Modèles et techniques en programmation parallèle hybride et multi-cœurs

Parallélisme multithreads (2)

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

18/11/2023

Techniques de programmation parallèle multithreads

- Programmation OpenMP grain fin
- Programmation OpenMP grain grossier
- Programmation OpenMP par tâches
- Autres outils : std::threads, TBB

Programmation OpenMP de type “grain fin”

Cela consiste à ajouter un pragma devant les boucles à paralléliser pour que OpenMP découpe automatiquement l'ensemble de valeurs de l'indice de boucle en groupes.

Chaque thread prend en charge un des groupes. Chaque groupe est (à peu près) de même taille (en nombre d'itérations).

Par exemple:

```
#pragma omp parallel for
for (i=0; i<n; i++)
  u[i] = f(a, v[i]);
```

Le programmeur a assez peu de possibilités pour adapter la répartition à un problème particulier

Avantages :

- Simple à programmer (si la boucle est simple).
- La différence entre le code parallèle et le code séquentiel est l'ajout de pragmas, donc on peut garder une seule version des sources.
- On peut paralléliser progressivement un code, choisir les boucles à paralléliser.

Désavantages :

- Les régions parallèles sont souvent plus nombreuses et plus petites, ce qui décroît souvent les performances.
- La répartition est équilibrée en nombre d'itérations pas en temps calcul, souvent certains threads ont fini avant les autres et sont inemployés.
- Les boucles complexes sont plus difficiles ou impossibles à paralléliser (boucles "while", boucles sur des indices non entiers, etc.)

Soit l'algorithme suivant: on part d'une valeur initiale de u et on calcule une suite de nT valeurs de u pour $t > 0$.

```
for (iT=0; iT < nT; iT++)
{
  dT = calcul_dt(u);
  for (iX=1; iX < nX-1; iX++)
    v[iX] = f(u[iX-1],u[iX],u[iX+1],dT)
  echange(u, v);
}
```

La boucle externe (sur `iT`) n'est pas parallélisable

- chaque itération modifie le même vecteur v,
- chaque itération dépend de toutes les précédentes.

Pour paralléliser, il ne reste que la boucle interne (sur iX):

```
11 for (iT=0; iT < nT; iT++)
12 {
13   dT = calcul_dt(u);
14   #pragma omp parallel for
15   for (iX=0; iX < nX; iX++)
16     v[iX]=f(u[iX-1],u[iX],u[iX+1],dT);
17   echange(u, v);
18 }
```

Il y a donc **nT régions parallèles** (avec création des threads à l'entrée - ligne 14 - et destruction des threads à la sortie - après la ligne 16).

Programmation OpenMP de type “grain grossier”

Cela consiste à :

- essayer de définir des régions parallèles plus grandes et moins nombreuses.

  Cela conduit souvent à des régions parallèles dans lesquelles une partie des instructions ne sont pas exécutées par tous les threads (utilisation des pragmas `single` *et* `master`*, ou des indices de threads :* `omp_get_thread_num()`*).*

- découper soi-même les boucles internes pour mieux tenir compte de la différence de temps calcul entre les itérations.

Parallélisme multithreads
grain fin (fine grain)

(threads $T_i, i > 0$ créés au début
de chaque région parallèle et détruits
à la fin de chaque région parallèle)

Parallélisme multithreads
grain grossier (coarse grain)

(threads $T_i, i > 0$ créés au début de l'execution
activés au début de chaque région parallèle
et mis en pause à la fin de chaque région parallèle)

Thread 0 actif pendant toute l'exécution

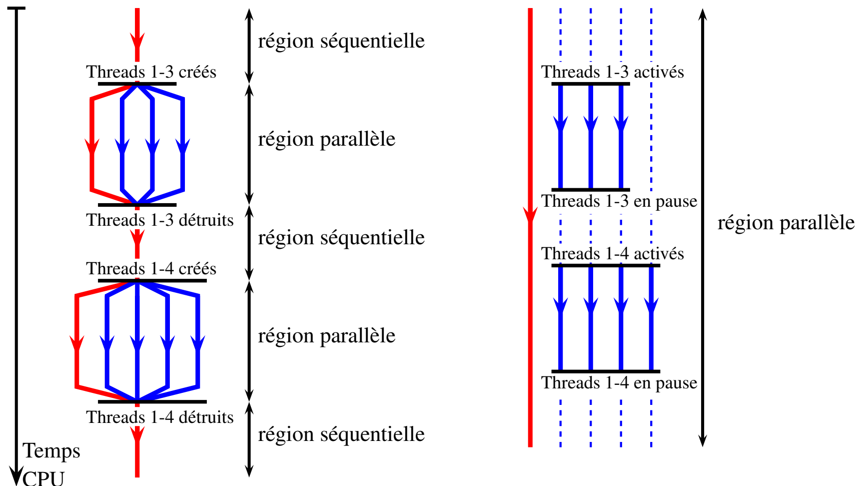

Illustration du premier point : on crée une seule région parallèle

```
1  #pragma omp parallel default(shared)
2  {
3    int iT, iX;
4    for (iT=0; iT < nT; iT++)
5    {
6      #pragma omp single
7      dT = calcul_dt(u);
8
9      #pragma omp for
10     for (iX=1; iX < nX-1; iX++)
11       v[iX] = f(u[iX-1],u[iX],u[iX+1],dT)
12
13     #pragma omp single
14     echange(u, v);
15   }
16 }
```

