Exemples OMP

Patrick Martineau

Version 1.0, 2018-03-06

1. Les bases

1.1. Vérification de la compilation par votre compilateur, exemple avec gcc

Soit le source suivant hello-simple.c avec une seule ligne pragma OMP simple. Vous pouvez compiler classiquement votre code :

```
gcc hello-simple.c -o hello-simple
```

ou bien en demandant l'interprétation des pragma openMP avec :

```
gcc -fopenmp hello-simple.c -o hello-simple
```

Lien sur hello-simple.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main(int argc, char *argv[]) {
int iam =0, np = 1;
#pragma omp parallel
    printf("Hello from one thread.\n");
}
```

1.2. Tirer partie de l'interprétation ou la non interprétation des clauses openMP

Le même code sert donc à la fois à obtenir un code séquentiel et un code parallèle. C'est un outil important qui permet de séparer :

1. la phase de conception et validation du code séquentiel,
2. de la phase de transformation en code parallèle (donc vous conservez en permanence la capacité à vérifier que vos modifications n'entraînent pas d'erreur à l'exécution)

1.3. Augmenter la portée de votre code

Pour permettre de recompiler votre source sous Windows/Linux/MacOS, vous pouvez utiliser le fichier omp_repair.h suivant, à la place de <omp.h>

Lien sur omp_repair.h.

```
Unresolved directive in Presentation-sourcesOMP.adoc -
include::./sources/omp_repair.h[]
```

En conclusion

- [x] Je sais compiler un code OMP
- [x] Je tire partie de l'accès au code séquentiel pour **vérifier** mes résultats
- [ ] Je maitrise le degré de parallélisme

2. Maitrise du parallélisme

Dans cette partie, on focalise sur la manière de créer des parties pouvant s'exécuter en parallèle.

2.1. Comment spécifier ce qui est parallèle de ce qui ne l'est pas ?

Le premier exemple a pour but de démarrer plusieurs threads en parallèle et de montrer que chacun a une identité propre, puisque chacun affiche un numéro (identifiant) différent. Chacun a donc une variable privée **iam** qui contient une valeur différente.

Lien sur hello.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main(int argc, char *argv[]) {
int iam =0, np = 1;
#pragma omp parallel private(iam, np)
    {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d \n", iam, np);
    }
}
```

Dans cet exemple, on peut s'étonner de demander à chaque thread de remplir la variable **np** avec le nombre de threads. Tous les threads obtiennent la même valeur. On aurait pu utiliser une variable partagée et demander à un seul thread de l'affecter avec le nombre de threads. Malheureusement, cela nous obligerait à synchroniser tous les threads, et donc on perdrait du temps…

Avec l'ajout d'une seule ligne, cet exemple a permis de créer plusieurs threads. Chacun exécute le code entre accolades. Les variables **iam** et **np** sont privées, c'est-à-dire que chaque thread a sa version de ces variables.

L'exemple suivant introduit une variable **b** partagée par plusieurs threads. Cette variable est donc

lue et modifiée simultanément, ou quasi-simultanément, par plusieurs threads. Ce programme peut donc conduire à un résultat incohérent.

Lien sur shared.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main(int argc, char *argv[]) {
  int b = 0;
#pragma omp parallel shared(b)
  {
        b++;
    printf("Incrementation par thread %d de %d \n", omp_get_thread_num(),
omp_get_num_threads());
  }
  printf("Valeur finale : %d\n", b);
}
```

Parfois, à l'intérieur d'une section parallèle, il y a une partie qui doit être exécutée une seule fois (par exemple, affecter **np** dans l'exemple hello.c).

Lien sur single.c.

```
#include <stdio.h>
#include "omp_repair.h"
#define SIZE 10
int main() {
int a,i,n=SIZE, b[SIZE];
#pragma omp parallel shared(a,b) private(i)
{
  #pragma omp single
  {
    a = 10;
    printf("Single construct executed by thread %d\n",
        omp_get_thread_num());
  }
  /* A barrier is automatically inserted here */
  #pragma omp for
  for (i=0; i<n; i++)
    b[i] = a;
} /*-- End of parallel region --*/
printf("After the parallel region:\n");
for (i=0; i<n; i++)
  printf("b[%d] = %d\n",i,b[i]);
}
```

Remarquons que ce pragma ne permet pas de préciser quel thread réalisera le block de code en question. On peut considérer que c'est le premier thread qui arrivera sur cette section qui

l'exécutera et que les suivants passeront directement aux instructions qui suivent ce block. De toute manière, tous les threads attendent que celui qui exécute la section ait fini avant de démarrer le code qui suit cette section "single".

L'exemple suivant permet de rapidement créer tous les threads en parallèle mais de définir séparément ce que chaque thread doit faire de son coté. S'il y a moins de section que de threads créés, alors certains threads se finissent immédiatement car ils n'ont rien à faire.

Lien sur sections.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main(int argc, char *argv[]) {
printf("Mon programme commence !\n");
#pragma omp parallel sections
{
#pragma omp section
    printf("Mon premier thread %d sur %d.\n",omp_get_thread_num(),
omp_get_num_threads());
#pragma omp section
    printf("Mon deuxième thread %d sur %d.\n",omp_get_thread_num(),
omp_get_num_threads());
}
}
```

- [x] Je sais créer des threads en parallèle
- [x] Je sais séparer code parallèle et code à exécuter par un seul thread
- [ ] Je maîtrise le partage des données

