12 Parallel Programming with OpenMP

OpenMP [?] can be seen as an extension of the C/C++ (and Fortran) language that takes advantage of the processing capabilities of current "multiprocessor" systems (multi-core × multi-chip). Through its use, it is possible to write code that utilizes all available processors, provided the code is "parallelizable."

Launched in 1997, OpenMP, now in version 3.X, allows for writing parallel programs in a simple and standardized way. Many current compilers include support for OpenMP, among them the GNU suite (gcc, g++ and gfortran), Intel compilers (icc, icpc, ifort), and the Portland Group compilers (pgc, pgCC, pgf77). These are available for operating systems like Linux, Windows, and Mac OS X.

This guide serves as an introduction to OpenMP, allowing you to explore some of its capabilities. The examples will be provided in C or C++. The best way to learn is by experimenting with the presented examples and introducing modifications on your own initiative.

12.1 The First Program

Before any explanation, I suggest copying the following code into an editor to create a simple program in OpenMP-C.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    #pragma omp parallel
    {
        printf("Hello OpenMP!\n");
    }
    return 0;
}
```

De seguida poderá compilá-lo com o gcc (de versão 4.2 ou superior), o icc, ou o pgc, da forma seguinte

- **gcc :** gcc -fopenmp hellomp.c -o hellomp
- **icc :** icc -openmp hellomp.c -o hellomp -cxxlib-icc
- **pgc :** pgc -mp hellomp.c -o hellomp

compilando e executando na máquina lsrc obtemos o seguinte.

```
pm@lsrc:~/OpenMP$ gcc -fopenmp hellomp.c -o hellomp
pm@lsrc:~/OpenMP$ ./hellomp
Hello OpenMP!
Hello OpenMP!
Hello OpenMP!
Hello OpenMP!
pm@lsrc:~/OpenMP$
```

Surprisingly, the message appeared 4 times. Why is that?

A normal program executes one line of code at a time, starting with the main function in a C program. In this case, we say there is a single thread of execution, the main thread. As we know, all programs have a main thread, and in most cases, this is the only thread that exists, so the program can only perform one task at a time.

In the code above, however, we define that the block of code containing the printf function belongs to a team of threads defined by the OpenMP "parallel" section. If nothing else is specified, the main thread will be divided into a team of as many threads as there are processors on the machine.

However, we can specify the desired number of threads by defining an environment variable before executing the program, as follows:

```
export OMP_NUM_THREADS=8
```

12.2 Directives (Pragmas)

A parallel section is defined in OpenMP using compiler directives. These directives are instructions to the compiler, indicating how to create the team of threads, how to associate threads with tasks, among other things. These directives are only considered if the program is compiled with OpenMP support. If such support is not present, the directives are ignored.

There are several directives available in OpenMP. Here, we will explore only a subset consisting of the following:

- **parallel:** used to create a parallel block of code that will be executed by a team of threads.
- **section:** used to specify different sections of code to be executed in parallel by different threads.
- **for:** used to specify loops where different iterations are executed by different threads.
- **critical:** used to define a block that can only be executed by one thread at a time.
- **reduction:** used to combine (reduce) the results of multiple local calculations into a single result.

These directives actually cause the compiler to introduce the necessary code to enable parallelism through OpenMP, but we can also use the support functions directly, as shown in the code below.

```c
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_num_threads() 0
    #define omp_get_thread_num() 0
#endif

int main(int argc, char **argv)
{
    int nthreads, thread_id;

    printf("I am the main thread.\n");

    #pragma omp parallel private(nthreads, thread_id)
    {
        nthreads = omp_get_num_threads();
        thread_id = omp_get_thread_num();

        printf("Hello. I am thread %d out of a team of %d\n",
                 thread_id, nthreads);
    }

    printf("Here I am, back to the main thread.\n");