Illustration du second point : on définit soi-même le découpage de la boucle interne

```
#pragma omp parallel default(shared)
{
  int iT, iX;
  int iTh = omp_get_thread_num();
  int nTh = omp_get_num_threads();
  int d = nX/nTh;
  int nX1 = iTh*d;
  int nX2 = n1 + d; if(iTh==nTh-1)nX2=nX;
  for (iT=0; iT < nT; iT++)
  {
    #pragma omp single
    dT = calcul_dt(u);
    for (iX=nX1; iX < nX2; iX++)
      v[iX] = f(u[iX-1],u[iX],u[iX+1],dT)
    #pragma omp single
    echange(u, v);
  }
```

Avantages :

- On crée moins de régions parallèles (une seule dans l'exemple).
- Le découpage de la boucle interne est fait une seule fois et c'est le développeur qui détermine ce découpage.

Suivant le type de problème, la programmation à grain grossier pourra être plus ou moins intéressante.

Désavantages :

- L'écriture du code est plus complexe (savoir quand mettre en pause certains threads par exemple).
- Il est moins facile de maintenir un source unique pour les versions séquentielle et parallèle.

Programmation OpenMP par tâches

On définit dans le codes des groupes d'instructions qui peuvent être exécutés en parallèle, qu'on appelle "tâches".

Il peut y avoir un nombre de tâches différent du nombre de threads (en général, il y aura plus de tâches que de threads).

Quand les tâches sont définies, le système les met dans une file d'attente. Les tâches sont exécutées dès qu'un cœur est disponible.

Avantages :

- On peut définir un nombre de tâches inconnu à l'avance (boucle "while" par exemple).
- Une tâche peut elle-même créer d'autres tâches (parallélisme à plusieurs niveaux).

Inconvénient :

- On ne sait pas exactement quand une tâche sera lancée, ce qui rend délicat la gestion des variables partagées.

Exemple : Calcul de la suite de Fibonacci définie par la récurrence

$$F_n = F_{n-1} + F_{n-2}$$

avec $F_0 = 0$ et $F_1 = 1$.

Exemple de code en C/C++ (fonction récursive):

```
long f(int n)
{
  if (n<2)
    return n;
  else
    return f(n-1) + f(n-2);
}
```

On peut remarquer que :

- dans f, découpage du travail seulement en 2 parties
- f est récursive, donc si on utilise des pragma dans f, elles seront utilisées de façon récursive

Le nombre de threads utilisés peut rapidement dépasser le nombre de cœurs disponibles, ce qui est assez compliqué à gérer.

Première version utilisant des sections OpenMP

```
long f(int n) {
  if (n < 2) return n;
  long a, b;

  omp_set_num_threads(2)
#pragma omp parallel sections
  {
  #pragma omp section
    a = f(n-1);
  #pragma omp section
    b = f(n-2);
  }
  return a + b;
}
```

Cette version va créer 2^L threads où L est le nombre de niveaux de récursivité.

Voir `Exemples3/omp_sections`.

L'exemple fonctionne relativement bien, mais l'utilisation des sections a 2 désavantages:

- Le nombre de sections est fixe pour chaque niveau de parallélisation.
- Il faut bien utiliser la fonction `omp_set_num_threads` sinon risque "d'étouffer" la machine

Seconde version utilisant les tâches OpenMP:

```
long fib_tasks(int n) {

  long i, j;
  if (n<2)
    return n;

#pragma omp task shared(i, n)
  i=fib_tasks(n-1);
#pragma omp task shared(j, n)
  j=fib_tasks(n-2);

#pragma omp taskwait

  return i+j;
}
```

Voir Exemples3/omp_tasks.

Cette version utilisant les tâches OpenMP permet d'enlever les désavantages des sections et de, plus, on peut contrôler le nombre total de threads utilisé par le code.

Mais:

- Faire attention au nombre de tâches créées (2 à la puissance n).
  En pratique on met une limite au nombre de niveaux de récursivité.
- Utiliser les résultats d'une tâche seulement quand on est sûr que la tâche est terminée : pas de barrière implicite à la fin de `omp task`
  Donc, il faut appeler `omp taskwait` avant d'utiliser `i` ou `j` (pour calculer `i+j`).

Le répertoire Exemples3/sinus contient 5 sous-répertoires qui tous comparent deux méthodes de calcul de $y = sin(x)$, soit par la fonction système soit une approximation par une formule de Taylor.

Remarque: on a artificiellement ralenti le calcul de la formule de Taylor pour mieux faire ressortir les temps de calcul.

- sinus_seq: calcul séquentiel
- sinus_omp_fine_grain: calcul OpenMP grain fin
- sinus_omp_coarse_grain: calcul OpenMP grain grossier
- sinus_omp_adaptatif: calcul OpenMP grain grossier avec équilibrage de charge
- sinus_omp_task: calcul OpenMP par tâches

Il existe plusieurs autres outils pour faire de la programmation multi-threads.

On commentera ici deux exemples qui utilisent

- `std::thread` : similaires à pthread mais de style plus C++ et avec des mécanismes plus évolués
- `tbb` : Threads Building Blocks, aussi spécifiques à C++ et plus souples, mais à installer séparemment

Exemple utilisant `std::thread`, voir `Exemples3/std_thread`

Commentaires

Exemple utilisant `tbb`, voir `Exemples3/tbb`

Commentaires