

Programmation Parallèle en Mémoire Partagée

Sophie Robert

UFR ST Département informatique

Le parallélisme et la mémoire partagée

Les caractéristiques

- Tous les threads partagent le même espace mémoire 
- Il faut partager les calculs 
- Règle de l'accès concurrent à la mémoire 

Première approche : Thread C++11

Programmation explicite des threads

- Une librairie pour créer et gérer les threads exécutant des instructions en parallèle.
- A la charge de celui qui écrit le programme.

```
#include <thread>
void printHello(int a){
    cout << " Hello World " << a+1 << " " << endl;
}
int main(int argc, char* argv[]){
    thread th;
    th = thread(printHello,1);
    th.join();
    return(0);
}
```

Première approche : Thread C++11

```
#include <thread>
#define NUM_THREADS 8
void printHelloId() {
    std::thread::id id = this_thread::get_id();
    cout << " Hello World from " << id << endl;
}
int main(int argc, char* argv[]) {
    thread th[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++)
        th[i] = thread(printHelloId);

    for (int i=0; i<NUM_THREADS; i++)
        th[i].join();
    return 0;
}
```

sophie@sebelso: ~/Enseignement/M1/ProgPar..

Fichier Édition Affichage Rechercher Terminal

```
(base) ProgParallele-./main
Hello World from Hello World from 1404284168824321404284
Hello World from Hello World from Hello World from 1404284168824321404284
72
140428400097024
1404284008489728
```

```
Hello World from 140428292638464
Hello World from 140428317816576
Hello World from 140428301031168
(base) ProgParallele-S
```

Première approche : Thread C++11

l'API

1. un constructeur lance un nouveau thread qui commence son exécution en invoquant **f** avec les arguments **args**

```
thread(Function&& f, Args&& ... args);
```

2. une fonction **join** pour la synchronisation. Le thread appelant est bloqué jusqu'à la fin de l'exécution du thread (*this).

```
void join();
```

```
th.join() attend la terminaison du thread th
```

Première approche : Thread C++11

L'accès concurrent

- la synchronisation par `join`
- différents verrous - `mutex` - `condition`

Précisions : Processeur/Cœur, Processus/Thread

Processeur/Cœur

- Un processeur par socket de la carte mère constitué d'un ou plusieurs cœurs qui partagent la mémoire de ce processeur.

Processus/Thread

- Un **Processus** est un programme en exécution et il inclut :
 - Un espace d'adressage (pile, tas, données, ...)
 - Des descripteurs des ressources allouées
 - Un état d'exécution
- Un **Thread** (processus léger) est aussi un programme en exécution mais il est lié à un processus :
 - Il n'inclut que sa pile et son état d'exécution
 - Tout le reste est partagé avec le processus père et tous les **threads** du même processus

Approche OpenMP

Programmation par annotations

- Un ensemble de directives `#pragma omp directive`
- Une bibliothèque de fonctions
- Des variables d'environnement
- Fortran, C, C++



<https://www.openmp.org/>

Approche OpenMP : Hello World

```
void main () {  
  
    int id = 0;  
    printf ("Hello(%d) ", id);  
    printf ("world(%d)\n", id);  
  
}
```

Approche OpenMP : Hello World

```
#include <omp.h>
void main () {

    #pragma omp parallel
    {
        int id = 0;
        printf ("Hello(%d) ", id);
        printf ("world(%d)\n", id);
    }
}
```

Compilation

```
g++ -std=c++11 -fopenmp hello.cpp -o hello
```

Approche OpenMP : Hello World

```
#include <omp.h>
void main () {

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf ("Hello(%d) ", id);
        printf ("world(%d)\n", id);
    }
}
```

