UNIVERSITÉ DE STRASBOURG

OpenMP

Cédric Bastoul

cedric.bastoul@unistra.fr

Université de Strasbourg

Références

Ce cours est librement basé sur les supports suivants, qu'on utilisera parfois directement :

- Standard OpenMP 4.0

  www.openmp.org/mp-documents/OpenMP4.0.0.pdf

- Cours OpenMP Oracle de R. van der Pas

  www.openmp.org/mp-documents/ntu-vanderpas.pdf

- Cours OpenMP IDRIS de J. Chergui et P.-F. Lavallée

  www.idris.fr/data/cours/parallel/openmp/IDRIS_OpenMP_cours.pdf

- Tutoriel OpenMP de B. Barney

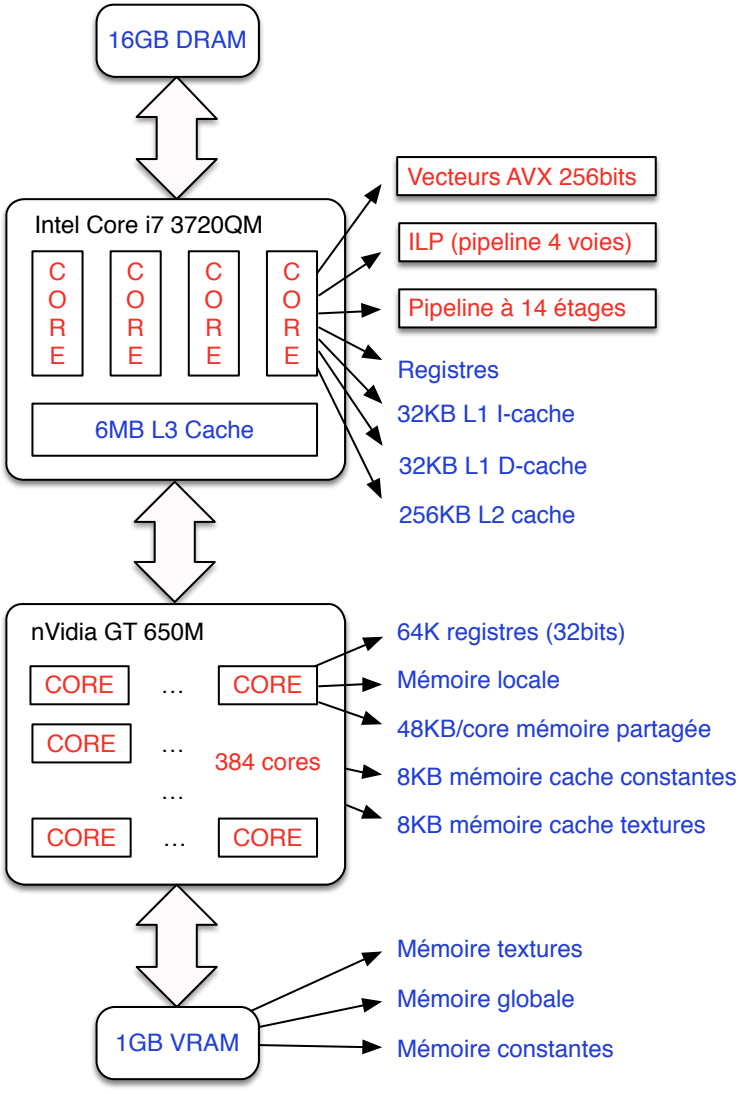
  computing.llnl.gov/tutorials/openMP

- Livre de B. Chapman, G. Jost et R van der Pas *Using OpenMP* MIT Press, 2008

Un mot sur la situation actuelle

Cet ordinateur :

- 2.7 milliards de transistors
- 5 types de parallélisme
- Au moins 4 modèles de programmation + APIs
- 15 types de mémoire



« Open Multi-Processing » API (Application Programming Interface) standard pour la programmation d'applications *parallèles* sur *architectures à mémoire partagée*

- Programmation basée sur les threads
- Directives pour les opérations vectorielles (OpenMP 4.0)
- Directives pour les accélérateurs matériels (OpenMP 4.0)

- Standard industriel mature et répandu
- Bonne performance si on s'y prend bien
- Effort de programmation minimal
- Portable

Historique

Architectures SMP (« Symmetric Multiprocessing » ou « Shared Memory multiProcessor ») depuis l'IBM System/360 modèle 67 (1966) programmées d'abord par des directives ad hoc

- 1991 Le groupe industriel Parallel Computing Forum définit un ensemble de directives pour le parallélisme de boucles pour Fortran (jamais normalisé)
- 1997 OpenMP 1.0 pour Fortran, C/C++ en 1998
- 2000 OpenMP 2.0 parallélisation de constructions de Fortran 1995
- 2008 OpenMP 3.0 concept de tâche
- 2013 OpenMP 4.0 SIMD, accélérateurs, etc.

Principe

Ajouter au code des directives pour indiquer au compilateur :

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

- En général, les directives sont facultatives (le programme est sémantiquement équivalent avec ou sans)
- La détection ou l'extraction du parallélisme est laissée à la charge du programmeur
- L'impact sur le code original séquentiel est souvent minime
- Mais pour gagner en performance il faut travailler un minimum

Positionnement

Principaux modèles de programmation parallèle :

- Architectures à mémoire partagée
  - *Intrinsics* instructions vectorielles assembleur (Intel SSE2, ARM NEON) très bas niveau
  - *Posix Threads* bibliothèque standardisée, bas niveau
  - ***OpenMP*** API standard de fait
  - *CUDA* plateforme propriétaire pour accélérateurs
  - *OpenCL* API et langage pour SMP + accélérateurs
- Architectures à mémoire distribuée
  - *Sockets* bibliothèque standardisée, bas niveau
  - ***MPI*** Message Passing Interface, bibliothèque standard de fait pour les architectures à mémoire distribuée (fonctionne aussi sur les SMP), remaniement important du code

Exemple : produit scalaire séquentiel [code]

```c
#include <stdio.h>
#define SIZE 256

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

  // Initialization
  sum = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  for (i = 0; i < SIZE; i++)
    sum = sum + a[i]*b[i];

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

Exemple : produit scalaire MPI

[code]

```
#include <stdio.h>
#include "mpi.h"
#define SIZE 256

int main(int argc, char* argv[]) {
  int i, numprocs, my_rank, my_first, my_last;
  double sum, sum_local, a[SIZE], b[SIZE];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  my_first = my_rank * SIZE/numprocs;
  my_last = (my_rank + 1) * SIZE/numprocs;

