Cours 1 Openmp (multi-threaded parallelisation)

* multithreading: method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them.

I. Documentation / base

`int omp_get_num_threads(void);` returns the number of threads in the current team (region parallèle)
(that is executing the parallel region to which the routine region binds)
If called from the sequential part of a program, this routine returns 1.

`int omp get_thread_num(void):` returns the thread number, within the current team, of the calling thread.
[0 ∨ get_num_threads - 1]

omp sections: distributes the work among threads bound to a defined parallel region

Example code 1:

```
#include <cstudio>
#include "unistd.h"
#include "omp.h"
... ← function definition of calcul2, calcul3() { sleep(3); return 3 }
int main () {
#pragma omp parallel num_threads(3) {
    int thid = omp_get_thread_num();
    int numth = omp_get_num_threads();
    printf ( "thid = %d , numth = %d \n ", thid, numth);

  #pragma omp sections {
    #pragma omp section
      { deux = calcul2(); }
    #pragma omp section
      { trois = calcul3(); }
    #pragma omp section
      { quatre = calcul4(); }
  }
}
neuf= deux + trois + quatre;
return 0;
}
```

II. omp for [manually / using omp for]

Example code [2]:

```
#include <cstdio>
#include < vector>
#include "omp.h"
#include "unistd.h"

int main() {
  int N = 100;
  std::vector<int> A(N);
  // code to parallelize
  // for(int i=0; i<N; i++) { A[i] = i; }
```

```
#pragma omp parallel num_threads(4) {
    int thid = omp_get_thread_num();
    int numth = omp_get_num_threads();

    // méthode 1: avec omp sections, chaque section parcours N/4 itérations
    #pragma omp sections {
        #pragma omp section{
            printf("hello thid = %d\n", thid);
            for(int i=0; i<N/4; i++) { A[i] = i; }
        }
        #pragma omp section {
            printf("hello thid = %d\n", thid);
            for(int i = N/4 ; i<2*N/4 ; i++) { A[i] = i; }
        }
        #pragma omp section{
            printf("hello thid = %d\n", thid);
            for(int i =2*N/4 ; i<3*N/4; i++) { A[i] = i; }
        }
        #pragma omp section {
            printf("hello thid = %d\n", thid);
            for(int i = 3*N/4 ; i<N; i++) { A[i] = i; }
        }

    // méthode 2: avec une seule boucle dont le début et la fin d'itération différent selon le thid
    int niter = N/numth;
    int debut = thid * niter
    int fin = (thid == numth-1) ? N:(thid+1) * niter
    for(int i = debut; i<fin; i++) { A[i] = i; }
    // int debut = thid * N/numth;
    // int fin = (thid+1) * N/numth;
    // int fin = (thid+1) * niter si N divisible par numth

    // méthode 3: pragma omp for
```

documentation:

ompfor : directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct

```
#pragma omp for
    for(int i=0; i<N; i++){
        printf("i = %d; thid = %d\n", i, thid);
        A[i] = i;
    }
}
for(int i=0; i<N; i++) { printf("A[%d] = %d\n", i, A[i]); }
return 0; }
```

output:

```
i = 0, thid = 0
i = 1, thid = 0
...
i = 24, thid = 0
i = 25, thid = 1
...
i = 49, thid = 1
i = 50, thid = 2
...
i = 74, thid = 2
i = 75, thid = 3
...
```

```
A[0] = 0
A[1] = 1
...
A[99] = 99
```

III. Conflits d'écriture

documentation:

pragma omp atomic: directive allows access of a specific memory location automatically. It ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location.

pragma omp for reduction(+:K): data sharing attribute clauses that can be used to perform some forms of (reduction([reduction-modifier,]reduction-identifier:list) recurrence calculations in parallel.

+, −, *, &, |, ^, && and ||

Example code [3]:

```
#include <cstdio>
#include <vector>
#include "omp.h"
#include "unistd.h"

int main() {
    int N = 1000000;
    int K=0;

    // code to parallelise:
    // for (int i=0; i<N; i++)  K=K+1;

    #pragma omp parallel num_threads(4) {
    //méthode 1 : lent car chaque operation est atomic.
        #pragma omp for
            for (int i = 0; i <1 ; i++) {
                #pragma omp atomic
                    K = K+1;
            }


    // méthode 3 :
    #pragma omp for reduction(+:K)
        for (int i=0;i<N ;i++) {
            K=K+1;
    } }
    printf("K=%d, N=%d\n", K,N);
    return 0; }
```

↓ opération, ↓ nom de la variable

+ codes du TP

```
// méthode 2 : rapide car il n'y a qu'une seule opération atomic à la fin
int Klocal =0;
# pragma omp for
    for(int i=0; i<N; i++) {
        Klocal = Klocal+1; }
# pragma omp atomic
    K = K+ Klocal;
```

Cours 2 (dernier cours de prog parallèle)

MPI:

Documentation:

MPI_Init: initialise MPI → int(*argc, char ***argv)
int MPI_Init(&argc, &argv)
↳(si main(int argc, char **argv)

MPI_Comm_rank: donne le rang du processus appelant dans un communicateur
int MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_Comm_size: donne la taille (nombre de processus) du groupe associé à un communicateur
int MPI_Comm_size(MPI_Comm comm, int *size)

MPI_Send: envoi d'un message point à point (blocking send)
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

MPI_Recv: réception bloquante (blocking recieve) d'un message point à point
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status) (buf: pointeur vers la zone mem à envoyer/recevoir)

