UMONS
Université de Mons

Faculté Polytechnique
Service d'Informatique

Informatique Parallèle et Distribuée

Notes de laboratoire

Pr. Manneback Pierre
Dr. Sidi Ahmed Mahmoudi
Ir. Françoi Rolan
Ir. Michel Bagein

POLYTECH MONS

Année académique 2017 - 2018
Version 2.0

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

UMONS
Université de Mons

Faculté Polytechnique
Service d'Informatique

Informatique Parallèle et Distribuée

Notes de laboratoire

Pr. Manneback Pierre
Dr. Sidi Ahmed Mahmoudi
Ir. Françoi Rolan
Ir. Michel Bagein

POLYTECH MONS

Année académique 2017 - 2018
Version 2.0

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

Contents

Chapter 1

OpenMP

1.1 Introduction

Ce chapitre présente OpenMP et ses différentes fonctionnalités. Après une brève introduction à OpenMP, nous présentons les différents composants qu'offre cette interface de programmation :

1. les directives OpenMP
2. les clauses OpenMP
3. les routines OpenMP

1.2 Présentation

OpenMP [1] est une interface de programmation pour le multithreading. Elle facilite la programmation d'applications parallèles à mémoire partagée en langage C/C++ ou Fortran depuis 1997. Les spécifications de cette interface sont développées par l'association sans but lucratif "OpenMP Architecture Review Board" (OpenMP ARB) [1] et implémentées par la librairie GOMP [2] inclue dans le compilateur GCC.

1.3 Utilité

La parallélisation d'un code source avec OpenMP s'effectue à l'aide de directives de compilation. Une directive de compilation est une ligne du code source qui débute par `#pragma` et qui contient des mots-clés indiquant au compilateur des actions à effectuer lors de la compilation. Toutes les directives propres à OpenMP commencent par le mot-clé `omp` (`#pragma omp`) et portent des indications sur la manière dont l'application doit être parallélisée. Cette parallélisation est effectuée par défaut à l'aide de threads POSIX (paradigme de mémoire partagée).

La parallélisation peut être spécifiée de trois manières :

1. Les boucles : la directive `#pragma omp for` spécifie que les itérations d'une boucle `for` doivent être exécutées en parallèle par plusieurs threads.
2. Les sections : les directives `#pragma omp sections` et `#pragma omp section` spécifient un ensemble de sections qui doivent être exécutées en parallèle.
3. Les tâches : la directive `#pragma omp task` délimite une partie du code qui doit être encapsulée dans une tâche et exécutée en parallèle par un thread différent du thread appelant.

Des clauses peuvent être jointes à ces directives pour affiner la parallélisation :

- `Private` : les threads générés se partagent le même espace mémoire et donc les même variables. `private` identifie les variables pour lesquelles les threads doivent avoir une copie locale.
- `If` : indique une condition à la parallélisation (nombre d'itérations de la boucle, taille d'une variable ...).
- `Reduction` : spécifie une opération de réduction (sommer les éléments d'un tableau réparti entre plusieurs threads dans un même emplacement mémoire ...).
- `Schedule` : spécifie l'ordonnancement des itérations de la boucle parallélisée.
- ...

Il existe également des directives influençant les flux d'exécution, comme :

- `Barrier` : place une barrière dans le code à laquelle tous les threads se synchronisent.
- `Master` : indique que seul le thread maître peut exécuter la section délimitée par la directive.
- `Critical` : délimite une section de code qui ne peut être exécutée que par un seul thread.
- ...

Enfin, des routines permettent de spécifier le nombre de threads devant être utilisé dans une section parallèle (par défaut, il y a autant de threads que de coeurs processeur), identifier le numéro du thread appelant, créer et utiliser des mutex ...

Le modèle de programmation d'OpenMP utilise le multithreading à mémoire partagée. Ce partage de la mémoire impose au programme de rester au sein d'un ordinateur unique[1], mais cela évite les transferts de données entre les fils d'exécution. Le partage de la mémoire peut entrainer l'apparition de sections critiques dans lesquelles les threads doivent être synchronisés pour éviter les conflits.

1.4 Les directives OpenMP

Cette section et les suivantes se basent sur les informations contenues dans [3] et [4].

1.4.1 `Parallel`

Syntaxe

```
#pragma omp parallel [clauses]
```

Les clauses employables sont prsentes la figure 1.1.

Description

La directive permet de spcifier une section de code qui sera excute en parallle par plusieurs threads. Lorsque le processus rencontre cette directive, il cre un ensemble de threads contenant autant de threads que de coeurs processeur. Le thread maitre est le thread numro 0. Les threads se synchronisent la fin de la section parallle (par exemple, aprs la fonction `sum` du listing 1.1).

Les directives comme `for`, `section` ... qui indiquent la manire de parallliser les calculs doivent toutes se trouver dans un bloc `parallel`.

Il est interdit de sortir du bloc `parallel` l'aide d'une instruction `break` ou autre.

[1] Sauf mise en place de mécanisme particulier comme un espace de mémoire global ou virtuellement partagé.

Illustration

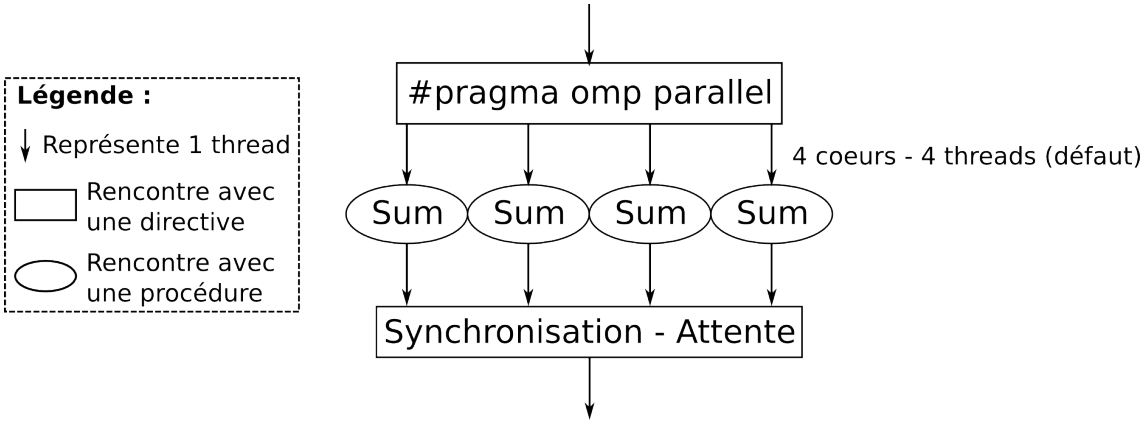

Figure 1.1: Parallel - Quatre threads effectuent chacun la fonction `sum`

Listing 1.1: Illustration de la directive `parallel` : somme des lments d'un tableau

```
1  void sum(){
2      int i, sum = 0;
3      for (i=0; i<1000; ++i)
4          sum += i;
5      printf("Thread %d a calcul la somme %d\n ", omp_get_thread_num(), sum);
6  }
7
8  int main(){
9      #pragma omp parallel
10     sum();
11     printf("Fin\n");
12 }
13
14 [fremals@fermi01 OpenMP]$ ./parallel_exe
15 Thread 0 a calcul la somme 499500
16 Thread 3 a calcul la somme 499500
17 Thread 2 a calcul la somme 499500
18 Thread 1 a calcul la somme 499500
19 Fin
```

1.4.2 For

Syntaxe

`#pragma omp for [clauses]`

Les clauses employables sont prsentes la figure 1.1.

Description

La directive `for` spcifie que les itrations d'une boucle doivent tre excutes en parallle par l'ensemble de threads. Par dfaut, OpenMP divise l'ensemble des itrations en autant d'ensembles conscutifs qu'il y a de threads. Chaque ensemble est excut par un thread. Une barrire implicite se situe la fin du bloc `for`. Le valeur du gardien de boucle ne peut pas changer au cours de l'excution des itrations de la boucle.

Il est possible d'crire directement `#pragma omp parallel for` comme prsent au listing 1.2.

Il est interdit de sortir du bloc `for` l'aide d'une instruction `break` ou autre.

Illustration

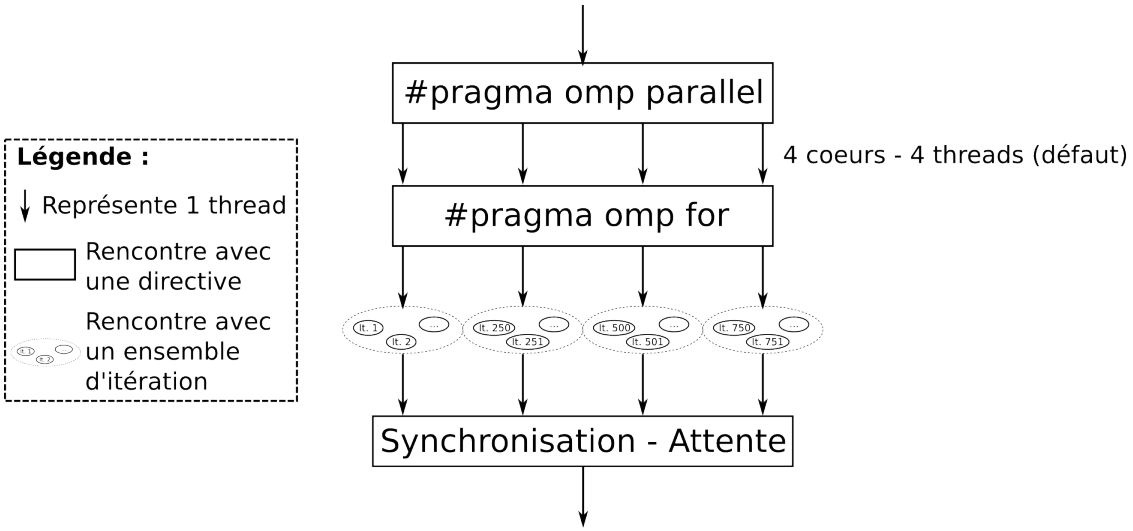

Figure 1.2: For - Quatre threads effectuent chacun 250 itrations de la boucle

Listing 1.2: Illustration de la directive `for` : addition de matrice

```
1  int main(){
2      int i, sum=0;
3      #pragma omp parallel for reduction (+:sum)
4      for (i =0; i<8; i++){
5          sum += i;
6          printf("Thread %d additionne %d\n " , omp_get_thread_num(), i);
7      }
8      printf("Somme finale : %d \n ", sum);
9  }
10
11 Alternative :
12
13 int main(){
14     int i, sum=0;
15     #pragma omp parallel
16     #pragma omp for reduction (+:sum)
17     for (i =0; i<8; i++){
18         sum += i;
19         printf("Thread %d additionne %d\n " , omp_get_thread_num(), i);
20     }
21     printf("Somme finale : %d \n ", sum);
22 }
23
24 [fremals@fermi01 OpenMP]$ ./for_exe
25 Thread 0 additionne 0
26 Thread 0 additionne 1
27 Thread 3 additionne 6
28 Thread 3 additionne 7
```

```
Thread 1 additionne 2
Thread 1 additionne 3
Thread 2 additionne 4
Thread 2 additionne 5
Somme finale : 28
```

Comme la variable `sum` est partage, il faut indiquer OpenMP qu'elle fait l'objet d'une opration de rduction. La clause `reduction(+:sum)` indique que chaque thread excute sa somme locale sur une copie locale de la variable `sum` et, une fois les itrations excutes, que les sommes locales sont additionnes dans la variable `sum` du thread maitre. Pour plus d'information sur cette clause, veuillez consulter la section 1.5.6.

1.4.3 Sections/Section

Syntaxe

```
#pragma omp sections [clauses]

    #pragma omp section
```

Les clauses employables sont prsentes la figure 1.1.

Description

Les directives `section` doivent se trouver au sein d'un bloc `sections`. Les diffrentes `section` appartenant un mme bloc `sections` sont excutes en parallle, chacune par un thread. Une barrire implicite se trouve la fin du bloc `sections`.

Il est interdit de sortir du bloc `section` l'aide d'une instruction `break` ou autre.

Illustration

Listing 1.3: Illustration des directives `sections` et `section` : addition de matrice

```
void computation1(int * tab, int size){
        int i;
        for(i=0; i<size; ++i)
                tab[i]+=i;
        printf("Thread %d a t r a i t  le tableau 1\n ", omp_get_thread_num());
}

void computation2(int * tab, int size){
        int i;
        for(i=0; i<size; ++i)
                tab[i]*=i;
        printf("Thread %d a t r a i t  le tableau 2\n ", omp_get_thread_num());
}

int main(){
        int tab1[3], tab2[6];
        int i;

        for(i=0;i<3;++i){
                tab1[i] = tab2[i] = i;
        }
        for(;i<6;++i){
                tab2[i] = i;
        }

```

```
26          #pragma omp parallel
27          #pragma omp sections
28          {
29              #pragma omp section
30              computation1(tab1, 3);
31              #pragma omp section
32              computation2(tab2,6);
33          }
34
35          for(i=0;i<3;++i){
36                  printf("Tab1[%d] : %d\n", i, tab1[i]);
37          }
38          for(i=0;i<6;++i){
39                  printf("Tab2[%d] : %d\n", i, tab2[i]);
40          }
41  }
42
43  [fremals@fermi01 OpenMP]$ ./section_exe
44  Thread 0 a t r a i t  le tableau 2
45  Thread 2 a t r a i t  le tableau 1
46  Tab1[0] : 0
47  Tab1[1] : 2
48  Tab1[2] : 4
49  Tab2[0] : 0
50  Tab2[1] : 1
51  Tab2[2] : 4
52  Tab2[3] : 9
53  Tab2[4] : 16
54  Tab2[5] : 25
```

1.4.4 Task

Syntaxe

`#pragma omp task [clauses]`

Les clauses employables sont prsentes la figure 1.1.

Description

La directive `task` dlimite une partie du code devant tre encapsule dans une tche. Tous les threads atteignant cette directive crent une nouvelle tche. Il faut donc veiller contrler le nombre de thread atteignant cette directive pour viter de crer plus de tche que ncessaire. Les tches sont excutes en parallle par l'ensemble de threads. Une tche n'est pas forcment excute par le thread qui l'a cre.

Il est interdit de sortir du bloc `task` l'aide d'une instruction `break` ou autre.

Illustration

Listing 1.4: Illustration de la directive `task` : excution de calculs intensifs en parallle

```
1  void computation1(int * tab, int size){
2          int i;
3          for(i=0; i<size; ++i)
4                  tab[i]+=i;
5          printf("Thread %d a t r a i t  le tableau 1\n ", omp_get_thread_num());
6  }
7
```

```
8  void computation2(int * tab, int size){
9          int i;
10         for(i=0; i<size; ++i)
11                 tab[i]*=i;
12         printf("Thread %d a  t r a i t  le tableau 2\n ", omp_get_thread_num());
13 }
14
15 int main(){
16         int tab1[3], tab2[6];
17         int i;
18
19         for(i=0;i<3;++i){
20                 tab1[i] = tab2[i] = i;
21         }
22         for(;i<6;++i){
23                 tab2[i] = i;
24         }
25
26         omp_set_num_threads(2);
27
28         #pragma omp parallel
29         {
30             #pragma omp task
31             computation1(tab1, 3);
32             #pragma omp task
33             computation2(tab2,6);
34         }
35
36         for(i=0;i<3;++i){
37                 printf("Tab1[%d] : %d\n", i, tab1[i]);
38         }
39         for(i=0;i<6;++i){
40                 printf("Tab2[%d] : %d\n", i, tab2[i]);
41         }
42 }
43
44 [fremals@fermi01 OpenMP]$ ./task_exe
45 Thread 0 a  t r a i t  le tableau 1
46 Thread 0 a  t r a i t  le tableau 1
47 Thread 0 a  t r a i t  le tableau 2
48 Thread 1 a  t r a i t  le tableau 2
49 Tab1[0] : 0
50 Tab1[1] : 3
51 Tab1[2] : 6
52 Tab2[0] : 0
53 Tab2[1] : 1
54 Tab2[2] : 8
55 Tab2[3] : 27
56 Tab2[4] : 64
57 Tab2[5] : 125
```

Grce la fonction `omp_set_num_threads`, les sections parallle sont excutes par un ensemble de 2 threads. Chacun des deux threads rencontre les directives `task` et cre deux tches. Nous vrifions que les tches ne sont pas forcment consommes par le thread qui les a cr.

1.4.5 Single

Syntaxe

```
#pragma omp single [clauses]
```

Les clauses employables sont prsentes la figure 1.1.

Description

La directive `single` dlimite une rgion du code qui est excute une seule fois par un seul thread. Une barrire implicite est place la fin de cette rgion.

Il est interdit de sortir du bloc `single` l'aide d'une instruction `break` ou autre.

