UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Parallel Computing (CPA)

Lab 1: Parallelization with OpenMP

Year 2023/24

Contents

Introduction

This laboratory work comprises 3 sessions, corresponding to each of the 3 sections of this document. The following table shows the files needed to develop each of the exercises.

Session 1	Numerical Integration	`integral.c`
Session 2	Image Processing	`imagenes.c`, `peppers.ppm`, `peppers-1k.ppm`
Session 3	Prime numbers	`primo_grande.c`, `primo_numeros.c`

The proposed exercises are intended to be done on the computers of the laboratory, with the Linux operating system, or accessing the same environment via the DSIC-LINUX remote desktop from `https://polilabsvpn.upv.es/`. It will also be necessary to connect from this environment to the `kahan` computing cluster via `ssh`, as described in section 1.3.

We start by creating a directory in the `W` unit for the source code of the lab session. We recommend that the path of that directory be short and without white spaces, because later we will need to type this path often. For instance, it could be `W/cpa/prac1` (from now on we assume that this is the path of the directory). We will save in this directory the `.c` and `.ppm` files for the lab session (directly, without subdirectories).

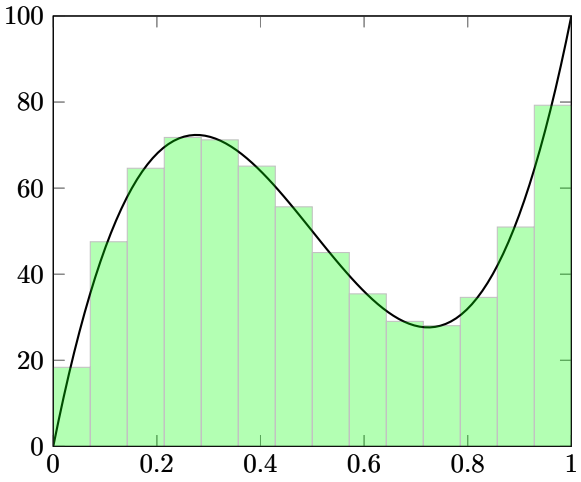

Figure 1: Geometric interpretation of the integral.

1 Numerical integration

In this first exercise, we will learn how to compile OpenMP parallel programs and how to parallelize simple loops. You will also learn how to run executable programs, both in your local machine and in the `kahan` computing cluster.

We consider here the integral of a given function $f(x)$ in the interval $[a, b]$:

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

In this exercise we will compute an approximation of the integral by summing the area of a set of rectangles that occupy an area similar to the one of the integral. Figure 1 shows an example of the approximation. This 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, $h = (b-a)/n$ is the width of the rectangles and, $x_i = a+h\cdot(i+0.5)$ is the midpoint of each rectangle's base. The accuracy of the approximation depends on the number of rectangles used.

The sequential code of the program is available in 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 summation of equation (1).

The objective of this laboratory exercise is to parallelize using OpenMP the two variants for the computation of the integral.

First, you have to compile the source code. For this purpose, open a terminal on the folder where file `integral.c` is stored and run the command:

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

If successful, the compiler will have created an executable in the same folder with the name `integral`. The meaning of the compiler arguments are:

- `-o executable_file`: the name of the executable file (output).

```
/* Computation of the integral for a function f. Variant 1 */
double calcula_integral1(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;
}
/* Computation of the integral for a function f. Variant 2 */
double calcula_integral2(double a, double b, int n)
{
   double x, h, s=0, result;
   int i;
   h=(b-a)/n;
   for (i=0; i<n; i++) {
      x=a;
      x+=h*(i+0.5);
      s+=f(x);
   }
   result = h*s;
   return result;
}
```

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

- `-lm`: link the executable with the library of mathematical functions. This option is needed when using mathematical functions such as `sin`, `cos`, `pow`, `exp`...
- `-Wall` (optional): show all the compile warnings.

Then, run the program. At runtime, an argument can be used to select the variant to be used (1 or 2). For example, to use the first variant you can use:

```
$ ./integral 1
```

The result of the integral will be shown on the screen. The result should be the same regardless of the variant chosen. Optionally, the program accepts the value n (number of rectangles) as a second argument (by default it uses 100000 rectangles). For example:

```
$ ./integral 1 500000
```

1.1 Parallelization of the first variant

The first step will be to modify the code in file `integral.c` to perform the computation of the integral in parallel using OpenMP. Instead of editing the original file, you should create a copy with a different name (e.g., `pintegral.c`).

