canaux

Application tâches avec dépendances

- Les threads sont créés au début et forment un *pool*
- Un runtime affecte une tâche disponible à un thread en attente, jusqu'à ce qu'il n'y ait plus de threads en attente, ou plus de tâche disponible
- Quand une tâche est finie, mise à jour de la disponibilité des tâches restantes selon les dépendances

Framework de tâches : exemples de runtimes

Application avec tâches dynamiques

- Initialement, un seul thread dans l'application
- À un certain point, possibilité de *spawner* un autre thread pour traiter 2 cas différents (ex : explorer les 2 branches d'un arbre)
- Utile surtout pour les algorithmes récursifs, comme l'exploration d'arbres, ou de configurations
- Problème : beaucoup de créations et de destructions de threads (couteux)

Application avec tâches dynamiques et nombre de threads fixe

- Même principe que précédemment, mais cette fois-ci le nombre de threads est fixé au début
- Lorsqu'un thread veut créer une tâche, il crée une description pour cette tâche et l'ajoute dans une file partagée
- Quand un thread a fini une tâche, il va consulter la file pour prendre la tâche suivante

OpenMP

Définition de OpenMP

- **Open Multi-Processing** API (Application Progamming Interface)
- Standard pour la programmation d'applications parallèles sur architectures à mémoire partagée

Caractéristiques principales

- Programmation basée sur les threads
- Directives pour les opérations vectorielles (OpenMP 4.X)
- Directives pour les accélérateurs matériels (OpenMP 4.X)

Avantages

- Standard mature et répandu
- Effort de programmation réduit
- portable (supporté par gcc et llvm)

Note : ces slides sont majoritairement repris d'une présentation de Cédric Bastoul

Principe

Ajout de directives pour indiquer au compilateur

- Quelles sont les instructions à exécuter en parallèle
- Comment distribuer les instructions et les données entre différents threads

Remarques

- En général, le fait d'ignorer les directives ne change pas la sémantique du programme
- La détection ou l'extraction du parallélisme est à la charge du programmeur

Exemple canonique : produit scalaire (séquentiel)

```c
#include <stdio.h>

#define SIZE 256

int main() {
  double sum;
  double a[SIZE];
  double b[SIZE];

  // Initialisation
  sum = 0.0;
  for (int i = 0; i < SIZE; i += 1) {
    a[i] = i * 0.5;
    b[i] = i * 2;
  }

  for (int i = 0; i < SIZE; i += 1) {
    sum += a[i] * b[i];
  }
  printf("sum = %g\n", sum);
  return 0;
}
```

Ici, canonique = le plus favorable à OpenMP

Exemple canonique : produit scalaire (MPI)

```c
#include <stdio.h>
#include "mpi.h"

#define SIZE 256

int main(int argc, char * argv[]) {
  int np; // number of threads
  int my_rank;
  int my_first;
  int my_last;

  double sum;
  double local_sum;
  double a[SIZE];
  double b[SIZE];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &
      my_rank);

  my_first = my_rank * SIZE / np;
  my_last = (my_rank + 1) * SIZE / np
      ;

  // Initialisation
  local_sum = 0.0;
  for (int i = 0; i < SIZE; i += 1) {
    a[i] = i * 0.5;
    b[i] = i * 2;
  }

  for (int i = 0; i < SIZE; i += 1) {
    local_sum += a[i] * b[i];
  }
  MPI_Allreduce(&local_sum, &sum, 1,
      MPI_DOUBLE, MPI_SUM,
      MPI_COMM_WORLD);

  if (my_rank == 0) {
    printf("sum = %g\n", sum);
  }
  MPI_Finalize();

  return 0;
}
```

