LAB REPORT

PARALLEL PROGRAMMING : LAB 02

Autors :		***Teacher :***
Etienne	**CORREGE**	**M.HAMELIN (ING5 SE Gr02)**
Arnaud	**SAINT CIRGUE**	

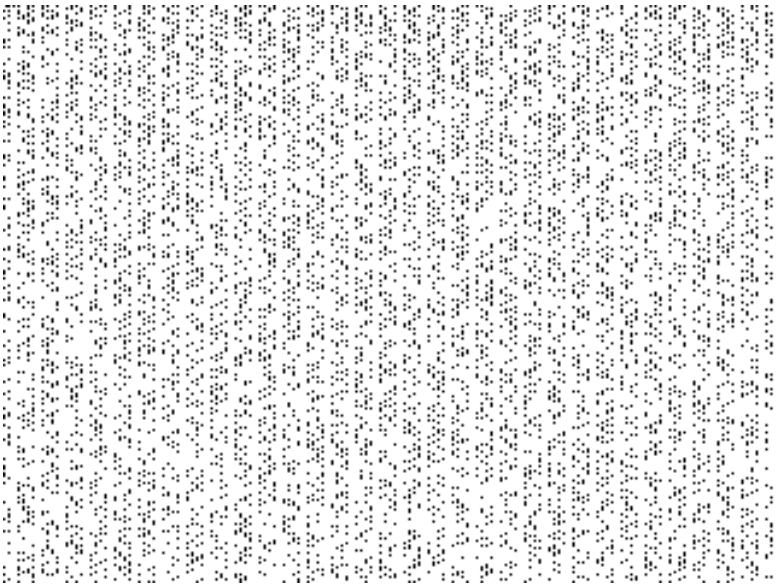

figure 0 : repartition of prime numbers from 1 to 76800

We certify that this work is original, the result of collaborative effort by the duo, and has been independently written.

Paris, 09/11/2023

Table of contents

0. Know your hardware

Q1: What is your OS/hypervisor system configuration?
How many CPU physical and logical cores does your PC run? How much RAM memory?
Use the commands lscpu, lsmem, cat /proc/cpuinfo, and cat /proc/meminfo to answer.

- Configuration VM (Arnaud Saint-Cirgue)

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  6
  On-line CPU(s) list:   0-5
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    CPU family:          6
    Model:               165
    Thread(s) per core:  1
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            2
    BogoMIPS:            5184.00
```

Figure 0.1 : cpu configuration on Arnaud Saint-Cirgue laptop

The computer has 2 CPU sockets. Each CPU socket has 6 physical cores. Hence, the computer has 6 physical cores in total. Each physical CPU core can run 1 thread. These threads are the core's logical capabilities. The total number of logical cores = CPU sockets × physical cores per socket × threads per physical core. Therefore, the computer has 1 × 6 × 1 = 6 logical cores in total.

Insert memory config screen

Figure 0.2 : memory configuration on Arnaud Saint-Cirgue laptop

By using the "cat /proc/meminfo" (Figure 0.4), we have a total of XGB RAM

- Configuration Dual Boot (Etienne Corrège)

```
etienne@Etibuntu:~$ lscpu
Architecture :                            x86_64
  Mode(s) opératoire(s) des processeurs :  32-bit, 64-bit
  Address sizes:                          39 bits physical, 48 bits virtual
  Boutisme :                              Little Endian
Processeur(s) :                           8
  Liste de processeur(s) en ligne :       0-7
Identifiant constructeur :                GenuineIntel
  Nom de modèle :                         Intel(R) Core(TM) i5-8250U CPU @ 1.6
                                          0GHz
    Famille de processeur :               6
    Modèle :                              142
    Thread(s) par cœur :                  2
    Cœur(s) par socket :                  4
    Socket(s) :                           1
```

Figure 0.3 : cpu configuration on Etienne Corrège laptop

With the same method, we have a total of 8 logical cores (Figure 0.3).

```
etienne@Etibuntu:~$ cat /proc/meminfo
MemTotal:        8000412 kB
MemFree:          649180 kB
MemAvailable:    4637456 kB
Buffers:          118732 kB
```

Figure 0.4 : memory configuration on Etienne Corrège laptop

By using the “cat /proc/meminfo” (Figure 0.4), we have a total of 8GB RAM.

1. Shared variables

1.1 Adding numbers in different threads

Q2: Read the original code in the sequential folder, compile it and run it on your target. Is the displayed result correct? Why?