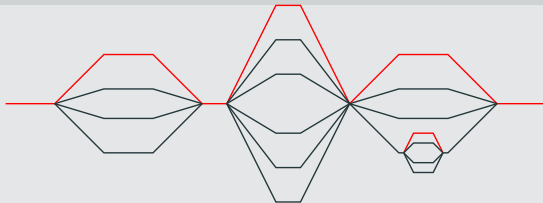
```
sophie@sebelso: ~/Enseignement/M1
Fichier Édition Affichage Recherche
(base) ProgParallele-./Hello
Hello(0) world(0)
Hello(7) world(7)
Hello(3) world(3)
Hello(4) world(4)
Hello(2) Hello(6) world(2)
Hello(1) world(1)
world(6)
Hello(5) world(5)
(base) ProgParallele-s
```

Le modèle d'exécution

Une équipe de Threads

- **Thread maître** : le processus créant la région parallèle (identifiant 0).
- **Équipe** : l'ensemble des threads qui participent à l'exécution parallèle.

Fork-Join

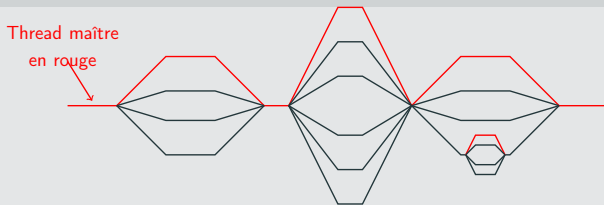


Le modèle d'exécution

Une équipe de Threads

- **Thread maître** : le processus créant la région parallèle (identifiant 0).
- **Équipe** : l'ensemble des threads qui participent à l'exécution parallèle.

Fork-Join

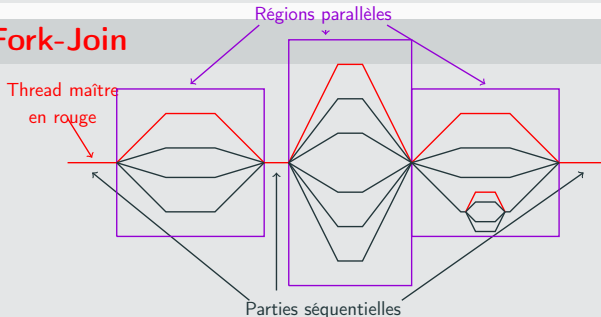


Le modèle d'exécution

Une équipe de Threads

- **Thread maître** : le processus créant la région parallèle (identifiant 0).
- **Équipe** : l'ensemble des threads qui participent à l'exécution parallèle.

Fork-Join

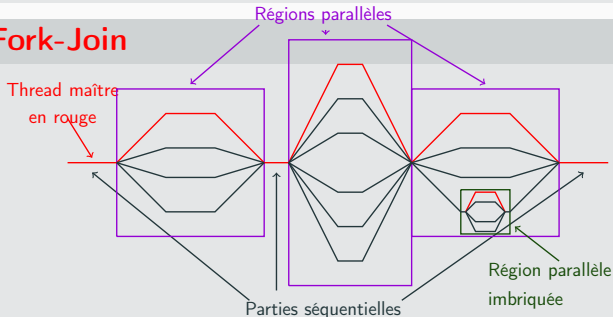


Le modèle d'exécution

Une équipe de Threads

- **Thread maître** : le processus créant la région parallèle (identifiant 0).
- **Équipe** : l'ensemble des threads qui participent à l'exécution parallèle.

Fork-Join



OpenMP : vue d'ensemble

Modèle multi-threadé en mémoire partagée

- Les threads **communiquent** en **partageant des variables**.
- Un partage non intentionnel de données produit des situations de compétition (race condition)
 - et des erreurs sur les résultats
- Pour empêcher les *race conditions* :
 - synchronisation
- La synchronisation coûte cher donc pour obtenir de la performance
 - modifier les accès aux données pour minimiser les besoins en synchronisation

Les régions parallèles

- En OpenMP, les threads sont créés avec la construction `parallel`
- Par exemple, pour créer une région parallèle de 4 threads :

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    f(id);
}
```

Interprétation

Chaque thread appelle `f(id)` pour `id = 0 à 3`.

Comment déterminer le nombre de processus

La taille max de l'équipe de threads

- Une variable d'environnement OMP_NUM_THREADS
- une fonction OpenMP (deprecated)

```
void omp_set_num_threads(int num_threads);
```

La taille d'une région parallèle

- par une clause lors de sa création