Exemple canonique : produit scalaire (Pthreads)

```c
#include <stdio.h>
#include <pthread.h>

#define SIZE 256
#define NTH 4 // NUM_THREADS
#define C (SIZE / NTH) // CHUNK

int id[NTH];
double sum;
double a[SIZE];
double b[SIZE];
pthread_t tid[NTH];
pthread_mutex_t mutex;

void * prod_scal(void * id) {
  int first = *(int *) id * C;
  int last = (*(int *) id + 1) * C;
  double local_sum = 0.0;
  for (int i = first; i < last;
       i += 1) {
    local_sum += a[i] * b[i];
  }
  pthread_mutex_lock(&mutex);
  sum += local_sum;
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

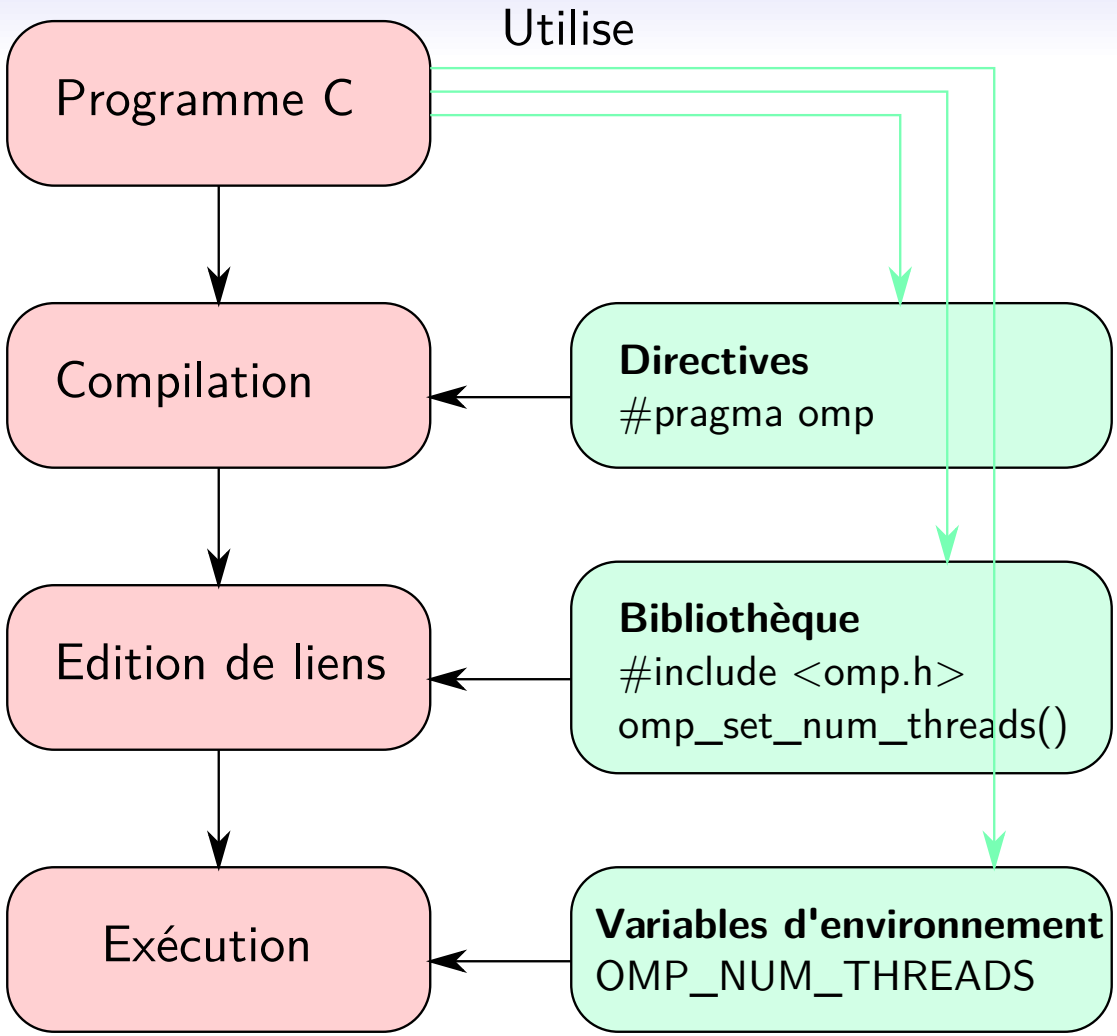
```c
int main() {
  // Initialisation
  sum = 0.0;
  for (int i = 0; i < SIZE; i += 1) {
    a[i] = i * 0.5;
    b[i] = i * 2;
  }
  pthread_mutex_init(&mutex, NULL);

  for (int i = 1; i < NTH; i += 1) {
    pthread_create(&tid[i], NULL,
        prod_scal, &id[i]);
  }
  id[0] = 0;
  prod_scal(&id[0]);
  for (int i = 1; i < NTH; i += 1) {
    pthread_join(tid[i], NULL);
  }
  pthread_mutex_destroy(&mutex);

  printf("sum = %g\n", sum);
  return 0;
}
```

Exemple canonique : produit scalaire (OpenMP)