```
Thread 1 start
Thread 0 start
Thread 2 start
Thread 3 start
Thread 3 stop
Thread 0 stop
Thread 1 stop
Thread 2 stop
SUM = 1147558492
real 14.63
user 55.03
sys 0.00
```

Figure 1.1.1 : print execution of “sum_value_threads.c” without mutex

When we first run the code we see that the output is a SUM = 1147558492. This answer does not seem correct and is not the output wished. Indeed, we want to make a SUM = 4000000000. We can explain this result because of the threads. The variable shared is declared as a global variable and when a thread modifies its value, there is a cohenrecy miss because the variable is not protected and several threads can write on the variable at the same time. This is why, in one cycle, the threads interrupt each other and don’t wait for one to finish before processing.

Q3: Measure the values of real, user, sys, (user+sys)/real of the original (in the sequential folder) code real execution.

As evaluated in the previous question, we have the different outputs :

Real	User	Sys	Ratio
14,63	55,03	0,00	3,76

Figure 1.1.2 : execution time on "sum_value_threads.c" code without mutex

Q4: Using thread synchronization functions, modify the code sum_value_threads.c (in the work folder) to solve the problem identified.

```
Thread 0 start
Thread 2 start
Thread 1 start
Thread 3 start
Thread 0 stop
Thread 2 stop
Thread 1 stop
Thread 3 stop
SUM = 4000000000
real 8.04
user 8.04
sys 0.00
```

Figure 1.1.3 : print execution of "sum_value_threads.c" with mutex

When implementing mutex, the problem is solved because it separates the threads and let the for loop finish for one thread before the next one works on the loop. When one thread finishes the computation of one loop, the shared value is safe again and the next thread can work on it.

Besides, depending on where we place our mutex in the code (inside the for loop or outside the loop), the execution time can significantly decrease or increase in the other case (which means that the processor fetches in the cache memory slot or in the ram).

1.2 Road to prime numbers

Q5: Give the values of real, user, sys, (user+sys) / real of the modified code. Compare the execution times real with those of Q3. Explain the difference between those measurements.

Real	User	Sys	Ratio
8,04	8,04	0,00	1

Figure 1.2.1 : execution time on "sum_value_threads.c" code with mutex

Using the mutex is significantly decreasing the real and user time because of the coherence miss. When using mutex, the processor fetches the value in the cache memory and not in the RAM, this is why the execution time significantly decreases (as one the reason, shorter distance to cover).

Q6: The function is_prime() will check if the number in argument is a prime number or not. Write this function and test it from 2 to 10000000 in the main() function to validate (expected : 664579)

```
ubuntu@ubuntu-2204:~/Desktop/Ece/Lab2/TP2/work/2-prime_numbers_threads$ ./prime_numbers_threads 1
Number of threads : 1
Thread 0 start
664579 prime numbers in thread 0
Thread 0 stop
NUMBER PRIMES = 664579
```

figure 1.2.2 : output of the prime_number_threads code with 1 thread (which is equal to a sequential execution)

Q7: Give the values of real, user, sys, (user+sys) / real, of the sequential code execution.

We get the following results

real	1,26
user	1,24
sys	0,01

Then we compute the coefficient, which gives this result : 0,99

Q8: Using the arguments argc and argv in the main() function, get the number of threads from the user. Write the thread_runner() function.

We decided to leave a default value of 4 threads hard coded in the code. When an argument is passed in the command line, the number written by the user replaces the default value as seen on the two below screens.

```
ubuntu@ubuntu-2204:~/Desktop/Ece/Lab2/TP2/work/2-prime_numbers_threads$ ./prime_numbers_threads
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
183072 prime numbers in thread 0
Thread 0 stop
165441 prime numbers in thread 1
Thread 1 stop
159748 prime numbers in thread 2
Thread 2 stop
156318 prime numbers in thread 3
Thread 3 stop
NUMBER PRIMES = 664579
```

(a)