  // Initialization
  sum_local = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  for (i = my_first; i < my_last; i++)
    sum_local = sum_local + a[i]*b[i];
  MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

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

Exemple : produit scalaire Pthreads [code]

```c
#include <stdio.h>
#include <pthread.h>
#define SIZE 256
#define NUM_THREADS 4
#define CHUNK SIZE/NUM_THREADS

int id[NUM_THREADS];
double sum, a[SIZE], b[SIZE];
pthread_t tid[NUM_THREADS];
pthread_mutex_t mutex_sum;

void* dot(void* id) {
  int i;
  int my_first = *(int*)id * CHUNK;
  int my_last = (*(int*)id + 1) * CHUNK;
  double sum_local = 0.;

  // Computation
  for (i = my_first; i < my_last; i++)
    sum_local = sum_local + a[i]*b[i];

  pthread_mutex_lock(&mutex_sum);
  sum = sum + sum_local;
  pthread_mutex_unlock(&mutex_sum);
  return NULL;
}
```

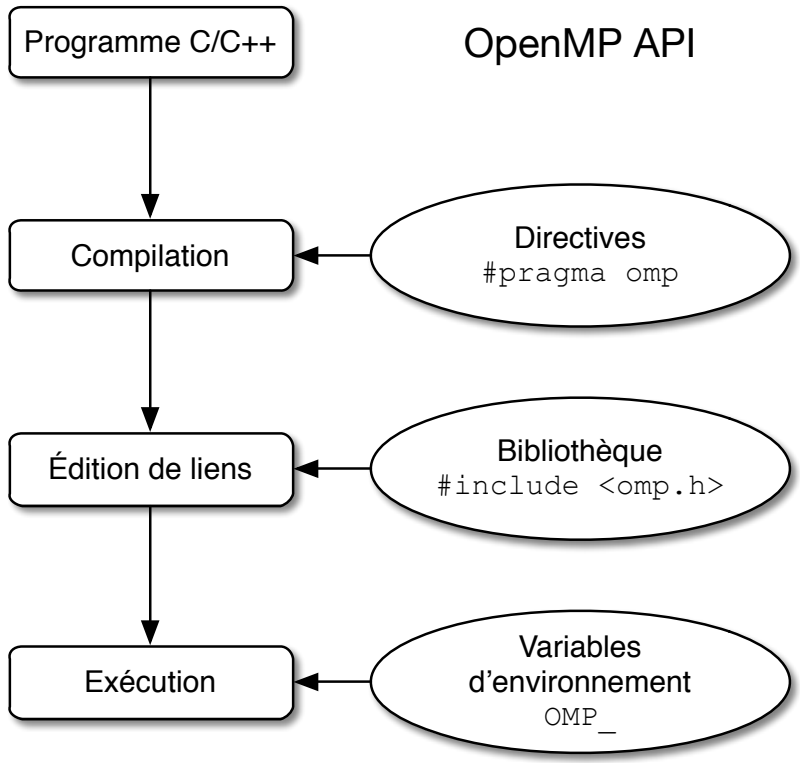
```c
int main() {
  int i;

  // Initialization
  sum = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  pthread_mutex_init(&mutex_sum, NULL);

  for (i = 0; i < NUM_THREADS; i++) {
    id[i] = i;
    pthread_create(&tid[i], NULL, dot,
                   (void*)&id[i]);
  }

  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

  pthread_mutex_destroy(&mutex_sum);

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

Exemple : produit scalaire OpenMP [code]

```
#include <stdio.h>
#define SIZE 256

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

  // Initialization
  sum = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  #pragma omp parallel for reduction(+:sum)
  for (i = 0; i < SIZE; i++) {
    sum = sum + a[i]*b[i];
  }

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

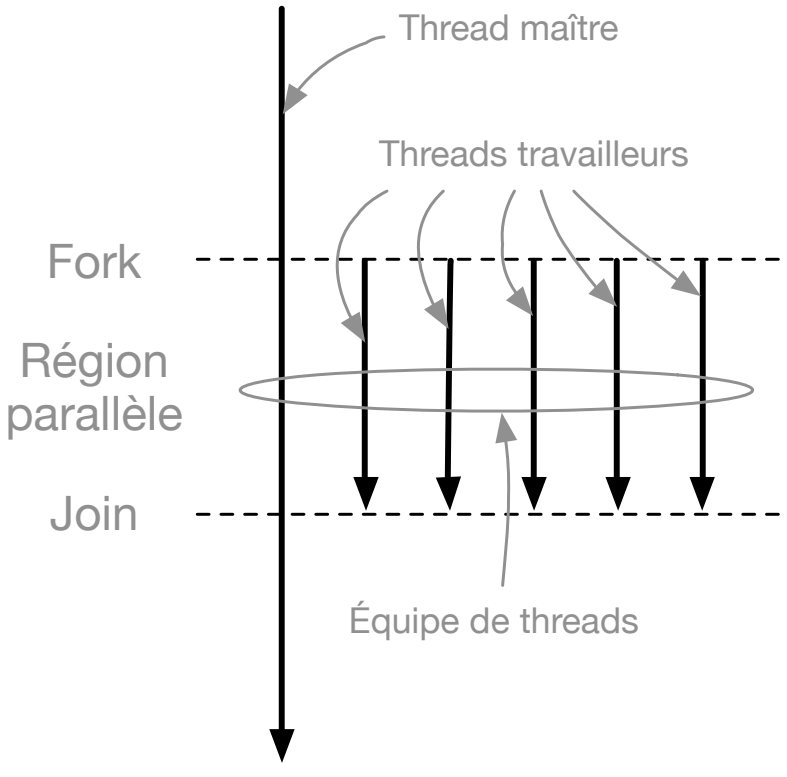
OpenMP API

1. *Directives* pour expliciter le parallélisme, les synchronisations et le statut des données (privées, partagées...)
2. *Bibliothèque* pour des fonctionnalités spécifiques (informations dynamiques, actions sur le runtime...)
3. *Variables d'environnement* pour influer sur le programme à exécuter (nombre de threads, stratégies d'ordonnancement...)



