Architecture Programmation Parallele
TD

TD2

```
// cree une equipe de threads
#pragma omp parallel

// variable d'environnement permettant de definir le nombre de threads pour une equipe.
OMP_NUM_THREADS

// meme chose mais avec une fonction
omp_set_num_threads(int nb)

// creer une equipe de 8 threads
#pragma omp parallel num_threads(8)


// parallelisation de la boucle for
// avec 4 tours de boucles pour chaque threads.
#pragma omp for schedule(static,4)
for(....)
{

}

// allocation static plus efficace !
// n'importe quel thread peut faire n'import quel paquet de 4
// explication de merde du prof ...
#pragma omp for schedule(dynamic,4)
for(;;)
{

}

// execute uniquement par un thread
#pragma omp single
(
}

// pour faire une barriere explicite
#pragma omp barrier
```

Exercice 1

```

void saxpy(float * array,int size, const float a, const float y)
{
```

```
    // on devrait s'occuper du nombre de threads pour que cela tombe juste avec la taille du
    // ce n'est pas fait ici
    #pragma omp parallel num_threads(8) private(i) shared(a,array,y)
    {
        #pragma omp for
        for(int i = 0; i<size; i++)
        {
            array[i] = (array[i] * a)+y;
        }
    }
}
```

Exercice 2

On vérifie que les boucles sont parallélisable. On voit que oui, donc ...

```

#prama omg parallel private(i,j) shared(colterm, rowterm, a, b,m,q,p)
{
    #pragma omp for
    for(i=0; i<m; i++)
    {
        rowterm[i] = 0.0 ;
        for(j=0; j<p; j++)
        {
            rowterm[i] += a[i][2*j] * a[i][2*j+1];
        }
    }

    #pragma omp for
    for(i=0; i < q ; i++)
    {
        colterm[i] = 0.0;
        for(j = 0; j < p ; j++)
        colterm[i] += b[2*j][i] * b[2*j+1][i];
    }
}
```

Exercice 3

```

float temp[N] = {100.0, 0.0,0.0,...,0.0};
int t, s;
/* calcul de l'evolution des temperatures */
#pragma omp parallel private(s) shared(temp, t)
{
    //on prend le nombre de threads
    p = omp_get_num_threads;
    for(t = 0; t < TMAX ; t++ )
    {
        #pragma omp for schedule((N-2)/p)
        for (s = 1; s < N-1 ; s++)
        temp[s] = (temp[s-1] + temps[s+1])/2.0 ;
    }
}
```

Exercice 4

Question 1

De base non le tri à bulle ne semble pas parallélisable.

Question 2

Possible car on lance les thread à chaque étapes, ils ne se font pas chier les uns les autres comme ça.

```
#pragma omp parallel private(i,temp)
{

    for(step = N; step > 0 ; step--)
    {
        if(step%2 == 0)
        {
            #pragma omp for
            for(i=0; i < N-1 ; i+=2)
            {
                if(Tab[i]>Tab[i+1]
                {
                    temp = Tab[i];
                    Tab[i] = Tab[i+1];
                    Tab[i+1] = temp;
                }
            }
        }
        else
        {
            #pragma omp for
            for(i=1; i < N-1 ; i+=2)
            {
                if(Tab[i]>Tab[i+1)
                    {
                        temp = Tab[i];
                        Tab[i] = Tab[i+1];
                        Tab[i+1] = temp;

                    }
                }
            }
        }

    }
```

Exercice 5

Question 1

```
    #pragma omp parallel private()
    {
        tid = omp_get_thread_num();
        p = om_get_thread_num();
        somme[tid] = 0;
        #pragma omp for schedule(static, n/p)
```

```
        for(i=0; i<n; i++)
        {
            somme[tid] = somme[tid] + v[i];
        }
        #pragma omp single
        sommeTotal = 0;
        for(i=0; i<p;i++)
        {
            sommeTotal += somme[i];
        }
```

Question 2

```
        #pragma omp parallel private(tid ,i ,v ,step , voisin) shared(n ,p ,somme ,globalsomm

        p = omp_get_num_threads();
        tid = omg_get_thread_num();
        somme[tid] = 0;

        #pragma omp for schedule(static, n/p)
        for(i=0; i < n; i++)
        {
            somme[tid] += v[i];
        }
        for(step = 1; step <= p ; step*=2)
        {
            voisin = somme[(tid+step)%p];
                #pragma omp barrier;
                somme[tid] += voisin;
            }
```

TD 3

Exercice 1

Problème section critique :

```
            #pragma omp critical <name>
            {

            }

            \#pragma omp atomic
            {

            }

            accés atomique :
            compare and swap
            do {
                old_count = count;
                newcount = old_count+1;
                success = compareandswap(&count, old_count, new_count);
            }while(!success)
```

compareandswap : si ce qu'il y a à l'addresse &*count* est égale à *old_count*, on écrit *new_count* à cette addresse.