```
ubuntu@ubuntu-2204:~/Desktop/Ece/Lab2/TP2/work/2-prime_numbers_threads$ ./prime_numbers_threads 8
Number of threads : 8
Thread 0 start
Thread 1 start
Thread 3 start
Thread 2 start
Thread 6 start
Thread 7 start
Thread 5 start
Thread 4 start
96469 prime numbers in thread 0
Thread 0 stop
86603 prime numbers in thread 1
Thread 1 stop
81796 prime numbers in thread 3
Thread 3 stop
80303 prime numbers in thread 4
Thread 4 stop
77729 prime numbers in thread 7
Thread 7 stop
83645 prime numbers in thread 2
Thread 2 stop
79445 prime numbers in thread 5
Thread 5 stop
78589 prime numbers in thread 6
Thread 6 stop
NUMBER PRIMES = 664579
```

(b)

Figure 1.2.3 : (a) execution print of prime_numbers.c with 4 threads and (b) 8 threads

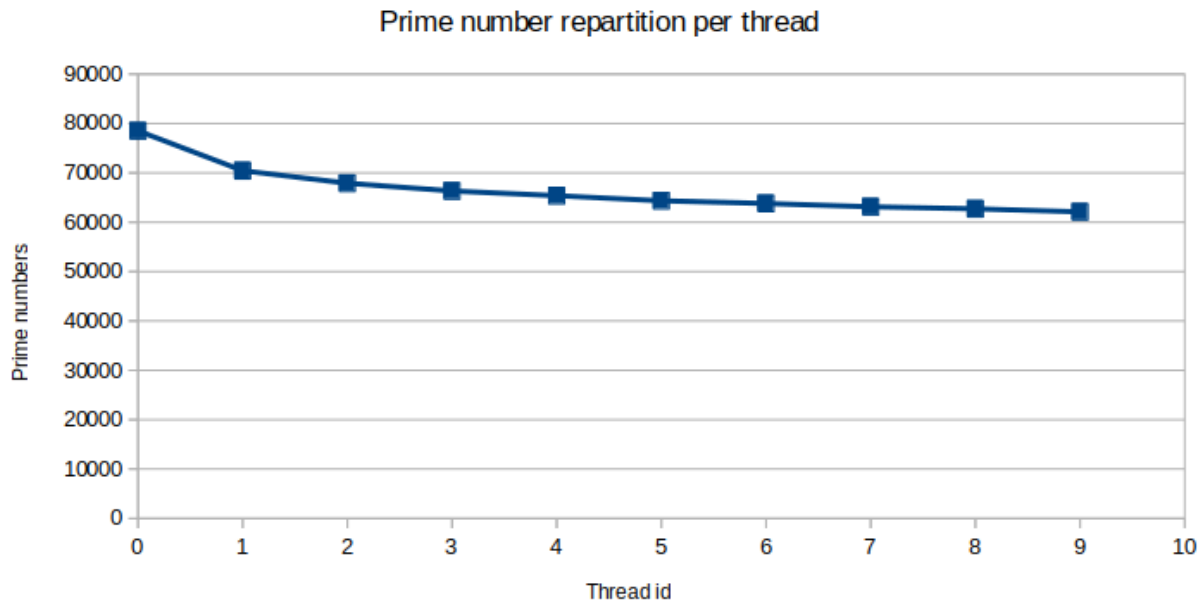

figure 1.2.4 : Graph showing the distribution of prime number per threads

In addition, we decided to plot a graph to see how the distribution of prime numbers is. As the primary numbers in our code are listed in ascending order, one after the other, we can see from the diagram that there are slightly more at the beginning of the real numbers and fewer as we move away from 0.