```c
#include <stdio.h>

#define SIZE 256

int main() {
  double sum;
  double a[SIZE];
  double b[SIZE];

  // Initialisation
  sum = 0.0;
  for (int i = 0; i < SIZE; i += 1) {
    a[i] = i * 0.5;
    b[i] = i * 2;
  }

  #pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < SIZE; i += 1) {
    sum += a[i] * b[i];
  }
  printf("sum = %g\n", sum);
  return 0;
}
```

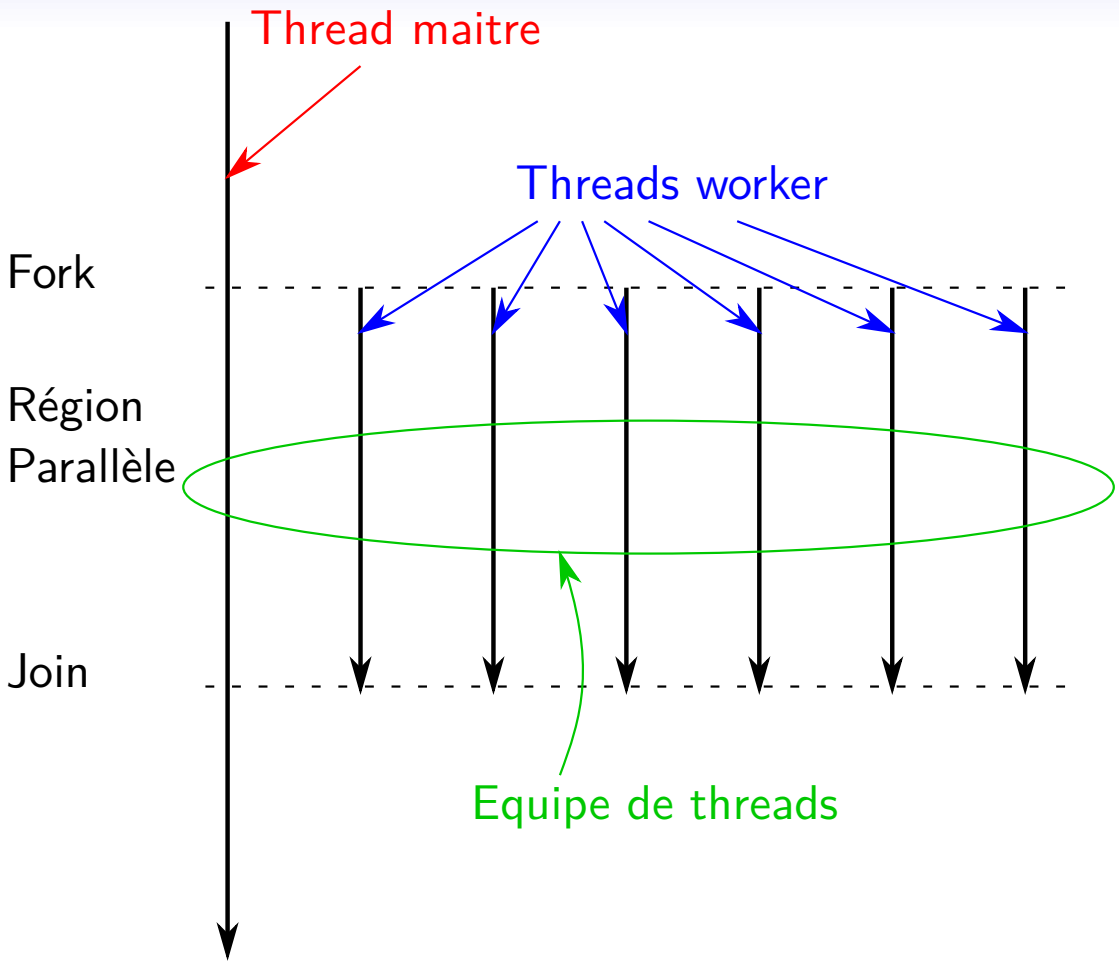
OpenMP API

- **Directives** pour expliciter le parallélisme, les synchronisations et le statut des données (privées, partagées)
- **Bibliothèque** de fonctions pour obtenir des informations à l'exécution et en contrôler certains aspects (ex : nombre de threads, ordonnancement)
- **Variables d'environnement** pour influer sur le programme à exécuter (même type d'informations que les fonctions de la bibliothèque)



Modèle d'exécution de OpenMP

- Les directives dans le code définissent des **régions parallèles**
- Durant l'exécution, le comportement respecte le modèle **fork-join** :
  - Le thread maitre crée des threads esclaves (ou *worker*) et forme une équipe avec eux
  - Les threads worker se terminent à la fin de la région parallèle
  - Le thread maitre continue son exécution
    - En réalité, les threads worker ne sont pas créés à chaque fois...



Modèle mémoire de OpenMP

Modèle à mémoire partagée

- Comme pour les pthreads, le modèle mémoire est celui de la mémoire partagée
- Les variables globales sont accessibles par tous les threads

Mais cas qui ne se pose pas en pthread : les variables locales

- Celles-ci peuvent être privées ou partagées (par défault : partagées)
- Si une variable est partagée, elles peut être lue et écrite par tous les threads (nécessite des mécanismes de synchro en plus pour éviter les race conditions, comme pour une variable globale)
- Si une variable est privée, c'est comme s'il s'agissait d'une nouvelle variable pour la section parallèle, chaque thread accédant à sa variable (la variable locale de la fonction originale n'est pas modifiée)

Principales directives de OpenMP

Construction de régions parallèles

- `parallel` : crée une région parallèle sur le modèle fork-join

Définition du code à partager

- `for` : partage les itérations d'une boucle parallèle
- `sections` : définit des blocs à exécuter en parallèle

Synchronisation (dans une région parallèle)

- `single` : déclare un bloc à exécuter par un seul thread
- `master` : déclare un bloc à exécuter par le thread maitre
- `critical` : déclare un bloc qui ne doit être exécuté que par un thread à la fois (section critique)
- `atomic` : déclare une instruction de lecture-écriture atomique (peut utiliser une primitive comme CAS plutôt qu'un lock)
- `barrier` : Barrière pour tous les threads

Directives de OpenMP

Format d'une directive