Question 1

```
#pragma omp parallel private(i,j) shared (A,B,moyenne,somme)
int A[N][P], B[N][P];
int somme = 0;
float moyenne;
#pragma omp for
for(int i = 0; i < N; i++)
for(int j = 0; j < N; j++)
#pragma omp atomic
somme += A[i][j];

#pragma omp single
moyenne = somme /(n*p);
#pragma omp for
for(i = 0; i < N; i++)
for(j=0; j<P; j++)
if(A[i][j] >= moyenne)
B[i][j] = 1
else
B[i][j] = 0;
```

Question b) :

```
#pragma omp parallel private(i,j) shared (A,B,moyenne,somme)
int A[N][P], B[N][P];
int somme = 0;
float moyenne;
#pragma omp for reduction(+: somme)
for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
        somme += A[i][j];

#pragma omp single
moyenne = somme /(n*p);
#pragma omp for
for(i = 0; i < N; i++)
    for(j=0; j<P; j++)
        if(A[i][j] >= moyenne)
            B[i][j] = 1
        else
            B[i][j] = 0;
```

1 Exerice 2

```
void calculer(float *A){
    int i,j, fini = 0;
    float diff = 0, temp;
    while(!fini)
    {
        diff = 0;
        for(i=0; i<n; i++)
        {
            for (j=0; j<n; j++)
            {
                temp = A[i][j];
                A[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i-1][j]
                + A[i][j+1] + A[i+1][j]);
                diff = diff + abs(A[i][j] - temp);
            }
        }
        if(diff / (n*n) < SEUIL)
```

```
                    fini = 1;
                }
            }
            solution:

            \begin{lstlisting}[language=C]
            void calculer(float *A){
                #pragma omp parallel private(i, j, temp) shared(A, diff, SEUIL, n, fini)
                int i,j, fini = 0;
                float diff = 0, temp;
                while(!fini)
                {
                    diff = 0;
                    #pragma omp for reduction(+: diff)
                    for(i=0; i<n; i++)
                    {
                        for (j=0; j<n; j++)
                        {
                            temp = A[i][j];
                            A[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i-1][j]
                            + A[i][j+1] + A[i+1][j]);
                            diff = diff + abs(A[i][j] - temp);
                        }
                    }
                    if(diff / (n*n) < SEUIL)
                    fini = 1;
                }
            }
```

Comme dans l'exercice avec les temperatures, on suppose que ça converge, donc pas de soucis si c'est pas vraiment parallelisable.

Exercice 3

```
            int max = 0,i;
            unsigned int A[N];

            #pragma omp parallel for
            for(i = 0; i < N; i++)
            {
                // pour eviter de passer par critical à chaque fois
                // ce qui ralentirait beaucoup la parallélisation.
                if(A[i] > max)
                {
                    #pragma omp critical{
                        if(A[i] > max)
                            max = A[i];
                    }
                }
            }
```

Exercice 4

```
int s, i, j;
for(s=0; s < NUM_STEPS ; s++)
{
    for(i=0; i<N; i++)
    {
        strength[i] = 0;
        for(j=0; j<N; j++)
        {
```

```
            strenght[i] += interaction(i,j);
        }
    }
    for(i=0; i<N; i++)
    {
        update(i);
    }
}
```

correction :

```
int s, i, j;
#pragma omp parallel private(s,i,j)
for(s=0; s < NUM_STEPS ; s++)
{
    #pragma omp for
    for(i=0; i<N; i++)
    {
        strength[i] = 0;
        for(j=0; j<N; j++)
        {
            strenght[i] += interaction(i,j);
        }
    }
    #pragma omp for
    for(i=0; i<N; i++)
    {
        update(i);
    }
}
```