```
#pragma omp parallel num_threads(x)  
{  
  ...  
}
```

- **x au plus la taille max de l'équipe**

Bonne pratique

hello.cpp

```
int main (){  
#pragma omp parallel num_threads(4)  
{  
    int id = omp_get_thread_num();  
    printf ("Hello(%d) ", id);  
}  
}
```

Compilation/Exécution

- `g++ -std=c++11 -fopenmp hello.cpp -o hello`
- `OMP_NUM_THREADS=16 ./hello`

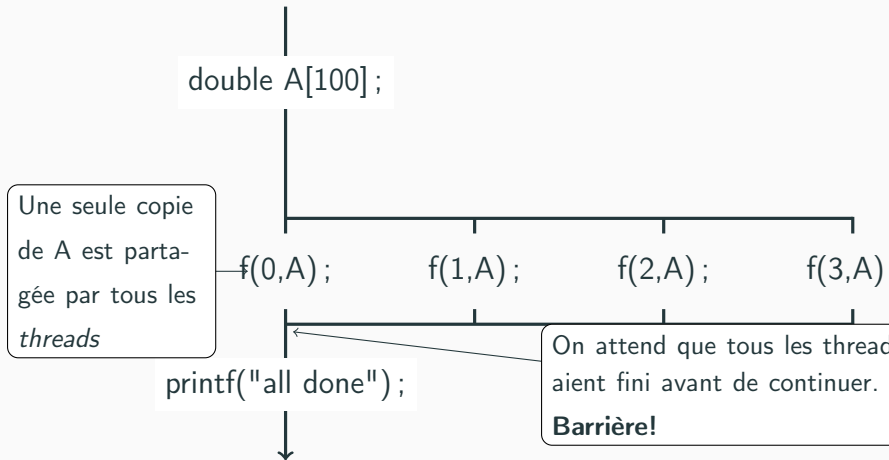
Les régions parallèles

```
double A[100];  
#pragma omp parallel num_threads(4)  
{  
    int id = omp_get_thread_num();  
    f(id, A);  
}  
printf("all done");
```

Interprétation

- Chaque thread appelle `f(id, A)` pour `id = 0 à 3`.
- Le tableau `A` est une **variable partagée** par tous les threads

Les régions parallèles

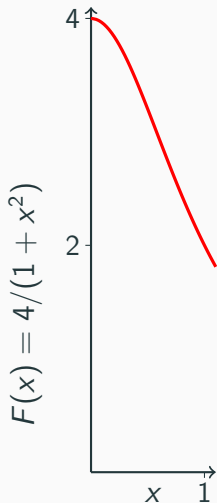


La taille de l'équipe et son identifiant

```
int omp_get_num_threads();  
int omp_get_thread_num();
```

1. nombre de threads dans l'équipe
2. id (ou rang) du thread

Un exemple : approximation de la valeur de π

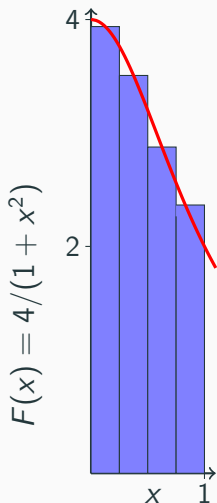


Intégration numérique

Modélisation mathématique, nous savons que :

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Un exemple : approximation de la valeur de π



Intégration numérique

Mathématiquement, nous savons que :

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Nous pouvons approcher l'intégrale par une somme d'aires de rectangles :

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Où chaque rectangle a une largeur Δx et une hauteur $F(x)$ au milieu de l'intervalle i .

Première version : En séquentiel

```
static long num_steps = 100000;
double step;

int main(){
    double x, pi, sum = 0.0;

    step = 1.0/(double)num_steps;

    for (int i=0 ; i < num_steps; i++){
        x = (i+0.5)*step;
        sum += 4.0 / (1.0 + x*x);
    }

    pi = sum * step;
}
```

Deuxième version : avec une région parallèle

```
static long nsteps = 100000;
int main(int , char* argv[]) {
    int nthreads = atoi(argv[1]);
    double pi=0.0, sum[nthreads], step=1.0/nsteps;