```
#pragma omp directive [clause [clause ...] ...]
```

Une directive est formée

- De l'élément de préprocessing (macro cpp) `#pragma omp`
- D'un nom de directive
- D'une liste de clauses optionnelles

Remarques

- Une directive s'applique sur la première instruction de la ligne suivante
  - Mais un bon code ne devrait jamais contenir plus d'une instruction par ligne... donc on peut dire qu'une directive s'applique sur la ligne suivante
  - Il s'agit d'une instruction au sens de la grammaire du C ; ne pas oublier la règle `INST` $\rightarrow$ `BLOCK` (comme pour les `if`)
- Comme pour les autres macros de cpp, une directive OpenMP peut s'étendre sur plusieurs ligne, en utilisant le caractère '\' en fin de ligne

La directive parallel (1/2)

Format de la directive `parallel`

```
#pragma omp parallel [clause [clause ...] ...]
{
  // Région parallèle
}
```

Fonctionnement

- Quand un thread rencontre une directive `parallel`, il crée un équipe de threads dont il devient le thread maitre de numéro 0 ; les threads de l'équipe exécutent tous l'instruction (ou le bloc)
- Le nombre de threads est déterminé, dans l'ordre, par : la clause `if`, la clause `num_threads`, le dernier appel à la primitive `omp_set_num_threads()`, la valeur de la variable d'environnement `OMP_NUM_THREADS`, la valeur par défaut.
- Rappel : il existe une barrière implicite à la fin de chaque région parallèle

La directive parallel (2/2)

Remarques

- Si une région parallèle contient un appel de fonction, les variables locales de cette fonction sont privées
- Il est interdit d'effectuer des branchements (`goto`) depuis ou vers une région parallèle
  - Mais aucun code (ou presque) ne devrait contenir de `goto`...
- Clauses possibles : `if`, `num_threads`, `private`, `shared`, `default`, `firstprivate`, `reduction`, `copyin`

La clause if

Format de la clause `if`