You can start by making the program show the number of threads used for its execution. To do so, edit the program as indicated in Figure 3. Basically, we store in a variable the number of threads (obtained by using the suitable OpenMP function) and then show that number. Take into account that you must include the `omp.h` header file (to use OpenMP functions).

To compile an OpenMP program, you must add the option `-fopenmp`, such as:

```
...
#include <omp.h>
...
int main(int argc, char *argv[]) {
   int nthreads;
   ...
   nthreads = ...;  /* Obtain the number of threads */
   printf("Number of threads: %d\n", nthreads);
   ...
}
```

Figure 3: (Incomplete) update to show the number of threads.

```
$ gcc -fopenmp -Wall -o pintegral pintegral.c -lm
```

To run the program using several threads (e.g. 4), you can use the `OMP_NUM_THREADS` environment variable, such as:

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

Bear in mind that **there must be no blank space** in "`OMP_NUM_THREADS=4`".

The program will show the number of threads on the screen. Does it display 1 thread instead of 4? If it does, take into account that the OpenMP function returns the number of **active** threads, and if it is called outside a parallel region, the number of active threads is just 1. You have to solve this issue so that the program works as expected. Additionally, take into account that the number of threads should only appear on the screen once.

The next step will be to modify the code in file `integral.c` to actually perform the computation of the integral in parallel using OpenMP. We will start with the first variant (`calcula_integral1`). A first approach could be to use the `parallel for` directive without considering if variables should be `private`, `shared` or some other type. After this change, you can compile and execute the program.

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 same as that of the original sequential code, and that it does not vary when using different numbers of threads. Execution should be repeated several times.

1.2 Parallelization of the second variant

We will proceed next with the parallelization of the second variant (`calcula_integral2`). As we can see in Figure 2, the code is practically identical to the first version, except for the use of an auxiliary variable `x`. Of course, that change should not affect the result of the computation.

You should parallelize this second version. Check again that the result is the same as that of the original sequential code, and that it does not vary when changing the number of threads.

1.3 Execution in the cluster

In this section we will use the `kahan` computing cluster to run the program, which will enable us to use a larger number of cores.

`kahan` is a cluster comprising 4 compute nodes, each of them equipped with 64 cores, and a front-end node where the users will log in to compile and submit the jobs. All the cores within a node share the memory

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=5:00
#SBATCH --partition=cpa

OMP_NUM_THREADS=3 ./pintegral 1
```

Figure 4: Script file for submitting the job to the queue system.

of that node, but cannot access the memory of other nodes. There is more information about `kahan` in the course materials.

To work with `kahan`, you should log into the front-end node by means of `ssh`:

```
$ ssh -Y -l login@alumno.upv.es kahan.dsic.upv.es
```

where `login` is your UPV user name. After running the command, you will be at the `home` directory of `kahan`.

If you run the `ls` command you will see that there is a directory `W`, that corresponds to your `W` unit at UPV. Remember that the files for the lab session should be located in a subdirectory of `W`, for instance `W/cpa/prac1`, as indicated in the Introduction section. Check that the following command lists the source code files for this lab session, among which there should be `pintegral.c` created previously:

```
$ ls W/cpa/prac1
```

Next we must compile the program, as we did previously, but this time in `kahan`. It is important to take into account that the generated executable must not be in the `W` directory, since that directory is not accessible from the compute nodes of `kahan`. For this, create a folder (with `mkdir`) where the executables will be placed, change to it (`cd`) and compile:

```
$ mkdir prac1
$ cd prac1
$ gcc -Wall -fopenmp -o pintegral ~/W/cpa/prac1/pintegral.c -lm
```

where we indicate the full path of file `pintegral.c` when compiling (the `~` character indicates the `home` directory and usually can be typed with `AltGr+4`).

Now you can run the program, for instance:

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

You have just run the program in the front-end, **not in the compute nodes** of `kahan`. Although it is possible to run a program on the front-end, you should do it for very short runs only. The front-end node must be used only to connect via `ssh`, compile and submit jobs to the cluster as explained next.

The execution of jobs in the cluster must be done by means of the **SLURM queue system**. In order to do that, we have to create a **job file**, which is basically a script with the options of the queue system followed by the commands that we want to execute. Figure 4 shows an example of job file in which an OpenMP program is run with 3 execution threads (see the last line in the script file). Lines starting with `#SBATCH` specify different options for the queue system. In this case the job will use the queue (partition) called *cpa*, using a node of the cluster (with its 64 cores) and with a maximum of 5 minutes for its execution.

Write the text of Figure 4 in a file (e.g., `jobopenmp.sh`) and save it in the directory `W/cpa/prac1`. Change the number of threads to be used when running the program, using the value that you wish. Would it make sense to change the number of nodes to a value greater than one?

Next we must launch the job to the queue system by using the command `sbatch`. Assuming that the job file is named `jobopenmp.sh`, the job can be submitted by executing the following command on the front-end (from the directory where the executable is located, in this case `~/prac1`):

```
$ sbatch ~/W/cpa/prac1/jobopenmp.sh
```

The job identifier will be displayed on the terminal.

Once submitted, the queue system will assign the necessary resources (one full node in our case) to the job, when those resources become available, keeping the job in a waiting status until then. In this way, the system ensures that the nodes allocated to a job will not be used by any other job.

What happens with the messages that the program should display? Those messages are not shown on the terminal, but they are stored in a file instead. For example, if the job id is 620, a new file will appear after its execution: `slurm-620.out`. We can see its contents using the `cat` command, like this:

```
$ cat slurm-620.out
Number of threads: 16
Value of the integral = 1.000000000041
```

We can also copy the generated file to the `W/cpa/prac1` directory, and once there visualize it with any text editor:

```
$ cp slurm-620.out ~/W/cpa/prac1
```

We can check the status of the queues using the command `squeue`, for instance:

```
$ squeue
      JOBID PARTITION       NAME        USER ST       TIME  NODES NODELIST(REASON)
       620     cpa       jobopenmp      john  R       0:01    1     kahan01
```

For each job, the status (ST) of the job is displayed, which can have among other values: queued or pending (PD), running (R) or successfully completed (CD).

A job can be cancelled using the `scancel` command:

```
$ scancel 620
```

1.4 Measurement of execution time

When developing parallel algorithms, it is important to measure the execution time of the program, or a part of it, to compare it to the sequential time and evaluate the speed-up obtained.

To measure the execution time of a program fragment we will use the `omp_get_wtime()` function from OpenMP, which returns the time elapsed (in seconds) since a fixed point in time. Figure 5 shows how to use this function.

Measure the execution time of the parallel program running on the cluster for 500000000 (500 million) rectangles, first using 1 thread and then 16 threads, checking how the time is reduced.

Can we increase the number of threads indefinitely, reducing the execution time forever? If not, which could be the maximum number of threads that could improve the performance?

2 Image processing

This practice exercise focuses on the implementation in parallel of an image filtering process using OpenMP. The objective of this practice exercise is to deepen your knowledge of OpenMP and the solution of dependencies among threads. The exercises should be implemented on the computers of the lab. The exercise will be based on the sequential version of a program that reads an image in `PPM` format (a portable text-based format), applies several filtering steps based on a weighted average 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 method is structured.

```
...
#include <omp.h>
...