    return 0;
}
```

The novelty here in terms of directives is the definition that the variables nthreads and thread_id are private, meaning that each thread has its own variable with this name instead of it being global and shared.

12.3 Directive *section*

The “section” directive defines a block of code that is associated with a single thread within a team. The following example illustrates its usage.

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

#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

void times_table(int n)
{
    int i, i_times_n, thread_id;

    thread_id = omp_get_thread_num();

    for (i=1; i<=n; ++i)
    {
        i_times_n = i * n;
        printf("Thread %d says %d times %d equals %d.\n",
               thread_id, i, n, i_times_n );

        sleep(1);
    }
}

void countdown()
{
    int i, thread_id;

    thread_id = omp_get_thread_num();

    for (i=10; i>=1; --i)
    {
        printf("Thread %d says %d...\n", thread_id, i);
        sleep(1);
    }

    printf("Thread %d says \"Lift off!\"\n", thread_id);
}

void long_loop()
{
    int i, thread_id;
    double sum = 0;

    thread_id = omp_get_thread_num();

    for (i=1; i<=10; ++i)
    {
        sum += (i*i);
        sleep(1);
    }

    printf("Thread %d says the sum of the long loop is %f\n",
           thread_id, sum);
}
```

```
int main(int argc, char **argv)
{
    printf("This␣is␣the␣main␣thread.\n");

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                times_table(12);
            }
            #pragma omp section
            {
                countdown();
            }
            #pragma omp section
            {
                long_loop();
            }
        }
    }

    printf("Back␣to␣the␣main␣thread.␣Goodbye!\n");

    return 0;
}
```

In this example, we have 3 threads: one for calling the `times_table()` function, another for the `countdown()` function, and another for `long_loop()`.

12.4 Directive *for*

OpenMP also allows parallelizing the execution of loops. For example, if we have a loop executing 1000 iterations, those iterations can be divided among multiple threads. Thus, with two threads, each would execute 500 iterations; with four threads, each would execute 250 iterations, and so on.

However, it is important to note that this is only possible if the iterations are independent, meaning they do not depend on each other's results and can therefore be executed in any order.

```
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

int main(int argc, char **argv)
{
    int i, thread_id, nloops;

    #pragma omp parallel private(thread_id, nloops)
    {
        nloops = 0;
        thread_id = omp_get_thread_num();

        #pragma omp for
        for (i=0; i<1000; ++i)
        {
                        if (nloops==0)
                                printf("Thread␣%d␣started␣with␣i=%d\n",thread_id,i);

                ++nloops;
        }

        thread_id = omp_get_thread_num();

        printf("Thread␣%d␣performed␣%d␣iterations␣of␣the␣loop.\n",
                thread_id, nloops );
    }

    return 0;
}
```

12.5 Directive *critical*

The critical directive allows defining a block of code that must be executed exclusively, meaning it can only be executed by one thread at a time. This is important when accessing shared resources, such as updating global variables. The following code is an example of this, where the variable `global_nloops` is updated by different threads. However, depending on execution conditions, the final result may not be as desired because the updates are not guaranteed to be atomic.

```
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

int main(int argc, char **argv)
{
    int i, thread_id;
    int global_nloops, private_nloops;

    global_nloops = 0;

    #pragma omp parallel private(private_nloops, thread_id)
    {
        private_nloops = 0;

        thread_id = omp_get_thread_num();

        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++private_nloops;
        }

        printf("Thread␣%d␣adding␣its␣iterations␣(%d)␣to␣the␣sum␣(%d)...\n",
               thread_id, private_nloops, global_nloops);

        global_nloops += private_nloops;

        printf("...total␣nloops␣now␣equals␣%d.\n", global_nloops);
    }

    printf("The␣total␣number␣of␣loop␣iterations␣is␣%d\n",
           global_nloops);

    return 0;
}
```

Solving this with the critical directive, we obtain the following code, where access to that same variable is restricted to one thread at a time.

```
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

int main(int argc, char **argv)
{
    int i, thread_id;
    int global_nloops, private_nloops;

    global_nloops = 0;

    #pragma omp parallel private(private_nloops, thread_id)
    {
        private_nloops = 0;

        thread_id = omp_get_thread_num();

        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++private_nloops;
        }

        #pragma omp critical
        {
            printf("Thread %d adding its iterations (%d) to the sum (%d)...\n",
                   thread_id, private_nloops, global_nloops);

            global_nloops += private_nloops;

            printf("...total nloops now equals %d.\n", global_nloops);
        }
    }

    printf("The total number of loop iterations is %d\n",
           global_nloops);
    return 0;
}
```

12.6 Directive *reduction*

Reduction is the process of combining the results of multiple “sub-calculations” into a single result. The “map-reduce” technique is, in fact, a very efficient way to perform complex calculations using parallel programming by dividing them into the parallel computation of several partial results, which are then combined (reduced) into a single result.