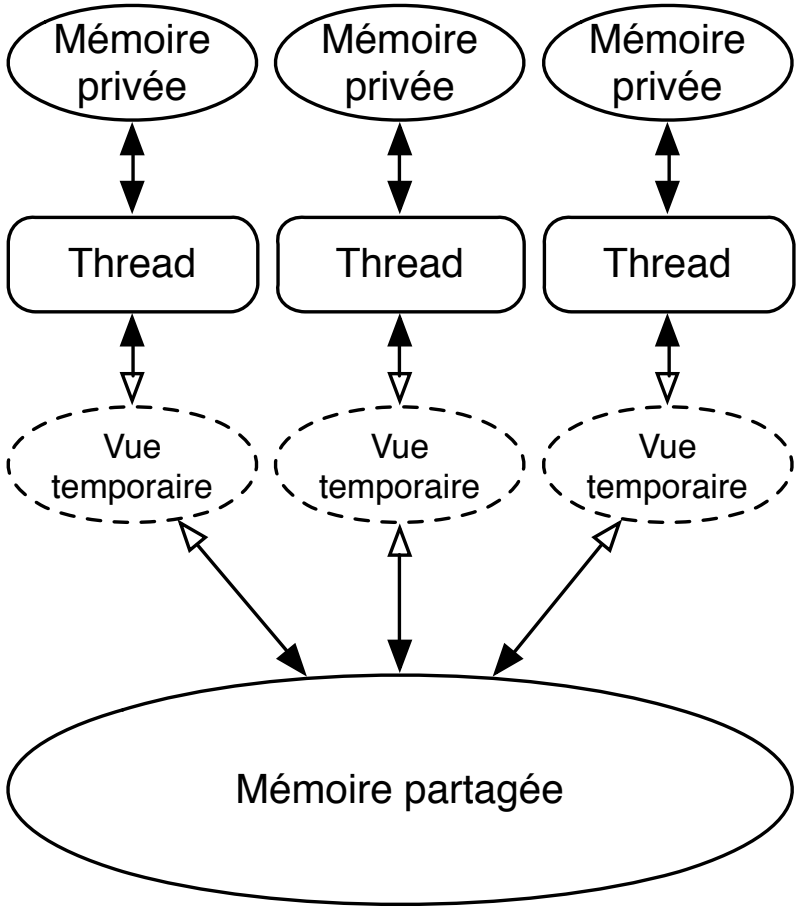
Modèle d'exécution OpenMP

1. L'utilisateur introduit des directives établissant des *régions parallèles*
2. Durant l'exécution, leur comportement respecte le modèle *fork-join* :
   - Le thread maître crée des threads travailleurs et forme une équipe avec eux
   - Les threads travailleurs se terminent avec la région parallèle
   - Le thread maître continue son exécution



Modèle mémoire OpenMP

- Tous les threads ont accès à la même *mémoire partagée*
- Chaque thread possède sa propre *mémoire privée*
- Les données partagées sont accessibles par tous les threads
- Les données privées sont accessibles seulement par le thread correspondant
- Les transferts de données sont transparents pour le programmeur



Directives OpenMP

Principales directives OpenMP

- Construction de régions parallèles
  - `parallel` : crée une région parallèle sur le modèle fork-join
- Partage du travail
  - `for` : partage des itérations d'une boucle parallèle
  - `sections` : définit des blocs à exécuter en parallèle
  - `single` : déclare un bloc à exécuter par un seul thread
- Synchronisation
  - `master` : déclare un bloc à exécuter par le thread maître
  - `critical` : bloc à n'exécuter qu'un thread à la fois
  - `atomic` : instruction dont l'écriture mémoire est atomique
  - `barrier` : attente que tous les threads arrivent à ce point
- Gestion de tâches
  - `task` : déclaration d'une tâche fille
  - `taskwait` : attente de la fin des tâches filles

Directives OpenMP

Format des directives en C/C++

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

Formées de quatres parties :

1. La sentinelle : `#pragma omp`
2. Un nom de directive valide
3. Une liste de *clauses* optionnelles (infos supplémentaires)
4. Un retour à la ligne

Règles générales :

- Attention à respecter la casse
- Une directive s'applique sur le bloc de code suivant
- Les directives longues peuvent se poursuivre à la ligne en utilisant le caractère antislash « \ » en fin de ligne

Directives OpenMP

Construction de régions parallèles

Directive `parallel` (1/2)

Format de la directive `parallel` en C/C++

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

Fonctionnement :

- Quand un thread rencontre une directive `parallel`, il crée une équipe de threads dont il devient le thread maître de numéro 0 ; les threads de l'équipe exécutent tous le bloc
- Le nombre de threads dépend, dans l'ordre, de l'évaluation de la clause `if`, de la clause `num_threads`, de la primitive `omp_set_num_threads()`, de la variable d'environnement `OMP_NUM_THREADS`, de la valeur par défaut
- Il existe une barrière implicite à la fin de la région parallèle

Directive `parallel` (2/2)

- Par défaut, le statut des variables est partagé dans la région parallèle
- Cependant, si la région contient des appels de fonction, leurs variables locales et automatiques sont de statut privé
- Il est interdit d'effectuer des branchements (`goto`) depuis ou vers une région parallèle
- Clauses possibles : `if`, `num_threads`, `private`, `shared`, `default`, `firstprivate`, `reduction`, `copyin`

Clause `if`

Format de la clause `if` en C/C++

```
if (/* Expression scalaire */)
```

- Quand elle est présente, une équipe de threads n'est créée que si l'expression scalaire est différente de zéro, sinon la région est exécutée séquentiellement par le thread maître

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, openMP\n");
   return 0;
}
```

Clause `num_threads`

Format de la clause `num_threads` en C/C++

```
num_threads(/* Expression entière */)
```

- Spécifie le nombre de threads à de l'équipe qui exécutera la prochaine région parallèle
- L'expression entière doit s'évaluer en une valeur entière positive

Exercice

Un premier exemple [code]

```
#include <stdio.h>

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

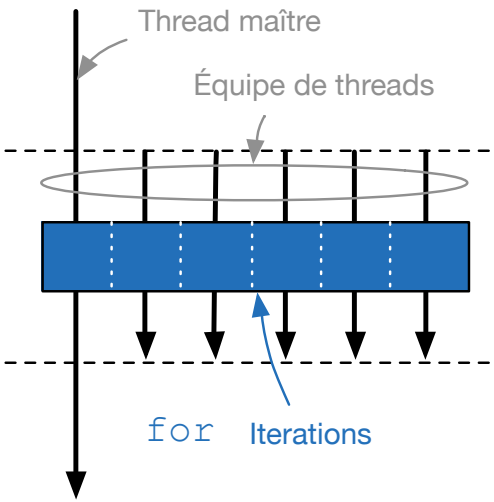
- Compilez ce programme avec et sans l'option `-fopenmp`
- Exécutez ce programme dans les deux cas
- Quel est le nombre de threads par défaut ? Est-ce raisonnable ?
- Changez le nombre de threads utilisé pour exécuter votre programme