Illustration

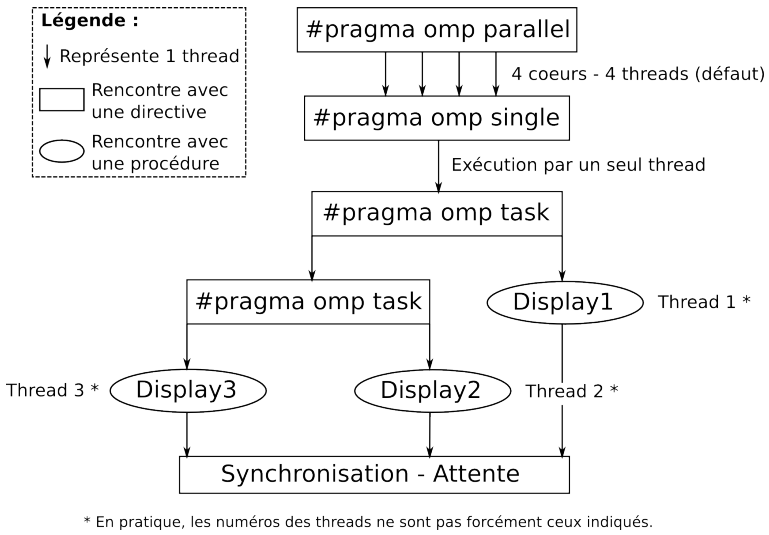

Figure 1.3: Droulement de la paralllisation lors de l'utilisation des directives `Task` et `Single`

Listing 1.5: Illustration de la directive `single` : cration de tches sans doublon

```
1  void computation1(int * tab, int size){
2          int i;
3          for(i=0; i<size; ++i)
4                  tab[i]+=i;
5          printf("Thread %d a t r a i t  le tableau 1\n ", omp_get_thread_num());
6  }
7
8  void computation2(int * tab, int size){
9          int i;
10         for(i=0; i<size; ++i)
11                 tab[i]*=i;
12         printf("Thread %d a t r a i t  le tableau 2\n ", omp_get_thread_num());
13 }
14
15 int main(){
16         int tab1[3], tab2[6];
17         int i;
18
19         for(i=0;i<3;++i){
20                 tab1[i] = tab2[i] = i;
21         }
22         for(;i<6;++i){
23                 tab2[i] = i;
24         }
25
26         omp_set_num_threads(2);
27
```

```
28          #pragma omp parallel
29          #pragma omp single
30          {
31              #pragma omp task
32              computation1(tab1, 3);
33              #pragma omp task
34              computation2(tab2,6);
35          }
36
37          for(i=0;i<3;++i){
38                  printf("Tab1[%d] : %d\n", i, tab1[i]);
39          }
40          for(i=0;i<6;++i){
41                  printf("Tab2[%d] : %d\n", i, tab2[i]);
42          }
43  }
44
45  [fremals@fermi01 OpenMP]$ ./single_exe
46  Thread 1 a t r a i t  le tableau 1
47  Thread 0 a t r a i t  le tableau 2
48  Tab1[0] : 0
49  Tab1[1] : 2
50  Tab1[2] : 4
51  Tab2[0] : 0
52  Tab2[1] : 1
53  Tab2[2] : 4
54  Tab2[3] : 9
55  Tab2[4] : 16
56  Tab2[5] : 25
```

La section crant les tches est excute par un seul thread et une seule tche est cre pour chacun des calculs. Les tches sont ensuite rparties entre les threads existant.

1.4.6 Barrier

Syntaxe

`#pragma omp barrier`

Description

La directive `barrier` synchronise les threads : lorsqu'un thread rencontre cette directive, il arrte son excution jusqu' ce que tous les threads l'aient atteint. Tous les threads de la section parallle doivent donc rencontrer cette directive sous peine de blocage du programme.

Illustration

Listing 1.6: Illustration de la directive `barrier` : mise en vidence des barrire implicites

```
1  int main(){
2      int i, sum=0;
3      #pragma omp parallel
4      {
5          double startTime = omp_get_wtime();
6          while( (omp_get_wtime() - startTime) < (double)(omp_get_thread_num()));
7          printf("%f - Thread %d  c o m p l t \n", omp_get_wtime(), omp_get_thread_num());
8          #pragma omp barrier
9          printf("%f -  B a r r i r e  p a s s e \n", omp_get_wtime());
10     }
```

```
}

[fremals@fermi01 OpenMP]$ ./barrier_exe
15599.287649 - Thread 0  c o m p l t
15600.287649 - Thread 1  c o m p l t
15601.287648 - Thread 2  c o m p l t
15602.287649 - Thread 3  c o m p l t
15602.287680 -  B a r r i r e  p a s s e
15602.287686 -  B a r r i r e  p a s s e
15602.287681 -  B a r r i r e  p a s s e
15602.287757 -  B a r r i r e  p a s s e
```

1.4.7 Master

Syntaxe

`#pragma omp master`

Description

La directive `master` dlimite une rgion qui ne sera excute que par le thread matre (les autres threads sautant la rgion et continuant excuter le reste du code). Il n'y a pas de barrire implicite la fin de la rgion.

Il est interdit de sortir du bloc `master` l'aide d'une instruction `break` ou autre.

Illustration

Listing 1.7: Illustration de la directive `Master` : affichage de rsultats

```
int main(){
    int i, a[5];
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;
        #pragma omp master
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);
        #pragma omp barrier
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}

[fremals@fermi01 OpenMP]$ ./master_exe
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
```

1.4.8 Critical

Syntaxe

```
#pragma omp critical (name)
```

Description

La directive dlimite une rgion de code qui sera excute par un seul thread la fois. Lorsqu'un thread atteint une section critique en cours d'excution par un autre thread, il attend jusqu' ce que ce dernier soit sorti de la rgion.

Les sections critiques peuvent tre nommes. Plusieurs sections critiques nommmes de la mme manire sont traites comme une seule et mme section. Les sections critiques portant des noms diffrents peuvent tre excutes en parallle. Toutes les sections critiques non nommes sont traites comme une mme section.

Il est interdit de sortir du bloc `critical` l'aide d'une instruction `break` ou autre.

Illustration

Listing 1.8: Illustration de la directive `Critical` : incrmentation d'une variable

```
1  int main(){
2      int x = 1, y=2;
3      #pragma omp parallel
4      {
5          #pragma omp critical (incr)
6          x += x;
7
8          #pragma omp critical (mul)
9          y *= y;
10     }
11     printf("x : %d, y : %d\n", x, y);
12 }
13
14 [fremals@fermi01 OpenMP]$ ./critical_exe
15 x : 16, y : 65536
```

1.4.9 Atomic

Syntaxe

```
#pragma omp atomic
```

Description

La directive `atomic` permet de spcifier la prsence d'une opration demandant une mise jour atomique de la mmoire. Cette directive fournit ainsi une section critique plus restreinte que celle procure par `critical`.

L'opration de mise jour ne peut tre quelconque, elle doit obir ces rgles :

1. au niveau syntaxique, elle s'crit `var op= expr`, `opop var` ou `var opop`
2. l'oprateur `op` peut tre :

   `+, -, *, /, ^, \&, |, <<, >>`

3. la variable doit tre scalaire

Illustration

Listing 1.9: Illustration de la directive `atomic` : incrmentations d'une variable

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

int main(){
    int x = 1;
    #pragma omp parallel
    {
        #pragma omp atomic
        ++x;
    }
    printf("x : %d\n", x);
}

[fremals@fermi01 OpenMP]$ ./atomic_exe
x : 5
```

1.4.10 `Taskwait`

Syntaxe

`#pragma omp taskwait`

Description

La directive `taskwait` spcifie que la tche qui la rencontre attend l'achvement des tches filles gnres depuis le dbut de la tche courrante.

1.5 Les clauses OpenMP

Les clauses OpenMP permettent d'affiner la parallélisation. Leurs compatibilités avec les directives sont présentées à la figure 1.1.

Table 1.1: Liste incomplète des clauses pouvant être appliquées aux directives

Clauses	Parallel	For	Sections	Single	Parallel For	Task
If	•				•	•
Private	•	•	•	•	•	•
Shared	•	•			•	•
Firstprivate	•	•	•	•	•	•
Lastprivate		•	•		•	
Reduction	•	•	•		•	
Default	•				•	•
Schedule		•			•	
Ordered		•			•	
Nowait		•	•	•		
Num_threads	•	•	•		•	

1.5.1 If

Syntaxe

`if(condition)`

Description

La clause `if` conditionne la paralllisation : si l'expression de la clause est value comme tant fausse, la rgion associe la directive est xcute squentiellement par un thread unique.

Illustration

Listing 1.10: Illustration de la clause `if`

```
1  int main(){
2      int cond = 0;
3      while(cond < 2){
4          #pragma omp parallel if (cond)
5          if (omp_in_parallel()){
6              #pragma omp single
7              printf("Condition : %d, parallelis (%d threads).\n", cond,
8                          omp_get_num_threads());
9          }
10         else{
11             printf("Condition : %d, serialis (%d thread).\n", cond,
12                 omp_get_num_threads());
13         }
14         ++cond;
15     }
16 }
17
18 [fremals@fermi01 OpenMP]$ ./if_exe
19 Condition : 0, serialis (1 thread).
20 Condition : 1, p a r a l l lis (4 threads).
```

1.5.2 Shared

Syntaxe

`shared (liste)`

Description

La clause `shared(liste)` indique la `liste` des variables qui doivent tre partages entre les diffrents threads. Les variables sont par dfaut partages. Une variable partage se trouve dans un seul espace mmoire qui est accessible par tous les threads. Les variables partages peuvent engendrer des sections critiques dont l'accs doit tre gr par le programmeur.

1.5.3 Private

Syntaxe

`private (liste)`

Description

La clause `private(liste)` indique la liste des variables qu'il faut rendre prives. Lorsqu'une variable est prive, chaque thread en possde une copie locale. C'est--dire qu'une nouvelle variable, non initialise et du mme type que la variable prive, est utilise au sein de chaque thread la place de la variable prive. Toutes les rfrences la variable originale sont remplaces par les rfrences la nouvelle variable. La valeur d'une variable prive n'est pas accessible en dehors de la zone dans laquelle elle est dclare.

Illustration

Listing 1.11: Illustration de la clause `private` : affichage du numro des threads

```
int main(){
    int pr1 = 921, pr2 = 473;
    #pragma omp parallel private(pr1, pr2)
    {
        #pragma omp master
        printf("pr1 : %d, pr2 : %d.\n", pr1, pr2);
        #pragma omp barrier
        pr1 = omp_get_thread_num();
        pr2 = -omp_get_thread_num();
        printf("%d - pr1 : %d, pr2 : %d\n", omp_get_thread_num(), pr1, pr2);
    }
    printf("pr1 : %d, pr2 : %d\n", pr1, pr2);
}

[fremals@fermi01 OpenMP]$ ./private_exe
pr1 : 32767, pr2 : 1535201808.
0 - pr1 : 0, pr2 : 0
3 - pr1 : 3, pr2 : -3
1 - pr1 : 1, pr2 : -1
2 - pr1 : 2, pr2 : -2
pr1 : 921, pr2 : 473
```

1.5.4 Firstprivate

Syntaxe

`firstprivate (list)`

Description

La clause `firstprivate` rend prives les variables de la `list` et les initialise la valeur qu'elles possdent avant de pntrer dans la rgion concerne.

Illustration

Listing 1.12: Illustration de la clause `firstprivate` : affichage du numro des threads

```
int main(){
    int pr1 = 921, pr2 = 473;
    #pragma omp parallel firstprivate(pr1, pr2)
    {
        #pragma omp master
        printf("pr1 : %d, pr2 : %d.\n", pr1, pr2);
```

```
    }
}

[fremals@fermi01 OpenMP]$ ./firstprivate_exe
pr1 : 921, pr2 : 473.
}
```

1.5.5 Lastprivate

Syntaxe

`lastprivate (list)`

Description

La clause **lastprivate** rend prives les variables de la **list** et copie la valeur que possde la variable prive la dernire itration ou section dans la variable d'origine.

Illustration

Listing 1.13: Illustration de la clause **lastprivate** : affichage du numro des threads

```
int main(){
    const int N = 10;
    double x, a[N];
    int i;

    #pragma omp parallel for lastprivate(x)
    for(i=0; i<=N; ++i){
        x = sin(i);
        a[i] = exp(x);
        printf(" I t ration  %d : %f - %f\n", i, x, a[i]);
    }
    printf("x : %f\n", x);
}

[fremals@fermi01 OpenMP]$ ./lastprivate_exe
 I t ration  0 : 0.000000 - 1.000000
 I t ration  1 : 0.841471 - 2.319777
 I t ration  4 : -0.756802 - 0.469164
 I t ration  5 : -0.958924 - 0.383305
 I t ration  2 : 0.909297 - 2.482578
 I t ration  3 : 0.141120 - 1.151563
x : -0.958924
}
```

1.5.6 Reduction

Syntaxe

`reduction (oprateur :  liste)`

Description

La clause **reduction** spcifie que les variables de la liste sont prives et qu'une opration de rduction, spcifie par l'oprateur, aura lieu sur elles la fin du bloc.

Les oprateurs possibles pour la rduction sont :

`+, *, -, &, ^, |, &&, or ||`

Illustration

Listing 1.14: Illustration de la clause `reduction`

```
1  int main(){
2      int sum, i, N=10;
3
4      #pragma omp parallel for reduction(+:sum)
5      for (i=0; i<N; i++) {
6          sum += i;
7      }
8      printf("sum : %d\n", sum);
9  }
10
11 [fremals@fermi01 OpenMP]$ ./reduction_exe
12 sum : 45
```

1.5.7 Default

Syntaxe

`default(shared|none)`

Description

La clause `default` spcifie la porte par dfaut des variables au sein de la rgion parallle. La clause `default(shared)` garde les variables partages et la clause `default(none)` entraine une erreur de compilation si une variable est utilise dans la rgion parallle sans que sa porte n'ait t dfinie l'aide d'une des clauses `Private`, `Shared`, `Firstprivate`, `Lastprivate` ou `Reduction`.

1.5.8 Schedule

Syntaxe

`schedule(type [, taille])`

Description

La clause `schedule` spcifie l'ordonnancement de l'excution des itrations d'une boucle parallelise. Il y a 4 types d'ordonnancement possibles :

1. `static` : l'ensemble des itrations est divis en blocs de taille spcifie par `taille` (si rien n'est spcifi, les itrations sont diviss en autant de blocs qu'il y a de threads pour les traiter ; les tailles de ces blocs peuvent tre lgrement diffrentes) et les blocs sont assigns statiquement aux diffrents thread selon l'algorithme Round-Robin dans l'ordre des numros des threads.

2. `dynamic` : l'ensemble des itrations est divis en blocs de taille spcifie par `taille` (si rien n'est spcifi, la taille par dfaut est 1) et les blocs sont dynamiquement attribus aux threads en attente (lorsqu'un thread a trait un bloc, il attend d'tre assign un autre bloc jusqu' ce qu'il n'y en aie plus).

3. `guided` : la taille des blocs d'itration est dynamique ; si la `taille` est mise 1, la taille des blocs d'itration est proportionelle au nombre d'itrations non assignes divis par le nombre de threads. Si la `taille` est mise la valeur k, la taille des blocs est de minimum k itrations, sauf pour le bloc final. La `taille` par dfaut est 1.

4. `runtime` : l'ordonnancement est dcide lors de l'excution via la variable d'environnement `OMP_SCHEDULE`. Il n'est pas possible de spcifier une `taille`.

Illustration

Listing 1.15: Illustration de la clause `schedule`

```
#define NUM_LOOPS 18
#define SLEEP_EVERY_N 6
#define STATIC_CHUNK 3

#define DYNAMIC_CHUNK 3

int main(int argc, char * argv){
    int i, nStatic[NUM_LOOPS], nDynamic[NUM_LOOPS], nGuided[NUM_LOOPS];

    #pragma omp parallel
    {
        #pragma omp for schedule(static, STATIC_CHUNK)
        for (i=0; i<NUM_LOOPS; ++i)
            nStatic[i] = omp_get_thread_num( );

        #pragma omp for schedule(dynamic, DYNAMIC_CHUNK)
        for (i=0; i<NUM_LOOPS; ++i)
            nDynamic[i] = omp_get_thread_num( );

        #pragma omp for schedule(guided)
        for (i=0; i<NUM_LOOPS; ++i)
            nGuided[i] = omp_get_thread_num( );
    }

    printf("Static : ");
    for(i=0; i<NUM_LOOPS; ++i)
        printf("%d ", nStatic[i]);
    printf("\n");

    printf("Dynamic : ");
    for(i=0; i<NUM_LOOPS; ++i)
        printf("%d ", nDynamic[i]);
    printf("\n");

    printf("Guided : ");
    for(i=0; i<NUM_LOOPS; ++i)
        printf("%d ", nGuided[i]);
    printf("\n");
}

[fremals@fermi01 OpenMP]$ ./schedule_exe
Static : 0 0 0 1 1 1 2 2 2 3 3 3 0 0 0 1 1 1
Dynamic : 0 0 0 2 2 2 1 1 1 3 3 3 0 0 0 3 3 3
Guided : 1 1 1 1 1 0 0 0 0 3 3 3 1 1 1 0 0 1
}
```

1.5.9 Ordered

Syntaxe

```
#pragma omp ordered
```

Description

La clause / directive **ordered** permet de spcifier que les itrations, ou une partie de celles-ci, doivent tre excutes squentiellement. Avant d'excuter ses itrations, un thread doit attendre que les itrations prcdentes soient acheves.

Illustration

Listing 1.16: Illustration de la clause **ordered**

```
1  int main(){
2      const int N = 5;
3      double x, a[N];
4      int i;
5
6      #pragma omp parallel for lastprivate(x) ordered
7      for(i=0; i<=N; ++i){
8          x = sin(i);
9          a[i] = exp(x);
10         printf(" I t ration  %d : %f - %f\n", i, x, a[i]);
11     }
12
13     printf("x : %f\n", x);
14 }
15
16 [fremals@fermi01 OpenMP]$ ./ordered_exe
17  I t ration  0 : 0.000000 - 1.000000
18  I t ration  1 : 0.841471 - 2.319777
19  I t ration  2 : 0.909297 - 2.482578
20  I t ration  3 : 0.141120 - 1.151563
21  I t ration  4 : -0.756802 - 0.469164
22  I t ration  5 : -0.958924 - 0.383305
23 x : -0.958924
```

1.5.10 Clause Nowait

Syntaxe

```
nowait
```

Description

La clause **nowait** permet d'annuler la barrire implicite en fin de bloc. Une fois qu'un thread n'a plus de travail, il ne se synchronise pas avec les autres threads et continue son excution.

Listing 1.17: Illustration de la clause **nowait**