```
if (Expression)
```

- La clause `if` permet de conditionner le fait qu'une région soit exécutée de manière parallèle
- La région est exécutée séquentiellement si `Expression` est évaluée à 0, parallèlement sinon
- `Expression` est une expression du même type que les expressions après les `#if` de cpp

Exemple d'utilisation de la clause if

```
#include <stdio.h>
#define PARALLEL 1 // 0 pour séquentiel, != 0 pour parallèle

int main() {
  #pragma omp parallel if (PARALLEL)
  printf("Hello\n");
  printf("World!\n");
  return 0;
}
```

La clause num_threads

Format de la clause `num_threads`

```
num_threads (Expression entiere)
```

- Spécifie le nombre de threads de l'équipe qui exécutera la région parallèle
- `Expression entiere` doit s'évaluer en une expression entière positive

Les directives de partage de travail

- Directives `for` et `sections`
- Sont incluses dans les régions parallèles
- Doivent être rencontrées par tous les threads ou aucun
- Impliquent une barrière à la fin de la construction (sauf si la clause `nowait` est spécifiée), mais pas au début

Directive for

- Répartit les itérations d'une boucle parallèle

Directive sections

- Répartit entre les threads des sections (blocs) spécifiés de manière explicite, chaque section est exécutée par un thread

La directive for

Format de la directive `for`

```
#pragma omp for [clause [clause ...] ...]
```

- Doit précéder une boucle for

- Indique que les itérations de la boucle qui suit la directive doivent être exécutée en parallèle par l'équipe de threads
- La variable d'itération est privée par défaut
- Les boucles doivent avoir une forme itérative simple, notamment :
  - Les bornes doivent être les mêmes pour les threads
  - Les boucles infinies ou `while` ne sont pas supportées
- Le programmeur est responsable de la sémantique, en particulier du fait qu'il n'y a pas de dépendances de données entre les itérations
- Clauses possibles : `nowait`, `schedule`, `ordered`, `firstprivate`, `lastprivate`, `reduction`, `collapse`

La clause nowait

Format de la clause nowait

`nowait`

- Enlève la barrière implicite en fin de construction de partage de travail (`for`, `sections`) et `single`
- Les threads finissant en avance peuvent exécuter les instructions suivantes sans attendre les autres
- Le programmeur doit s'assurer que la sémantique du programme est préservée
  - Peut nécessiter une directive `barrier` un peu plus tard...

La clause schedule

Format de la clause schedule

`schedule(type [, chunk])`

- Spécifie la politique de partage des itérations
- 5 valeurs possibles pour `type` :
  - `static` : les itérations sont divisées en paquets de `chunk` itérations consécutives ; les paquets sont assignés aux threads en round-robin ; si chunk n'est pas précisé, des paquets de tailles similaires sont créés (un par thread)
  - `dynamic` : chaque thread prend un paquet de `chunk` itérations consécutives dès qu'il n'a pas de travail (le dernier paquet peut être plus petit) ; si `chunk` n'est pas précisé, il vaut 1
  - `guided` : Même chose que `dynamic`, mais la taille des paquets décroit de manière exponentielle (augmenter le parallélisme à la fin en minimisant le nombre de demande de paquets)
  - `runtime` : Le choix de la politique est reporté au moment de l'exécution, par exemple par la variable d'environnement `OMP_SCHEDULE`
  - `auto` : Le choix de la politique est laissé au compilateur et/ou au runtime
- Le choix de la politique peut être crucial pour les performances

La directive sections

Format de la directive sections

```
#pragma omp sections [clause [clause ...] ...]
{
  #pragma omp section
  // Bloc 0
  ...
  #pragma omp section
  // Bloc n
  ...
}
```

- Indique que les instructions dans les différentes sections doivent être exécutées en parallèle par l'équipe de threads : ce ne sont pas des sections critiques (races conditions entre les accès aux variables globales)
- Chaque section n'est exécutée qu'une seule fois, mais plusieurs sections peuvent être exécutées par le même thread (même si le nombre de sections est égal au nombre de threads)
- Le contenu du bloc après `sections` ne peut être rempli que d'instructions (blocs) comportant la directive `section` (sinon, quelle sémantique ?)
- Clauses possibles : `private`, `firstprivate`, `lastprivate`, `shared`, `reduction`, `nowait`

La directive single

Format de la directive single