    #pragma omp parallel num_threads(nthreads)
    {
        double x;
        int id = omp_get_thread_num();
        sum[id]=0.0;
        for (int i=id ; i < nsteps; i+=nthreads){
            x = (i+0.5)*step;
            sum[id] += 4.0 / (1.0 + x*x); }
    }
    for (int i = 0; i < nthreads; i++)
        pi += sum[i];
    pi = pi * step; }
```

La visibilité des variables 😞

```
double pi=0.0, sum[nthreads], step=1.0/nsteps;  
#pragma omp parallel num_threads(nthreads)  
{  
    int id;  
    double x;
```

Interprétation

- `pi` avant la région parallèle est une variable **partagée** accessible en lecture et écriture par tous les threads dans la région parallèle
- `x` déclarée dans la région parallèle est une variable **locale** à chaque thread. Elle n'existe plus à la sortie de la région parallèle.

Deuxième version : Éviter les accès concurrents

```
double sum[nthreads];
```

Interprétation

Promotion du scalaire vers un tableau de taille `nthreads` pour que chaque thread écrive dans une case différente

```
id = omp_get_thread_num();  
for (int i=id ; i < nsteps; i+=nthreads){  
    x = (i+0.5)*step;  
    sum[id] += 4.0 / (1.0 + x*x);}
```

Interprétation

Technique usuelle pour obtenir une distribution cyclique des itérations d'une boucle.

Comment gérer les accès concurrents

Les constructions de haut niveau

- Critical
- Atomic
- Barrière

Les constructions de bas niveau (prochain cours)

- Flush
- Verrous

Les sections critiques

Un seul thread à la fois peut entrer dans une région

critical

```
float res;  
#pragma omp parallel num_threads(  
    nthreads)  
{  
    float B; int i, id;  
    id = omp_get_thread_num();  
    for(i=id; i<niters; i+=nthreads){  
        B = big_job(i);  
#pragma omp critical  
    {  
        consume (B, res);  
    }  
}
```

Interprétation

Les threads
attendent leur tour.
Un seul appel à
consume() à la
fois.

La construction Atomic

Atomic offre une exclusion mutuelle mais s'applique uniquement sur les mises à jour d'un espace mémoire (ici, la mise à jour de `x` par exemple).

```
double x = 0.0;
#pragma omp parallel
{
    double tmp, B;
    B = dolt();
    tmp = big(B);

#pragma omp atomic
    x += tmp;
}
```

interprétation

- **Atomic** ne protège que les lectures/écritures de `x`.
- une seule instruction dans le bloc structuré

Retour au calcul de π

```
double sum[nthreads], step = 1.0/(double)
    num_steps;
#pragma omp parallel num_threads(nthreads)
{
    int i, id;
    double x;
    id = omp_get_thread_num();
    sum[id]=0.0;
    for (int i=id ; i < num_steps; i+=nthreads){
        x = (i+0.5)*step;
        sum[id] += 4.0 / (1.0 + x*x);
    }
}
double pi = 0.0;
for (int i = 0; i < nthreads; i++)
    pi += sum[i] * step;
```

Retour au calcul de π

False sharing

- Si des éléments indépendants sont situés sur la même ligne de cache, chaque mise à jour déclenche un déplacement de la ligne de cache d'un thread à un autre.
 - On appelle ceci du *false sharing*.
- Promouvoir un scalaire en tableau, les éléments restent contigus en mémoire et partagent leurs lignes de cache
 - Résultat : faible passage à l'échelle
- **Solution :**
 - Lorsque les mises à jour sont fréquentes, travailler avec des copies locales, plutôt qu'avec un tableau indexé par chaque thread
 - Il faut faire du *padding* pour que les éléments utilisés soient sur des lignes de cache distinctes

Solution au False sharing