Directives OpenMP

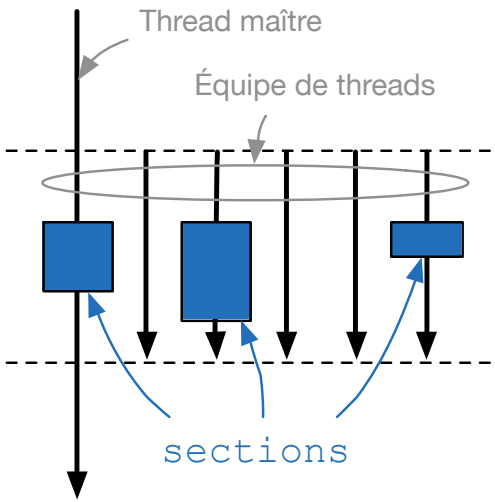
Partage de travail

Directives de partage de travail

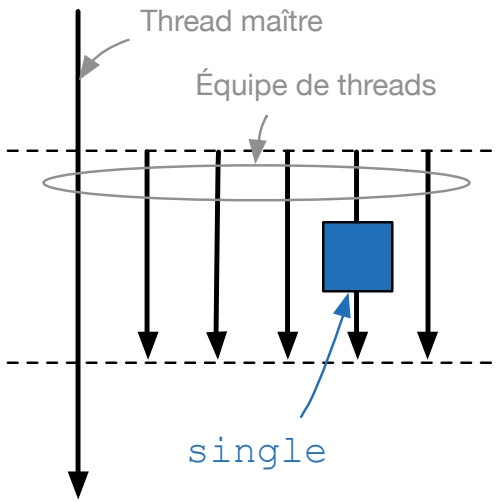
- Incluses dans les régions parallèles
- Doivent être rencontrées par tous les threads ou aucun
- Répartissent le travail entre les différents threads
- Impliquent une barrière à la fin de la construction (sauf si la clause `nowait` est spécifiée)



`for` : répartit les itérations d'une boucle parallèle



`sections` : répartit suivant des sections prédéfinies



`single` : un seul thread exécute une section prédéfinie

Directive `for`

Format de la directive `for` en C/C++

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

- Indique que les itérations de la boucle qui suit la directive doivent être exécutées en parallèle par l'équipe de threads
- La variable d'itération est privée par défaut
- Les bornes doivent être les mêmes pour tous les threads
- Les boucles infinies ou `while` ne sont pas supportées
- Rappel : le programmeur est responsable de la sémantique
- Clauses possibles : `schedule`, `ordered`, `private`, `firstprivate`, `lastprivate`, `reduction`, `collapse`, `nowait`

Exercice

- Écrire un programme C effectuant la somme de chaque élément d'un tableau et d'un scalaire dans un deuxième tableau
- Parallélisez ce programme avec OpenMP

Clause `schedule` (1/2)

Format de la clause `schedule` en C/C++

```
schedule(type[, chunk])
```

- Spécifie la politique de partage des itérations
- 5 types possibles :

static Les itérations sont divisées en blocs de `chunk` itérations consécutives ; les blocs sont assignés aux threads en round-robin ; si `chunk` n'est pas précisé, des blocs de tailles similaires sont créés, un par thread

dynamic Chaque thread demande un bloc de `chunk` itérations consécutives dès qu'il n'a pas de travail (le dernier bloc peut être plus petit) ; si `chunk` n'est pas précisé, il vaut 1

Clause `schedule` (2/2)

guided Même chose que `dynamic` mais la taille des blocs décroît exponentiellement ; si `chunk` vaut plus que 1, il correspond au nombre minimum d'itérations dans un chunk (sauf pour le dernier)

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 est critique pour les performances

Exercice (1/2)

Lisez, étudiez, compilez et exécutez le code suivant [code]

```
#include <stdio.h>
#include <omp.h>
#define SIZE  100
#define CHUNK  10

int main() {
  int i, tid;
  double a[SIZE], b[SIZE], c[SIZE];

  for (i = 0; i < SIZE; i++)
    a[i] = b[i] = i;

  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    if (tid == 0)
      printf("Nb threads = %d\n", omp_get_num_threads());
    printf("Thread %d: starting...\n", tid);

    #pragma omp for schedule(dynamic, CHUNK)
    for (i = 0; i < SIZE; i++) {
      c[i] = a[i] + b[i];
      printf("Thread %d: c[%2d] = %g\n", tid, i, c[i]);
    }
  }
  return 0;
}
```

Exercice (2/2)

- Analysez le programme : quelles sont les instructions exécutées par tous les threads ? Par un seul thread ?
- Exécutez le programme plusieurs fois. Que penser de l'ordre d'exécution des instructions ?
- Redirigez la sortie de l'exécutable sur l'utilitaire `sort`. Exécutez et observez la répartition des itérations.
- Recommencer plusieurs fois. La répartition est-elle stable ?
- Changer la politique d'ordonnancement par `static`. Exécutez plusieurs fois. La répartition est-elle stable ?
- Discutez des effets de la politique d'ordonnacement sur les performances.