int main(int argc, char *argv[]) {
   double t1, t2;
   ...
   t1 = omp_get_wtime();
   ...  /* Fragment of code to measure */
   t2 = omp_get_wtime();
   printf("Elapsed time: %f\n", t2-t1);
}
```

Figure 5: Measuring the execution time of a code fragment.

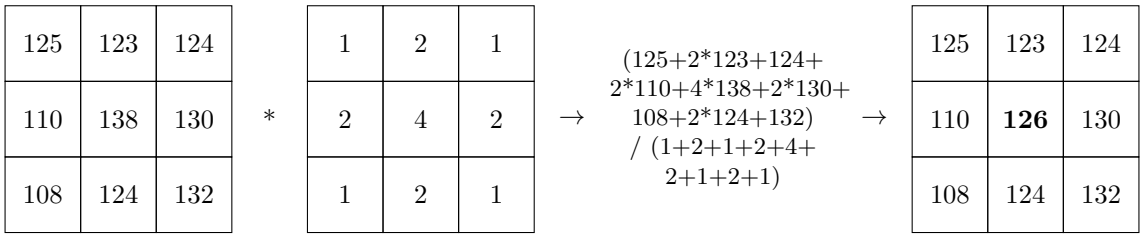

Figure 6: Model for the application of the weighted average in the image filtering. From left to right, original values, filtering mask, weighting operation and final value (in bold face).

2.1 Problem description

Image filtering consists in substituting the values of the pixels in an image by values depending on the values of its neighbors. Image filtering can be used for noise reduction, for sharpening or blurring an image, etc.

Average filtering consists in substituting the value of each pixel by the average of the values of neighboring pixels. The neighbors of a pixel are the pixels located, in both cartesian directions, not further than a maximum distance called radius. Average filtering notably reduces random noise, but introduces a significant blurring effect. In a weighted average filter, a mask is used that weighs the values of the neighboring pixels, following a parabolic or linear interpolation. This filtering provides better results than a simple average, but it has a higher computational cost. Finally, the filtering is an iterative process that may involve several sequential steps.

2.2 Sequential version

The material for the student includes the file `imagenes.c` with the sequential implementation of the filtering. The filtering used gives a weight of one to the pixels further away from the center and increases the values as they get closer to the center. Figure 6 shows a schema of the filter.

This figure shows an original image (left) where filtering is applied to the central pixel (with a color value of 138). The filtering is based on the square mask shown on its right. Applying the filter implies performing the computations shown in the figure, obtaining the resulting image shown on the right side. This filtering is performed at every pixel of the image.

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 repeated several times over the whole image, therefore requiring five nested loops: steps, rows, columns, radius per rows, radius per columns, as depicted in Figure 7.

```
for (p=0;p<pasos;p++) {
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++) {
      resultado.r = 0;
      resultado.g = 0;
      resultado.b = 0;
      tot=0;
      for (k=max(0,i-radio);k<=min(n-1,i+radio);k++) {
         for (l=max(0,j-radio);l<=min(m-1,j+radio);l++) {
           v = ppdBloque[k-i+radio][l-j+radio];
           resultado.r += ppsImagenOrg[k][l].r*v;
           resultado.g += ppsImagenOrg[k][l].g*v;
           resultado.b += ppsImagenOrg[k][l].b*v;
           tot+=v;
         }
      }
      resultado.r /= tot;
      resultado.g /= tot;
      resultado.b /= tot;
      ppsImagenDst[i][j].r = resultado.r;
      ppsImagenDst[i][j].g = resultado.g;
      ppsImagenDst[i][j].b = resultado.b;
    }
  }
  memcpy(ppsImagenOrg[0],ppsImagenDst[0],n*m*sizeof(struct pixel));
}
```

Figure 7: Main loops in the image processing.

```
P3       <- Identifier for the format (ppm, color RGB)
512 512  <- Image size (number of columns and number of rows)
255      <- Intensity depth
224 137 125 225 135 ... <- 512x512x3 values. Each pixel is coded with 3 consecutive values (R,G,B)
```

Figure 8: PPM image file format.

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`[1] or `display` (available in Linux). The image format is shown in Figure 8 and can be displayed using the commands `head`, `more` or `less`.

Therefore, the program reads the contents of a file whose name is indicated by the constant `IMAGEN_ENTRADA`, and it will apply the filtering as many times as the constant `NUM_PASOS` specifies, using the value of `VAL_RADIO` for the radius. Finally, it will write the filtered image in the file indicated in `IMAGEN_SALIDA`. Memory allocation is performed by the reading function, ensuring that all the pixels of the image are consecutively stored in memory.

You are asked to verify the correct operation of the program. Compile and run the program on the local machine. You will see that the program creates the output file `peppers-fil.ppm` (the name specified in `IMAGEN_SALIDA`), which is the result of applying the filter to the file `peppers.ppm`, a well-known test image from a popular benchmark repository[2] (see Figure 9).

Copy the file `peppers-fil.ppm` to a file named `ref.ppm`. This file is the output of the original program, which will be used as the reference to be compared to the output of the modified programs that will be requested later on, in order to check their correct behavior.

[1] `http://www.irfanview.com`

[2] `https://links.uwaterloo.ca/Repository.html`

Figure 9: Reference image (`peppers.ppm`) before (left) and after (right) applying a one-step filtering with a radius of 5.

Next, modify the program to show the number of threads that it is using, as you did in the previous session, and also the execution time of the function that performs the filtering of the image.

Compile and execute the program on the cluster. For this, as explained in section 1.3, you must connect via `ssh` to the front-end node of `kahan`, change to directory `~/prac1` and compile. In this case you must also copy the image files from the source code folder (`~/W/cpa/prac1`) to the executables folder (`~/prac1`), for instance by means of the command:

