Introduction to OPENMP

1 Basic programs

The first example of a parallel program is

```c
#include <stdio.h>
int main(void)
{
#pragma omp parallel
{
    printf("Hello, world.\n");
}
    printf("bonjour, monde.\n");
return 0;
}
```

1) Compile this code with gcc with the option -fopenmp and run

2) How many Hello World do you have?
   Set the number of threads by using the variable OMP_NUM_THREADS to 2, 8, 20
   Do you have some changes?

3) Compile without the option -fopenmp and run again. What happens?
   The second code illustrates the loop parallelization

```c
#include <stdio.h>
#include <math.h>
#include<stdlib.h>
double essai(double x)
{
    return(5.0+10.0*x+x*x*exp(x)*log(x+0.1)+sqrt(abs(x)));
}
int main()
{
    const int NITER=200000000;
    double *a;
    a=(double *) malloc(sizeof(double)*NITER);
    if (a== NULL) exit(1);
#pragma omp parallel for
    for (int j=0;j<NITER;j++)
    { 
a[j] = essai(j*0.01);
    }
exit(0);
}
```

4) Compile and run the code.

5) In order to measure the efficiency of the parallelization, type
time progexec
   where progexec is the executable.
   By setting the variable OMP_NUM_THREADS to 1, 2, 4, 6 8, 16 collect the different results
   (user time, real time) in a file and plot the two evolution. Comment your graphics.
2 Internal functions

The following code is able to collect internal information

```c
#include <stdio.h>
#include <math.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>

double essai(double x)
{
    return(5.0+10.0*x+x*x*exp(x)*log(x+0.1)+sqrt(abs(x)));
}
int main()
{
    const int NITER=200000000;
    double *a;
    clock_t debut=clock();
    double deb,end;
    a=(double *) malloc(sizeof(double)*NITER);
    #pragma omp parallel
    if(omp_get_thread_num() == 0)
    {
        deb=omp_get_wtime();printf("threads %d \n",omp_get_num_threads());
    }
    if (a== NULL) exit(1);
    #pragma omp parallel for
    for (int j=0;j<NITER;j++)
    {\n        a[j] = essai(j*0.01);
    
    if(omp_get_thread_num() == 0)
    {
        end=omp_get_wtime();
        printf("omp elapsed time %e \n",(end-deb));
    clock_t fin=clock();
        printf("global elapsed time %e \n",(double) (fin-debut)/CLOCKS_PER_SEC);
        exit(0);
    }
    
    #pragma omp parallel
    if(omp_get_thread_num() == 0)
    {
        deb=omp_get_wtime();printf("threads %d \n",omp_get_num_threads());
    }
    if (a== NULL) exit(1);
    #pragma omp parallel for
    for (int j=0;j<NITER;j++)
    {\n        a[j] = essai(j*0.01);
    
    if(omp_get_thread_num() == 0)
    {
        end=omp_get_wtime();
        printf("omp elapsed time %e \n",(end-deb));
    clock_t fin=clock();
        printf("global elapsed time %e \n",(double) (fin-debut)/CLOCKS_PER_SEC);
        exit(0);
    }
```

6) Compile and run the code.

7) Suppress the line #pragma omp parallel. Recompile and rerun. What happens?

3 Reduction

The following code computes the π number by using a numerical evaluation of an integral by a rectangle method. Each thread computes a part of the loop and a reduction instruction is performed

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```
#include<omp.h>
double f( double a ) { return (4.0 / (1.0 + a*a)); }
int main( int argc, char *argv[])
{
    int n = 1000000000;
    double PI25DT = 3.141592653589793238462643;
    double pi, h, sum=0.0;
    double startwtime = 0.0, endwtime;
    h = 1.0 / (double) n;
    #pragma omp parallel
    if (omp_get_thread_num() == 0) {startwtime = omp_get_wtime();}
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i <= n; i ++)
    {
        double x = h * ((double)i - 0.5);
        sum += f(x);
    }
    if (omp_get_thread_num() == 0)
    {
        pi = h * sum;
        printf("pi is approximately %20.15e, Error is %e\n", pi, fabs(pi - PI25DT));
        endwtime = omp_get_wtime();
        printf("omp_get_wtime(): %f\n",endwtime-startwtime);
        exit(0);
    }
}

8) Compile and run the code.
9) Increase the thread number from 1 to 8 Collect data and plot the wall time versus the number of threads
10) Open a second terminal and type top. Rerun the program for a thread number of 1, 2 and 4

4 Synchronisation clauses

Because the different threads share the same memory, it is necessary to check that two (or more) threads do not attempt to write in the same location at the same time. This is important to avoid conflict and consequently to avoid unexpected bugs in your code. In the pragma directive of a loop, you can specify the nature of variables used in your code. A simple rule consists to declare interval variables within the loop.

In order to show this characteristic of parallel coding, you change the previous code as follows:
— Remove the declaration for x within the loop
— add the line double x : after the line double pi,...
11) Compile and run for 4 threads. What happens?

12) Add the clause private (x) at the end of the pragma omp parallel for Compile and run again.

13) Remove the clause private(x) and compile with the option -O3. Compile and run again. What happens?