```
1  int main(){
2      int i = 0, N = 3;
3
4      #pragma omp parallel
```

```
5     {
6         #pragma omp for
7         for (i=0; i<N; ++i)
8             printf("Boucle 1,  i t ration  %d\n", i);
9
10        #pragma omp for nowait
11        for (i=0; i<N; ++i)
12            printf("Boucle 2,  i t ration  %d\n", i);
13
14        #pragma omp for
15        for (i=0; i<N; ++i)
16            printf("Boucle 3,  i t ration  %d\n", i);
17    }
18 }
19
20 [fremals@fermi01 OpenMP]$ ./nowait_exe
21 Boucle 1,  i t ration  0
22 Boucle 1,  i t ration  2
23 Boucle 1,  i t ration  1
24 Boucle 2,  i t ration  0
25 Boucle 3,  i t ration  0
26 Boucle 2,  i t ration  2
27 Boucle 3,  i t ration  2
28 Boucle 2,  i t ration  1
29 Boucle 3,  i t ration  1
```

1.6 Les routines OpenMP

De nombreuses routines permettent de paramétrer et d'obtenir des informations sur la parallélisation durant l'exécution du programme (le header `omp.h` doit être inclus pour pouvoir les utiliser) :

- `void omp_set_num_threads(int num_threads)` : spécifie la taille de l'ensemble de threads (entier positif) utilisé pour exécuter les régions de code parallèles. Cette routine doit être appelée depuis une section de code séquentiel.
- `int omp_get_num_threads(void)` : renvoie le nombre de threads en cours d'exécution.
- `int omp_get_thread_num(void)` : renvoie le numéro du thread appelant (le thread maître a le numéro 0).
- `int omp_get_num_procs(void)` : renvoie le nombre de coeurs, d'unités de travail que possède la machine.
- `void omp_set_dynamic(int dynamic_threads)` : permet d'activer ou de désactiver la possibilité d'ajuster la taille de l'ensemble de threads exécutant les régions parallèles (le mode dynamique est activé si la variable `dynamic_threads` est positive non nulle). Cette routine doit être appelée depuis une section de code séquentiel.
- `int omp_get_dynamic(void)` : renvoie 0 si l'ajustement dynamique des threads est activé et une valeur différente de zéro sinon.
- `double omp_get_wtime(void)` : renvoie le nombre de secondes écoulées depuis un certain point dans le passé
- `int omp_in_parallel(void)` : indique si la section de code exécutée est parallélisée (la valeur de retour est différente de zéro si c'est le cas et égale à zéro sinon).

- `void omp_init_lock(omp_lock_t *lock)` : initialise un verrou associé à la variable `lock` 1.18.
- `void omp_destroy_lock(omp_lock_t *lock)` : désassocie la variable `lock` de tout vérouillage.
- `void omp_set_lock(omp_lock_t *lock)` : force le thread appelant à attendre jusqu'à ce que le verrou soit libre (la variable `lock` doit avoir été au préalable initialisé).
- `void omp_unset_lock(omp_lock_t *lock)` : libère le verrou du thread appelant.
- `int omp_test_lock(omp_lock_t *lock)` : tente de vérouiller le verrou mais ne bloque pas si celui-ci l'est déjà.

Listing 1.18: Illustration du verrouillage au sein d'OpenMP

```
1  int main() {
2      omp_lock_t my_lock;
3      omp_init_lock(&my_lock);
4
5      #pragma omp parallel num_threads(4)
6      {
7          int tid = omp_get_thread_num( );
8          int i, j;
9
10
11         omp_set_lock(&my_lock);
12         printf("Thread %d - starting locked region\n", tid);
13         sleep(1);
14         printf("Thread %d - ending locked region\n", tid);
15         omp_unset_lock(&my_lock);
16
17     }
18
19     omp_destroy_lock(&my_lock);
20 }
21
22 [fremals@fermi01 OpenMP]$ ./lock_exe
23 Thread 2 - Entrant dans la r gion critique
24 Thread 2 - Sortant de la r gion critique
25 Thread 3 - Entrant dans la r gion critique
26 Thread 3 - Sortant de la r gion critique
27 Thread 0 - Entrant dans la r gion critique
28 Thread 0 - Sortant de la r gion critique
29 Thread 1 - Entrant dans la r gion critique
30 Thread 1 - Sortant de la r gion critique
```

Chapter 2

OpenMPI

2.1 Introduction

Ce chapitre introduit l'utilisation d'OpenMPI. Il commence par prsenter brivement MPI et Open-MPI puis prsente certaines fonctions OpenMPI.

2.1.1 Prsentation

MPI (Message Passing Interface) est une API[1] spcifiant la paralllisation de processus sur des systmes mmoire distribue. Les communications entre processus s'effectue par mission/rception de messages. Le projet a commenc en 1992 et en est aujourd'hui la version 2.0.

Open MPI est un projet open-source, fruit de la fusion de plusieurs projets, qui a vu le jour en 2003. Le but recherch au travers cette fusion est de fournir la meilleure librairie MPI. Ce projet permet d'intgrer les fonctions MPI des codes C/C++ ou Fortran, de compiler les codes modifis et d'excuter les programmes sur des systmes multi-coeurs mmoire distribue.

Le but de cette API est de permettre la paralllisation sur un ordinateur, sur plusieurs ordinateurs, sur plusieurs noeuds d'un cluster ... Contrairement OpenMP, la mmoire est distribue et les multiples processus MPI possdent leur propre mmoire. Les communications inter-processus sont assures par le paradigme de passage de messages (send/receive), ceci rendant les transferts de donnes plus coteux qu'avec OpenMP. C'est pourquoi il peut parfois tre intressant de mlanger OpenMP et MPI au sein d'un mme programme : MPI distribue alors les grosses tches entre les diffrentes machines, et OpenMP paralllise ces tches au sein de chaque machine.

OpenMPI offre des mthodes permettant l'initialisation et la terminaison des processus MPI. La portion de code comprise entre ces deux fonctions sera excute autant de fois qu'il y a de processus. L'utilisateur utilise le rang des processus pour filtrer et attribuer les traitements aux processus. Gnralement, un mcanisme maitre/esclave est utilis de telle manire que le maitre partage les donnes traiter entre les esclaves (et lui-mme) et intgre finalement le rsultat final.

De nombreuses fonctions permettant l'change de donnes entre les processus ont t implmentes. Il est ainsi possible d'envoyer des messages d'un metteur vers un destinataire de manire synchrone, de manire asynchrone, d'envoyer des messages d'un metteur vers un ensemble de processus (broadcasting) ou d'effectuer l'inverse (opration de rduction) ... Il y a galement une fonction permettant la synchronisation des processus.

[1] Application Programming Interface - Interface de programmation

2.1.2 Communicateurs

OpenMPI permet d'organiser des groupes de processus grce des communicateurs. Les communicateurs permettent d'organiser les communications entre les processus MPI. Il permettent de crer des groupes de processus et ainsi de dfinir les destinataires ou les metteurs de messages pour des fonctions telles que `MPI_Reduce`, `MPI_Bcast` ...

Dans le cadre de ce cours, nous n'tudierons pas la gestion des communicateurs[2]. Voici un ensemble de communicateurs prexistant dans la librairie OpenMPI et qui pourront servir lors de l'emploi des fonctions de communication :

- `MPI_COMM_WORLD` : communicateurs contenant tous les processus MPI.
- `MPI_COMM_SELF` : contient uniquement le processus appelant (communication intra-processus).
- `MPI_COMM_NULL` : communicateur `NULL` (utile pour les fonctions o il est permis de passer un communicateur `NULL`).

2.1.3 Compilation et excution

La compilation d'un programme utilisant OpenMPI est particulire car elle ncessite l'emploi de la commande `mpicc`. Cette commande englobe le compilateur `gcc` et permet de compiler du code `C` et de lier au programme les librairies `MPI` ncessaires. Pour compiler un code, il suffit d'utiliser la ligne de commandes :

```
mpicc prog.c -o prog
```

Pour excuter un programme, il faut utiliser la commande `mpirun`. Ce script inspecte le systme et dmarre les jobs MPI en fonction du matriel disponible. La commande :

```
mpirun -np 2 prog
```

permet de lancer un programme `prog` qui sera excut par deux processus (np = number of processes).

Il est possible d'excuter des processus sur une autre machine que celle appelant le programme. Pour ce faire, il suffit d'indiquer le nom des machines htes cibles :

```
mpirun -np 2 --host host1,host2 prog
```

```
mpirun -np 2 --hostfile machines prog
```

L'option `--hostfile` requiert l'emploi d'un fichier, ici nomm `machines`, qui contient le nom des htes (un hte par ligne).

Comme les transferts de donnes s'effectue via une connexion SSH, il est intressant d'installer une cl SSH sur les htes. Ceci permet d'viter de devoir saisir le mot de passe ncessaire la connexion chaque excution. Voici la procdure suivre pour placer une cl SSH sur un hte :

```
[xxx@s-iglab-07a ~]$ ssh-keygen % cration de la cl
Generating public/private rsa key pair.
Enter file in which to save the key (/home/xxx/.ssh/id_rsa): % saisir Entre
Created directory '/home/xxx/.ssh'.
```

[2] Les personnes intresses peuvent examiner la fonction `MPI_Comm_create` qui permet la cration de communicateurs.

```
Enter passphrase (empty for no passphrase): % saisir Entre
Enter same passphrase again: % saisir Entre
Your identification has been saved in /home/xxx/.ssh/id_rsa.
Your public key has been saved in /home/xxx/.ssh/id_rsa.pub.
The key fingerprint is:
8e:e0:51:29:64:4d:27:60:c1:37:18:f0:00:e0:eb:e2 temp@s-iglab-07a
[xxx@s-iglab-07a ~]$ ssh-copy-id xxx@s-iglab-08a % envoie de la cl sur l'hte
s-iglab-08a
The authenticity of host 's-iglab-08a (10.107.45.38)' can't be established.
RSA key fingerprint is 60:53:7e:94:a1:7a:43:ab:e1:48:fa:61:64:00:68:75.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 's-iglab-08a,10.107.45.38' (RSA) to the list of
known hosts.
xxx@s-iglab-08a's password: % saisir le mot de passe de la session hte
Now try logging into the machine, with "ssh 'xxx@s-iglab-08a'", and check in:

  ~/.ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.
```

Il faut galement distribuer le fichier excutable sur la machines hte. Pour ce faire, excutez la commande :
`[xxx@s-iglab-07a  ]$ scp ./prog xxx@s-iglab-07a:~`
et le fichier sera alors plac dans le dossier de l'utilisateur `xxx` sur l'hte `s-iglab-07a`.

2.1.4 Appel bloquant - Appel non bloquant

Lorsqu'une routine est bloquante, son retour se fait une fois sa fonctionnalit excute. Si la routine est non-bloquante, son retour est immdiat, mme si sa fonctionnalit n'est pas compltement ralis.

2.2 Librairie de fonctions

Cette section se base sur les informations contenues dans [5], [6] et [7].

2.2.1 MPI_Init

Syntaxe C

```
#include <mpi.h>
      int MPI_Init(int *argc, char ***argv)
```

Description

Initialise l'environnement d'excution de MPI. L'appel cette fonction doit ncessairement prcder l'emploi de toute autre fonction d'OpenMPI. OpenMPI ne pouvant tre initialis qu'une fois, toute autre appel cette fonction rsultera en un chec.

Paramtres d'entre

`argc` : pointeur vers le nombre d'arguments.
`argv` : tableau d'arguments.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

`MPI_ERR_OTHER` : indique la tentative d'un second appel la routine `MPI_Init`, ce qui n'est pas permis.

Illustration

Listing 2.1: Illustration de la fonction `MPI_Init`

```
1 #include <mpi.h>
2
3 int main(int argc, char * argv[]){
4     MPI_Init(&argc, &argv);
5     /*Programme principal*/
6     MPI_Finalize();
7 }
```

2.2.2 MPI_Finalize

Syntaxe C

```
#include <mpi.h>
   int MPI_Finalize()
```

Description

Tous les processus doivent appeler cette routine avant de se terminer. Une fois l'appel cette routine effectu, il n'est plus possible d'appeler d'autres fonctions MPI, mme `MPI_Init` ou `MPI_Init_thread`[3].

Dans le cas des systmes multithread, il est conseill que le thread appelant `MPI_Finalize` soit le thread ayant appel `MPI_Init` ou `MPI_Init_thread`.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

Illustration

Listing 2.2: Illustration de la fonction `MPI_Finalize`

```
1 #include <mpi.h>
2
3 int main(int argc, char * argv[]){
4     MPI_Init(&argc, &argv);
5     /*Programme principal*/
6     MPI_Finalize();
7 }
```

[3]Les exceptions cette rgle sont les fonctions `MPI_Get_version` (indique la version d'OpenMPI utilise), `MPI_Initialized` (indique si `MPI_Init` a dj t appel), et `MPI_Finalized` (indique si `MPI_Finalize` a dj t appel).

2.2.3 MPI_Get_processor_name

Syntaxe C

```
#include <mpi.h>
   int MPI_Get_processor_name(char *name, int *resultlen)
```

Description

Cette routine permet de rcuprer le nom du processeur et donc d'identifier la machine hte. Le paramtre `name` contiendra le nom du processeur et doit au moins pouvoir accueillir `MPI_MAX_PROCESSOR_NAME` caractres. Le second paramtre contient le nombre de caractres du nom du processeur.

Paramtres de sortie

`name` : un identifiant unique pour le noeud de travail.
`resultlen` : la longueur de la chane pointe par `name`.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

Illustration

Listing 2.3: Illustration de la fonction `MPI_Get_processor_name`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char * argv[])
5  {
6      int size = 0, len = 0;
7      char name[MPI_MAX_PROCESSOR_NAME];
8      MPI_Init(&argc,&argv);
9      MPI_Get_processor_name(name, &len);
10     printf("Processor : %s (%d  c a r a c t r e s ).\n", name, len);
11     MPI_Finalize();
12 }
13
14 [fremals@fermi01 OpenMPI]$ mpirun -np 1 get_proc
15 Processor : fermi01 (7  c a r a c t r e s ).
```

2.2.4 MPI_Comm_size

Syntaxe C

```
#include <mpi.h>
   int MPI_Comm_size(MPI_Comm comm, int *size)
```

Description

Indique le nombre de processus se trouvant dans le groupe associ au communicateur `comm`.

Paramtres d'entre

`comm` : communicateur.

Paramtres de sortie

`size` : nombre de processus dans le groupe `comm`.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

`MPI_ERR_COMM` : le communicateur n'est pas valide. Une erreur frquente est d'utiliser un communicateur NULL.

`MPI_ERR_ARG` : un argument n'est pas valide.

Illustration

Listing 2.4: Illustration de la fonction `MPI_Comm_size`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char * argv[])
5  {
6      int size = 0;
7      MPI_Init(&argc,&argv);
8
9      MPI_Comm_size(MPI_COMM_WORLD,&size);
10     printf("Il y a %d processus dans le groupe\n", size);
11
12     MPI_Finalize();
13 }
14
15 [fremals@fermi01 OpenMPI]$ mpirun -np 2 ./comm_size
16 Il y a 2 processus dans le groupe
17 Il y a 2 processus dans le groupe
```

2.2.5 MPI_Comm_rank

Syntaxe C

```
#include <mpi.h>
   int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Description

Indique le rang (le numro) du processus appelant au sein du groupe de processus associ au communicateur `comm`. Le rang permet d'identifier un processus MPI et de lui assigner un ensemble d'instructions particulier.

Paramtres d'entre

`comm` : communicateur.

Paramtres de sortie

`rank` : rang du processus appelant au sein du groupe de processus associ `comm`.

Valeur de retour

MPI_SUCCESS : la routine s'est excute sans erreur.

MPI_ERR_COMM : le communicateur n'est pas valide. Une erreur frquente est d'utiliser un communicateur NULL.

Illustration

Listing 2.5: Illustration de la fonction MPI_Comm_rank

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

int main(int argc, char * argv[])
{
    int size = 0, rank = 0;
    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("[%d] Il y a %d processus dans le groupe.\n", rank, size);

    MPI_Finalize();
}

[fremals@fermi01 OpenMPI]$ mpirun -np 2 ./comm_rank
[0] Il y a 2 processus dans le groupe.
[1] Il y a 2 processus dans le groupe.
```

2.2.6 MPI_Send

Syntaxe C

```
#include <mpi.h>
   int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

Description

Effectue l'envoie bloquant d'un message. La routine bloque jusqu'au transfert du message du buffer d'envoi buf au buffer de rception (la boite au lettre du destinataire).

Paramtres d'entre

buf : l'adresse de l'emplacement mmoire contenant le message envoyer.

count : nombre d'lments de type datatype envoyer (entier non-ngatif).

datatype : type des lments envoys. Les types existant sont : MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG, MPI_LONG_DOUBLE[4].

dest : rang du processus destinataire dans le communicateur comm.

tag : une tiquette pour le message choisie par le programmeur, permet de filtrer les messages.

comm : communicateur.

[4]Il est possible de crer des types de donnes drivs, des types de donnes construits partir des types fournis par MPI, l'aide des routines MPI_Type_contiguous, MPI_Type_vector, MPI_Type_indexed, MPI_Type_struct, MPI_Type_commit, MPI_Type_free ...

Valeur de retour

MPI_SUCCESS : la routine s'est excute sans erreur.

MPI_ERR_COMM : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

MPI_ERR_COUNT : l'argument count est incorrect.

MPI_ERR_TYPE : l'argument datatype est incorrect.

MPI_ERR_TAG : l'argument tag est incorrect. Les tags doivent tre non-ngatifs ; la plus grande valeur permise pour ce paramtre est disponible dans l'attribut MPI_TAG_UB.

MPI_ERR_RANK : rang du destinataire incorrect. Le rang doit tre compris entre 0 et la taille du groupe du commincateur dcrment d'une unit.