```
double pi = 0.;
double step = 1.0/(double)num_steps;
#pragma omp parallel num_threads(nthreads)
{
    double x, sum; <————
    int id = omp_get_thread_num();
    sum=0.0; <————
    for (int i=id ; i < num_steps; i+=nthreads){
        x = (i+0.5)*step;
        sum += 4.0 / (1.0 + x*x); <————
    }
    #pragma omp critical
        pi += sum * step; <————
}
```

Solution au false sharing

Variable locale

```
#pragma omp parallel num_threads(nthreads)
{
    int i, id, nthrds;
    double x, sum; <————
    ...
}
```

Interprétation

- On crée un scalaire local à chaque thread pour accumuler les sommes partielles
- Pas de tableau, donc pas de *false sharing*

Solution au false sharing

Calcul en local

```
sum=0.0; <————  
for (int i=id ; i < num_steps; i+=nthreads){  
    x = (i+0.5)*step;  
    sum += 4.0 / (1.0 + x*x); <————  
}
```

Interprétation

- Pas d'accès concurrent

Solution au false sharing

Résultat final

```
#pragma omp atomic  
    pi += sum * step;  
}
```

Interprétation

- Il faut protéger la somme dans une section critique.

SPMD vs partage de travail/collaboration

- La construction **parallel** crée un programme SPMD (*Single Program Multiple Data*), c-à-d chaque thread exécute le même code.
- Comment diviser un chemin d'exécution entre plusieurs threads formant une équipe ?
- **C'est de la collaboration (*Worksharing*)**
 - A la main en différenciant le comportement de chaque thread en fonction de son identifiant
 - Gestion des boucles pour partager les itérations
 - Sections
 - Tasks

La directive **for**

Collaboration dans une boucle

La construction d'une collaboration sur une boucle répartit les itérations de la boucle en plusieurs threads dans une équipe.

Exemple

```
#pragma omp parallel num_threads(nthreads)
{

    #pragma omp for
    for (int i = 0; i < N; i++)
        calcul_savant(i);
}
```

Comment partager le travail d'une boucle ?

Séquentiel

```
for (int i=0, i < N; i++) a[i] = a[i] + b[i];
```

OpenMP Région parallèle

```
#pragma omp parallel num_threads(nthreads)
{
    int id, i, istart, iend;
    id = omp_get_thread_num();
    istart = id * N/nthreads; // N divisible par
                             nthreads
    iend = (id+1) * N/nthreads;
    for (i=istart; i<iend; i++)
        a[i] = a[i] + b[i];
}
```

Comment partager le travail d'une boucle ?

Séquentiel

```
for (int i=0; i < N; i++) a[i] = a[i] + b[i];
```

OpenMP : région parallel + construction for de collaboration

```
#pragma omp parallel num_threads(nthreads)
{
    #pragma omp for
    for (int i=0; i < N; i++) a[i] = a[i] + b[i];
}
```

Comment partager le travail d'une boucle ?

OpenMP : construction combinée

```
#pragma omp parallel for num_threads(nthreads)  
for (int i=0; i < N; i++) a[i] = a[i] + b[i];
```

Paralléliser les boucles

Approche de base

- Trouver les boucles de calcul intensif
- Rendre les itérations de la boucle **indépendantes** afin qu'elles puissent s'exécuter dans n'importe quel ordre.
- Placer la directive OpenMP appropriée et c'est terminé !



Exemple



```
int A[max];  
int j = 5;  
for (int i=0; i<max; i++){  
    A[i] = big(j);  
    j+=2;  
}
```

Paralléliser les boucles

Approche de base

- Trouver les boucles de calcul intensif
- Rendre les itérations de la boucle **indépendantes** afin qu'elles puissent s'exécuter dans n'importe quel ordre.
- Placer la directive OpenMP appropriée et c'est terminé !

Example



```
int A[N];  
#pragma omp parallel for  
for (int i=0; i<N; i++){  
    int j = 5+2*i; <————  
    A[i] = big(j);  
}
```

Réduction

Comment gérer ce cas ?