```
$ cp ~/W/cpa/prac1/*.ppm ~/prac1/
```

Finally, create a job file and launch the file with `sbatch` (again, see section 1.3).

2.3 Parallel implementation

There are different approximations for the parallel implementation using OpenMP, depending on which loop is chosen for its parallelization. The work in this laboratory exercise will focus on analyzing the five loops and deciding (and testing) which loops can be parallelized. Each parallelization will require identifying the variables that must be shared or private. To do so, the student must do the following steps for each loop:

1. Analyze if the different iterations of a loop have any inter-dependency (e.g. if the second iteration uses as input results produced in the first iteration), and in that case, if those dependencies can be overcome or solved using an OpenMP clause (e.g. in the case of summations). If the loop cannot be parallelized, skip the next steps.

2. Write the directives required to parallelize the loop, paying attention to the scope of the variables (which variables should be private to each thread and which ones should be shared among the threads).

3. Run the modified program (preferably on the cluster).

4. Check that the file generated by the modified program (`peppers-fil.ppm`) is exactly the same as the one produced by the original program (`ref.ppm`). To do so, run the command:

```
Sequential time: _____________

Parallel versions execution time:
              Version 1  Version 2  ...
  2 threads   _________  _________  ...
  8 threads   _________  _________  ...
 32 threads   _________  _________  ...
```

Figure 10: Time measurement sheet.

```
$ cmp peppers-fil.ppm ref.ppm
```

If both files are the same, the previous command will not show any message.

Take these pieces of advice into account:

- Start from the innermost loop. As there are fewer variables involved, it may be simpler to start from this loop, and progressively consider the outer loops successively until you reach the outermost loop.
- The `reduction` clause cannot be applied to a variable of type `struct`. In case that you need so, substitute the `struct` by one variable per each of the members.
- A parallel version may produce correct results but be inefficient and slow, even taking longer when the number of threads is increased. Therefore, we suggest using only 2 threads when running the parallel versions for now.
- Take into account that the cluster is shared among all the students, so you should not fill up the queue with your jobs. You may run several commands in the same script file.

Would parallelizing two loops at the same time make sense?

Obtain a set of results similar to the ones shown in Figure 10, in which we obtain the execution time of the sequential version and the execution time of each parallel version for different numbers of threads. Take into account that if a parallel version takes longer with 2 threads than the sequential version, it will get even worse with more threads, so you should not execute it with more than 2 threads.

It can be seen that there are parallel versions that are more efficient than others. Which are the most efficient ones? Why?

3 Prime numbers

The objective of this lab exercise is 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.

In this case, the parallelization will not be "trivial". Sometimes, using OpenMP is not as straightforward as including a pair of directives. When possible, this is desirable since it brings clarity, simplicity and platform independence. However, sometimes we must carefully think about the problem and explicitly indicate the distribution of the work (loop iterations) among the threads. This is what we should do in this exercise.

3.1 Sequential Algorithm

The classical sequential algorithm to find out if a number is prime is shown in Figure 11. It involves checking if the number can be exactly divided by any number below it (different from 1). In this case, the number is not prime.

```
Function prime (n)
  If n is even 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 11: 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 12: Algorithm to be parallelized: It searches the largest prime that can be stored in an unsigned integer of 8 bytes.

Checking whether a number is prime or not using this algorithm has a small computing cost, provided that the number is not too large. Note that the loop ends as soon as an exact divisor is found (in this case it is a composite number so no further checking is required). Therefore, the largest computational cost will be reached when the number to be checked is large and prime, or composite (not prime) with large factors.

With the objective of working with a code that has a higher computational cost, we will extend the problem to finding the largest prime number that can be stored in an integer variable of 8 bytes. Parallelizing problems with low computational cost is not useful except for basic learning and typically leads to low performance and efficiency.

The process will start with the largest number that can be stored in an unsigned integer of 8 bytes and will decrease it until a prime number is found, using the previous algorithm to check if each number is prime. The largest prime will be odd, so we can decrease it by two in each iteration in order to skip even numbers, which are obviously not prime. The algorithm is shown in Figure 12.