Illustration

Listing 2.6: Illustration de la fonction MPI_Send

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char * argv[])
5  {
6      int size = 0, rank = 0;
7      MPI_Init(&argc,&argv);
8
9      MPI_Comm_size(MPI_COMM_WORLD,&size);
10     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11     if(rank == 0){
12         int tosend = 5;
13         MPI_Ssend(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
14         printf("[%d] Message  p o s t .\n", rank);
15     }
16     if(rank==1){
17         sleep(1);
18         MPI_Status status;
19         int torecv;
20         printf("[%d] Lecture de la boite au lettre.\n", rank);
21         MPI_Recv(&torecv, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
22         printf("[%d] Message  r e u  : %d.\n", rank, torecv);
23     }
24     MPI_Finalize();
25 }
26
27 [fremals@fermi01 OpenMPI]$ mpirun -np 2 send
28 [0] Envoie du message.
29 [0] Message  p o s t .
30 [1] Lecture de la boite au lettre.
31 [1] Message  r e u  : 5.
```

2.2.7 MPI_Isend

Syntaxe C

```
#include <mpi.h>
   int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm, MPI_Request *request)
```

Description

Commence l'envoi non-bloquant d'un message. L'appel alloue un objet reprsentant la requte de communication et l'associe la variable `request`. Cette variable peut ensuite tre utilise pour connaitre l'tat de la communication (`MPI_Test`) ou attendre son achvement (`MPI_Wait`).

L'appel `MPI_Isend` indique au systme qu'il peut commencer la copie des donnes en dehors du buffer d'envoi. Le buffer d'envoi ne doit pas tre accd tant que cette copie n'est pas finie.

Paramtres d'entre

`buf` : l'adresse de l'emplacement mmoire contenant le message envoyer.

`count` : nombre d'lments de type `datatype` envoyer (entier non-ngatif).

`datatype` : type des lments envoys. Les types existant sont : `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG`, `MPI_LONG_DOUBLE`[5].

`dest` : rang du processus destinataire dans le communicateur `comm`.

`tag` : une tiquette pour le message choisie par le programmeur et permettant de filtrer les messages l'arrive. La plus grande valeur possible pour ce paramtre est contenue dans la constante `MPI_TAG_UB`.

`comm` : nom du communicateur.

Paramtres de sortie

`request` : variable permettant de connaitre l'tat de la communication.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

`MPI_ERR_COMM` : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

`MPI_ERR_COUNT` : l'argument `count` est incorrect.

`MPI_ERR_TYPE` : l'argument `datatype` est incorrect.

`MPI_ERR_TAG` : l'argument `tag` est incorrect. Les tags doivent tre non-ngatifs.

`MPI_ERR_RANK` : rang du destinataire incorrect. Le rang doit tre compris entre 0 et la taille du groupe du commincateur dcrment d'une unit.

Illustration

Listing 2.7: Illustration de la fonction `MPI_Isend`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int computation(){
5          return rand()%10;
6  }
7
8  int main(int argc, char * argv[])
9  {
10     int size = 0, rank = 0;
11     MPI_Init(&argc,&argv);
12
```

[5] Il est possible de crer des types de donnes drivs, des types de donnes construits partir des types fournis par MPI, l'aide des routines `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_indexed`, `MPI_Type_struct`, `MPI_Type_commit`, `MPI_Type_free` ...

```
13      MPI_Comm_size(MPI_COMM_WORLD,&size);
14      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
15      MPI_Request sendState;
16      if(rank == 0){
17          sleep(1);
18          int tosend = computation(), temp;
19          MPI_Isend(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &sendState);
20          while(tosend!=9){
21              temp = computation(); //le buffer d'envoie ne peut  tre  modifi
22              MPI_Wait(&sendState, MPI_STATUS_IGNORE);
23              tosend = temp;
24              MPI_Isend(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &sendState);
25          }
26          MPI_Wait(&sendState, MPI_STATUS_IGNORE);
27      }
28      if(rank==1){
29          MPI_Status status;
30          int torecv=0;
31          while(torecv!=9){
32              MPI_Recv(&torecv, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
33              printf("Valeur calcule : %d\n", torecv);
34          }
35      }
36      MPI_Finalize();
37  }
38
39  [fremals@fermi01 OpenMPI]$ mpirun -np 2 isend
40  Valeur calcule : 3
41  Valeur calcule : 6
42  Valeur calcule : 7
43  Valeur calcule : 9
```

2.2.8 MPI_Recv

Syntaxe C

```
#include <mpi.h>
   int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

Description

Effectue la rception bloquante d'un message. La routine bloque jusqu' la rception du message. Un appel `Recv` peut rendre la main avant l'appel au `Send` correspondant.

Un message peut tre reu uniquement s'il est adress au processus rcepteur, et si les valeurs de la source, de l'tiquette et du communicateur correspondent celles spcifies dans la fonction de rception. Pour la source et l'tiquette, il existe deux valeurs, MPI_ANY_SOURCE et MPI_ANY_TAG, qui spcifient la rception de messages en provenance de n'importe quelle source (appartenant au groupe du communicateur spcifi) et portant n'importe quelle tiquette.

Un processus peut s'envoyer un message soi-mme, mais cela ncessite l'emploi d'une fonction d'envoie asynchrone pour viter tout blocage.

Paramtres d'entre

`count` : nombre maximum d'lments recevoir de type `datatype` (entier non-ngatif).
`datatype` : type des lments reus. Les types existant sont : `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`,

MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG, MPI_LONG_DOUBLE[6].

dest : rang du processus source dans le communicateur comm.

tag : une tiquette pour le message choisie par le programmeur et permettant de filtrer les messages. S'il n'y a pas besoin de filtrer les messages, le tag MPI_ANY_TAG peut tre utilis. La plus grande valeur possible pour ce paramtre est contenue dans la constante MPI_TAG_UB.

comm : communicateur.

Paramtres de sortie

buf : l'adresse de l'emplacement mmoire o le message doit tre stock.
status : structure contenant des informations sur le message reu :

- status.MPI_TAG : l'tiquette du message reu.
- status.MPI_SOURCE : l'metteur du message reu.

Valeur de retour

MPI_SUCCESS : la routine s'est excute sans erreur.

MPI_ERR_COMM : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

MPI_ERR_COUNT : l'argument count est incorrect.

MPI_ERR_TYPE : l'argument datatype est incorrect.

MPI_ERR_TAG : l'argument tag est incorrect. Les tags doivent tre non-ngatifs ; la plus grande valeur permise pour ce paramtre est disponible dans l'attribut MPI_TAG_UB. Dans le cas de la rception, l'tiquette peut valoir MPI_ANY_TAG.

MPI_ERR_RANK : rang de la source incorrect. Le rang doit tre compris entre 0 et la taille du groupe du commincateur dcrment d'une unit. Dans le cas de la rception, le rang peut valoir MPI_ANY_SOURCE.

Illustration

Listing 2.8: Illustration de la fonction MPI_Recv

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char * argv[])
5  {
6      int size = 0, rank = 0;
7      MPI_Init(&argc,&argv);
8
9      MPI_Comm_size(MPI_COMM_WORLD,&size);
10     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11     if(rank == 0){
12         int tosend = 5;
13         MPI_Ssend(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
14         printf("[%d] Message  p o s t .\n", rank);
15     }
16     if(rank==1){
17         sleep(1);
18         MPI_Status status;
19         int torecv;
```

[6]Il est possible de crer des types de donnes drivs, des types de donnes construits partir des types fournis par MPI, l'aide des routines MPI_Type_contiguous, MPI_Type_vector, MPI_Type_indexed, MPI_Type_struct, MPI_Type_commit, MPI_Type_free ...

```
20          printf("[%d] Lecture de la boite au lettre.\n", rank);
21          MPI_Recv(&torecv, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
22          printf("[%d] Message  r e u  : %d.\n", rank, torecv);
23      }
24      MPI_Finalize();
25  }
26
27  [fremals@fermi01 OpenMPI]$ mpirun -np 2 recv
28  [1] Lecture de la boite au lettre.
29  [0] Envoie du message.
30  [0] Message  p o s t .
31  [1] Message  r e u  : 5 (source : 0,   tiquette   : 5).
```

2.2.9 MPI_Irecv

Syntaxe C

```
#include <mpi.h>
   int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Request *request))
```

Description

Lance la rception non bloquante d'un message. L'appel alloue un objet reprsentant une requte de communication et l'associe la variable `request`. Cette variable peut tre ultrieurement utilise pour connaitre l'tat de la communication (`MPI_Test`) ou attendre son achvement (`MPI_Wait`).

L'appel `MPI_Irecv` indique au systme d'exploitation qu'il peut commencer crire les donnes dans le buffer de rception. Le buffer de rception ne doit pas tre accd tant que cette copie n'est pas finie.

Paramtres d'entre

`count` : nombre maximum d'lments de type `datatype` recevoir (entier non-ngatif).
`datatype` : type des lments reus. Les types existant sont : `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG`, `MPI_LONG_DOUBLE`[7].
`dest` : rang du processus source dans le communicateur `comm`.
`tag` : une tiquette pour le message choisie par le dveloppeur et permettant de filtrer les messages. S'il n'y a pas besoin de filtrer les messages, le tag `MPI_ANY_TAG` peut tre utilis. La plus grande valeur possible pour ce paramtre est contenue dans la constante `MPI_TAG_UB`.
`comm` : nom du communicateur.

Paramtres de sortie

`buf` : l'adresse de l'emplacement mmoire o le message doit tre stock.
`status` : structure contenant des informations sur le messages reu :

- status.MPI_TAG : l'tiquette du message reu.
- status.MPI_SOURCE : l'metteur du message reu.

[7] Il est possible de crer des types de donnes drivs, des types de donnes construits partir des types fournis par MPI, l'aide des routines `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_indexed`, `MPI_Type_struct`, `MPI_Type_commit`, `MPI_Type_free` ...

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

`MPI_ERR_COMM` : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

`MPI_ERR_COUNT` : l'argument `count` est incorrect.

`MPI_ERR_TYPE` : l'argument `datatype` est incorrect.

`MPI_ERR_TAG` : l'argument `tag` est incorrect. Les tags doivent tre non-ngatifs.

`MPI_ERR_RANK` : rang de la source incorrect. Le rang doit tre compris entre 0 et la taille du groupe du commincateur dcrment d'une unit. Dans le cas de la rception, le rang peut valoir `MPI_ANY_SOURCE`.

Illustration

Listing 2.9: Illustration de la fonction `MPI_Irecv`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char * argv[])
5  {
6      int size = 0, rank = 0;
7      MPI_Init(&argc,&argv);
8
9      MPI_Comm_size(MPI_COMM_WORLD,&size);
10     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11     if(rank == 0){
12         sleep(1);
13         int tosend = 5;
14         MPI_Send(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
15     }
16     if(rank==1){
17         MPI_Status status;
18         int torecv;
19         MPI_Request recvState;
20         MPI_Irecv(&torecv, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &recvState);
21         int recved = 0;
22         while(!recved){
23             MPI_Test(&recvState, &recved, MPI_STATUS_IGNORE);
24             printf("Etat de la   r ception   : %d\n", recved);
25                 if(!recved)
26                     sleep(0);
27         }
28     }
29     MPI_Finalize();
30 }
31
32 [fremals@fermi01 OpenMPI]$ mpirun -np 2 ./irecv
33 Etat de la   r ception   : 0
34 Etat de la   r ception   : 0
35 ...
36 Etat de la   r ception   : 0
37 Etat de la   r ception   : 0
38 Etat de la   r ception   : 1
```

2.2.10 MPI_Bcast

Syntaxe C

```
#include <mpi.h>
   int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm)
```

Description

Cette routine transmet un message du processus source (de rang `root`) vers tous les processus du communicateur, le processus source y compris. Tous les processus du groupe doivent l'appeler en passant les mme valeurs pour les variables `root` et `comm`. La taille des donnes (`size*sizeof(datatype)`) doit galement tre identique pour les diffrents processus. La valeur de la variable `buffer` du processus source est copie dans le `buffer` des processus associs au communicateur.

Paramtres d'entre

`buffer` : pointeur vers la zone mmoire qui contient/reoit les donnes.
`count` : nombre d'lments transfrer.
`datatype` : type des lments reus. Les types existant sont : `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG`, `MPI_LONG_DOUBLE`[8].
`root` : rang du processus source.
`comm` : communicateur.

Paramtres de sortie

`buffer` : pointeur vers la zone mmoire qui contient/reoit les donnes.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.
`MPI_ERR_COMM` : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.
`MPI_ERR_COUNT` : l'argument `count` est incorrect. L'argument ne peut pas tre ngatif.
`MPI_ERR_TYPE` : l'argument `datatype` est incorrect, le type n'est pas connu de MPI.
`MPI_ERR_BUFFER` : le pointeur `buffer` est incorrect. La zone de mmoire pointe doit tre alloue.
`MPI_ERR_ROOT` : le rang du processus source est incorrect.

Illustration

Listing 2.10: Illustration de la fonction `MPI_Bcast`

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int computation(){
5         return rand()%10;
```

[8] Il est possible de crer des types de donnes drivs, des types de donnes construits partir des types fournis par MPI, l'aide des routines `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_indexed`, `MPI_Type_struct`, `MPI_Type_commit`, `MPI_Type_free` ...

```
}

int main(int argc, char * argv[])
{
    int size = 0, rank = 0;
    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Request sendState;
    if(rank == 0){
        int tobcast = 4;
        MPI_Bcast(&tobcast, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf("[%d] Valeur diffuse : %d\n", rank, tobcast);
    }
    else if(rank != 0){
        int tobcast = 0;
        MPI_Bcast(&tobcast, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf("[%d] Valeur diffuse : %d\n", rank, tobcast);
    }
    MPI_Finalize();
}

[fremals@fermi01 OpenMPI]$ mpirun -np 3 ./bcast
[0] Valeur diffuse : 4
[1] Valeur diffuse : 4
[2] Valeur diffuse : 4
```

2.2.11 MPI_Reduce

Syntaxe C

```
#include <mpi.h>
   int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm)
```

Description

`MPI_REDUCE` effectue une opration de rduction (somme, max, ET logique ...) sur un ensemble de valeurs fournies par les processus du communicateur. La routine rduit, en utilisant l'opration `op`, les lments contenus dans les buffers `sendbuf` et place le rsultat dans le buffer `recvbuf` du processus de rang `root`. La taille des donnes se trouvant dans le buffer d'entre et de sortie est dfinie par les arguments `count` et `datatype`. Tous les processus faisant appel `MPI_REDUCE` pour une mme opration de rduction doivent donner des valeurs communes pour les arguments `count`, `datatype`, `op`, `root` et `comm`.

Cette opration est bloquante.

Paramtres d'entre

`sendbuf` : adresse d'un des lments qui subiront l'opration de rduction.

`count` : nombre d'lments dans le buffer `sendbuf` d'un processus (diffrent du nombre total d'lments qui seront rduits).

`datatype` : type des lments reus. Les types existant sont : `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG`, `MPI_LONG_DOUBLE`[9].

[9] Il est possible de crer des types de donnes drivs, des types de donnes construits partir des types fournis par MPI, l'aide des routines `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_indexed`, `MPI_Type_struct`,

`op` : opration de rduction appliquer. Les oprations existantes sont : `MPI_MAX` (maximum), `MPI_MIN` (minimum), `MPI_SUM` (somme), `MPI_PROD` (produit), `MPI_LAND` (ET logique), `MPI_BAND` (ET bit--bit), `MPI_LOR` (OU logique), `MPI_BOR` (OU bit--bit), `MPI_LXOR` (OU exclusif logique), `MPI_BXOR` (OU exclusif bit--bit), `MPI_MAXLOC` et `MPI_MINLOC` (ces deux dernires oprations permettent de rcuprer le maximum et le rang du processus qui le dtient).

`root` : rang du processus qui recevra le rsultat.

`comm` : communicateur.

Paramtres de sortie

`recvbuf` : adresse de la zone mmoire qui contiendra le rsultat (doit tre diffrente de `sendbuf`).

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

`MPI_ERR_COMM` : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

`MPI_ERR_COUNT` : l'argument `count` est incorrect. L'argument ne peut pas tre ngatif.

`MPI_ERR_TYPE` : l'argument `datatype` est incorrect, le type n'est pas connu de MPI.

`MPI_ERR_BUFFER` : le pointeur `buffer` est incorrect. La zone de mmoire pointe doit tre alloue. Cette erreur peut galement apparaitre s'il y a crnelage (aliasing)[10] entre deux buffers.

`MPI_ERR_ROOT` : le rang du processus source est incorrect.

Illustration

Listing 2.11: Illustration de la fonction `MPI_Reduce`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char * argv[])
5  {
6      int size = 0, rank = 0;
7      MPI_Init(&argc,&argv);
8
9      MPI_Comm_size(MPI_COMM_WORLD,&size);
10     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11     MPI_Request sendState;
12
13     srand(rank);
14     int toreduce = rand()%10, reduced=0;
15     printf("[%d] Valeur : %d\n", rank, toreduce);
16     MPI_Reduce(&toreduce, &reduced, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
17     if(rank==0)
18         printf("[%d] Valeur max : %d\n", rank, reduced);
19     MPI_Finalize();
20 }
21
22 [fremals@fermi01 OpenMPI]$ mpirun -np 5 reduce
23 [0] Valeur : 3
24 [4] Valeur : 1
25 [1] Valeur : 3
26 [3] Valeur : 6
27 [2] Valeur : 0
```

`MPI_Type_commit`, `MPI_Type_free` ...

[10] Le crnelage est le recouvrement partiel ou total de deux zones mmoire. Ex. : une premire zone mmoire pointe vers un tableau de 10 lments tandis qu'une seconde zone mmoire pointe vers le 3 lment du tableau.

```
28 [0] Valeur max : 6
```

2.2.12 MPI_Scatter

Syntaxe C

```
#include <mpi.h>
   int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Description

Le processus `root` distribue les donnes contenues dans `sendbuf` entre tous les processus du groupe dans l'ordre de leur rang. Chaque processus, le `root` y compris, aura ainsi `recvcount` lment de type `recvtype` (dont la taille doit tre identique celle de `sendcount` lments de taille `sendtype`) dans leur buffer `recvbuf`. Les arguments `root` et `comm` doivent avoir la mme valeur pour tous les processus appelant.

Cette opration est bloquante.

Paramtres d'entre

`sendbuf` : adresse des lments distribuer (cet argument est significatif pour le processus `root` uniquement).

`sendcount` : nombre d'lments distribuer (cet argument est significatif pour le processus `root` uniquement).

`sendtype` : type des lments distribuer. Les types existant sont : `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG`, `MPI_LONG_DOUBLE`[11] (cet argument est significatif pour le processus `root` uniquement).

`recvcount` : nombre d'lments reu.

`recvtype` : type des lments distribus.

`root` : rang du processus qui distribue les donnes.

`comm` : communicateur.

Paramtres de sortie

`recvbuf` : adresse de la zone mmoire qui contiendra les lments distribus.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

`MPI_ERR_COMM` : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

`MPI_ERR_COUNT` : l'argument `count` est incorrect. L'argument ne peut pas tre ngatif.

`MPI_ERR_TYPE` : l'argument `datatype` est incorrect, le type n'est pas connu de MPI.

`MPI_ERR_BUFFER` : le pointeur `buffer` est incorrect. La zone de mmoire pointe doit tre alloue.

