Supercomputing

Hands-On 2: Parallelization with OpenMP

2022

Contents

Introduction

This Hands-on comprises 3 sessions. Next table shows the documents and files needed to develop each one of the exercises.

Session 1	Numeric Integration	`integral.c`
Session 2	Image Processing	`image.c`, `lenna.ppm`
Session 3	Prime numbers	`big_prime.c`, `count_primes.c`

In most cases, as functions such as `sqrt` are used, the mathematical software library is needed. To compile the code you will need to link with the mathematical software library by including `-lm` in the compiling line.

Please remind that in order to compile OpenMP programs, we should include the proper compilation option, such as:

```
$ gcc integral.c -o integral -fopenmp -lm
```

To execute an OpenMP program with several threads, you can use the following example (changing the number of threads accordingly):

```
$ OMP_NUM_THREADS=4 ./integral 1
```

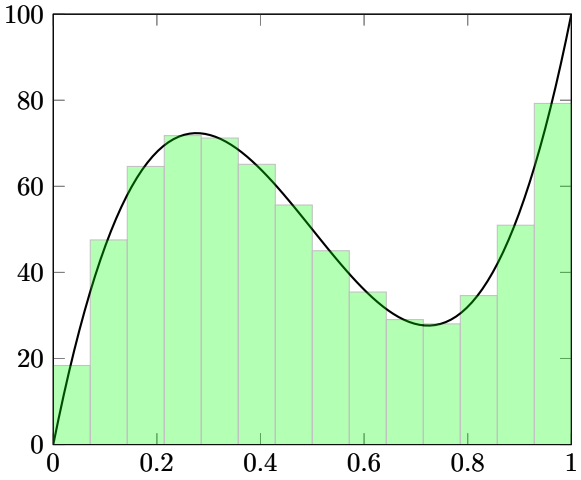

Figure 1: Geometric interpretation of the integral.

1 Numeric integration

The definite integral of a given function $f(x)$ in the interval $[a, b]$ can be defined as:

$$\int_a^b f(x)dx.$$

In this exercise we will compute an approximation of the integral by summing the area of a set of rectangles that occupy a similar area as the one of the integral. Figure 1 shows an example of the approximation. These approximation can be computed using the following expression:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(x_i) \cdot h = h \cdot \sum_{i=0}^{n-1} f(x_i), \tag{1}$$

where n is the number of rectangles used, $(b-a)/n$ is the width of the rectangles and, $x_i = a + h \cdot (i + 0.5)$ is the medium point of each rectangle. The accuracy of the approximation depends on the number of rectangles used.

The sequential code of the program is available in the file `integral.c`. Figure 2 shows an extract of such code. In particular, we can see that there are two different functions for computing the integral. Both approximations are quite similar, and both include a loop that implements the summatory of the Equation (1).

The objective of this Hands-on is to parallelize using OpenMP the computation of the integral.

First, compile and execute the program.

```
$ ./integral 1
```

The result of the integral will be shown on the screen. Optionally, the program accepts the value n as a second argument.

1.1 Parallelization the code

The next step will be to modify the code in the file `integral.c` to perform the computation of the integral in parallel using OpenMP. A first approach could be to use the directive `parallel for` without considering if variables should be `private` or `shared`. By default, all variables are `shared`. After this change, you can compile and execute the program.

```
/* Calculates the integral of the function between point a and b.*/
double calculate_integral_1(double a, double b, int n) {
   double h, s = 0, result;
   int i;
   h = (b - a) / n;
   for (i = 0; i < n; i++) {
      s += f(a + h * (i + 0.5));
   }
   result = h * s;
   return result;
}
```

Figure 2: Sequential code for the computation of the integral.

We can check that the result obtained is incorrect. The problem is that the scope of the variables may be incorrect, resulting in race conditions. To solve this, we should correctly indicate the scope of the variables within the loop, by using clauses such as `private` or `reduction`, if necessary.

Once the code has been corrected, it can be compiled and executed again. We should check that the result is the sames as using a single thread (or without the `-fopenmp` compiler option) or as using different numbers of threads. Execution should be repeated several times [1].