```
#pragma omp single [clause [clause ...] ...]
{
  // Bloc
}
```

- Spécifie que le l'instruction suivante sera exécutée par un seul thread
- On ne peut pas prévoir quel thread exécutera le bloc
- Doit être rencontrée par tous les threads ou aucun
- Implique une barrière à la fin de la construction
- Utile pour les parties de code non thread-safe (exemple : lecture d'un fichier)
- Clauses possibles : `private`, `firstprivate`, `copyprivate`, `shared`, `nowait`

Raccourcis parallel for/parallel sections

Format de la directive parallel for

```
#pragma omp parallel for [clause [clause ...] ...]
for (...) {
   ...
}
```

Format de la directive parallel sections

```
#pragma omp parallel sections [clause [clause ...] ...]
{
   #pragma omp section
   // Bloc 0
   ...
   #pragma omp section
   // Bloc n
   ...
}
```

- Ces directives créent une région parallèle avec une seule construction parallèle
- Clauses possibles : toutes celles possibles pour les deux directives correspondantes, sauf `nowait` (n'aurait pas de sens)

Directives orphelines

- L'influence d'une région parallèle porte sur le bloc de code qui la suit directement (étendue statique) et sur les fonctions appelées dedans (étendue dynamique)
- Possible de rencontrer une directive en dehors de l'étendue statique ; une telle directive est appelée **directive orpheline**
- Une directive orpheline est considérée si elle est liée à une région parallèle au moment de son exécution, et ignorée sinon
- Dans l'exemple ci-contre, le `for` est bien exécuté de manière parallèle

omp for orphelin

```
#define SIZE 1024

void init(int v[SIZE]) {
  #pragma omp for
  for (int i = 0; i < SIZE; i += 1) {
    v[i] = 0;
  }
}

int main() {
  int vec[SIZE];
  #pragma omp parallel
  init(vec);
  return 0;
}
```

Directives imbriquées

Possible d'imbriquer des régions parallèles

- L'implémentation peut ignorer les régions internes
- Le niveau d'imbrication est a priori arbitraire
- Peu utile en général ; à éviter

Exemple de directives imbriquées

```
#include <stdio.h>
#include <stdbool.h>
#include <omp.h>

int main() {
  omp_set_nested(true);
  #pragma omp parallel num_threads(2)
  {
    #pragma omp parallel num_threads(2)
    printf("Hello world\n");
  }
  return 0;
}
```

Clauses de statut des variables

- OpenMP cible les architectures à mémoire partagée ; la plupart des variables sont donc partagées par défaut
- On peut contrôler le statut de partage des données :
  - Quelles données des sections séquentielles sont transférées dans les régions parallèles
  - Quelles données seront visibles par tous les threads ou privées à chaque thread
- Principale clauses :
  - `private` : définit une liste de variables privées au thread
  - `firstprivate` : `private` avec initialisation
  - `lastprivate` : `private` avec mise à jour finale
  - `shared` : définit une liste de variables partagées
  - `default` : définit le statut par défaut
  - `reduction` : définit une liste de variables à réduire (selon la définition de réduction vue précédemment)

Clause private

Format de la clause private

```
private(<liste de variables>)
```

- Définit une liste de variables à placer en mémoire privée
- Il n'y a pas de lien entre une variable de la liste à l'intérieur de la région parallèle et la variable à l'extérieur de la région parallèle
- Toutes les références dans la région parallèle sont vers les variables privées
- Un thread ne peut pas accéder aux variables privées d'un autre thread
- La valeur de début d'une variable privée est indéfinie au début de la région parallèle
- La valeur d'une variable privée à la fin de la région parallèle est perdue

Clause firstprivate

Format de la clause firstprivate

```
firstprivate(<liste de variables>)
```

- Combine le comportement de la clause `private` avec une initialisation des variables à l'entrée de la région parallèle
- Les variables sont initialisées avec la valeur de la variable correspondante à cet endroit du programme

Clause lastprivate

Format de la clause lastprivate