[11] Il est possible de crer des types de donnes drivs, des types de donnes construits partir des types fournis par MPI, l'aide des routines `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_indexed`, `MPI_Type_struct`, `MPI_Type_commit`, `MPI_Type_free` ...

Illustration

Listing 2.12: Illustration de la fonction `MPI_Scatter`

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

int main(int argc, char * argv[])
{
    int size = 0, rank = 0, i;

    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    int * data;
    int myData;

    if(rank == 0){
        data = (int *) malloc(size*sizeof(int));
        for(i=0; i<size; ++i){
            data[i] = i;
            printf("[%d] Data[%d] : %d\n", rank, i, data[i]);
        }
    }

    MPI_Scatter(data, 1, MPI_INT, &myData, 1, MPI_INT, 0, MPI_COMM_WORLD);

    ++ myData;
    printf("[%d] Data : %d\n", rank, myData);


    MPI_Gather(&myData, 1, MPI_INT, data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if(rank==0){
        for(i=0; i<size; ++i){
            printf("[%d] Data[%d] : %d\n", rank, i, data[i]);
        }
    }

    MPI_Finalize();
}

[fremals@fermi01 OpenMPI]$ mpirun -np 3 scatter
[0] Data[0] : 0
[0] Data[1] : 1
[0] Data[2] : 2
[0] Data : 1
[0] Data[0] : 1
[0] Data[1] : 2
[0] Data[2] : 3
[1] Data : 2
[2] Data : 3
```

2.2.13 MPI_Gather

Syntaxe C

```
#include <mpi.h>
   int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Description

Cette opration est le dual de `MPI_Scatter`. Chaque processus (le processus de rang `root` inclus) envoie le contenu du buffer `sendbuf` au processus de rang `root`. Les donnes sont concatnes dans le buffer de rception `recvbuf` dans l'ordre des rangs des processus. Les arguments `sendcount` et `sendtype` du processus `i` doivent indiquer une taille identique celle indique par les arguments `recvcount` et `recvtype` du processus `root` (`recvcount` indique donc le nombre d'lment reu de la part d'un processus, et non le nombre total d'lments reus). Les arguments `root` et `comm` doivent avoir la mme valeur pour tous les processus appelant.

Cette opration est bloquante.

Paramtres d'entre

`sendbuf` : adresse des lments regrouper.
`sendcount` : nombre d'lments transfrer au `root`.
`sendtype` : type des lments regrouper. Les types existant sont : `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG`, `MPI_LONG_DOUBLE`[12].
`recvcount` : nombre d'lments envoys par chaque processus et reu par le `root` (cet argument est significatif pour le processus `root` uniquement).
`recvtype` : type des lments regroups (cet argument est significatif pour le processus `root` uniquement).
`root` : rang du processus qui reoit les donnes.
`comm` : communicateur.

Paramtres de sortie

`recvbuf` : adresse de la zone mmoire qui contiendra les lments concatns (seul le processus `root` doit fournir une adresse valide).

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.
`MPI_ERR_COMM` : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.
`MPI_ERR_COUNT` : l'argument `count` est incorrect. L'argument ne peut pas tre ngatif.
`MPI_ERR_TYPE` : l'argument `datatype` est incorrect, le type n'est pas connu de MPI.
`MPI_ERR_BUFFER` : le pointeur `buffer` est incorrect. La zone de mmoire pointe doit tre alloue.

2.2.14 MPI_Wait

Syntaxe C

```
#include <mpi.h>
   int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

[12] Il est possible de crer des types de donnes drivs, des types de donnes construits partir des types fournis par MPI, l'aide des routines `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_indexed`, `MPI_Type_struct`, `MPI_Type_commit`, `MPI_Type_free` ...

Description

Un appel `MPI_Wait` rend la main quand l'opration identifie par la variable `request` est acheve. L'objet reprsentant la requte de communication est dsalloue par cette routine et la variable `request` est mise la valeur `MPI_REQUEST_NULL`.

La variable `status` contient des informations sur l'opration acheve. Le champ `MPI_ERROR` de cette variable contient des informations lorsque `MPI_Testall`, `MPI_Testsome`, `MPI_Waitall` ou `MPI_Waitsome` a renvoy l'erreur `MPI_ERR_IN_STATUS`. Dans le cas des routines d'envoi, `status` est utile uniquement pour la fonction `MPI_Test_cancelled` ou si l'erreur `MPI_ERR_STATUS` a t retourne. Le valeur du champ `MPI_ERROR` vaut alors `MPI_SUCCESS` pour chaque opration d'envoie ou de rception russie, ou `MPI_ERR_PENDING` pour les oprations qui ont chou ou qui sont inacheves. S'il n'est pas ncessaire d'examiner la variable `status`, celle-ci peut tre remplace par la constante `MPI_STATUS_IGNORE`.

Cette opration est bloquante.

Paramtres d'entre

`request` : identifie l'opration attendre.

Paramtres de sortie

`status` : structure contenant des informations sur l'opration ; le paramtre peut tre remplac par `MPI_STATUS_IGNORE`.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.
`MPI_ERR_REQUEST` : l'argument `request` est incorrect.
`MPI_ERR_ARG` : un argument est incorrect.

Illustration

Listing 2.13: Illustration de la fonction `MPI_Wait`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int computation(){
5          return rand()%10;
6  }
7
8  int main(int argc, char * argv[])
9  {
10     int size = 0, rank = 0;
11     MPI_Init(&argc,&argv);
12
13     MPI_Comm_size(MPI_COMM_WORLD,&size);
14     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
15     MPI_Request sendState;
16     if(rank == 0){
17         sleep(1);
18         int tosend = computation(), temp;
19         MPI_Isend(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &sendState);
20         while(tosend!=9){
21             temp = computation(); //le buffer d'envoie ne peut  tre  modifi
22             MPI_Wait(&sendState, MPI_STATUS_IGNORE);
```

```
23              tosend = temp;
24              MPI_Isend(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &sendState);
25          }
26          MPI_Wait(&sendState, MPI_STATUS_IGNORE);
27      }
28      if(rank==1){
29          MPI_Status status;
30          int torecv=0;
31          while(torecv!=9){
32              MPI_Recv(&torecv, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
33              printf("Valeur calcule : %d\n", torecv);
34          }
35      }
36      MPI_Finalize();
37  }
38
39  [fremals@fermi01 OpenMPI]$ mpirun -np 2 wait
40  Valeur calcule : 3
41  Valeur calcule : 6
42  Valeur calcule : 7
43  Valeur calcule : 9
```

2.2.15 MPI_Test

Syntaxe C

```
#include <mpi.h>
   int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Description

La routine renseigne sur l'tat d'une opration identifie par `request`. `flag` contient 1 si l'opration s'est termine et 0 sinon.

La variable `status` contient des informations sur l'opration acheve. Le champ `MPI_ERROR` de cette variable contient des informations lorsque `MPI_Testall`, `MPI_Testsome`, `MPI_Waitall` ou `MPI_Waitsome` a renvoy l'erreur `MPI_ERR_IN_STATUS`. Dans le cas des routines d'envoi, `status` est utile uniquement pour la fonction `MPI_Test_cancelled` ou si l'erreur `MPI_ERR_STATUS` a t retourne. Le valeur du champ `MPI_ERROR` vaut alors `MPI_SUCCESS` pour chaque opration d'envoie ou de rception russie, ou `MPI_ERR_PENDING` pour les oprations qui ont chou ou qui sont inacheves. S'il n'est pas ncessaire d'examiner la variable `status`, celle-ci peut tre remplace par la constante `MPI_STATUS_IGNORE`.

Cette opration est non-bloquante.

Paramtres d'entre

`request` : identifie l'opration sonde.

Paramtres de sortie

`flag` : indique si l'opration s'est acheve (=1) ou non (=0).
`status` : structure contenant des informations sur l'opration ; le paramtre peut tre remplac par `MPI_STATUS_IGNORE`.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.
`MPI_ERR_REQUEST` : l'argument **`request`** est incorrect.
`MPI_ERR_ARG` : un argument est incorrect.

Illustration

Listing 2.14: Illustration de la fonction `MPI_Test`

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

int main(int argc, char * argv[])
{
    int size = 0, rank = 0;
    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank == 0){
        sleep(1);
        int tosend = 5;
        MPI_Send(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    if(rank==1){
        MPI_Status status;
        int torecv;
        MPI_Request recvState;
        MPI_Irecv(&torecv, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &recvState);
        int recved = 0;
        while(!recved){
            MPI_Test(&recvState, &recved, MPI_STATUS_IGNORE);
            printf("Etat de la   r ception   : %d\n", recved);
                if(!recved)
                    sleep(0);
        }
    }
    MPI_Finalize();
}

[fremals@fermi01 OpenMPI]$ mpirun -np 2 ./irecv
Etat de la   r ception   : 0
Etat de la   r ception   : 0
...
Etat de la   r ception   : 0
Etat de la   r ception   : 0
Etat de la   r ception   : 1
```

2.2.16 `MPI_Probe`

Syntaxe C

```
#include <mpi.h>
   int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Description

L'appel MPI_Probe est bloquant jusqu' l'envoi par le processus de rang source d'un message portant l'tiquette tag dans le communicateur comm. Le message n'est pas reu, mais cela permet l'utilisateur de dcider comment le recevoir en se basant sur les informations prsentes dans status. Il peut, par exemple, allouer de la mmoire pour le buffer de rception en fonction de la longueur du message (MPI_Get_Count permet de connaitre le nombre d'lments envoys).

Si le champ status ne doit pas tre examin, il peut tre remplac par la constante MPI_STATUS_IGNORE.

Paramtres d'entre

source : rang du processus source ou MPI_ANY_SOURCE.

tag : valeur de l'tiquette du message attendu ou MPI_ANY_TAG. La plus grande valeur possible pour ce paramtre est contenue dans la constante MPI_TAG_UB.

comm : communicateur.

Paramtres de sortie

status : structure contenant des informations sur le message sond.

Valeur de retour

MPI_SUCCESS : la routine s'est excute sans erreur.

MPI_ERR_COMM : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

MPI_ERR_TAG : l'argument tag est incorrect. Les tags doivent tre non-ngatifs ; la plus grande valeur permise pour ce paramtre est disponible dans l'attribut MPI_TAG_UB. Dans le cas de la rception, l'tiquette peut valoir MPI_ANY_TAG.

MPI_ERR_RANK : rand du destinataire incorrect. Le rang doit tre compris entre 0 et la taille du groupe du commincateur dcrment d'une unit. Dans le cas de la rception, le rang peut valoir MPI_ANY_SOURCE.

Illustration

Listing 2.15: Illustration de la fonction MPI_Probe

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  int main(int argc, char * argv[])
6  {
7      srand(5);
8      int size = 0, rank = 0;
9      MPI_Init(&argc,&argv);
10
11     MPI_Comm_size(MPI_COMM_WORLD,&size);
12     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
13     if(rank == 0){
14         MPI_Status status;
15         int compute1=0;
16         double compute2=0;
17         while(compute1!=9 || compute2<8){
18             MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
19             switch(status.MPI_TAG){
```

```
                case 1 :
                    MPI_Recv(&compute1, 1, MPI_INT, status.MPI_SOURCE,
                                status.MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    printf("Premier calcul : %d\n", compute1);
                    break;
                case 2 :
                    MPI_Recv(&compute2, 1, MPI_DOUBLE, status.MPI_SOURCE,
                                status.MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    printf("Deuxime calcul : %f\n", compute2);
                    break;
            }
        }
    }
    else if(rank==1){
        int compute1 = 0;
        while(compute1!=9){
            compute1 = computation1();
            printf("Compute1 : %d\n", compute1);
            MPI_Send(&compute1, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        }
    }
    else if(rank==2){
        double compute2 = 0;
        while(compute2<8){
            compute2 = computation2();
            printf("Compute2 : %f\n", compute2);
            MPI_Send(&compute2, 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
}

[fremals@fermi01 Listings]$ mpirun -np 3 ./probe
Deuxime calcul : 2.747456
Deuxime calcul : 0.464678
Deuxime calcul : 9.927552
Compute2 : 2.747456
Compute2 : 0.464678
Compute2 : 9.927552
Compute1 : 5
Compute1 : 5
...
Compute1 : 0
Compute1 : 9
Premier calcul : 5
Premier calcul : 5
...
Premier calcul : 0
Premier calcul : 9
```

2.2.17 MPI_Iprobe

Syntaxe C

```
#include <mpi.h>
   int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
MPI_Status *status)
```

Description

L'appel `MPI_Iprobe` permet de vrifier sans blocage la prsence d'un message entrant qui porte l'tiquette `tag` et qui est mis par le processus de rang `source` dans le communicateur `comm`. Le paramtre `flag` est mis 1 si un message peut tre reu ; le paramtre `status` contient alors les mme informations qu'aurait fournit un appel `MPI_Recv()`. Le message n'est pas reu, mais cela permet l'utilisateur de dcider comment le recevoir en se basant sur les informations prsentes dans `status` (il peut, par exemple, allouer de la mmoire pour le buffer de rception en fonction de la longueur du message).

Si le champ `status` ne doit pas tre examin, il peut tre remplac par la constante `MPI_STATUS_IGNORE`.

Paramtres d'entre

`source` : rang du processus source ou `MPI_ANY_SOURCE`.

`tag` : valeur de l'tiquette du message attendu ou `MPI_ANY_TAG`. La plus grande valeur possible pour ce paramtre est contenue dans la constante `MPI_TAG_UB`.

`comm` : communicateur.

Paramtres de sortie

`status` : structure contenant des informations sur le message sond.

`flag` : boolen indiquant la prsence ou non d'un message entrant correspondant aux critres indiqus.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

`MPI_ERR_COMM` : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

`MPI_ERR_TAG` : l'argument `tag` est incorrect. Les tags doivent tre non-ngatifs.

`MPI_ERR_RANK` : rang du destinataire incorrect. Le rang doit tre compris entre 0 et la taille du groupe du commincateur dcrment d'une unit. Dans le cas de la rception, le rang peut valoir `MPI_ANY_SOURCE`.

Illustration

Listing 2.16: Illustration de la fonction `MPI_Iprobe`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main( int argc, char * argv[] )
5  {
6      int rank;
7      int sendMsg = 123;
8      int recvMsg = 0;
9      int flag = 0;
10     int count;
11     MPI_Status status;
12     MPI_Request request;
13     int errs = 0;
14
15     MPI_Init(&argc, &argv);
16     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
17
18      if(rank == 0)
19      {
20          MPI_Isend(&sendMsg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
21          while(!flag)
22          {
23              MPI_Iprobe(0, 0, MPI_COMM_WORLD, &flag, &status);
24          }
25          MPI_Get_count(&status, MPI_INT, &count);
26          if(count != 1)
27          {
28              errs++;
29          }
30          MPI_Recv(&recvMsg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
31          if (recvMsg != 123)
32          {
33              errs++;
34          }
35          MPI_Wait(&request, &status);
36      }
37
38      MPI_Finalize();
39      return errs;
40  }
```

2.2.18 MPI_Get_count

Syntaxe C

```
#include <mpi.h>
   int MPI_Get_count( MPI_Status *status, MPI_Datatype datatype, int *count )
```

Description

Cette routine permet de connaitre le nombre d'lments en cours d'envoi pour un transfert identifi par la variable `status`.

Paramtres d'entre

`status` : le statut retourn par une opration de rception ou de sondage.
`datatype` : le type des lments reus.

Paramtres de sortie

`count` : le nombre d'lments qui seront reus. Si la taille de `datatype` vaut 0, `count` est mis 0 ; si la taille des donnes dans `status` n'est pas un multiple de la taille de `datatype`, `count` vaut `MPI_UNDEFINED`.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.

Illustration

Listing 2.17: Illustration de la fonction `MPI_Get_count`

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

int computation1(){
        return rand()%10;
}

double computation2(){
        return (double) rand()/RAND_MAX*10;
}

int main(int argc, char * argv[])
{
    srand(5);
    int size = 0, rank = 0;
    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank == 0){
        MPI_Status status;
        int count;

        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_INT, &count);
        printf("[%d] Taille du message : %d\n", rank, count);

        int * numbers = (int *) malloc(count*sizeof(int));
        MPI_Recv(numbers, count, MPI_INT, status.MPI_SOURCE, status.MPI_TAG,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        free(numbers);
    }
    else if(rank==1){
        int len = rand()%10, i;
        int * numbers = (int *) malloc(len*sizeof(int));
        for(i=0; i<len;++i)
                numbers[i] = i;
        printf("[%d] Taille du message : %d\n", rank, len);
        MPI_Send(numbers, len, MPI_INT, 0, 1, MPI_COMM_WORLD);
        free(numbers);
    }
    MPI_Finalize();
}

[fremals@fermi01 OpenMPI]$ mpirun -np 2 get_count
[1] Taille du message : 5
[0] Taille du message : 5
```

2.2.19 MPI_Barrier

Syntaxe C

```c
#include <mpi.h>
   int MPI_Barrier(MPI_Comm comm)
```

Description

L'appel `MPI_Barrier` bloque l'excution du processus appelant jusqu' ce que tous les processus du communicateur aient atteint cette routine.

Paramtres d'entre

`comm` : communicateur.

Valeur de retour

`MPI_SUCCESS` : la routine s'est excute sans erreur.
`MPI_ERR_COMM` : le communicateur est incorrect. Une erreur frquente est d'utiliser un communicateur NULL.

2.2.20 `MPI_Wtime`

Syntaxe C

```
#include <mpi.h>
   double MPI_Wtime();
```

Description

Cette fonction renvoie le nombre flottant de secondes coules depuis un certain moment dans le pass. Pour mesurer un laps de temps, il est donc ncessaire de faire deux mesures et de soustraire la premire la seconde.

Valeur de retour

Retourne le nombre flottant de secondes coules depuis un certain point dans le pass.