```
double moy=0.0, A[N];  
for (int i = 0; i < N; i++)  
    moy += A[i];  
moy /= N;
```

Situation très fréquente

- Toutes les valeurs sont **accumulées** dans une seule variable. Il y a une vraie dépendance entre les itérations de la boucle qui ne peut pas être supprimée trivialement.
- C'est une situation connue appelée réduction.

Réduction avec OpenMP

La clause reduction

reduction (**op** : **list**)

Dans une construction de collaboration ou parallel

- Une copie locale de chaque variable de **list** est créée et initialisée en fonction de l'opérateur **op** (0 pour "+" par exemple.).
- Le compilateur trouve les expressions de réduction standard qui contiennent **op** et les utilise pour mettre à jour la copie.
- Les copies locales sont réduites à une valeur unique et combinées dans la valeur partagée d'origine.

Réduction avec OpenMP

La clause reduction

reduction (**op** : **list**)

Dans une construction de collaboration ou parallel

- Les variables de **list** doivent être partagées dans la région parallèle englobante

Exemple

```
double moy=0.0, A[N];  
#pragma omp parallel for reduction(+:moy)  
for (int i = 0; i < N; i++)  
    moy += A[i];  
moy /= N;
```

Partage de données

```
double A[10];  
int main(){  
    int index[10];  
    #pragma omp parallel  
        work(index);  
    printf("%d\n", index[0]);  
}
```

```
extern double A[10];  
void work(int* index){  
    double temp[10];  
    static int count;  
    ...  
    ...  
}
```

partagé ou local

- A, index et count sont partagés par tous les threads.
- temp est local à chaque thread

Changer les attributs de stockage d'une variable

- Il est possible de changer les attributs de stockage des données pour une construction omp en utilisant les clauses suivantes :
 - **shared** : partagée par tous les threads.
 - **private** : locale à chaque thread (non partagée → privée).
 - **firstprivate** : comme private mais initialisée, pour chaque thread avec sa valeur avant la construction OpenMP.
 - **lastprivate** : comme private mais **une** dernière valeur de la donnée dans une boucle parallèle est transmise à la variable partagée en dehors de la boucle.

Clause private

```
void wrong(){  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 1000; j++)  
        tmp += j;  
  
    printf ("tmp = %d\n", tmp);  
}
```

private(tmp)

- Crée une nouvelle copie locale de `tmp` pour chaque thread.

Attention

`tmp` n'est pas initialisée. Valeur de sortie ?? 🙄

Clause firstprivate

```
void useless(){  
    int tmp = 0;  
    #pragma omp parallel for firstprivate(tmp)  
    for (int j = 0; j < 1000; j++)  
        tmp += j;  
    printf ("tmp = %d\n", tmp);  
}
```

firstprivate

- Initialise chaque copie privée avec la valeur correspondante du **thread maître** (Cas particulier de private).

Clause firstprivate

```
void useless(){  
    int tmp = 0;  
    #pragma omp parallel for firstprivate(tmp)  
    for (int j = 0; j < 1000; j++)  
        tmp += j;  
    printf ("tmp = %d\n", tmp);  
}
```

firstprivate

- Initialise chaque copie privée avec la valeur correspondante du **thread maître** (Cas particulier de private).

Interprétation

- Chaque thread obtient son propre tmp (ici tmp=0)

Clause lastprivate

```
void closer(){
    int tmp = 0;
    #pragma omp parallel for firstprivate(tmp)
                                lastprivate(tmp)
        for (int j = 0; j < 1000; j++)
            tmp += j;
    printf ("tmp = %d\n", tmp);
}
```

lastprivate

- Passe la valeur d'une variable privée à la dernière itération à la variable partagée (Cas particulier de private).

Interprétation

tmp=0 initialement et à la sortie recopie d'une des variables privées

Politique par défaut

clause `default(shared/private/firstprivate)`

- Le statut par défaut d'une variable pour laquelle on ne définit pas explicitement le statut
- Si on ne précise pas un statut par défaut les variables dont le statut n'est pas précisé sont `shared`

Politique par défaut

Cas particulier : clause `default(none)`