MPI_Finalize: termine MPI
int MPI_finalize(void)

constantes MPI:

MPI_STATUS_IGNORE: cte permettant d'indiquer qu'on ne souhaite pas récupérer le MPI_status

MPI_COMM_WORLD: communicateur prédéfini contenant tous les processus MPI disponibles

Cours:

Un message MPI est caractérisé par le triplet:
(communicator, rank source/dest, tag)
et par une signature de données (count, datatype)

un MPI_Recv 'match' un MPI_Send si:
- même communicator
- source attendue par le recieve correspond au dest du send (ou wildcard)
- le tag correspond (ou wildcard)
- la réception a un tampon buf assez grand pour contenir le message (sinon troncature)

Example Code:

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

int calcul2() { sleep(2); return 2;}
int calcul3() { sleep(3); return 3;}
int calcul4() { sleep(4); return 4;}

int main (int argc, char **argv) {
    MPI_Init(&argc, &argv);
```

```
// récupérer l'identifiant (rank) et le nombre de processus (size)
int rank, size;
MPI_Comm_rank (MPI_COMM_WORLD , &rank );
MPI_Comm_size (MPI_COMM_WORLD , &size );

int deux, trois , quatre, neuf;
// le processus 0 lance calcul2() ensuite reçoit les valeurs du calcul3() et calcul4() des processus 1 et 2
// enfin, il calcul la valeur neuf, puis l'envoie aux processus 1 et 2

    if (rank ==0 ) {
        deux = calcul2();   buf count datatype source tag
        MPI_Recv ( &trois , 1, MPI_INT, 1 , 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv ( &quatre 1, MPI_INT,2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
        neuf =  deux + trois + quatre;
        MPI_Send ( &neuf, 1 , MPI_INT, 1 , 0, MPI_COMM_WORLD);
        MPI_Send ( &neuf , 1 , MPI_INT, 2 ,0, MPI_COMM_WORLD);
    }
    // processus 1 effectue calcul3(), envoie la valeur au processus 0, reçoit la val neuf finale
    else if (rank == 1 ){
         trois = calcul3();
         MPI_Send ( &trois , 1, MPI_INT , 0, 0, MPI_COMM_WORLD);
         MPI_Recv ( &neuf , 1 , MPI_INT , 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    // processus 2 effectue calcul4(), envoie la valeur au processus 0, reçoit la val neuf finale
    else if (rank ==2){
         quatre = calcul4();
         MPI_Send ( &quatre, 1, MPI_INT , 0, 0, MPI_COMM_WORLD);
         MPI_Recv ( &neuf , 1 , MPI_INT , 0, 0, MPI_COMM_WORLD , MPI_STATUS_IGNORE);
    }

    print ("processus : %d/%d , neuf: %d\n" ,rank, size, neuf);
    MPI_Finalize();
    return 0;
}
```

AUX: (SIMD)

vectoriser des calculs sur tableaux de float

1) SIMD et registres AUX

SIMD: Single instruction, Multiple data
(une seule instruction CPU opère sur plusieurs valeurs en parallèle)

AVX / AVX2: registres vectoriels de 256 bits

→ pour des float (32 bits): 256 / 32 = 8 ⇒ un registre — m256 contient 8 floats
↳ type C/C++ représentant un vecteur de 8 floats

- chargement / stockage :

_mm256_loadu_ps(float const* p) : charge 8 floats depuis p (non aligné)

_mm256_storeu_ps(float* p, __m256 a) : stocke 8 floats vers p (non aligné)

- arithmétique elt par elt :

_mm256_add_ps(a,b)

_mm256_sub_ps(a,b)

_mm256_mul_ps(a,b)

_mm256_div_ps(a,b)

Exemple code :

Objectif : Calculer $\forall i \in \{0, \dots, 7\}$, $z[i] = x[i] + y[i]$
en une seule opération vectorielle sur 8 éléments

```
// compilation : g++ -O2 aux-exemple.cpp -o aux-exemple-maux2
// execution ./aux-exemple

# include <cstdio>
# include <immintrin.h>

int main() {
   float x[8] = {0, 1, 2, 3, 4, 5, 6, 7};
   float y[8] = {2, 2, 2, 2, 2, 2, 2, 2};
   float z[8];

// définir les variables AUX contenant 8 flottants chacune
__m256 vecx, vecy, vecz;

 vecx = _mm256_loadu_ps(&x[0]);  // copie x[0] ... x[7] dans un registre AUX
 vecy = _mm256_loadu_ps(&y[0]);  // copie y[0] ... y[7] dans un registre AUX
 vecz = _mm256_add_ps(vecx, vecy); // addition elt par elt
_mm256_storeu_ps(&z[0], vecz); // stocker les 8 elts de vecz dans le tableau z

for(int i=0; i<8; i++) std::printf("%f ", z[i])
 std::printf("\n");
 return 0;
}
```

cas général : tableau de taille N

idée : traiter 8 par 8, puis finir les derniers éléments si N n'est pas multiple de 8

```
__m256 vecx, vecy, vecz, int i=0;  (si N mod 8 ≠ 0)
for (; i+7 < N ; i = i+8){
   vecx = _mm256_loadu_ps(&x[i]);
   vecy = _mm256_loadu_ps(&y[i]);
   vecz = _mm256_add_ps(vecx, vecy);
   _mm256_storeu_ps(&z[i], vecz);
}
```

```
for(; i<N; i++){
   z[i] = x[i] + y[i];
}
```