We can see that there are differences in the number of prime numbers in each thread. To optimize that, we need an algorithm that can share out the same amount of prime numbers in each thread. We thought of sharing out the numbers in the threads cutting numbers in lines of 10.

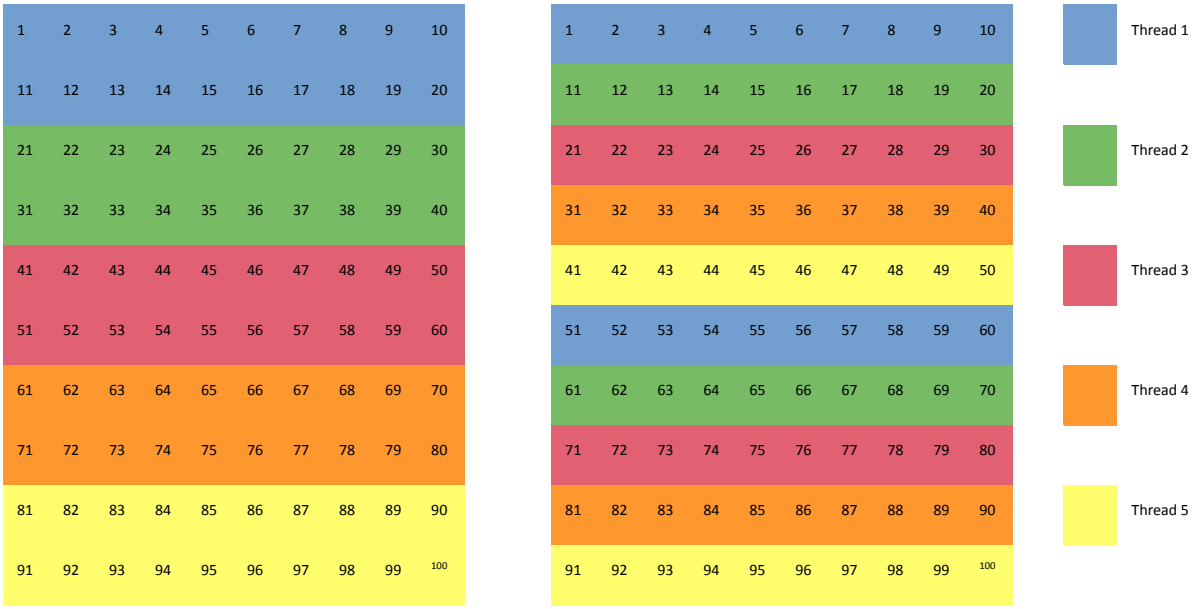

figure 1.2.5 : old distribution of threads versus new distribution

In some cases the result was very satisfying, for example, when we use 10 threads, every thread has 66k (+-100) prime numbers. In other cases, like 3 or 6 threads, we had deviations of 10k numbers which is very significant. We had to test with various numbers, other than 10 and we obtain the following table (figure 1.2.6)

		Number of threads								
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	1	0	0	0	0	0	0	0	0	0
	2	1	0	1	0	0	0	1	0	0
	3	1	1	1	0	1	0	1	1	0
	4	1	0	1	0	0	0	1	0	0
	5	1	0	1	1	0	0	1	0	1
	6	1	1	1	0	1	0	1	1	0
	7	1	1	1	0	1	1	1	1	0
	8	1	0	1	0	0	0	1	0	0
	9	1	0	1	0	0	0	1	0	0
	10	1	0	1	1	0	0	1	0	1
Jump value	**11**	1	0	1	1	0	0	1	0	1
	12	1	1	1	0	1	0	1	1	0
	13	1	1	1	0	1	0	1	1	0
	14	1	1	1	0	1	1	1	1	0
	15	1	1	1	1	1	0	1	1	1