OpenMP provides a way to perform this operation (often complex) efficiently through the reduce directiv

```
reduce( operator : lista de variaveis)
```

where the operator can be any binary operator (e.g., +, -, *), and the variable list can be a single variable or a list of variables used for the reduction. For example:

```
reduction( + : sum )
```

In this case, the compiler will combine the partial results from each thread into the variable sum. The following example illustrates its usage.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;
    int private_nloops, nloops;

    nloops = 0;

    #pragma omp parallel private(private_nloops) \
                         reduction(+ : nloops)
    {

        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++nloops;
        }

            }

    printf("The total number of loop iterations is %d\n",
           nloops);

    return 0;
}
```

12.7 Before the class

1. Test all the example programs provided above and analyze their behavior.
2. Visit the site to learn how to address issues related to memory consistency.
3. Prepare solutions for the following two exercises. You can test them on the machines from mpi-01.deec.lan to mpi-05.deec.lan, which are accessible on the DEEC network (they can be accessed externally via VPN[1]).

12.8 In Class

These two exercises should be fully completed during the class.

Exercise 12.1
In any table of derivatives, you will find that

$$\frac{d\,(\tan^{-1}(x))}{dx} = \frac{1}{1+x^2}.$$

Since $\tan(0) = 0$ and $\tan(\pi/4) = 1$, if we calculate the integral

$$\int_0^1 \frac{1}{1+x^2} = \tan^{-1}(1) - \tan^{-1}(0) = \frac{\pi}{4} - 0 \tag{12.1}$$

Thus, we can obtain the value of $\pi/4$ through the integral above, which can in turn be approximated using the trapezoidal rule. This approximation consists of dividing the integration interval into subintervals, where the function is approximated by straight-line segments connecting the function values at the endpoints of these subintervals. Finally, the integral of the function is approximated by summing the areas of the resulting trapezoids. Consequently, the approximation improves as the number of subdivisions increases.

$$\int_a^b f(x) = \sum_{i=0}^{k-1} (x_{i+1} - x_i)\,\frac{(f(x_i) + f(x_{i+1}))}{2},$$

where the interval is divided into k subintervals, with $d = (b-a)/k$ and $x_i = a + i \times d$. Simplifying, we get

$$\int_a^b f(x) = \frac{d}{2} \sum_{i=0}^{k-1} f(x_i) + f(x_{i+1}). \tag{12.2}$$

[1] https://helpdesk.deec.uc.pt/help/pt-pt/19-vpn

1. Write a program to approximate the value of π by evaluating the integral 12.1 using equation 12.2. The program should prompt the user for the number of intervals to use in the approximation and, when presenting the calculated value, also display the error relative to the reference value 3.141592653589793238462643.

2. Run the previous program and record the execution times for various values of k. Execution times can be obtained using the command `time ./program_pi`.

3. Modify the previous program to take advantage of multiple cores available on the machine using OpenMP. The program should indicate how many threads are being used.

4. Record the execution times for the same k values used in part 2 and discuss the results obtained.

Exercise 12.2

In this exercise, we aim to simulate the distribution of temperatures in a lake and its evolution over time. Assuming the entire lake starts at a uniform temperature of 16 degrees Celsius, we will introduce a hot spot at one end of the lake and observe how the temperature propagates across the surface.

To perform the simulation, we will divide the lake's area into square cells (the size of the cells is irrelevant for this task). The temperature of a cell at time step k is calculated as the average of the temperature of the cell itself and its four neighbors (excluding the corners) at time step $k-1$.

1. Write a program that takes 5 parameters: number of rows, number of columns, heat row, heat column, and number of epochs. The program should generate one image file for each epoch, where the pixel values correspond to the temperature values of the corresponding cells. The file names should clearly indicate the epoch they represent. For example, the file lake004.pgm corresponds to the fourth simulated epoch.

   Note: To generate the file, use the following function:

```
 /* This function saves a rectangular array of integer
values  as an image of grey levels. Note that only
values between 0 and 255 are valid. */
void savepgm(int * img,int rx, int ry, char * fname){
        FILE * fp;
        int color,i,j;
        fp=fopen(fname,"w");
        /* header for PPM output */
        fprintf(fp,"P5\n# CREATOR: AC Course, DEEC-UC\n");
        fprintf(fp,"%d %d\n255\n",rx,ry);

        for (i=0;i<ry;i++){
                for (j=0;j<rx;j++){
                        color=img[i*rx+j];
                        if (color>255)
                                color=255;
                        else if (color <0)
                                color=0;
                        fputc((char)(color),fp);
                }
        }
        fclose(fp);
}
```

2. Using OpenMP, modify the previous program so that the temperature calculations are distributed across the available cores.

3. Modify the previous program so that writing to the file is handled by a single thread, while the others calculate the next epoch.

4. Analyze the execution times for each case.