```
lastprivate(<liste de variables>)
```

- Combine le comportement de la clause `private` avec une mise à jour des variables originales à la fin de la région parallèle
- Les variables sont mises à jour avec la valeur de la variable privée du thread qui exécute la dernière itération d'une boucle, ou la dernière section par rapport à l'exécution séquentielle

Clause shared

Format de la clause shared

```
shared(<liste de variables>)
```

- Définit une liste de variables partagées, i.e. accessible en lecture et en écriture par tous les threads (comme pour les variables globales avec les pthreads)
- Tous les threads d'une même équipe peuvent accéder aux variables partagées simultanément (sauf si une directive OpenMP l'interdit, comme `atomic` ou `critical`)
- Les modifications d'une variable partagée sont visibles par tous les threads de l'équipe (mais pas toujours immédiatement, sauf si une directive OpenMP le précise, comme `flush`)

Clause default

Format de la clause default

`default(shared|none)`

- Permet à l'utilisateur de changer le statut par défaut des variables de la région parallèle (hors variables locales des fonctions appelées)
- Choisir `none` impose au programme de spécifier le statut de chaque variable
  - Permet d'éviter de partager des variables par erreur

Clause reduction

Format de la clause reduction

```
reduction(<operateur>: <liste de variables>)
```

- Réalise une réduction sur les variables de la liste
- Une copie privée de chaque variable dans la liste est créée pour chaque thread ; à la fin de la région, l'opérateur de réduction est appliqué aux variables privées et le résultat est écrit dans la variable du programme correspondante
- `<operateur>` peut valoir : `+, -, *, &, |, ^, &&, ||`
- Les variables dans la liste doivent être partagées
- Attention à la stabilité numérique (variables flottantes)
- Les variables de la liste ne peuvent être utilisées que dans des instructions de forme particulière (voir standard OpenMP, page 167)

Directive threadprivate

Format de la directive threadprivate

```
// Déclaration des variables globales et statiques
#pragma omp threadprivate(<liste de variables globales ou statiques>)
```

- Spécifie que les variables listées seront privées et persistantes à chaque thread au travers de l'exécution de toutes les régions parallèles
- La valeur des variables au début de la première région parallèle n'est pas définie, sauf si la clause `copyin` est utilisée
- Ensuite, les variables sont préservées
- La directive doit suivre la déclaration des variables globales ou statique concernées
- Le nombre de threads doit être fixe (`omp_set_dynamic(false)`)
- Clauses possibles : aucune

Clause copyin

Format de la clause copyin

```
copyin(<liste de variables déclarées threadprivate>)
```

- Spécifie que les valeurs des variables `threadprivate` du thread maitre présentes dans la liste devront être copiées dans les variables privées correspondantes des threads worker au début de la région parallèle
- Rappel : cette clause s'applique à la directive `parallel`

Clause copyprivate

Format de la clause copyprivate

```
copyprivate(<liste de variables déclarées threadprivate>)
```

- Demande la copie des valeurs des variables privées d'un thread dans les variables privées correspondantes des autres threads d'une même équipe
- Utilisable seulement avec la directive `single`
- La copie a lieu après l'exécution du bloc associé à la directive single et avant la sortie de barrière de fin de bloc
- Ne peut pas être utilisée conjointement à la directive `nowait`

Mécanismes de synchronisation

- **Barrière** : attendre que tous les threads aient atteint un point donné de l'exécution avant de continuer
  - Implicite en fin de construction OpenMP (sans `nowait`)
  - Directive `barrier`
- **Ordonnancement** : garantir un ordre global d'exécution
  - Clause `ordered`
  - Sert surtout pour le debug
- **Exclusion mutuelle** : s'assurer qu'une seule tâche à la fois exécute une portion de code
  - Directive `critical`
  - Directive `atomic`
- **Attribution** : affecter un traitement à un thread donné
  - Directive `master`
- **Verrou** : parce qu'OpenMP grandissant, il y a de plus en plus de demande pour le faire évoluer vers plus de souplesse (et donc une écriture de programme proche des pthreads)
  - Fonctions de la bibliothèque OpenMP

Directive barrier

Format de la directive barrier