	16	1	0	1	0	0	0	1	0	0
	17	1	0	1	0	0	0	1	0	0
	18	1	1	1	0	1	0	1	1	0
	19	1	1	1	0	1	0	1	1	0
	20	1	0	1	1	0	1	1	0	1
	30	1	1	1	1	1	0	1	1	1

figure 1.2.6 : Table of validate distribution for different step in each thread

Finally we decided to make a mix of 14 and 15 numbers by lines so the distribution can always be good for 1 to 10 threads. We can't say that it's valid for a greater quantity of 10 threads.

Q9: Trace the curve of real, as a function of the number of threads $real = f(nth)$

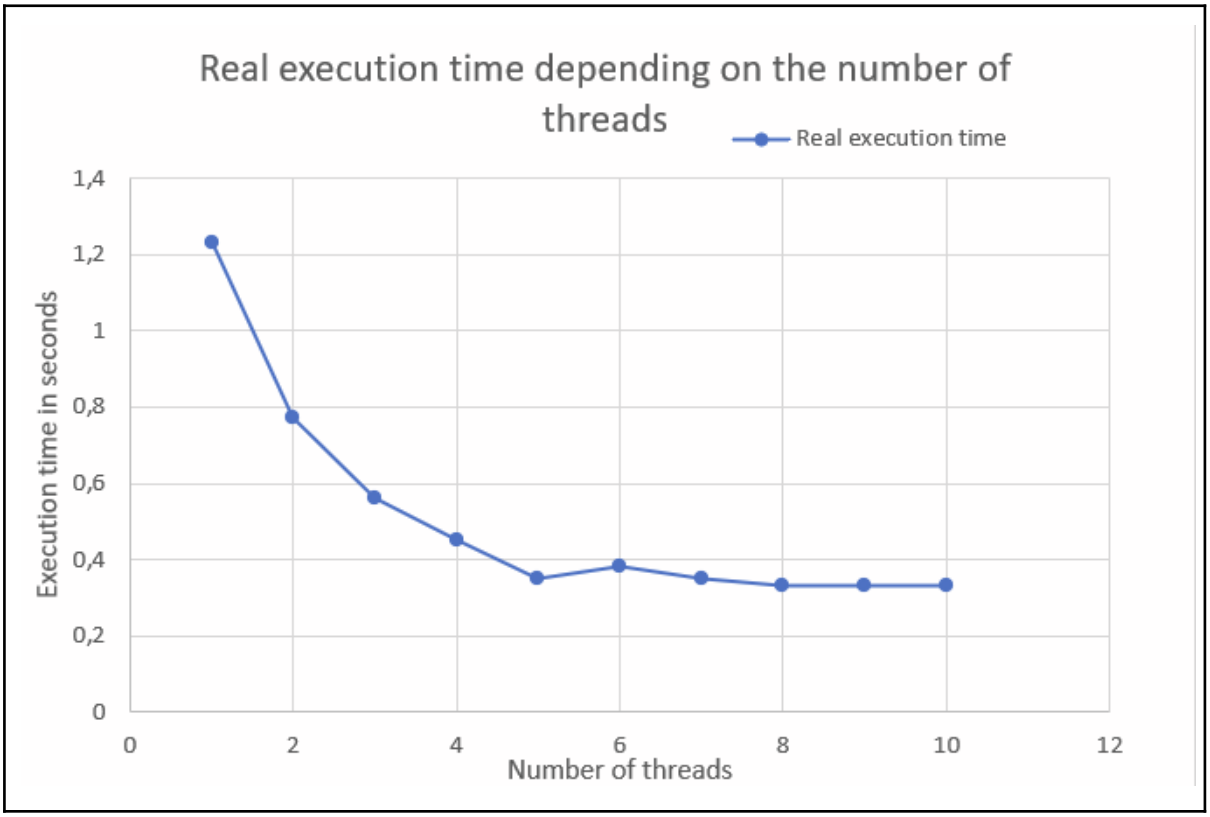

figure 1.2.7 : Graph showing the evolution of the real execution time depending on the number of threads used

As we can see on the graph, the execution time does not decrease when we reach around the number of processors (here 6). In our case, it tends to 0,34 seconds.

After rethinking our code, we have decided to test another way of dividing prime numbers by threads. You can see below the new results of our code compared with the first output.

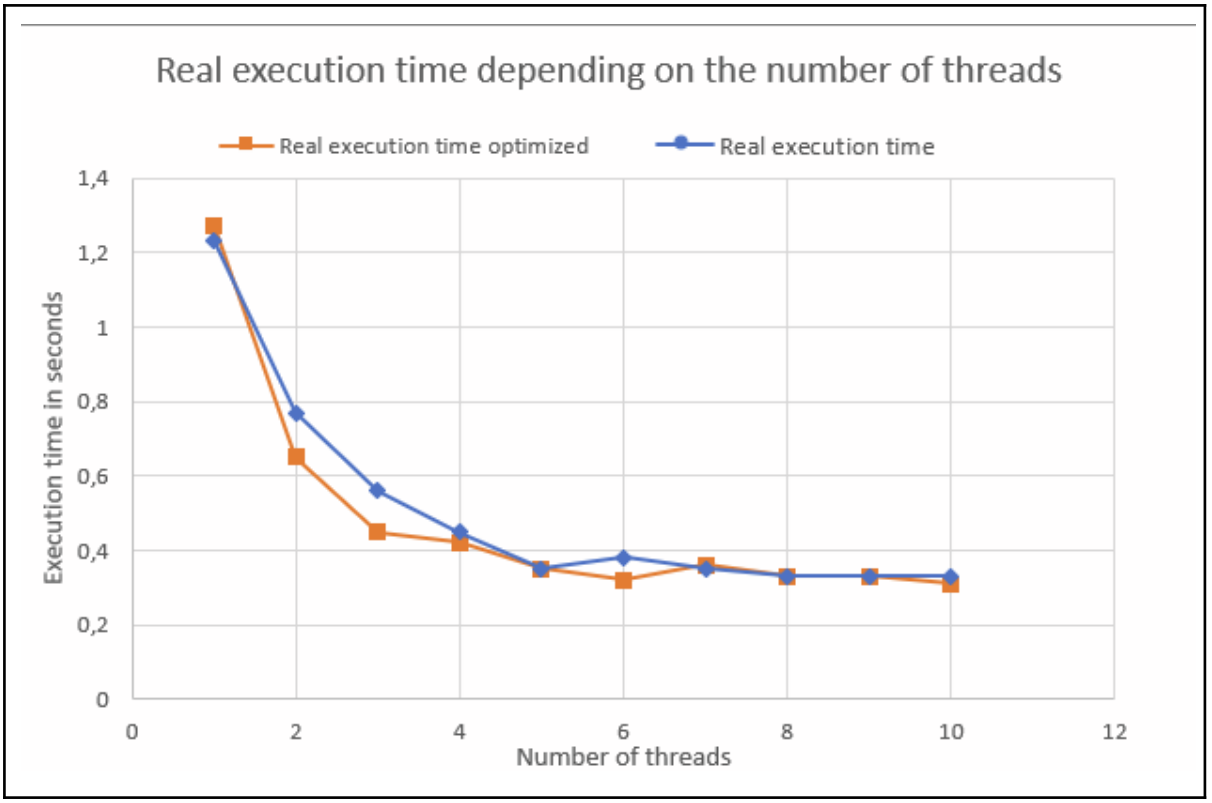

figure 1.2.7 : Graph comparing the result of the 2 different way of coding the prime_number_threads program