Clause `collapse`

Format de la clause `collapse` en C/C++

```
collapse(/* Expression entière strictement positive */)
```

- Spécifie le nombre de boucles associées à une directive `for` (1 par défaut)
- Si l'expression entière vaut plus que 1, les itérations de toutes les boucles associées sont groupées pour former un unique espace d'itération qui sera réparti entre les threads
- L'ordre des itérations de la boucle groupée correspond à l'ordre des itérations des boucles originales

```
#pragma omp for collapse(2)
for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    f(i, j);
```

≡

```
#pragma omp for
for (i = 0; i < 100; i++)
  f(i/10, i%10);
```

Clause `nowait`

Format de la clause `nowait` en C/C++

nowait

- Retire la barrière implicite en fin de construction de partage de travail
- 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

Exercice (1/2)

Lisez, étudiez, compilez et exécutez le code suivant [code]

```
#include <stdio.h>
#define SIZE 100

int main() {
  int i;
  double a[SIZE], b[SIZE], c[SIZE], d[SIZE];

  for (i = 0; i < SIZE; i++)
    a[i] = b[i] = i;

  #pragma omp parallel
  {
    #pragma omp for schedule(static) nowait
    for (i = 0; i < SIZE; i++)
      c[i] = a[i] + b[i];

    #pragma omp for schedule(static)
    for (i = 0; i < SIZE; i++)
      d[i] = a[i] + c[i];
  }

  for (i = 0; i < SIZE; i++)
    printf("%g␣", d[i]);
  printf("\n");
  return 0;
}
```

Exercice (2/2)

- Exécutez le programme plusieurs fois. Les résultats semblent-ils incohérents ?
- Analysez le programme : quelles itérations vont être exécutées par quels threads (le détail de la politique `static` page 57 du standard OpenMP vous aidera) ?
- Après analyse, l'utilisation de la clause `nowait` vous semble-t-elle raisonnable ?
- Changez la politique d'ordonnancement pour la seconde boucle à `guided`.
- Exécutez le programme plusieurs fois. Les résultats semblent-ils incohérents ? Si vous n'avez pas vu le problème, cherchez mieux ;-) !

Directive `sections`

Format de la directive `sections` en C/C++

```
#pragma omp sections [clause [clause] ...]
{
  #pragma omp section
   // Bloc 1
   ...
  #pragma omp section
   // Bloc N
}
```

- Indique que les instructions dans les différentes sections doivent être exécutées en parallèle par l'équipe de threads
- Chaque section n'est exécutée qu'une seule fois
- Les sections doivent être définies dans l'étendue statique
- Clauses possibles : `private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`

Directive `single`

Format de la directive `single` en C/C++

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

- Spécifie que le bloc d'instructions suivant la directive sera exécuté par un seul thread
- On ne peut pas prévoir quel thread exécutera le bloc
- Utile pour les parties de code non thread-safe (par exemple les entrées/sorties)
- Clauses possibles : `private`, `firstprivate`, `copyprivate`, `nowait`

Raccourcis `parallel for/sections`

Format de la directive `parallel for` en C/C++

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

Format de la directive `parallel sections` en C/C++

```
#pragma omp parallel sections [clause [clause] ...]
{
  #pragma omp section
  // Bloc 1
  ...
  #pragma omp section
  // Bloc N
}
```

- Créent une région parallèle avec une seule construction
- Clauses possibles : union des clauses sauf `nowait`

Directives OpenMP

Clauses de statut des variables

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 et comment
  - Quelles données seront visibles par tous les threads ou privées à chaque thread
- Principales clauses :
  - `private` : définit une liste de variables privées
  - `firstprivate` : `private` avec initialisation automatique
  - `lastprivate` : `private` avec mise à jour automatique
  - `shared` : définit une liste de variables partagées
  - `default` : change le statut par défaut
  - `reduction` : définit une liste de variables à réduire

Clause `private`

Format de la clause `private` en C/C++

```
private(/* Liste de variables */)
```

- Définit une liste de variables à placer en mémoire privée
- Il n'y a pas de lien avec les variables originales
- Toutes les références dans la région parallèle seront vers les variables privées
- Un thread ne peut accéder les variables privées d'un autre thread
- Les modifications d'une variable privée sont visibles seulement par le thread propriétaire
- Les valeurs de début et de fin sont indéfinies

Exercice

Exemple d'utilisation de la clause `private` [code]

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
  int val;
  #pragma omp parallel //private(val)
  {
    val = rand();
    sleep(1);
    printf("My val : %d\n", val);
  }
  return 0;
}
```

- Compilez et exécutez ce code avec et sans `private(val)`
- Qu'observez-vous et pourquoi ?
- Est-ce risqué même avec la clause `private` ?

Clause `firstprivate`

Format de la clause `firstprivate` en C/C++

```
firstprivate(/* Liste de variables */)
```

- Combine le comportement de la clause `private` avec une initialisation automatique
- Les variables listées sont initialisées avec leur valeur au moment de l'entrée dans la région parallèle

Clause `lastprivate`

Format de la clause `lastprivate` en C/C++

```
lastprivate(/* Liste de variables */)
```

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

Clause `shared`

Format de la clause `shared` en C/C++

```
shared(/* Liste de variables */)
```

- Définit une liste de variables à placer en mémoire partagée
- Il n'y a qu'une instance de chaque variable partagée
- 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` en C/C++

```
default(shared | none)
```

- Permet à l'utilisateur de changer le statut par défaut des variables de la région parallèle (hors variables locales et automatiques des fonctions appelées)
- Choisir `none` impose au programmeur de spécifier le statut de chaque variable

Clause `reduction`

Format de la clause `reduction` en C/C++

```
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 construction, l'opérateur de réduction est appliqué aux variables privées et le résultat est écrit dans la variable partagée correspondante
- `operateur` peut valoir `+`, `-`, `*`, `&`, `|`, `^`, `&&` ou `||`
- Les variables dans la liste doivent être partagées
- Attention à la stabilité numérique
- La variable ne peut être utilisée que dans des instructions de forme particulière (voir standard OpenMP, page 167)

Exercice

- Écrire un programme C calculant la somme des éléments d'un tableau.
- Parallélisez ce programme avec OpenMP.
- Comparer le temps d'exécution séquentielle et le temps d'exécution parallèle.

Directive `threadprivate`

Format de la directive `threadprivate` en C/C++

```
// Déclaration de variables globales et/ou statiques
#pragma omp threadprivate(/* Liste de variables globales/statiques */)
```

- Spécifie que les variables listées seront privées *et persistantes* à chaque thread au travers de l'exécution de multiples régions parallèles
- La valeur des variables n'est pas spécifiée dans la première région parallèle sauf si la clause `copyin` est utilisée
- Ensuite, les variables sont préservées
- La directive doit suivre la déclaration des variables globales ou statiques concernées
- Le nombre de threads doit être fixe (`omp_set_dynamic(0)`)
- Clauses possibles : aucune

Exercice

Étudiez, exécutez ce code et discutez les résultats [code]