Illustration

Listing 2.18: Illustration de la fonction `MPI_Wtime`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char * argv[])
5  {
6      int size = 0, rank = 0;
7      MPI_Init(&argc,&argv);
8
9      MPI_Comm_size(MPI_COMM_WORLD,&size);
10     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11     if(rank == 0){
12         int tosend = 5;
13         MPI_Ssend(&tosend, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
14         printf("[%d] Message  p o s t .\n", rank);
15     }
16     if(rank==1){
17         double start, end;
18         start = MPI_Wtime();
19         sleep(1);
20         MPI_Status status;
21         int torecv;
22         printf("[%d] Lecture de la boite au lettre.\n", rank);
23         MPI_Recv(&torecv, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
24         end = MPI_Wtime();
25         printf("[%d] Message  r e u  : %d (%f s     coules    ).\n", rank, torecv,
26             end-start);
27     }
28     MPI_Finalize();
```

```
29 }
30
31 [fremals@fermi01 OpenMPI]$ mpirun -np 2 wtime
32 [0] Message  p o s t  .
33 [1] Lecture de la boite au lettre.
34 [1] Message  r e u  : 5 (1.000319 s     coules    ).
```

Chapter 3

Les processeurs graphiques et CUDA

3.1 Introduction

Les ordinateurs actuels prsentent une architecture complexe pouvant associer des processeurs centraux multi-cœurs (CPU - Central Processing Unit multicore), ainsi que des processeurs graphiques (GPU - Graphics Processing Unit) de plus en plus aptes excuter des calculs gnralistes. Ce paragraphe est prsent en trois parties:

1. Architecture des processeurs centraux et graphiques.
2. Mmoires des processeurs graphiques.
3. Langages de programmation sur GPU.
4. Outils d'exploitation des architectures htrognes (multi-CPU/multi-GPU).

3.2 Architecture des processeurs centraux et graphiques

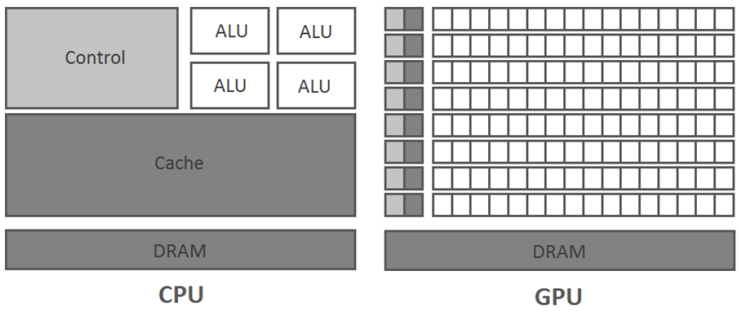

Figure 3.1: Prsentation des architectures des CPUs et des GPUs

Les processeurs centraux et les processeurs graphiques[1] sont reprsents la figure 3.1. Le processeur central est compos d'une Unit Arithmtique et Logique (ALU, Arithmetic and Logical Unit) qui effectue principalement les calculs mathmatiques (addition, soustraction, multiplication et division) et logiques (OR, AND, NOT, XOR, tests binaires ...) ; d'une unit de contrle, ou squenceur, qui synchronise les diffrents lments du processeur (il charge les instructions du

[1]Les processeurs graphiques dont nous parlons dans ce document correspondent aux modles dvelopps par NVIDIA. Les processeurs graphiques AMD/ATI peuvent contenir des lments qui diffrent.

programme en mmoire, les dcode et les fait excuter par les ALUs de manire squentielle) ; et de plusieures niveaux de mmoire cache qui permet l'acclration des traitements (mmoire rapide mais coteuse, proche du CPU). Enfin, la DRAM est la mmoire vive o les donnes sont places lorsqu'elles sont traites. Cette mmoire ne se trouve pas sur le processeur mais au niveau de la carte mre, sous forme de barette mmoire.

Pour les CPU, nous appelons "cœur processeur" la combinaison d'une ALU et d'un contrleur. Plus il y en a, et plus il est possible d'effectuer des calculs simultanment[2].

Les processeurs graphiques sont dcoups en multi-processeurs (SM, Streaming Multiprocessor), chaque multi-processeurs tant composs de 8, 16 ou 32 cœurs (SM, Streaming Processor). Les cœurs des GPUs sont moins performants que ceux des CPUs, mais leur grand nombre (128 dans l'exemple) permet le traitement simultane d'un grand ensemble de donnes et de rendre ainsi le GPU plus performant que le CPU. Chaque SM est ici un processeur compos de 8, 16 ou 32 ALUs, d'une unit de contrle et d'une zone de mmoire cache. Les ALUs des GPUs sont gnralement moins performantes que celles des CPUs (vitesse d'horloge[3]). Cette structure matrielle particulire a t dvloppe pour maximiser les traitements graphiques typiques des cartes vidos, o les mme traitements sont simultanment appliqus sur des donnes diffrentes (mode de fonctionnement SIMD, Single Instruction Multiple Data). Ainsi, toutes les ALUs d'un mme SM effectuent la mme instruction au mme moment.

Les GPU nouvelle gnration (Fermi) apportent des nouvelles amliorations, notamment les SM indpendants. Des programmes diffrents peuvent s'excuter concouramment sur les SMs.

3.3 Structures des mmoires ddies aux processeurs graphiques

La structure mmoire ddie au processeur graphique est appele mmoire device. Elle est physiquement situe sur la carte mmoire proche du processeur graphique. Un ordinateur peut possder plusieures cartes d'extension, ou device, dont les cartes graphiques font parties. Par opposition, la mmoire ddie aux CPUs est appele mmoire hte car c'est la mmoire situe sur la carte mre de l'ordinateur; celle qui hberge les cartes d'extension.

La structure mmoire ddie aux processeurs graphiques est complexe mais importante matriser. En effet, c'est l'utilisateur qui a la charge du placement des donnes dans les diffrents types de la mmoire device. Il y a cinq types de mmoire (cf. fig. 3.2) :

1. Mmoire globale : correspond la mmoire RAM du processeur graphique. C'est l'un des types de mmoire accessible par le CPU. La capacit de cette mmoire varie gnralement de 512 Mo 4 Go en fonction du type de la carte graphique. Cette mmoire de masse se trouvant sur la carte graphique a une latence d'accs aux donnes importante : de 400 600 cycles d'horloge.

2. Mmoire partage : cette mmoire se trouve dans le SM. Une donne place dans la mmoire partage d'un SM n'est accessible que par n'importe lequel de ses cœurs. Cette mmoire est limite 16 ko par SM et possde une latence de 4 cycles d'horloge. Si une donne est accde en lecture par tous les threads tournant sur le SM, un systme de broadcast matriel permet une propagation de la donne sans qu'il y aie conflit d'accs. Il y aura par contre conflit d'accs si certains threads, mais pas tous, accdent une mme donne.

3. Mmoire constante : cette mmoire est localise dans le SM. Ce type de mmoire est accessible par le CPU en criture mais uniquement en lecture par le GPU. Elle est physiquement optimise pour le cas o tous les threads s'excutant accde la mme donne (par exemple,

[2]Par dfaut, le paralllisme est obtenu dans le cadre de ce cours en utilisant les threads.
[3]Environ 1,5 GHz pour les GPUs et environ 3 GHz pour les CPUs

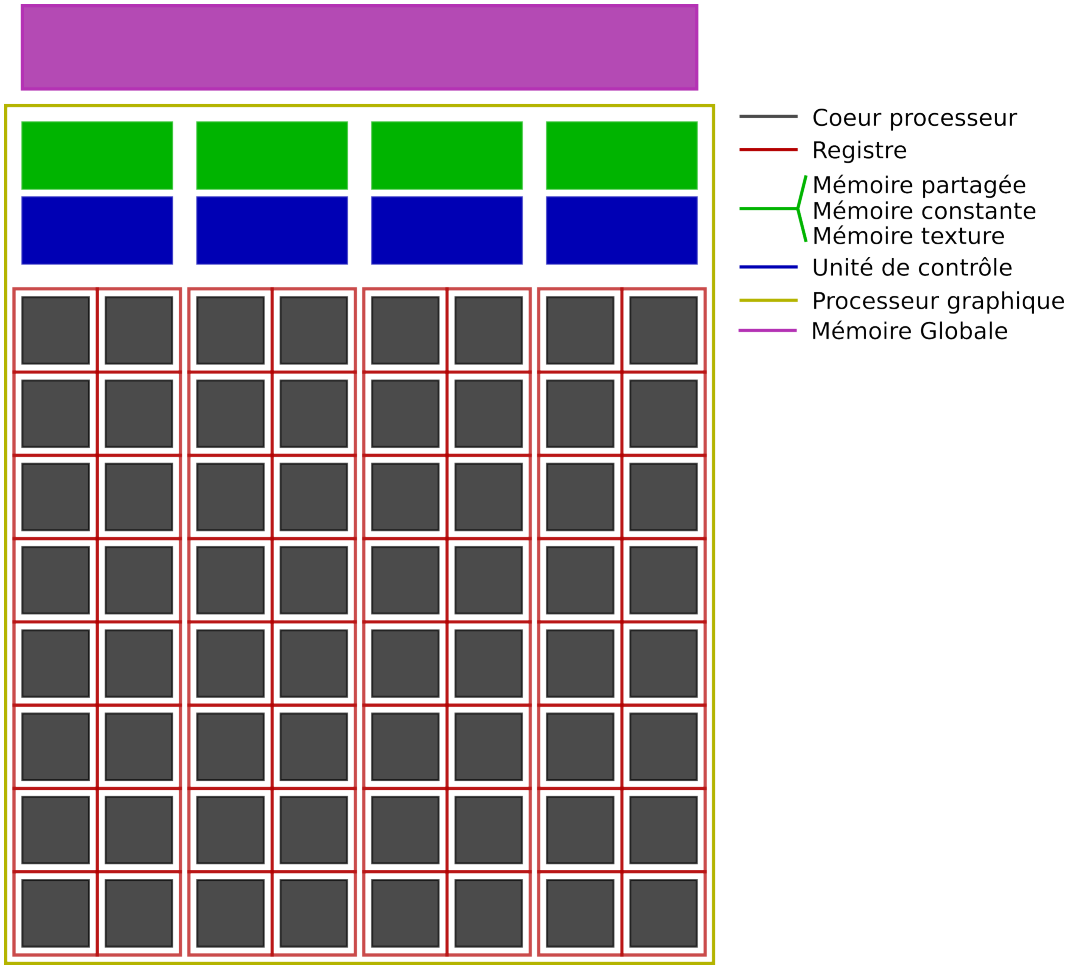

Figure 3.2: Prsentation de la structure des mmoires d'un GPU de 64 cœurs

c'est le cas de la constante π), sa latence est alors de 1 cycle. Lorsque les threads accdent simultanment des adresses diffrentes de la mmoire constante, les accs sont srialiss[4]. La capacit de cette mmoire est limite 64 ko par GPU et 8 ko par SM.

4. Mmoire texture: tout comme la mmoire constante, cette mmoire est en lecture seule pour le GPU et accessible en criture par le CPU. Le cot de la lecture est aussi trs faible (4 cycles d'horloge). Les textures permettent vraiment de simplifier le traitement d'images : elles permettent la mise en œuvre de filtrages bilinaires et trilinaires trs facilement et l'accs alatoire ais aux pixels.

5. Registre : cette mmoire correspond aux registres de travail des ALUs, ce qui en fait la mmoire qui possde la latence la plus basse (1 cycle d'horloge). Il y a 8192 registres de quatre octets par SM.

6. Mmoire locale : cette mmoire, de mme type que la mmoire globale (latence de 400 600 cycles d'horloge) est utilise comme back up, ou swap, de registres lorsque le nombre de registres est insuffisant pour un programme. Elle a le mme rle que le swap disk pour les CPUs. Sa taille est dtermine la compilation. En gnral, son emploi est bannir pour viter les mauvaises performances.

La Table 3.1 dcrit les proprits des diffrentes mmoires du GPU. Les tailles donnes sont des

[4]Effectus les uns aprs les autres.

ordres de grandeur, car elles peuvent varier selon le modle de carte graphique, le nombre de registres disponibles par thread dpend de la programmation de l'algorithme.

Type de mmoire	Utilit	Taille	Latence (cycles)
Registres	Propre chaque thread	8192 × 4 octets	1
Locale	Complte les registres	Indtermine	400 600
Partage	Communication entre threads	16 ko	4
Constante	Lecture seule	8 ko	1 ou plus
Textures	Utile en traitement d'images	1 ko par unit de calcul	4
Globale	Mmoire principale	jusqu' 4 Go	400 600

Table 3.1: Proprits des mmoires du GPU

3.4 Les langages de programmation GPU

Comme montr dans l'introduction, les GPUs ont une architecture massivement parallle. La possibilit de pouvoir les utiliser pour faire autre chose que le rendu d'images a attir l'attention de nombreux chercheurs. Cependant, jusqu'au dbut des annes 2000, les GPU n'taient pas facilement programmables. Ils ne pouvaient qu'excuter une suite fixe de traitements appliqus aux objets gomtriques pour obtenir le rendu 2D ou 3D, cette suite de traitements appele *"pipeline graphique"* tait fige. Seules quelques expriences marginales utilisant certaines fonctions cbles du hardware furent menes et permirent d'acclrer quelques tches simples avec ce qu'on appelle les Shaders.

Les Shaders sont des programmes qui permettent de paramtrer une partie du pipeline graphique. Apparus la fin des annes 90, ils ont commenc tre beaucoup plus utiliss partir de 2001. Depuis, ils n'ont pas cess d'voluer et d'offrir de plus en plus de possibilits aux programmeurs grce des langages permettant la programmation de ces Shaders. OpenGL et Direct3D sont deux Shaders trs populaires, de plus en plus utiliss dans la cration des applications de gnration d'images 3D.

Les dernires annes ont t marques par l'ouverture la programmation des GPUs avec l'avnement de nouveaux langages de programmation. CUDA et OpenCL sont les plus utiliss actuellement ; ils permettent de porter des algorithmes gnriques sur les processeurs graphiques (GPGPU - General-Purpose computing on Graphics Processing Units). Nous prsentons ci-dessous une description de la bibliothque graphique OpenGL (rendu d'images 2D/3D) et des environnements de programmation sur GPU (CUDA et OpenCL). Notons qu'une desciption plus approfondie de CUDA est prsente puisqu'il reprsente le language de programmation GPU le plus utilis actuellement, en raison de son efficiacit et de sa compatibilit avec les cartes les plus utilises : NVIDIA.

3.4.1 OpenGL

OpenGL (Open Graphic Library) [8] est une bibliothque graphique ouverte, reconnue comme un standard et dveloppe depuis 1992 par Silicon Graphics [9], un des tnors de l'informatique graphique. Il a eu ses lettres de noblesse dans l'imagerie scientifique et dans la production

cinmatographique. L'aspect ouvert d'OpenGL lui a permis d'tre port sur des plateformes et systmes d'exploitation varis. Son concurrent direct est la bibliothque Direct 3D dveloppe par Microsoft et fortement reconnue dans le monde du jeu vido.

L'objectif majeur d'OpenGL est de construire et de visualiser des scnes 2D/3D (images de synthse, photos, etc.) sur cran. OpenGL permet l'interactivit avec l'utilisateur en grant les priphriques d'entres (clavier, souris, etc.) pour modifier la scne (dplacement des objets, changement des points de vue de la camra, etc.) et de visualiser l'effet immdiatement. OpenGL est prconis dans les milieux scientifiques et acadmiques du fait de son ouverture, de sa souplesse d'utilisation et de sa disponibilit sur des plateformes varies.

3.4.2 CUDA : programmation des processeurs graphiques NVIDIA

L'API CUDA (Compute Unified Device Architecture) reprsente l'une des technologies les plus utilises pour la programmation parallle sur GPU. CUDA a t dvelopp par nVidia en 2007, permettant de programmer les processeurs graphique dans un langage proche du C standard[5]

Utiliser un processeur graphique pour excuter un programme sous CUDA est un procd recqurant plusieures tapes (cf. figure 3.3). La premire tape est facultative et consiste slectionner le processeur graphique qui sera utilis pour le traitement. Cette tape est utile uniquement si la machine possde plusieurs GPUs et que l'utilisateur veut soit les utiliser en mme temps, soit utiliser un autre GPU que celui assign par dfaut. La seconde tape est de rserver l'espace ncessaire, au niveau de la mmoire globale du device, pour toute donne d'entre et/ou de sortie. Les donnes d'entre sont ensuite transfres de la mmoire hte vers la mmoire device alloue, les rendant ainsi disponibles pour leur traitement sur le processeur graphique. L'utilisateur doit ensuite spcifier le nombre de threads lancer sue le GPU ainsi que leur agencement. Les traitements sont alors excuts par les diffrents cœurs du processeur graphique. Enfin, les rsultats sont rapatries sur la mmoire hte et l'espace mmoire device alloue peut tre libre.

Chacune des ces tapes demandent des connaissances propres aux processeurs graphiques et au langage de programmation CUDA. Celles-ci sont prsentes en dtail dans les sept points suivants:

1. **Slection du processeur graphique:** Un processeur graphique est par dfaut assign pour effectuer les traitements recquis par le processeur central, mais l'utilisateur peut choisir de changer de processeur s'il en possde plusieurs. Pour ce faire, il faut utiliser les fonctions `cudaGetDeviceCount` et `cudaSetDevice` :

```
cudaError_t cudaGetDeviceCount(int * count)

cudaError_t cudaSetDevice(int device)
```

   La premire fonction place le nombre de processeurs graphiques prsents dans la machine au sein de la variable `count`. La seconde fonction permet de slectionner le GPU et la variable `device` reprsente l'identifiant du processeur qui sera assign pour les calculs. Celui-ci est compris entre 0 et `count`-1.

2. **Allocation de la mmoire:** Avant toute chose, il faut allouer la mmoire device ncessaire aux traitements sur le GPU (au minimum pour les donnes d'entre et de sortie). Pour se faire, CUDA implmente sa propre version de la fonction `malloc`[6] :