```
#pragma omp barrier
```

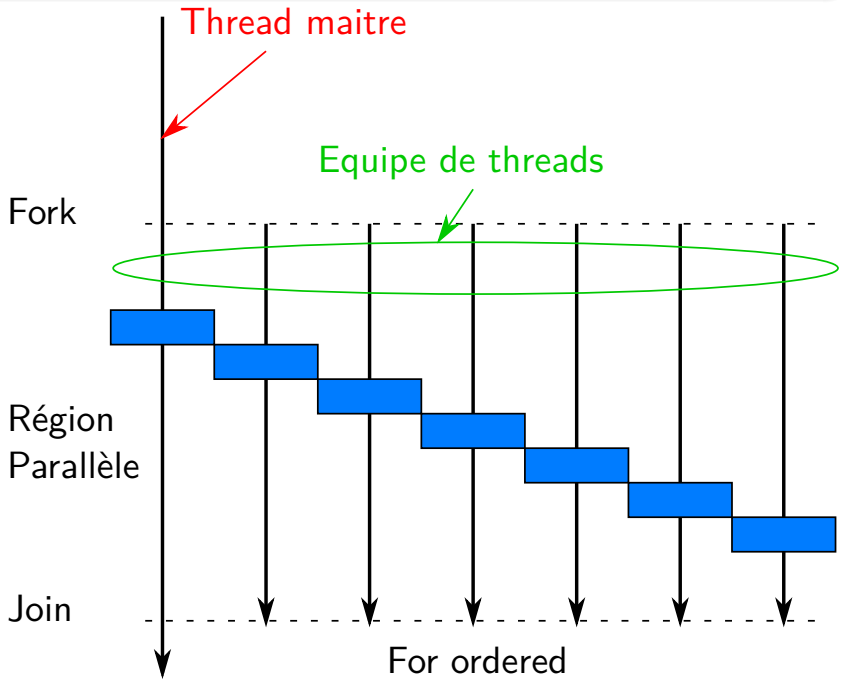
- Synchronisation entre tous les threads d'une équipe
- Doit être rencontrée par tous les threads ou aucun (sinon : deadlock)
- Clauses possibles : aucune

Directive ordered

Format de la directive ordered

```
#pragma omp ordered
```

- Seulement dans un `for` où l'on a indiqué la clause `ordered`
- Spécifie que les exécutions du bloc suivant la directive devront respecter l'ordre séquentiel
- Les threads s'attendent si besoin pour respecter cet ordre
- Les parties de la boucle non gardées par cette directive peuvent s'exécuter en parallèle



Directive critical

Format de la directive critical

```
#pragma omp critical [nom]
{
  // Bloc
}
```

- Spécifie que le bloc d'instructions suivant la directive doit être exécuté un seul thread à la fois
- Si un thread exécute un bloc protégé par la directive `critical` et qu'un second arrive à ce bloc, alors le second devra attendre que le premier ait terminé avant de commencer l'exécution du bloc
- Les blocs précédés de directives `critical` avec un même nom sont exécutés en exclusion mutuelle ($\Leftrightarrow$ lock identique)
- Clauses possibles : aucune

Directive atomic

Format de la directive atomic

```
#pragma omp atomic
...  = ... ; // instruction d'affectation
```

- Spécifie que l'affectation (évaluation et écriture de la variable affectée) suivant la directive doit être réalisée de manière atomique
- Plus efficace que `critical` dans ce cas
- Formes d'instruction particulières : voir standard OpenMP pour les détails
- Clauses possibles : aucune

Directive master

Format de la directive master

```
#pragma omp master
{
  // Bloc
}
```

- Spécifie que le bloc d'instructions suivant la directive sera exécuté par le seul thread maitre, les autres threads passent cette section de code
- Il n'y a pas de barrière implicite ni à l'entrée ni à la fin du bloc
- Clauses possibles : aucune

Directive flush

Format de la directive flush

```
#pragma omp flush(<Liste de variables partagées>)
```

- Instruction de barrière mémoire dans OpenMP
- Implicite après une région parallèle, un partage de travail (hors `nowait`), une section critique ou un verrou
- Je ne vois personnellement aucun cas où cela est utile en OpenMP
- Clauses possibles : aucune