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

int tid, tprivate, rprivate;
#pragma omp threadprivate(tprivate)

int main() {
  // On interdit explicitement les threads dynamiques
  omp_set_dynamic(0);

  printf("Région parallèle 1\n");
  #pragma omp parallel private(tid, rprivate)
  {
    tid = omp_get_thread_num();
    tprivate = tid;
    rprivate = tid;
    printf("Thread %d: tprivate=%d rprivate=%d\n", tid, tprivate, rprivate);
  }

  printf("Région parallèle 2\n");
  #pragma omp parallel private(tid, rprivate)
  {
    tid = omp_get_thread_num();
    printf("Thread %d: tprivate=%d rprivate=%d\n", tid, tprivate, rprivate);
  }
  return 0;
}
```

Clause `copyin`

Format de la clause `copyin` en C/C++

```
copyin(/* Liste de variables déclarées threadprivate */)
```

- Spécifie que les valeurs des variables `threadprivate` du thread maître présentes dans la liste devront être copiées dans les variables privées correspondantes des threads travailleurs en début de région parallèle

Clause `copyprivate`

Format de la clause `copyprivate` en C/C++

```
copyprivate(/* Liste de variables déclarées private */)
```

- 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`
- Ne peut être utilisée en conjonction avec la clause `nowait`

Directives OpenMP

Synchronisation

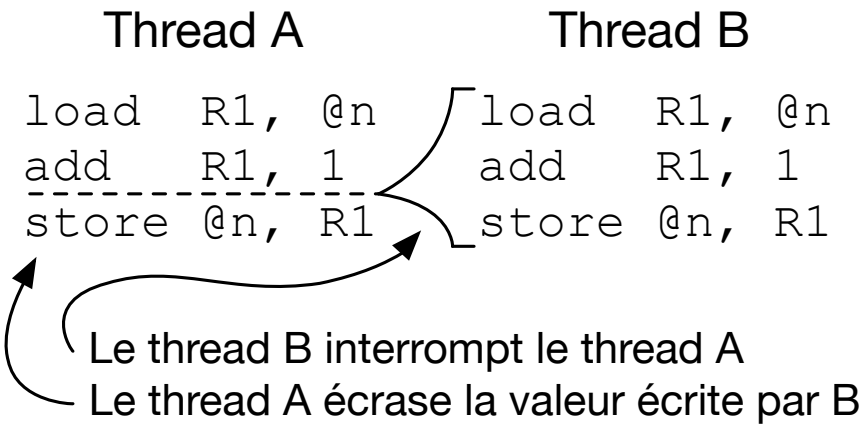
Erreur type : *data race*

Risque de data race [code]

```
#include <stdio.h>
#define MAX 1000

int main() {
  int i, n = 0;

  #pragma omp parallel for
  for (i = 0; i < MAX; i++)
    n++;

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

À l'exécution du programme ci-contre, la valeur affichée peut être inférieure à `MAX` :

- Accès concurents à `n`
- Incrémentation non atomique
- Incrémentations « perdues »



Erreur type : défaut de cohérence

Risque d'incohérence [code]

```
#include <stdio.h>

int main() {
  int fin = 0;

  #pragma omp parallel sections
  {
    #pragma omp section
    {
      while (!fin)
        printf("Pas fini\n");
    }
    #pragma omp section
    {
      fin = 1;
      printf("Fini\n");
    }
  }
  return 0;
}
```

À l'exécution du programme ci-contre, « `Pas fini` » peut être affiché après « `Fini` » :

- Interruption de la première section entre l'évaluation de `fin` et l'affichage (data race)
- Utilisation d'une vue temporaire obsolète de la mémoire partagée par le thread exécutant la première section (défaut de cohérence)

Erreur type : non synchronisation

Non synchronisation [code]

```
#include <stdio.h>
#include <omp.h>
int main() {
  double total, part1, part2;
  #pragma omp parallel \
              num_threads(2)
  {
    int tid;
    tid = omp_get_thread_num();
    if (tid == 0)
      part1 = 25;
    if (tid == 1)
      part2 = 17;
    if (tid == 0) {
      total = part1 + part2;
      printf("%g\n", total);
    }
  }
  return 0;
}
```

À l'exécution du programme ci-contre, la valeur affiché peut être différente de 42

- Le thread 0 n'attend pas le thread 1 pour faire le calcul et l'affichage

Mécanismes de synchronisation

- *Barrière* pour attendre que tous les threads aient atteint un point donné de l'exécution avant de continuer
  - Implicite en fin de construction OpenMP (hors `nowait`)
  - Directive `barrier`
- *Ordonnancement* pour garantir un ordre global d'exécution
  - Clause `ordered`
- *Exclusion mutuelle* pour assurer qu'une seule tâche à la fois exécute une certaine partie de code
  - Directive `critical`
  - Directive `atomic`
- *Attribution* pour affecter un traitement à un thread donné
  - Directive `master`
- *Verrou* pour ordonner l'exécution d'au moins deux threads
  - Fonctions de la bibliothèque OpenMP

Directive `barrier`

Format de la directive `barrier` en C/C++

```
#pragma omp barrier
```

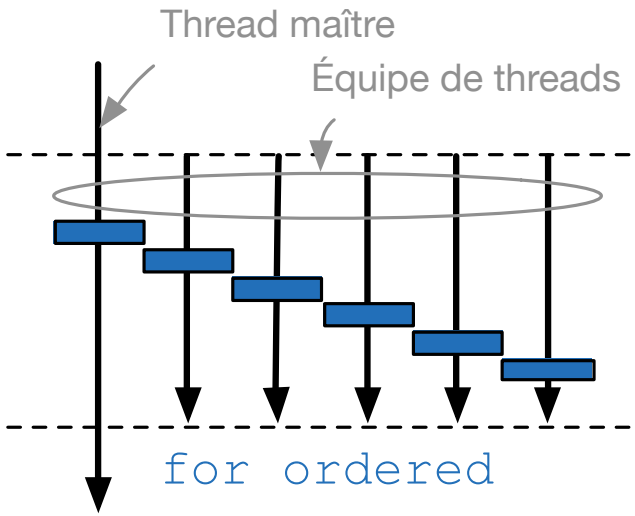
- Synchronisation entre tous les threads d'une équipe
- Quand un thread arrive à la directive `barrier` il attend que tous les autres threads y soient arrivés ; quand cela arrive, les threads poursuivent leur exécution en parallèle
- Doit être rencontrée par tous les threads ou aucun : attention aux deadlocks
- Clauses possibles : aucune