```
int A[N];  
int j;  
#pragma omp parallel for default(none)  
for (int i=0; i<N; i++){  
    j = 5+2*i;  
    A[i] = big(j);  
}
```

Interprétation

- A la compilation erreurs sur `A` et `j`
- il faut explicitement définir le statut des variables utilisées dans la région parallèle.

Politique par défaut

```
clause default(none)
```

```
int A[N];
```

```
int j;
```

```
#pragma omp parallel for default(none)  
                                shared(A) private(j)
```

```
for (int i=0; i<N; i++){
```

```
    j = 5+2*i;
```

```
    A[i] = big(j);
```

```
}
```

Dernier retour sur le calcul de π !

La version séquentielle

```
static long num_steps = 100000;
double step;

int main(){
    double x, pi, sum = 0.0;
    step = 1.0/(double)num_steps;

    for (int i=0 ; i < num_steps; i++){
        x = (i+0.5)*step;
        sum += 4.0 / (1.0 + x*x);
    }

    pi = sum * step;
}
```

Dernier retour sur le calcul de π

Une nouvelle version parallèle

```
#include <omp.h>
static long nsteps = 100000;
int main(int , char* argv[]){
    double x, pi, sum = 0.0;
    int nthreads = atoi(argv[1]);
    double step = 1.0/(double)nsteps;
    #pragma omp parallel for private(x) reduction(+:
        sum) num_threads(nthreads)
    for (int i=0 ; i < nsteps; i++){
        x = (i+0.5)*step;
        sum += 4.0 / (1.0 + x*x);
    }
    pi = sum * step;
}
```

Conclusion sur π !

1 ligne ajoutée

```
#include <omp.h>
...
....
    #pragma omp parallel for private(x) reduction(+:
        sum) num_threads(nthreads)
    ...
```

Efficace et pas cher

Nous avons créé un programme parallèle sans changer une ligne de code. En ajoutant uniquement 1 ligne simple.

Les barrières de synchronisation : par l'exemple

```
#pragma omp parallel shared(A,B,C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    A[id] = big_calc4(id);
} <————
```

Barrière implicite région parallèle

On ne sort de la région parallèle `omp parallel` que lorsque tous les threads ont terminé.

Les barrières de synchronisation : par l'exemple

```
#pragma omp parallel shared(A,B,C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp for          <————
    for (i = 0; i < N; i++)
        C[i] = big_calc3(i,A);
    A[id] = big_calc4(id);
}
```

Barrière implicite omp for

On ne sort de la boucle pour les instructions suivantes que lorsque tous les threads ont terminé leurs itérations

Les barrières de synchronisation : par l'exemple

```
#pragma omp parallel shared(A,B,C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp for
    for (i = 0; i < N; i++)
        C[i] = big_calc3(i,A);
    #pragma omp for nowait <————
    for (i = 0; i < N; i++)
        B[i] = big_calc2(C,i);
    A[id] = big_calc4(id);
}
```

On peut supprimer une barrière implicite

`nowait` est une clause pour `omp parallel` ou `omp for` qui permet de supprimer la barrière implicite.

Les barrières de synchronisation : par l'exemple

```
#pragma omp parallel shared(A,B,C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier <————
    #pragma omp for
    for (i = 0; i < N; i++)
        C[i] = big_calc3(i,A);
    #pragma omp for nowait
    for (i = 0; i < N; i++)
        B[i] = big_calc2(C,i);
    A[id] = big_calc4(id);
}
```

Une barrière explicite

Il est possible de mettre une barrière explicite grâce à la directive `omp barrier`. **C'est très coûteux**

Synchronisation : Construction master

```
#pragma omp master
```

- La construction master annonce un bloc structuré qui est uniquement exécuté par le thread maître
- Les autres threads l'ignorent (**aucune synchronisation**)

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    {
        do_specific_things_master();
    }
    do_many_other_things
}
```

Synchronisation : Construction master

```
#pragma omp master
```

- La construction master annonce un bloc structuré qui est uniquement exécuté par le thread maître
- Il faut gérer la synchronisation si nécessaire par des barrières explicites

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    {
        exchange_boundaries();
    }
    #pragma omp barrier
    do_many_other_things
```

Construction single

```
#pragma omp single
```

- La construction `single` annonce un bloc structuré qui sera exécuté par un seul thread (pas nécessairement le thread maître)
- Il y a une **barrière implicite** à la fin d'un bloc `single`
- On peut enlever la barrière avec la clause `nowait`.

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    {
        exchange_boundaries();
    }
}
```

Construction `omp for` et la clause `schedule`

La clause `schedule` modifie comment les itérations de la boucle seront projetées sur les threads.