2.2. Comment paramétrer le nombres des threads en parallèle

Il y a trois manières d'agir sur le degré de parallélisme :

1. Au niveau du shell : la compilation au format parallèle ayant été réalisée, on peut utiliser une variable d'environnement pour préciser le nombre de threads à créer dans **les processus fils** :

   ```
   export OMP_NUM_THREADS=8
   ```

2. Au niveau du processus : au début du code, on peut préciser le degré de parallélisme à utiliser tout le long de l'exécution de ce **processus** :

   ```
   omp_set_num_threads(NumThreads);
   ```

3. Au niveau du block : en utilisant une clause du pragma parallel qui permet de préciser le degré

de parallélisme pour la **section** parallèle

```
#pragma omp parallel num_threads(NumTreads)
```

En conclusion

- [x] Je contrôle le degré de parallélisme

3. La synchronisation

Le pragma barrier permet de préciser un point de synchronisation commun à tous les threads au milieu d'une section parallèle. Tous les threads seront bloqués à ce point du code tant que tous les threads ne sont pas arrivés à ce point. Un équivalent serait de séparer cette section parallèle en deux sections parallèles successives mais on perdrait les valeurs des données locales.

Lien sur barrier.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main() {
  int tid;
#pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Thread %d sleeping\n", tid);
    sleep(tid);
    printf("Thread %d woke up, but waiting for others\n", tid);
#pragma omp barrier
    printf("Thread %d ready to go\n", tid);
  }
}
```

4. La concurrence

A partir du moment où plusieurs threads s'exécutent simultanément, le problème est de décider si chaque thread a accès à sa version d'une variable (variable privée) ou si la variable est partagée et tous les threads accèdent à la même case mémoire. Dans l'exemple suivant, la clause private permet de préciser les variables privées. En C, par défaut la variable est partagée mais je vous conseille de le préciser avec **shared()**.

Lien sur private.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main() {
  int b = 20, tid = 0;
#pragma omp parallel private(b, tid)
  {
    b++;
    tid = omp_get_thread_num();
    printf("Thread %d: b = %d\n", tid, b);
  }
  printf("%d\n", b);
}
```

Dans cet exemple, on remarque que chaque thread n'a pas une initialisation de **b** à 20. Chaque thread ajoute 1 à 'sa variable **b**'. Une fois tous les threads achevés, la variable **b** affichée est toujours égale à 20.

Pour préciser la valeur initiale d'une variable privée, on peut affecter celle-ci au début du thread ou utiliser **firstprivate()** pour 'recopier' la valeur présente avant le début de la section parallèle.

Lien sur firstprivate.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main() {
  int b = 20, tid = 0;
#pragma omp parallel private(tid) firstprivate(b)
  {
    b++;
    tid = omp_get_thread_num();
    printf("Thread %d: b = %d\n", tid, b);
  }
  printf("b = %d\n", b);
}
```

L'exemple suivant permet d'indiquer quelle valeur sera recopiée dans la variable, une fois les threads parallèles achevés.

Lien sur lastprivate.c.

```
#include <stdio.h>
#include "omp_repair.h"
#define SIZE 20
int main() {
  int i,a, n=SIZE;
#pragma omp parallel for private(i) lastprivate(a)
  for (i=0; i<n; i++)
  {
    a = i+1;
    printf("Thread %d a = %d i = %d\n", omp_get_thread_num(),a,i);
  }
printf("Value of a after parallel for: a = %d\n",a);
}
```

Dans le cas des variables partagées, on peut rapidement mettre en évidence un problème de cohérence. Pour celà, il suffit de préciser un nombre de threads important, par exemple 1000, avant l'exécution du programme suivant.

Lien sur shared.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main(int argc, char *argv[]) {
  int b = 0;
#pragma omp parallel shared(b)
  {
        b++;
    printf("Incrementation par thread %d de %d \n", omp_get_thread_num(),
omp_get_num_threads());
  }
  printf("Valeur finale : %d\n", b);
}
```

Le problème mis en évidence avec ce résultat incohérent est qu'une variable partagée (ici **b**) doit être protégée si on veut que plusieurs threads la modifient.

On a besoin de garantir l'exclusion mutuelle entre plusieurs threads pour réaliser ces modifications. Deux solutions sont proposées en fonction de la granularité de la section critique (le nombre et la complexité des opérations). Pour une opération simple :

Lien sur atomic.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main() {
int count = 0;
#pragma omp parallel shared(count)
{
  #pragma omp atomic
  //count = count + 1; //Not working?
  count++;
}
printf("%d\n", count);
}
```

Pour un block d'instructions plus complexe :

Lien sur critical.c.

```
#include <stdio.h>
#include "omp_repair.h"
int main() {
  int b = 0;
#pragma omp parallel shared(b)
  {
#pragma omp critical
    {
      int local; /* une variable locale au thread */
      local=b; /* recopie le contenu de la variable partagée localement */
      local++; /* incrémente la variable locale*/
      b=local; /* recopie le résulat dans la variable partagée*/
    }
  }
  printf("%d\n", b);
}
```

5. Conclusion

Il reste à étudier les exemples suivants.

Pour "déplier" une boucle **for** simplement :

- parallelfor.c
- parallelfor+.c
- schedule.c

Pour consolider un résultat global sur la base de résultats partiels obtenus par les différents threads :

- reduction+.c
- reduction.c

Pour conserver un ordre correspondant à l'ordre séquentiel d'une boucle sur une partie de la section:

- ordered.c
- ordered2.c