The student should read and analyze the program provided in file `primo_grande.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 (8 bytes).

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's try. What happens?

```
#pragma omp parallel ...
{
   for (i = ...; p && i <= ...; i += ...)
      if (n % i == 0) p = 0;
}
```

Figure 13: Parallelization scheme for function `primo`.

Actually, OpenMP does not allow the use of the `parallel for` directive in this case. Bear in mind that the `parallel for` directive stands for a `parallel` directive followed by a `for` directive. This second directive automatically splits the loop iterations among the threads, but it needs that the start, end and increment of the loop are perfectly defined. In the `primo` function, the start value and the increment of the loop variable (`i`) are well defined, but the final value is not known a priori. It may reach value `s` or it can end before if `p` is set to false. Therefore, we can neither use the `for` nor the `parallel for` directives.

Actually, the impossibility to use the `for` directive is because the loop termination condition contains an element that checks if the number is prime. What would happen if we remove this verification? The function will still be correct, but which problem will arise? (N.B. If you cannot realize where the problem is, you may remove this condition and run the program in parallel, but it is advisable to start on a smaller factor, as the execution time will increase very much).

Once the need of keeping such condition in the termination loop is understood, we must find an alternative way to implement the parallelism.

Our approach will be to implement an explicit splitting of the iterations of the loop among the threads. That is, we will perform explicitly what the `for` directive does automatically.

The parallelization of the loop with look like in Figure 13, where each loop in the parallel region must do a subset of the iterations of the original loop. The iterations assigned to each thread are defined by the initial value of `i`, the increment and the final value. Therefore, you need to modify some of these elements to ensure that each thread processes a different subset of iterations.

For example, assuming `s`=19, the values that the variable `i` will take along the loop are: 3, 5, 7, 9, 11, 13, 15, 17, 19. A way to split the iterations could be to assign a consecutive block of iterations to each thread. Given 3 threads, the distribution will be:

Thread 0	3	5	7
Thread 1	9	11	13
Thread 2	15	17	19

However, another way to split the iterations would be using a cyclic distribution:

Thread 0	3	9	15
Thread 1	5	11	17
Thread 2	7	13	19

In this case, cyclic distribution is easier to implement than block distribution, so we recommend it. Obviously, your implementation must work correctly for any value of `s` and number of threads. You will need to use OpenMP functions to get the number of threads and the thread identifier. For the sake of efficiency, do not call those functions in each loop iteration. Write the new parallel version and measure the execution time.

It is important to keep the exit condition in the loop when an exact divisor is found. In this way, if implemented correctly, when any thread discovers that a number is not prime, all other threads will eventually stop.

Note: It is advisable to add the `volatile` modifier to the variable that is used for loop control: `volatile int p;` The `volatile` modifier in the C language indicates that the compiler must not optimize the access to that variable (i.e., it must not load it in registers so that any access to it is done effectively on memory), in such a way that its modification from one thread will be visible earlier in the rest of threads[3].

[3]This behavior can also be accomplished using the `flush` sentence in OpenMP.

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

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 100,000,000. [N.B.: If the program takes too long, a smaller limit can be chosen. In order to obtain a good performance improvement, the sequential version should take around 1 or 2 minutes at least.]

The algorithm to implementing this process is described in Figure 14. Check the sequential version of this algorithm, measuring its execution time.

Given that a parallel version of the algorithm for checking if a number is prime is already available, it is trivial to solve this problem in parallel by just using that parallel version.

However, this approach has low performance. The reason is that the initial prime numbers are very small and computing each of them requires very little processing time, so there is no gain in splitting the workload among the threads.

A better strategy is to parallelize the loop on the main program (which is straightforward using OpenMP directives) and directly use the sequential version of the function `prime`. You should develop a parallel version based on that idea and measure the execution time. Finally, different scheduling strategies for the loop should be checked, using at least the following ones:

- Static without specifying the *chunk* size.
- Static with *chunk* size of 1.
- Dynamic.

Remember that the scheduling can be defined using two different ways:

1. Directly defining the scheduling in the OpenMP `for` directive using the clause `schedule` (such as `schedule(static,6)`).
2. Using the clause `schedule(runtime)` in the `for` directive, and giving a value to the environment variable `OMP_SCHEDULE` (e.g. `OMP_SCHEDULE="static,6"`). In this way you can change the scheduling without recompiling the program.