- `schedule(static[, chunk])`
Distribue des blocs d'itérations de taille `chunk` à chaque thread
- `schedule(dynamic[, chunk])`
Chacun son tour, chaque thread prend `chunk` itérations dans une liste jusqu'à ce que toutes les itérations aient été réalisées.
- `schedule(guided[, chunk])`
Les threads récupèrent dynamiquement des blocs d'itérations. La taille des blocs est d'abord grosse puis

Schedule

Quand se servir de quel ordonnancement ?

Ordonnancement	Quand s'en servir ?
STATIC	Calcul pré-déterminé et prédictible par le programmeur
DYNAMIC	Non prédictible, quantité de travail par itération variable
GUIDED	Cas spécifique d'ordonnancement dynamique pour réduire la surcharge (<i>overhead</i>) introduite

`schedule(static[,chunk])`

- Si `chunk` n'est pas spécifié les itérations sont réparties "équitablement" sur les threads
- Sinon chaque thread effectue `chunk` itérations de manière cyclique.

```
static long num_steps = 100000;
omp_set_num_threads(8);
#pragma omp parallel for schedule(static,10)
                        private(x) reduction
                        (+:sum)
for (int i=0 ; i < num_steps; i++){
    x = (i+0.5)*step;
    sum += 4.0 / (1.0 + x*x);
}
pi = sum * step;
```

- thread 0 : itérations 0->9 80->89 160->169

`schedule(dynamic[,chunk])`

- `chunk` par défaut vaut un
- Sinon chaque thread demande `chunk` itérations à exécuter et lorsqu'il a terminé demande un nouveau paquet de `chunk` itérations.

```
static long num_steps = 100000;
omp_set_num_threads(8);
#pragma omp parallel for schedule(dynamic,10)
                        private(x) reduction
                        (+:sum)
for (int i=0 ; i < num_steps; i++){
    x = (i+0.5)*step;
    sum += 4.0 / (1.0 + x*x);
}
pi = sum * step;
```

Boucles imbriquées

```
#pragma omp parallel for shared(A,B,C)
for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
        C[i*n+j] = A[i*n+j]+B[i*n+j];
```

interprétation

Le calcul des lignes de C sont réparties sur les threads

```
for (int i=0; i<n; i++)
#pragma omp parallel for shared(A,B,C)
    for (int j=0; j<m; j++)
        C[i*n+j] = A[i*n+j]+B[i*n+j];
```

interprétation

Le calcul de chaque ligne de C est réparti sur les threads

Boucles imbriquées : la clause `collapse`

```
#pragma omp parallel for shared(A,B,C) collapse(2)
for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
        C[i*n+j] = A[i*n+j]+B[i*n+j];
```

La clause `collapse` permet de spécifier le nombre de boucles qui sont réunies pour le partage du travail

interprétation sur notre exemple

Le calcul des éléments de la matrice `C` seront répartis sur les threads.

Les variables d'environnement OpenMP

Fonction ou clause	Variable d'environnement
<code>omp_set_num_threads</code>	<code>OMP_NUM_THREADS=8</code>
<code>schedule(static,chk)</code>	<code>OMP_SCHEDULE=STATIC,chk</code>
<code>schedule(dynamic,chk)</code>	<code>OMP_SCHEDULE=DYNAMIC,chk</code>

Pour wooclap

```
int N = grand;  
int A[N] ; // valeurs rand entre 0 et 100  
#pragma omp parallel for schedule(??,10)  
for (int i=0; i<N; i++)  
    if (A[i]<50)  
        for (int j=0; j<100; j++)  
            A[i] += rand()%10;
```