Clause `ordered`

Format de la directive `ordered` en C/C++

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

- 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 parallèle non gardées par cette directive peuvent s'exécuter en parallèle
- Il faut aussi indiquer la clause `ordered` à la construction `for`



Directive `critical`

Format de la directive `critical` en C/C++

```
#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
- Clauses possibles : aucune

Directive `atomic`

Format de la directive `atomic` en C/C++

```
#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 la directive `critical` dans ce cas
- Formes d'instruction particulières : voir standard OpenMP 4.0 page 127 pour les détails
- Clauses possibles : aucune

Directive `master`

Format de la directive `master` en C/C++

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

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

Directive `flush`

Format de la directive `flush` en C/C++

```
#pragma omp flush (/* Liste de variables partagées */)
```

- Spécifie que la vue temporaire du thread qui la rencontre doit être cohérente avec l'état de la mémoire partagée pour chaque variable de la liste
- Implicite après une région parallèle, un partage de travail (hors `nowait`), une section critique ou un verrou
- À faire après écriture dans un thread et avant lecture dans un autre pour partager une variable de manière cohérente
- Indispensable même sur un système à cohérence de cache
- Clauses possibles : aucune

Exercice

Corriger les codes donnés en exemple des erreurs type :

- Data race
- Défaut de cohérence
- Défaut de synchronisation

Exercice

- Récupérer les codes suivants (crédit Dominique Béréziat) : [code] et [code]
  - Calcule les points de l'ensemble de Mandelbrot
  - Enregistre le résultat sous forme d'image au format ras
- Lisez ce programme et étudiez son parallélisme
- Implémentez une (ou des) version(s) parallèle(s) avec OpenMP

Directives OpenMP

Gestion des tâches

Concept de tâche

Une tâche OpenMP est un bloc de travail indépendant qui devra être exécuté par un des thread d'une même équipe

- Les directives de partage de travail créent des tâches de manière implicite
- Il est possible de créer des tâches explicitement
  - Parallélisation de boucles non-itératives
  - Parallélisation de code récursif
- Un « pool » de tâches existe pour chaque région parallèle
- Un thread qui crée une tâche l'ajoute à ce pool ; ce ne sera pas nécessairement lui qui l'exécutera
- Les threads exécutent les tâches du pool à la première barrière qu'ils rencontrent

Directive `task`

Format de la directive `task` en C/C++

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

- Crée une nouvelle tâche composée du bloc d'instruction suivant la directive et de l'environnement de données au moment de la création de la tâche
- La tâche créée est soit exécutée immédiatement par le thread qui l'a créée (voir clauses), soit ajouté au pool
- Clauses possibles : `if` (sémantique différente de la directive `parallel`), `final`, `untied`, `default`, `mergeable`, `private`, `firstprivate`, `shared`

Clauses de la directive `task`

Format de la clause `if` en C/C++

```
if (/* Expression scalaire */)
```

- Exécution immédiate par le thread si vrai, pool si faux

Format de la clause `final` en C/C++

```
final (/* Expression scalaire */)
```

- Les sous-tâches seront intégrées à la tâche si vrai

Format de la clause `untied` en C/C++

```
untied
```

- Tout thread peut reprendre la tâche si elle est suspendue

Format de la clause `mergeable` en C/C++

```
mergeable
```

- La tâche est combinable si elle est immédiate ou intégrée

Directive `taskwait`

Format de la directive `taskwait` en C/C++

```
#pragma omp taskwait
```

- Spécifie un point d'attente de la terminaison de toutes les sous-tâches créées par le thread rencontrant la directive
- Il s'agît d'une barrière spécifique aux tâches
- Clauses possibles : aucune

Exercice : affichages possibles ? [code 1, 2, 3, 4]

```c
#include <stdio.h>
int main() {

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

  return 0;
}
```

```c
#include <stdio.h>
int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task
      printf("Hello,\n");
      #pragma omp task
      printf("world!\n");
    }
  }
  return 0;
}
```

```c
#include <stdio.h>
int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task
      printf("Hello,\n");
      #pragma omp task
      printf("world!\n");
      printf("Bye\n");
    }
  }

  return 0;
}
```

```c
#include <stdio.h>
int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task
      printf("Hello,\n");
      #pragma omp task
      printf("world!\n");
      #pragma omp taskwait
      printf("Bye\n");
    }
  }
  return 0;
}
```

Exercice

- Parallélisez ce code de calcul des nombres de Fibonacci
- Comparez les performances avec le code séquentiel

Calcul des termes de la suite de Fibonacci [code]

```
#include <stdio.h>
#include <stdlib.h>

int fibo(int n) {
  if (n < 2)
    return n;

  return fibo(n-1) + fibo(n-2);
}

int main(int argc, char* argv[]) {
  int n = atoi(argv[1]);
  printf("fibo(%d) = %d\n", n , fibo(n));
  return 0;
}
```

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*)

- Directives en dehors de l'étendue statique dites « orphelines »
- Liées à la région parallèle qui les exécute immédiatement
- Ignorées à l'exécution si non liées à une région parallèle

omp for orphelin [code]

```
#include <stdio.h>
#define SIZE 1024

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

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

Directives imbriquées

Il est possible d'imbriquer les régions parallèles

- L'implémentation peut ignorer les régions internes
- Niveau d'imbrication a priori arbitraire
- Attention aux performances

Directives imbriquées [code]

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

int main() {
  omp_set_nested(1);
  #pragma omp parallel num_threads(2)
  {
    #pragma omp parallel num_threads(2)
    printf("Hello, world\n");
  }
  return 0;
}
```

Bibliothèque OpenMP

Bibliothèque OpenMP

- Fonctions liées à l'environnement d'exécution
  - Modification du comportement à l'exécution (e.g., politique d'ordonnancement)
  - Surveillance du runtime (e.g., nombre total de threads, numéro de thread etc.)
- Fonctions utilitaires d'intérêt général, même en dehors de la parallélisation car portables
  - Mesure du temps
  - Mécanisme de verrou

Préserver l'indépendance à OpenMP

_OPENMP défini lorsque le compilateur prend en charge OpenMP

- Utiliser des directives préprocesseur pour écrire un code avec et sans support d'OpenMP
- Définir des macros pour chaque fonction OpenMP utilisée

Exemple de compilation conditionnelle [code]

```
#include <stdio.h>
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif

int main() {
  #pragma omp parallel
   printf("Hello from thread %d\n", omp_get_thread_num());
   return 0;
}
```