2 Image processing

This practical focuses on the implementation in parallel of an image filtering process based on the median using OpenMP. This API implements a shared-memory parallel model. The objective of this practical is to deepen on the knowledge of OpenMP and the dependencies among processes. The exercise will be based on the sequential version of a program that reads an image in the `PPM` format (a portable-text based format), applies several filtering steps based on the averaged median with variable radius and writes the resulting image in a file using the same format. The result of the exercise will be a parallel code using OpenMP that exploits the parallelism of the different loops in which the median process consist of.

2.1 Problem description

Image filtering consists of substituting the values of the pixels in an image by values depending on the values of the neighbors. The image filtering can be used to reduce noise, focus, blur an image, etc. The neighbors of an image are the pixels located, in both directions, not further than a maximum value called the radius. Median filtering notably reduces white noise but introduces a blurring effect. In averaged median, a mask is used to weight the different values of the pixels close to the pixel being processed, following a parabolic or linear adjustment. This filtering provides better results than the simple median, but it has a higher computational cost. Finally, filtering is an iterative process that may involve several sequential steps.

2.2 Sequential version

The student should download from the resources in drive the file `image.c` with the sequential implementation of the filtering. The filtering used gives a weight of one to the pixels further from the center and increases the values as they get closer to the center. Figure 3 shows an schema of the filter.

The algorithm follows the two dimensions of the image, and for each pixel, two inner loops are applied that go through the pixels that are located not further than the radius for both dimensions. The limits of the image are checked to avoid surpassing the lower and upper limits of the image. The image filtering is

[1] **Important:** The students should ensure that the program is effectively using the number of threads we have indicated. This can be achieved by calling the `omp_get_num_threads` and printing the results on the screen.

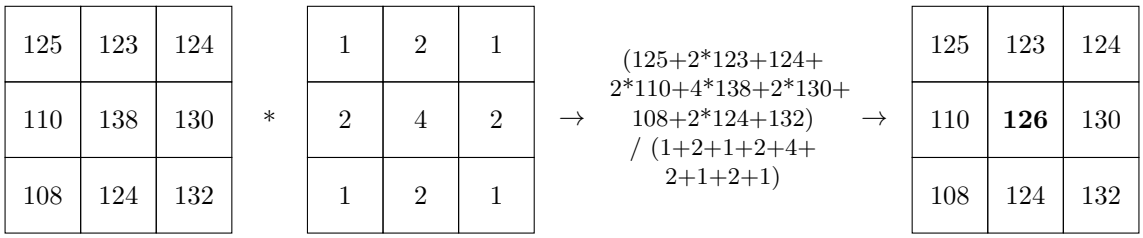

Figure 3: Model for the application of the weighted median in the image filtering. From left to right, original values, filtering mask, weighting operation and in bold face final value.

repeated several times for all the images, therefore requiring five nested loops: steps, rows, columns, radius per row, radius per column, as depicted in Figure 4.

For reading and writing the image, the `ppm` format is used. This format is simple and images can be displayed using different programs, such as `irfanview`[2] or `display` (available in Linux). The image content is represented in text format starting by an identifier (P3), the dimensions of the image, the levels of intensity (normally 255), and 3 values per pixel corresponding to the Red, Green, and Blue channels. Figure 5 shows this format, which can be inspected by using commands such as `head`, `more` or `less`.

Therefore, the program reads the content of a file whose name is indicated by the constant `INPUT_IMAGE`, it will apply the filtering as many times as the constant `NUM_STEPS` indicates, using the value of `RADIUS` for the radius. Finally, it will write the filtered image in the file named `OUTPUT_IMAGE`. Memory allocation is performed by the reading function, ensuring that all the pixels of the image are consecutively stored in memory. The student should verify the correct performance of the program and adjust the values of `NUM_STEPS` and `RADIUS`, in a way that a reasonable cost is required. A well-known test image coming from a popular benchmark is provided [3], see Figure 6, although students are welcome to use their own ones [4].