```
cudaError_t cudaMalloc (void ** devPtr, size_t size)
```

[5] NVIDIA CUDA. https://developer.nvidia.com/cuda-zone

[6] La fonction `malloc` permet de rserver, sur la mmoire hte, un espace mmoire d'une taille spcifie par le dveloppeur

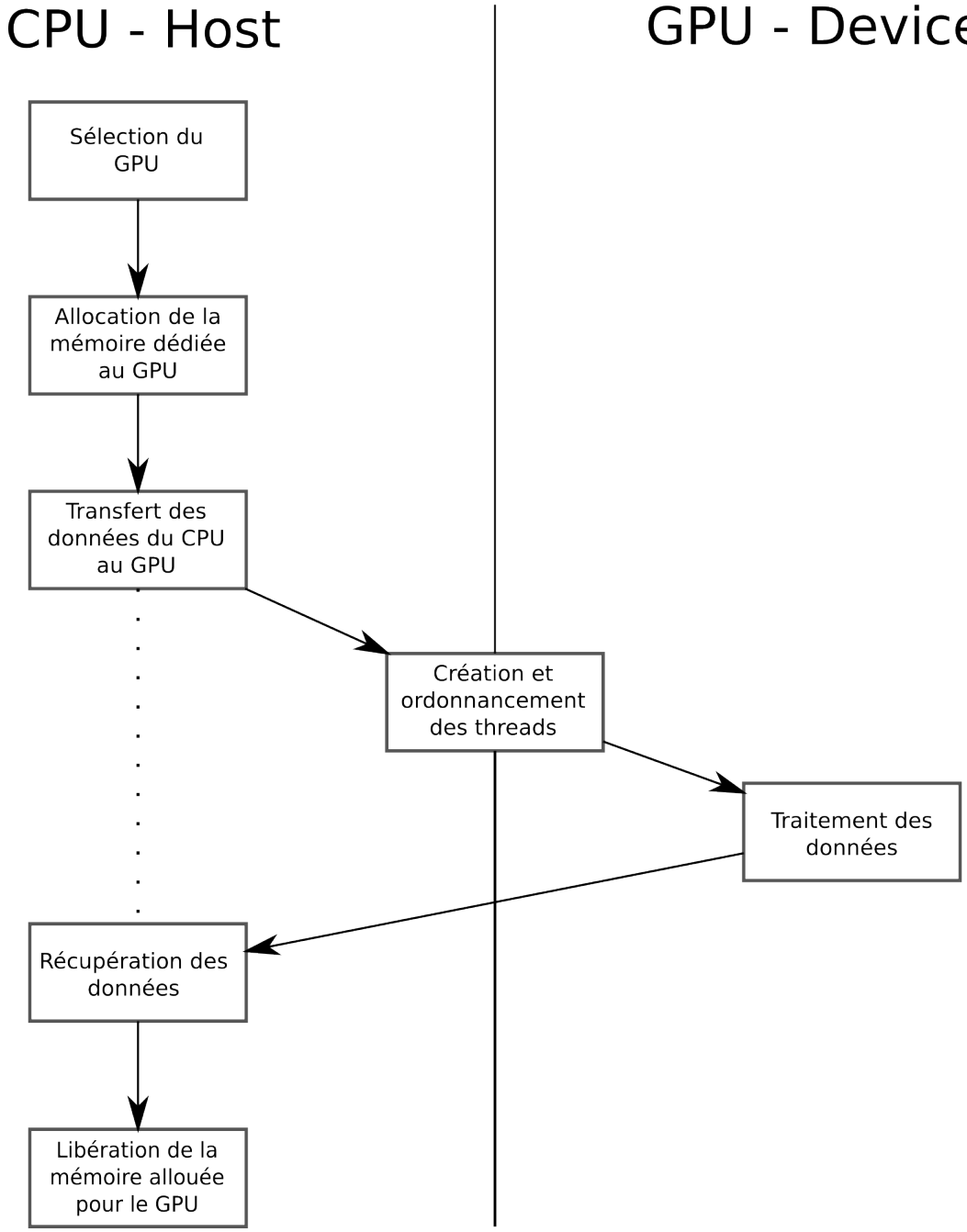

Figure 3.3: Etapes ncessaire un traitement sur processeur graphique

La variable `devPtr` est un pointeur qui recevra l'adresse du dbut de la zone mmoire device alloue et `size` dtermine la taille demande de l'espace mmoire.

3. **Transfert des donnes CPU ⇒ GPU:** Pour effectuer le transfert des donnes entre la mmoire hte et la mmoire device, CUDA implmente sa propre version de la fonction `memcpy`[7] :

```
cudaError_t cudaMemcpy(void * dst, const void * src, size_t count, enum
                       cudaMemcpyKind kind)
```

[7]La fonction memcpy permet de copier, sur le processeur graphique, les donnes d'un espace mmoire dans un autre

La variable `dst` est un pointeur vers l'espace mmoire qui accueillera les donnes, `src` est un pointeur vers l'espace mmoire qui contient les donnes, `count` indique la taille en octet des donnes transfrer (l'espace mmoire de `src` `src+count` seront copies vers l'espace mmoire `dst` `dst+count`), et `kind` indique le type de transfert effectuer. `kind` connait quatre valeurs :

(a) cudaMemcpyHostToHost : la donne est copi depuis un espace mmoire hte vers un espace mmoire hte (cette fonction correspond un appel `memcpy`)

(b) cudaMemcpyHostToDevice : la donne est copie depuis un espace mmoire hte vers un espace mmoire device (c'est le type qui est utilis dans le cadre de ce paragraphe)

(c) cudaMemcpyDeviceToHost : la donne est copie depuis un espace mmoire device vers un espace mmoire hte

(d) cudaMemcpyDeviceToDevice : la donne est copie depuis un espace mmoire device vers un espace mmoire device

4. **L'apppel du kernel : le partitionnement des threads:** Le GPU fonctionne selon le mode SIMD et est ainsi conu pour que tous les cœurs excutent simultanment les mme instructions. La fonction dfinissant la suite de ces instructions est appele "kernel".

   Le processeur graphique fonctionne uniquement avec des threads, il n'y a pas de notion de processus. Lors du lancement du kernel, le dveloppeur indique le nombre de threads qui seront crs (valeur constante, fige lors de la compilation) et tout ces threads excuteront ce mme kernel. Les contrleurs des multiprocesseurs grent les threads par paquets (warp[8]) de 32 :

   - Pour les GPUs possdant des SM de 8 cœurs : un warp est actif la fois, chaque cœur gre quatre threads
   - Pour les GPUs possdant des SM de 32 cœurs : deux warps sont actifs la fois, chaque cœur gre deux threads

   Le contrleur du SM grent les threads par paquet de 32. L'utilisateur peut galement regrouper les threads au sein de blocs avec un maximum de 768 ou 1024 threads par blocs (selon le GPU, les ntres permettent un maximum de 768). La rpartition de la mmoire partage est dfinie par la taille du bloc : les donnes qui sont stockes dans la mmoire partage sont accessibles par tous les threads d'un mme bloc. Lors de l'appel au kernel, l'utilisateur dfinit le nombre de blocs et le nombre de threads par blocs.

   Lors de l'excution, les blocs sont assigns aux SM de manire quivalente l'assignation des processus sur les processeurs centraux par le systme d'exploitation[9]. Les blocs sont dcomposs en warp et ceux-ci sont excuts par les SP. Lorsqu'un warp devient inactif (synchronisation, accs mmoire) le warp est interchang avec un warp en attente d'excution. Ce mcanisme permet de recouvrir les temps d'inactivit des warps par les temps d'activit d'autres warps. Son efficacit dpendra du nombre de threads, et donc de warps, assign un SM et de la dure des temps d'inactivit. Plus les temps d'inactivit sont long, plus le nombre de threads doit tre lev pour obtenir une bonne couverture, et donc de meilleures performances.

   Pour connaitre le nombre de blocs assigns au SM, il suffit de rsoudre ce systme d'quations :

[8] Ce terme provient des mtiers tisser, il dsigne un ensemble de fils (thread = fil) ; un warp CUDA est ainsi un ensemble de 32 threads. Source.

[9] Cf. algorithme de Round-Robin

- $p * b \leq 768$, 1024 *ou* 10536 : maximum 768, 1024 ou 1536 threads (selon le GPU) peuvent tre attribu un SM
- $r * p * b \leq R$: il ne faut pas dpasser la quantit de registres disponibles
- $s * b \leq S$: il ne faut pas dpasser la quantit de mmoire partage disponible
- $b \leq 8$: limite matrielle
- $p \leq 512$: limite matrielle

Avec :

- R : l'ensemble des registres disponibles pour un SM
- r : le nombre de registres utiliss par un thread
- S : la quantit de mmoire partage disponible dans un SM
- s : la quantit de mmoire partage utilise par un bloc
- b : le nombre de blocs assigns un SM
- p : le nombre de threads par bloc

Dans ce systme de contraintes, r et s sont fixs la compilation du kernel et peuvent tre fonction de p. b tant galement dpendant de p, c'est le nombre de threads par bloc qui permet d'ajuster la rpartition des blocs sur les SM.

Listing 3.1: Appel d'un kernel : addition de vecteurs

```
1
2  #define THREAD_PER_BLOCK 32
3
4  __global__ void addition(int * a, int * b, int * c, unsigned int limit_d)
5  {
6      int id = blockIdx.x * THREAD_PER_BLOCK + threadIdx.x;
7      if(id<limit_d)
8          c[id] = a[id] + b[id];
9  }
10
11 int main(){
12     ...
13
14     // allocation de la  m moire
15     cudaMalloc((void**) &a_d, size * sizeof(int));
16     cudaMalloc((void**) &b_d, size * sizeof(int));
17     cudaMalloc((void**) &c_d, size * sizeof(int));
18
19     // transfert des  d o n n es  vers le processeur graphique
20     cudaMemcpy(a_d, a_h, size * sizeof(int), cudaMemcpyHostToDevice);
21     cudaMemcpy(b_d, b_h, size * sizeof(int), cudaMemcpyHostToDevice);
22
23     // appel du kernel
24     unsigned int blocks = (size) / THREAD_PER_BLOCK +
25         ((size) % THREAD_PER_BLOCK > 0);
26     addition<<<blocks,THREAD_PER_BLOCK>>>(a_d,b_d,c_d,size);
27
28     ...
29 }
```

L'allocation de la mmoire, le transfert des donnes et l'appel un kernel sont illustrs au listing 3.1. L'appel du kernel est effectu la ligne 26 et se prsente sous la forme :

```
kernel<<<Nombre_de_blocs,Nombre_de_threads_par_bloc>>>(arguments);
```

Pour mieux coller la reprsentation des donnes, il est possible de dfinir le nombre de blocs et le nombre de threads par bloc sous une reprsentation en 2D ou 3D. Il existe un type `dim3` qui permet cette reprsentation :

```
dim3 var(x,y,z)
```

En ralit, lors de l'appel du kernel, les valeurs `Nombre_de_blocs` et `Nombre_de_threads_par_bloc` sont stockes dans deux variables de type `dim3`. Ainsi, l'exemple prsent au listing 3.1 est un raccourci pour l'appel prsent au listing 3.2.

Listing 3.2: Appel d'un kernel en utilisant des variables de type `dim3`

```
1
2           ...
3
4       // appel du kernel
5       dim3 blocks_3D(blocks,1,1);
6       dim3 threads_3D(THREAD_PER_BLOCK,1,1);
7       addition<<<blocks_3D,threads_3D>>>(a_d,b_d,c_d,size);
8
9       ...
10  }
```

La figure 3.4 prsente la rpartition des threads avec deux variables `dim3`, initialises `(2,2,1)`. Cette fonctionnalit permet de prparer une grille de blocs et threads dont la gomtrie correspond celle des donnes. Les `dim3` sont intressantes lorsque les donnes sont organises en structure possdant plusieurs dimensions (matrice, cube ...). Ceci facilite le calcul de l'index d'accs aux donnes dans chacun des threads.

5. **Le kernel:**

En-tte de la fonction

Il existe trois types d'appel de kernel, chacun tant spcifi par un prfixe en tte de la dfinition du kernel :

- `__global__` : indique que le kernel est appel depuis le CPU et excut sur le GPU
- `__device__` : indique que le kernel est appel depuis le GPU et excut par le GPU
- `__host__` : indique que le kernel est appele depuis le CPU et excute par le CPU (correspond aux procdures traditionnelles d'un CPU)

Il y a des limitations par rapport la programmation sur CPU : un kernel ne peut tre rcursif et il ne peut retourner aucune valeur. Un exemple est prsent au listing 3.1, la ligne 4.

Identification du thread

Les cœurs du GPU effectuent tous le mme code et l'identifiant des threads est le seul lment qui permet de les diffrencier. Pour identifier le thread au sein du kernel, CUDA fournit trois donnes de type `dim3` qui identifient un bloc au sein de la grille et un thread au sein de son bloc :

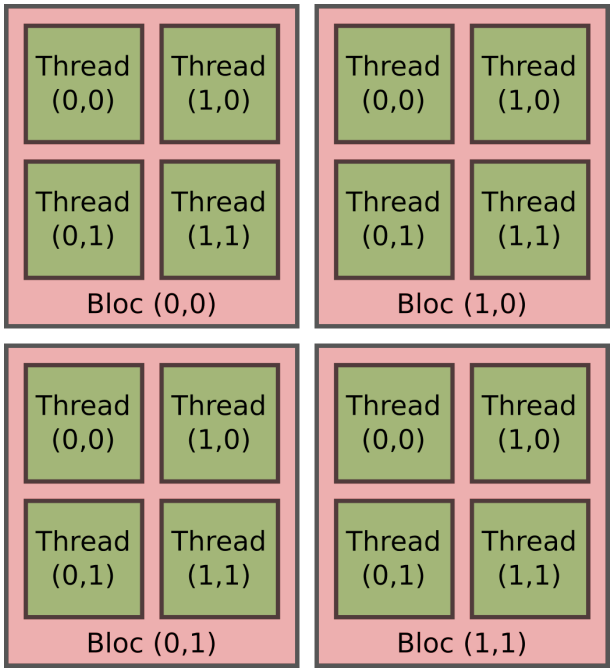

Figure 3.4: Rsultat de l'appel `kernel«<(2,2,1),(2,2,1)»>(arguments);`

- `threadId` : fournit l'index du thread dans le bloc.
- `blockId` : fournit l'index du bloc dans la grille.
- `blockDim` : fournit le nombre de threads par blocs.

Dans le cas d'une matrice (image 2D ...), il suffit d'utiliser ce code pour localiser l'lment associ au thread :

```
int Col = blockId.x * blockDim.x + threadId.x;

int Row = blockId.y * blockDim.y + threadId.y;
```

Dans le cas d'une image 3D, il est intressant d'utiliser les trois dimensions :

```
int Col = blockId.x * blockDim.x + threadId.x;

int Row = blockId.y * blockDim.y + threadId.y;