Now, the execution time tends to 0,33 seconds and we have a better speedup and the best execution time is reached when we use 6 CPUs, which makes sense.

Q10: Using Amdahl's law, estimate the fraction of "perfectly parallel" code as a function of the number of threads and plot those results.

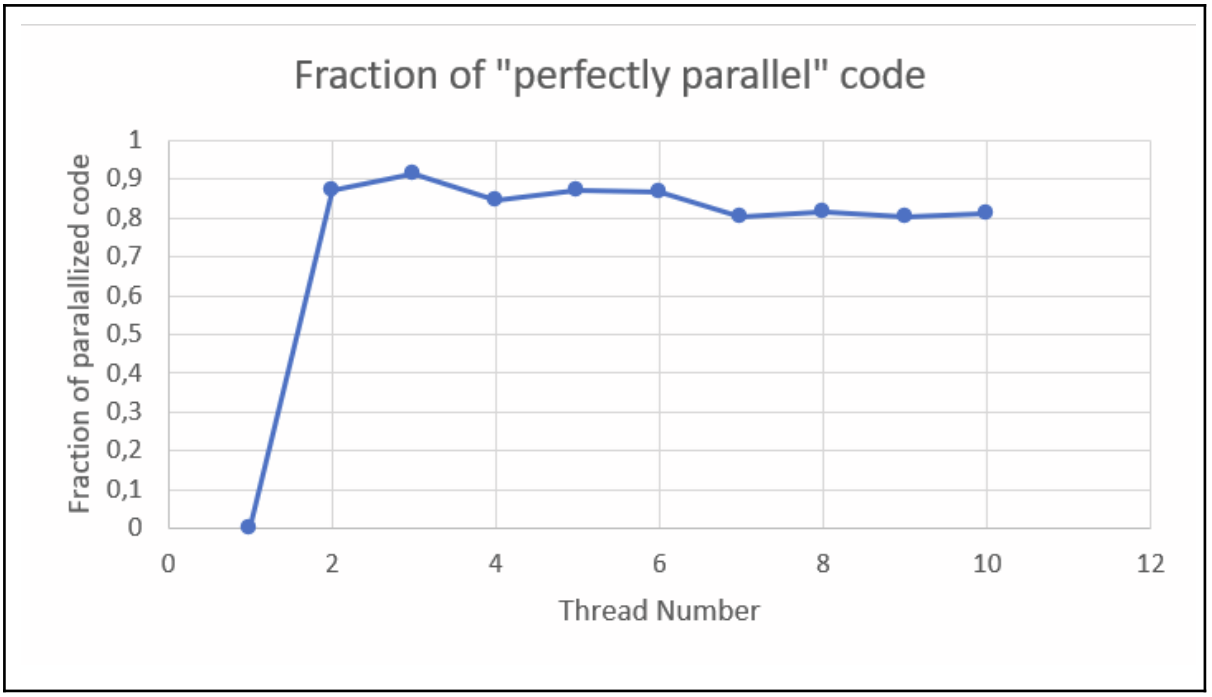

figure 1.2.8 : Graph showing the evolution of the efficiency of the parallelized code

The first value of P is equal to 0 because there is only 1 thread, which is equal to a sequential run. Besides, we can see that the value of p tends to 85 % if we look at the data until 6 threads. The optimization is less efficient after 6 threads because I only have 6 processors on my computer.

We think that it is weird that the best optimization is for 3 threads and not 6 threads (which is equal to our number of CPUs). It is maybe because we did not optimize the code well. Also, when not using the mains socket to power the computer, the performance drops to 70 - 75%

2. Producer/consumer problem

Q11: Build, then run, the prod-cons program using the "make time" command. Explain the difference between the total produced and consumed.