2.3 Parallel implementation

There are different approximations for the parallel implementation using OpenMP. This Hands-on will focus on analyzing the five loops and deciding (and testing) which loops will lead to a better performance gain. Each parallelization will require identifying the variables that must be shared or private. To do so, the student must:

- To analyze if the different iterations of the loops have any inter-dependency (e.g., if the second iteration uses as input results produced in the first iteration). If dependencies exist, and if those can be overcome or directly coded (such as sums).
- Once the feasible loops for parallelization are identified, shared and private variables must be identified. The shared and private variables list will be detailed in the corresponding OpenMP defines.
- Once the different parallel approaches are defined, they will be implemented using the corresponding OpenMP directives.
- Experimentation will be performed considering the different combinations of the loops. Code should be instrumented with the corresponding timing measuring (`omp_get_wtime`).
- The results will differ depending on the loops to be parallelized. A table should be used to collect the processing time for each one of the loops or reasonable combinations of them. Results will be plotted on a graph.

[2] `http://www.irfanview.com`
[3] `http://en.wikipedia.org/wiki/Standard_test_image`
[4] Higher than 4 megapixels is recommendable.

```
for (s = 0; s < steps; s++) {
  for (i = 0; i < width; i++) {
    for (j = 0; j < height; j++) {
      result.r = 0;
      result.g = 0;
      result.b = 0;
      total = 0;
      for (k = max(0, i - radius); k <= min(width - 1, i + radius); k++) {
        for (l = max(0, j - radius); l <= min(height - 1, j + radius); l++) {
          filter_factor = filter_block[k - i + radius][l - j + radius];
          result.r += src[k][l].r * filter_factor;
          result.g += src[k][l].g * filter_factor;
          result.b += src[k][l].b * filter_factor;
          total += filter_factor;
        }
      }
      result.r /= total;
      result.g /= total;
      result.b /= total;
      dst[i][j].r = result.r;
      dst[i][j].g = result.g;
      dst[i][j].b = result.b;
    }
  }
  if (s + 1 < steps)
    memcpy(src[0], dst[0], width * height * sizeof(struct pixel));
}
```

Figure 4: Main loops in the image processing.

```
P3       <- Constant string that defines the format (ppm, RGB colour)
512 512  <- Image dimensions (row and column)
255      <- Maximum intensity level
224 137 125 225 135 ... <- 512x512x3 values. Each pixel is coded with 3 consecutive values (R,G,B)
```

Figure 5: Content of a ppm image file.

Figure 6: Reference image (`lenna.ppm`) before (left) and after applying a filtering step of radius 5 (right).

3 Prime numbers

This Hands-on aims to solve in parallel the well-known problem of checking whether a number is prime or not. Although other more efficient versions do exist, we will use, in this case, the typical sequential approach. This approach is simple and easier for parallelization. However, in this case, the parallelization will not be as simple as in the previous exercises. Sometimes, using OpenMP is not as straightforward as including a pair of directives. This is desirable since it brings clarity, simplicity, and platform independence when possible. However, we must carefully think about the program topology and explicitly indicate the load balancing among threads.

3.1 Sequential algorithm

The classic sequential algorithm to know if a number is prime is shown in Figure 7. It consist on checking if the number can be exactly divided by any number below it (different from 1). In this case, the number is not prime.

Checking whether a number is prime or not using this algorithm has a low computing cost for small numbers [5]. Parallelizing problems with the low computational cost is not efficient except for basic learning and usually leads to low performance and efficiency. To work with a code with a higher computational cost, we will extend the problem to find the most significant prime number stored in an integer variable of 8 bytes. The process will start with the most considerable number stored in an unsigned integer of 8 bytes, and we will decrease it until a prime number is found. The largest prime will be odd, so we could decrease it by two since actual values are not prime. The algorithm is shown in Figure 8.

The student should read and analyze the program provided in the file `big_prime.c`. This program uses the previous algorithms to search and show on the screen the largest prime number that can be stored in an unsigned integer variable.