Fonctions affectant le runtime

`void omp_set_num_threads(int n);`	Fixe le nombre de threads pour la prochaine région parallèle à `n`
`void omp_set_dynamic(int bool);`	Active ou désactive l'ajustement automatique du nombre de threads
`void omp_set_nested(int bool);`	Active ou désactive le support de régions parallèles imbriquées
`void omp_set_max_active_levels(int n);`	Fixe le nombre maximum de régions parallèles imbriquables à `n`
`void omp_set_schedule(omp_sched_t type, int chunk);`	Fixe la politique l'ordonnancement quand `runtime` a été choisie à `type` (1 pour `static`, 2 pour `dynamic`, 3 pour `guided` ou 4 pour `auto`) et spécifie la valeur de `chunk`

Fonctions monitorant le runtime 1/2

`int omp_get_num_threads();`	Retourne le nombre de threads dans l'équipe en cours d'exécution
`int omp_get_dynamic();`	S'évalue à vrai si l'ajustement automatique du nombre de thread est activé, à faux sinon
`int omp_get_nested();`	S'évalue à vrai si le support de régions parallèles imbriquées est activé, à faux sinon
`int omp_get_max_active_levels();`	Retourne le nombre maximum de régions parallèles imbriquables
`void omp_get_schedule( omp_sched_t* type, int* chunk);`	Retourne le type de la politique d'ordonnancement et la valeur de chunk au moment de l'appel (voir `opm_set_schedule()` pour les valeurs possibles)

Fonctions monitorant le runtime 2/2

`int omp_get_thread_num();`	Retourne le numéro du thread courant dans l'équipe courante
`int omp_get_num_procs();`	Retourne le nombre de processeurs disponibles
`int omp_in_parallel();`	S'évalue à vrai si une région parallèle est en cours d'exécution, à faux sinon
`int omp_in_final();`	S'évalue à vrai si la tâche courante est `final`, à faux sinon

Et bien d'autres encore (voir la norme)...

Fonctions de mesure du temps

`double omp_get_wtime();`	Retourne le temps écoulé en secondes depuis un temps de référence
`double omp_get_wtick();`	Retourne le temps écoulé en secondes entre deux « tops » d'horloge (indique la précision de la mesure du temps)

- Mesures de temps comparables seulement dans un même thread
- Fonctions portables très utiles même en dehors du parallélisme

Verrous OpenMP

Deux types de verrous et leurs fonctions associées :

- `omp_lock_t` pour les verrous simples
- `omp_nest_lock_t` pour les verrous à tours, pouvant être verrouillés plusieurs fois par un même thread et devant être déverrouillés autant de fois par ce thread pour être levés

- Alternatives plus flexibles à `atomic` et `critical`
- Verrous portables entre Unix et Windows
- Attention à bien les initialiser avec les fonctions adaptées
- Attention à ne pas verrouiller plusieurs fois un verrou simple
- Préférer `atomic` ou `critical` quand c'est possible

Fonctions sur les verrous

`void omp_init_lock(omp_lock_t* l);`	Initialise un verrou
`void omp_destroy_lock(omp_lock_t* l);`	Détruit un verrou
`void omp_set_lock(omp_lock_t* l);`	Positionne un (tour de) verrou, la tâche appelante est suspendue jusqu'au verrouillage effectif
`void omp_unset_lock(omp_lock_t* l);`	Libère un (tour de) verrou
`int omp_test_lock(omp_lock_t* l);`	Tente de positionner un verrou sans suspendre la tâche ; s'évalue à vrai si la tentative réussit, à faux sinon

- Partout `nest_lock` au lieu de `lock` pour les verrous à tours

Exemple d'utilisation d'un verrou [code]

```
#include <stdio.h>
#include <omp.h>
#define MAX 10
int main() {
  int i;
  omp_lock_t lock;
  omp_init_lock(&lock);

  #pragma omp parallel sections private(i)
  {
    #pragma omp section
    for (i = 0; i < MAX; i++) {
      omp_set_lock(&lock);
      printf("Thread A: locked work\n");
      omp_unset_lock(&lock);
    }

    #pragma omp section
    for (i = 0; i < MAX; i++) {
      if (omp_test_lock(&lock)) {
        printf("Thread B: locked work\n");
        omp_unset_lock(&lock);
      } else {
        printf("Thread B: alternative work\n");
      }
    }
  }

  omp_destroy_lock(&lock);
  return 0;
}
```

Variables d'environnement OpenMP

Généralités sur les variables d'environnement

- Variables dynamiques utilisées par les processus pour communiquer
- Prises en compte par le runtime OpenMP avec une priorité plus faible que les fonctions de la bibliothèque elles-mêmes de priorité plus faible que les directives en cas de conflit
- Affichage de la valeur d'une variable `NOM_VAR`
  - Unix : `echo $NOM_VAR`
  - Windows : `set %NOM_VAR%`
- Affectation d'une valeur `VAL` à une variable `NOM_VAR`
  - Unix : `NOM_VAR=VAL`
  - Windows : `set NOM_VAR=VAL`

Variables d'environnement OpenMP

`OMP_NUM_THREADS`	Entier : nombre de threads à utiliser dans les régions parallèles
`OMP_SCHEDULE`	`type,[chunk]` : définit la politique d'ordonnancement à utiliser, voir clause `schedule`
`OMP_DYNAMIC`	`true` ou `false` : autorise ou interdit le runtime à ajuster dynamiquement le nombre de threads
`OMP_NESTED`	`true` ou `false` : active ou désactive le parallélisme imbriqué
`OMP_MAX_ACTIVE_LEVELS`	Entier : nombre maximum de niveaux actifs de parallélisme imbriqué
`OMP_THREAD_LIMIT`	Entier : nombre de threads maximum à utiliser par un programme OpenMP

Et bien d'autres encore (voir la norme)...

Quelques règles de bonne conduite

- Utiliser la clause `default(none)`
  - Pour ne pas déclarer par erreur une variable partagée
- Définir les régions parallèles hors des boucles si possible
  - Pour ne pas perdre du temps à créer et détruire les threads
- Ne pas imposer de synchronisations inutiles
  - Penser à la clause `nowait`
- Utiliser les synchronisations les plus adaptées
  - Utiliser `atomic` ou `reduction` plutôt que `critical`
- Équilibrer la charge entre les threads
  - Utiliser les possibilités de la clause `schedule`
- Donner suffisamment de travail aux threads
  - Penser à la clause `if`