```
ubuntu@ubuntu-2204:~/Desktop/Ece/Lab2/TP2/source/3-prod_cons$ make time
gcc -O0 -g       -I./    -Wall           -o fifo.o -c fifo.c
gcc -O0 -g       -I./    -Wall           -o prod-cons.o -c prod-cons.c
gcc -o prod-cons fifo.o prod-cons.o -lm -lpthread
/usr/bin/time -p ./prod-cons 2 10
Start with 2 producers, and 10 consumers
Producer #0 starts
                Consumer thread #0
Producer #1 starts
                Consumer thread #2
                Consumer thread #1
                Consumer thread #5
                Consumer thread #4
                Consumer thread #8
                Consumer thread #3
                Consumer thread #7
                Consumer thread #9
                Consumer thread #6
Producer #0 finishes, produced 500000 items
Producer #1 finishes, produced 500000 items
                Consumer #5: received 322742 items
                Consumer #1: received 602616 items
                Consumer #6: received 68402 items
                Consumer #9: received 136513 items
                Consumer #8: received 280864 items
                Consumer #3: received 128000 items
                Consumer #2: received 201649 items
                Consumer #0: received 195959 items
                Consumer #7: received 106812 items
                Consumer #4: received 284245 items
Total: 1000000 produced, 2327802 consumed. Lost/found: -1327802 (-132%)
real 0.17
user 0.85
sys 0.00
```

figure 2.1.1 : Output of the prod-cons program.

In this simulation we have 2 producers and different consumers. In the end, there are 1000000 goods produced and 2327802 goods consumed. There is a dramatic situation because the different consumer threads called the FIFO in the same time which caused the over-consumption.

Q12: In the work directory, change the implementation of the fifo module so that concurrent accesses to shared data are protected with a mutex. Verify that your implementation is now correct, and try to improve performance.

We added mutex to the fifo structure, created it in the fifo_new function, and destroyed it in the fifo_destroy function. Finally we implement the lock and unlock mutex in the push and pop functions.

```
Start with 2 producers, and 10 consumers
Producer #0 starts
Producer #1 starts
                Consumer thread #0
                Consumer thread #1
                Consumer thread #2
                Consumer thread #3
                Consumer thread #4
                Consumer thread #5
                Consumer thread #6
                Consumer thread #7
                Consumer thread #8
                Consumer thread #9
Producer #1 finishes, produced 500000 items
Producer #0 finishes, produced 500000 items
                Consumer #7: received 97659 items
                Consumer #8: received 100979 items
                Consumer #1: received 102020 items
                Consumer #0: received 102825 items
                Consumer #9: received 102229 items
                Consumer #3: received 100095 items
                Consumer #2: received 100529 items
                Consumer #6: received 99404 items
                Consumer #4: received 95379 items
                Consumer #5: received 98881 items
Total: 1000000 produced, 1000000 consumed. Lost/found: 0 (00%)
real 1.80
user 5.48
sys 7.60
```

figure 2.1.2 : console from the prod_cons time execution

After implementing mutex, the total good produced is equal to the total good consumed, 0% of lost.