[5] This loop ends when an exact divisor is found. Therefore, the higher computational cost will be when the number to be checked is large and either prime or not prime but composed of large factors.

```
Function prime (n)
  If n is par and it is not number 2 then
    p <- false
  else
    p <- true
  End
  If p then
    s <- square root of n
    i <- 3
    While p  and  i <= s
      If remainder of n divided by i is 0 then
        p <- false
      End
      i <- i + 2
    End
  End
  return p
End
```

Figure 7: Sequential algorithm to determine if n is prime.

```
Function largest_prime
  n <- largest integer
  While n is not a prime
    n <- n - 2
  End
  return n
End
```

Figure 8: Algorithm to be parallelized: It searchers the largest prime that can be stored in an unsigned integer with 8 bytes.

```
Function count_primes(last)
  n <- 2 (skip 1 and 2)
  i <- 3
  While i <= last
    If  i is prime then
      n <- n + 1
    End
    i <- i + 2
  End
  return n
End
```

Figure 9: Algorithm that counts the number of prime numbers between 1 and a given value.

3.2 Parallel algorithm

The student should implement a parallel version of the function that checks whether a number is prime or not, using OpenMP. Since part of the function is a `for` loop, it seems straightforward to use a `parallel for` directive. Let is try. What has happened?

Actually, OpenMP does not allow the direct parallelization of a `for` loop, unless starting, ending and increment are perfectly defined. Since the loop in function `prime` can be terminated before (when we found an exact divisor of the number), it is not feasible to parallelize the loop using the `parallel for` directive. OpenMP cannot parallelize the loop if we include the checking of the prime condition inside the loop. First idea would be to remove the condition, but it will bring other problems. (N.B. If you cannot realise where is the problem, you may remove this condition and run the program in parallel, but it is advisable to start on an smaller factor).

We must use the primitives and functions of OpenMP to get the number of threads and to execute them in parallel, assigning each thread to a part of the iterations. Therefore, we should find an alternative way to implement parallelism. One approach is to transform this loop in a `while` loop and implement an explicit splitting of the iterations of the loop among the threads. You should implement these new versions and measure the execution time required. Implementing a cyclic distribution of blocks of consecutive iterations will be interesting.

Keeping the exit condition in the loop is important when an exact divisor is found. In this way, if implemented correctly, when each thread discovers that a number is not prime, all other threads should stop immediately. The code should be implemented ensuring that:

1. If compiled without using OpenMP, it should work correctly.
2. If executed using only one thread, it should work correctly and produce the same result regardless the number of threads used.

3.3 Counting primes

The final exercise for this problem will be to compute the total number of prime numbers between 1 and a large number, such as 100000000 [6]. The algorithm for implementing this process is described in Figure 9.

Check the sequential version of this algorithm, measuring the execution time. Given that a parallel version of the algorithm for checking if a number is prime is already available solving this problem in parallel by simply using this parallel version is trivial. The student should implement this version and measure the execution time.

One can easily see that the performance is not so good. When computing small prime numbers, few data are to be split, and the first iterations in parallel are not efficiently distributed among the different threads.

[6] If the program takes longer, a smaller limit can be chosen. In order to obtain good performance improvement, the sequential version should take around 1 or 2 minutes at least.

It will be desirable to skip the first iterations, execute them sequentially, and execute in parallel only those iterations reasonable assuming that they have a high computing cost. This can be achieved by using the `if` clause in the parallel region, which can use multiple threads only above a specific value. The student should check this option, measuring the computing time for different numbers of n.

However, this approach also has low performance. The final strategy will be to parallelize the loop on the main program and directly use the sequential version of the `count_primes`. Then, you should measure the time using this new algorithm. Finally, different distributions of the loop should be checked, using at least the following scheduling strategies:

- Static without *chunk*,
- Static with *chunk* 1,
- Dynamic.

Take into account that the scheduling can be defined by using the environment variable `OMP_SCHEDULE` (e.g., `export OMP_SCHEDULE="static,6"`). In order to be able to change the scheduling from the environment, we must indicate the the `runtime` scheduling type.

References

[1] M. Boratto. *Hands-On Supercomputing with Parallel Computing*. Available: `https://github.com/muriloboratto/Hands-On-Supercomputing-with-Parallel-Computing`. 2022.

[2] B. Chapman, G. Jost and R. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007, USA.