int Depth = blockId.z * blockDim.z + threadId.z;
```

La gestion de la mmoire

Comme cela a t prsent la section 3.3, le processeur graphique est pourvu de diffrents niveaux de mmoires, plus ou moins proches des cœurs, qui permettent de mieux grer la latence d'accs aux donnes. Certaines de ces mmoires sont alloues mme le kernel :

- La mmoire partage : cette mmoire permet de partager des donnes entre tous les threads d'un mme bloc. Pour dclarer une variable dans la mmoire partage, il suffit de faire prcder sa dclaration au sein du kernel par le mot-cl `__shared__`. La fonction `__syncthreads()` permet de synchroniser les threads d'un bloc et de s'assurer que les blocs de mmoire partage sont mis jour par tous les threads. Il y a trois cas d'accs optimiss la mmoire partage :

- Le plus rapide : tous les threads accdent en lecture la mme donne. Un systme de broadcast est mis en œuvre par le matriel.
- Des threads conscutifs accdent des donnes conscutives : le dcodage de l'adresse de la donne n'est ralis qu'une seule fois, les adresses suivantes sont incrmentes par le matriel.
- Le cas gnral : les threads accdent aux donnes de faon alatoire et chaque accs une donne ncessite un dcodage spcifique de son adresse.

La dure de vie des donnes en mmoire partage est celle du bloc.

- Les registres : cette mmoire est propre un thread et possde sa dure de vie. Toute variable dclare sans prfixe dans un kernel est priori place dans un registre.

Il existe d'autres mmoires possdant une latence plus faible que la mmoire globale, mais devant tre initialises avant l'appel au kernel :

- La mmoire constante : cette mmoire est accessible par tous les threads ainsi que par l'hte. Elle possde la dure de vie de l'application. Une variable est dans la mmoire constante lorsque sa dclaration est prcde du mot cl `__constant__`. Une telle variable ne peut tre dclare dans le corps d'une fonction. Cette mmoire agit comme la mmoire partage pour ce qui en est de son accs : il y a lecture sans conflit de son contenu si tous les threads accdent un lment unique (broadcast), et il y a srialisation des accs si des lments diffrents sont accds simultanment par diffrents threads. Elle est surtout utilise pour les scalaires dont la valeur ne changera pas en cours d'excution. Le transfert d'une donne dans une telle mmoire ncessite l'appel par le CPU la fonction :
  `cudaMemcpyToSymbol(const char * symbol, const void * src, size_t count, size_t offset, enum cudaMemcpyKind kind)`
  Cette fonction copie les `count` octets prsent l'adresse indique par `src` vers `symbol` + `offset`. `symbol` reprsente soit une variable rsidant en mmoire constante ou globale, soit la chane de caractres identifiant la variable dclare en mmoire globale ou constante. `kind` peut prendre comme valeur `cudaMemcpyHostToDevice` ou `cudaMemcpyDeviceToDevice`.
- La mmoire texture : c'est une mmoire optimise pour un espace deux dimensions. Les threads d'un mme warp lisant des adresses proches auront des performances optimales. La lecture des mmoires par le mcanisme des textures peut tre une alternative avantageuse la lecture depuis les mmoires globale et constante. L'initialisation de la texture se fait partir de l'hte par la fonction :
  `cudaError_t cudaBindTextureToArray(const struct texture<T,dim,readMode> & dst, const struct cudaArray * src, const struct cudaChannelFormatDesc & desc)`
  Cette fonction permet de charger la mmoire texture `dst` sur le device par le tableau de deux dimensions `src` se trouvant en mmoire hte. `desc` desc dcrit la manire de lire la mmoire texture (cf. documentation ou M. Sidi). `dst` est une variable du type prdfini texture :
  `struct texture<type, dim, cudaReadModeElementType> dst`
  Tel que `type` reprsente le type donnes (entier, virgule flottante ...), `dim` reprsente le nombre de dimensions de la texture (gnralement gale 2) et `cudaReadModeElementType` est le mode de lecture. La lecture de la texture par le kernel se fait via la fonction :
  `tex2D(dst, x, y)`
  Tel que `dst` reprsente la texture initialise et dclare en haut et `x` et `y` reprsentent les coordonnes des lments l'intrieur de la texture.

Synchronisation des threads

Il est possible de synchroniser les threads d'un bloc l'aide de la fonction `__syncthreads()`. Il n'est actuellement pas possible de synchroniser l'ensemble des blocs de threads l'intrieur d'un kernel, mais la version 4.0 de CUDA, qui sortira prochainement, permettra une telle synchronisation.

Branchements

Les processeurs graphiques sont construits de telles manires que tous les cœurs d'un SM effectuent la mme opration un moment donn. S'il y a un branchement (`if`, un `while` offrant un nombre d'itration diffrent selon le thread ...) au sein d'un warp, les branches divergentes sont excutes les unes aprs les autres. Ainsi, si un `if` offre une divergence au sein d'un warp, il faut deux fois plus de temps pour excuter le warp. Les branchements sont donc viter.

La fusion des accs la mmoire globale

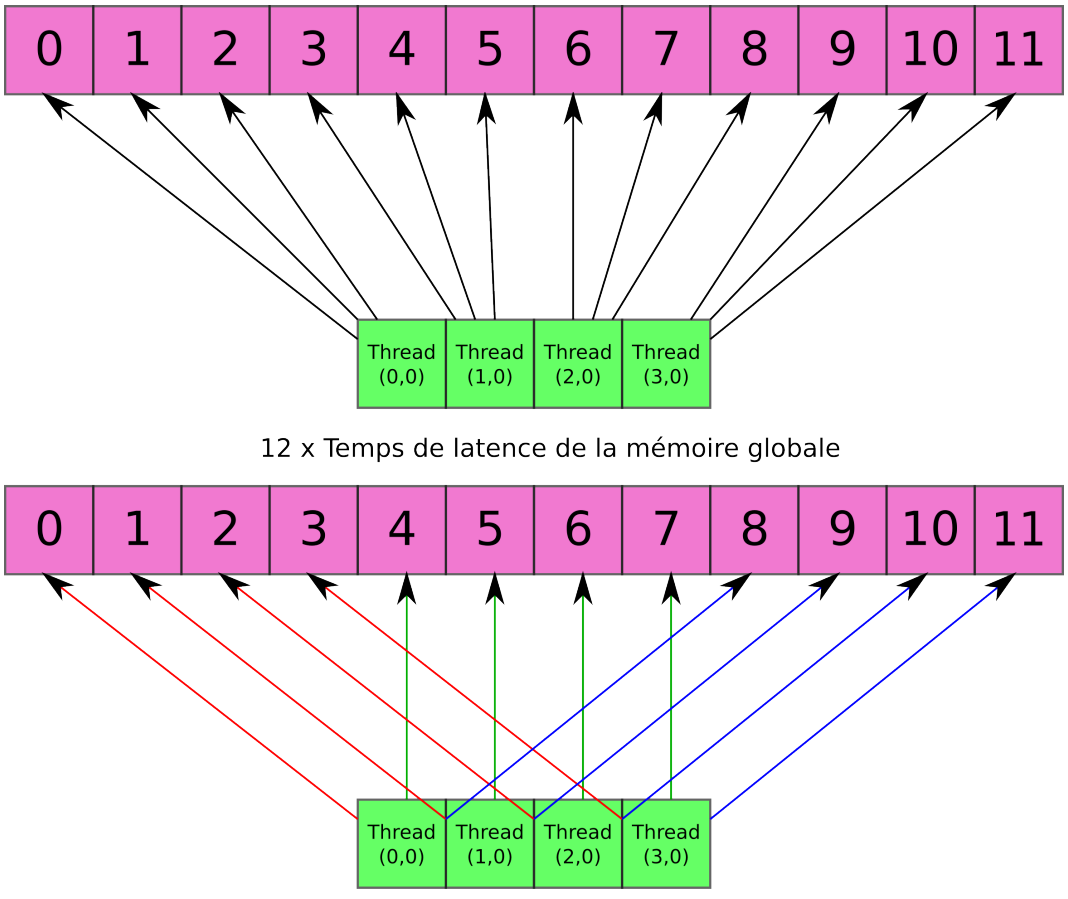

Figure 3.5: Prsentation de l'accs fusionn la mmoire globale : si des threads adjacents accdent des espaces mmoire non-adjacents, les accs sont effectus les uns aprs les autres ; si les threads accdent des espaces mmoires adjacents, les accs sont fusionns et les donnes sont charges en une fois.

L'accs la mmoire globale implique une latence leve de plus de 400 cycles d'horloge. Lorsque N threads adjacents d'un warp chargent N donnes adjacentes (cf. figure 3.5), les N accs sont fusionns en un accs N lments. Ceci permet de diviser la latence par quasiment N.

6. **Transfert des donnes GPU ⇒ CPU:**

   La rcupration des rsultats sur le processeur central est assez simple, il suffit d'utiliser nouveau la fonction `cudaMemcpy` prsente au point 3 en utilisant la constante `cudaMemcpyDeviceToHost`.

7. **Libration des ressources:**

   Enfin, toute ressource alloue doit tre libre. Pour ce faire, CUDA fournit une version GPU de la fonction `free`[10] :

   ```
   cudaError_t cudaFree((void * devPtr))
   ```

   `devPtr` indique l'espace mmoire librer.

   Un exemple de transfert de donnes vers le CPU et de libration d'espace moire est prsent au listing 3.3.

Listing 3.3: Addition de vecteurs : rcupration du rsultat et libration des ressources

```
int main(){
    ...

    //transfert des  d o n n es  vers le processeur central
    cudaMemcpy(c_h, c_d, size * sizeof(int), cudaMemcpyDeviceToHost);

    // l i b ration  des espaces  m moire  a l l o u s
    cudaFree(a_d);
    cudaFree(b_d);
    cudaFree(c_d);

    ...
}
```

8. **Exemple complet**

Listing 3.4: Addition de vecteurs : rcupration du rsultat et libration des ressources

```
#include <sys/time.h>
#include <stdio.h>

#define THREAD_PER_BLOCK 512

__global__ void addition(int * a, int * b, int * c, unsigned int limit_d)
{
    //  r  cupration  de l'identifiant du thread
    int id = blockIdx.x * THREAD_PER_BLOCK + threadIdx.x;
    //si la taille n'est pas un multiple de la taille des warp,
    //on  v rifie  tre  dans les limite des tableaux
    if(id<limit_d){
        //addition avec les  a c c s  aux  d o n n es  f u s i o n n es
        c[id] = a[id] + b[id];
    }
}
```

[10] La fonction `free` permet de librer une ressource alloue sur une processeur central.

```
17
18
19 int main(int agrc, char * argv[]){
20     //allocation des ressources sur le processeur central
21     unsigned int size = atoi(argv[1]), i;
22     int * a_h = (int *) malloc(size * sizeof(int)), * a_d;
23     int * b_h = (int *) malloc(size * sizeof(int)), * b_d;
24     int * c_h = (int *) malloc(size * sizeof(int)), * c_d;
25
26     //allocation des ressources sur le processeur graphique
27     cudaMalloc((void**) &a_d, size * sizeof(int));
28     cudaMalloc((void**) &b_d, size * sizeof(int));
29     cudaMalloc((void**) &c_d, size * sizeof(int));
30     srand(1234);
31     for(i=0;i<size;++i){
32         a_h[i] = rand()%size;
33         b_h[i] = rand()%size;
34     }
35
36     //allocation des ressources pour mesurer le temps
37     unsigned long long time = 0;
38     struct timeval tv1,tv2;
39     gettimeofday(&tv1, NULL);
40
41     //transfert des  d o n n es  sur le GPU
42     cudaMemcpy(a_d, a_h, size * sizeof(int), cudaMemcpyHostToDevice);
43     cudaMemcpy(b_d, b_h, size * sizeof(int), cudaMemcpyHostToDevice);
44
45     //appel au kernel
46     unsigned int blocks = (size) / THREAD_PER_BLOCK +
47         ((size) % THREAD_PER_BLOCK > 0);
48     addition<<<blocks,THREAD_PER_BLOCK>>>(a_d,b_d,c_d,size);
49
50     //transfert des   r sultats  sur le CPU
51     cudaMemcpy(c_h, c_d, size * sizeof(int), cudaMemcpyDeviceToHost);
52
53     //calcul du temps     coul
54     gettimeofday(&tv2, NULL);
55     time+=(tv2.tv_sec-tv1.tv_sec) * 1000000L + (tv2.tv_usec-tv1.tv_usec);
56     printf("Temps de calcul GPU : %llu usec \n", time);
57
58     // l i b ration  des ressource sur le GPU
59     cudaFree(a_d);
60     cudaFree(b_d);
61     cudaFree(c_d);
62
63     // l i b ration  des ressource sur le CPU
64     free(a_h);
65     free(b_h);
66     free(c_h);
67 }
```

Pour ceux qui veulent aller plus loin avec CUDA, ils peuvent consulter quelques exemples d'utilisation de GPU via CUDA pour le traitement d'images et de vidos dans [10],[11] et[12].

3.4.3 OpenCL

OpenCL (Open Computing Language) se veut au GPGPU ce qu'OpenGL est au rendu 3D, savoir un standard ouvert. OpenCL se compose d'une API et d'un langage de programmation driv du C. Son but est de faciliter la programmation d'applications utilisant les divers processeurs

disponibles dans les machines : les CPU, les GPU, et peut tre tendu vers d'autres types de processeurs.

Le projet de cration d'OpenCL a t initi par Apple, qui en a ensuite confi la gestion au Khronos Group [13]. Le projet a depuis t rejoint par les principaux acteurs du domaine : AMD, nVidia et Intel. Leurs futurs produits devraient tre compatibles avec les spcifications du standard. On peut donc s'attendre ce qu'OpenCL devienne un standard trs utilis. La premire version est disponible depuis Avril 2010. OpenCL bnficie essentiellement de son indpendance vis--vis des API 3D, sa compatibilit avec les cartes de tous les constructeurs ainsi que sa portabilit sur diffrentes plateformes.

Pour ceux qui veulent aller plus loin avec OpenCL, ils peuvent consulter quelques exemples d'utilisation de GPU via CUDA pour le traitement d'images mdicales dans dans [14]

3.5 Exploitation des architectures multi-cœurs htrognes

Les processeurs graphiques ont permis de fournir une solution trs efficace pour l'accl-ration des applications intensives en calcul. Cependant, cette solution peut tre encore amliore par l'exploitation simultane des cœurs CPU et GPU multiples. Dans ce contexte, il existe diffrents travaux pour l'exploitation des plateformes multi-cœurs et htrognes, tels que: StarPU, StarSS et GrandCentralDispatch.

3.5.1 StarPU

La librairie StarPU [15], dveloppe l'INRIA Bordeaux (France), permet d'offrir un support excutif unifi pour exploiter les architectures multi-cœurs htrognes, tout en s'affranchissant des difficults lies la gestion des transferts de donnes. StarPU propose par ailleurs plusieurs stratgies d'ordonnancement efficaces et offre la possibilit d'en concevoir aisment de nouvelles. StarPU se base pour son fonctionnement sur deux structures principales : la codelet et les tches.

- **La codelet:** permet de prciser sur quelles architectures le noyau de calcul peut s'effectuer, ainsi que ses implmentations associes (CPU, GPU ...).
- **Les tches:** elles consistent appliquer la codelet sur l'ensemble des donnes, en utilisant la stratgie d'ordonnancement slectionne (StarPU fournit une interface qui permet le partitionnement des donnes).

L'volution d'une tche depuis sa soumission jusqu' la notification de sa terminaison l'application peut tre rsum en six tats:

1. **Soumission :** StarPU reoit toutes les tches soumises partir de l'application.
2. **Slection des ressources de calcul :** l'ordonnanceur distribue les tches selon la politique d'ordonnancement slectionne. Si la tche a t attribue au GPU : la description de la tche est donc envoye au pilote associ ce GPU (chaque unit de calcul est associe un pilote spcifique). Ds que disponible, le pilote rclame une nouvelle tche l'ordonnanceur.
3. **Chargement des donnes :** pour les tches attribues au GPU, les donnes doivent tre transfres vers la carte graphique. StarPU permet de masquer ces transferts puisqu'il dispose d'une bibliothque qui met en œuvre une mmoire partage virtuelle (ou DSM pour Distributed Shared Memory). La programmation des mouvements de donnes est donc transparente pour le programmeur.
4. **Calcul StarPU :** une fois les donnes charges, Les tches peuvent tre excutes sur les ressources htrognes de calcul indiques dans la codelet.

5. **Rapatriement des rsultats :** une fois les calculs achevs, les rsulats peuvent tre rapatris vers la mmoire hte.

6. **Libration des ressources :** une fois que la tche est termine, StarPU en est notifi et lance une autre tche en fonction des dpendances des tches et de sa politique d'ordonnancement.

Un exemple d'utilisation de StarPU pour le traitement d'images est dcrit dans [16],[17] et [18].

3.5.2 StarSs

StarSs [19] a t dvelopp l'universit de Catalunia (Barcelone. Espagne). Il permet de fournir un modle de programmation flexible pour les architectures multi-cœurs. Ainsi, il est compos de six composantes principales: CellSs, SPMSs, GPUSs, ClearSpeedSs, ClusterSs et GridSs.

1. **CellSs :** permet une exploitation automatique du paralllisme fonctionnel partir d'un programme squentiel travers les lments de traitement de l'architecture Cell BE.

2. **SPMSs :** permet d'exploiter de manire automatique et parallle les cœurs multiples des processeurs SMP (Symmetric multiprocessing) partir d'un code squentiel crit par le programmeur.

3. **ClearSpeedSs :** permet aussi une exploitation automatique et parallle des cœurs multiples des architectures ClearSpeed.

4. **GPUSs :** permet l'exploitation des processeurs graphiques (GPU) multiples.

5. **ClusterSs :** permet l'exploitation parallle des serveurs prsents dans les clusters.

6. **GridSs :** permet une exploitation parallle des ressources de calcul disponibles dans une grille.

3.5.3 GrandCentralDispatch

Grand Central Dispatch [20] est une technologie dveloppe par Apple pour mieux exploiter les processeurs multi-cœurs dans les plateformes Mac. Cette nouvelle architecture permet aux dveloppeurs d'utiliser pleinement la puissance de calcul des processeurs multi-cœurs. Elle travaille en distribuant de manire efficace les traitements (processus) aux diffrents cœurs. Le code de Grand Central Dispatch est ouvert depuis septembre 2009.

Une comparaison entre ces technologies d'exploitation de plateformes muli-cœurs (StaPU, StarSs et GrandCentralDispatch) nous permet de distinguer que StarPu est la plus prometteuse grce son exploitation de l'intgralit des ressources htrognes de calcul (multi-CPU/multi-GPU), l'inverse de StarSs et GrandCentralDispatch qui ne permettent qu'une exploitation distincte des cœurs GPUs multiples ou CPU multiples.

Bibliography

[1] OpenMP ARB. The OpenMP API specification for parallel programming. http://openmp.org/wp/.

[2] GCC team. Welcome to the home of GOMP. http://gcc.gnu.org/projects/gomp/.

[3] Dartmouth College. What is OpenMP ? http://www.dartmouth.edu/~rc/classes/intro_openmp/.

[4] Lawrence Livermore National Laboratory Blaise Barney. OpenMP. https://computing.llnl.gov/tutorials/openMP/.

[5] DeinoMPI. The Great and Terrible implementation of MPI-2. http://mpi.deino.net/mpi_functions/.

[6] Argonne National Laboratory. Web pages for MPI and MPE. http://www.mcs.anl.gov/research/projects/mpi/www/.

[7] Open MPI. Open MPI v1.5.4 documentation. http://www.open-mpi.org/doc/v1.5/.

[8] OpenGL. OpenGL Architecture Review Board: ARB vertex program. Revision 45. http://oss.sgi.com/projects/ogl-sample/registry/, 2004.

[9] SGI. Silicon Graphics. OpenGL-the industry's foundation for high performance graphics. http://www.sgi.com/products/software/opengl/, 1992.

[10] P. d. C. Possa, S. A. Mahmoudi, N. Harb, and C. Valderrama. A new self-adapting architecture for feature detection. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 643–646, Aug 2012.

[11] S. A. Mahmoudi and P. Manneback. Multi-gpu based event detection and localization using high definition videos. In *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, pages 81–86, April 2014.

[12] Sidi Ahmed Mahmoudi, Michal Kierzynka, and Pierre Manneback. *Real-Time GPU-Based Motion Detection and Tracking Using Full HD Videos*, pages 12–21. Springer International Publishing, Cham, 2013.

[13] KHRONOS GROUP. Khronos group : Open standards for media authoring and acceleration. http://www.khronos.org/about/, 2000.

[14] Mohamed Amine Larhmam, Sidi Ahmed Mahmoudi, Mohammed Benjelloun, Saïd Mahmoudi, and Pierre Manneback. *A Portable Multi-CPU/Multi-GPU Based Vertebra Localization in Sagittal MR Images*, pages 209–218. Springer International Publishing, Cham, 2014.

[15] Cdric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andr Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *In Concurrency and Computation: Practice and Experience, Euro-Par 2009, best papers issue*, pages 863–874, 2009.

[16] S.A. Mahmoudi, P. Manneback, C. Augonnet, S. Thibault, et al. Traitements d'images sur architectures parallèles et hétérogènes. *Technique et Science Informatiques, Revue des sciences et technologies de l'information*, 2012.

[17] S.A. Mahmoudi, P. Manneback, C. Augonnet, and S. Thibault. Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes. In *20ème Rencontres francophones du parallélisme (RenPar'20)*, 2011.

[18] Sidi Ahmed Mahmoudi and Pierre Manneback. *Multi-CPU/Multi-GPU Based Framework for Multimedia Processing*, pages 54–65. Springer International Publishing, Cham, 2015.

[19] Eduard Ayguad, Rosa M Badia, Francisco D Igual, Jesus Labarta, Rafael Mayo, and Enrique S Quintana-Orti. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. *Proceedings of the 15th International Euro-Par Conference on Parallel Processing. Euro-Par'09*, pages 851–862, 2009.

[20] Apple. Grand Central Dispatch. A better way to do multicore, 2009.