Fundaments of HPC
Second Assignment

Nicola Domenis

December 11, 2019

1 Introduction

We present the second assignment in the course of FHPC. We will discuss about:

Exercise Zero
Comparison of an OpenMP program that implements the "touch by one"policy versus the implementation of a "touch by all" policy;

Exercise One
Rewriting an MPI code about a montecarlo estimation of pi using OpenMP and comparing the two.

2 Exercise 0

We start by observing two slightly different codes: 01_array_sum.c and 04_touch_by_all.c. They implement the same algorithm: summing the first N natural integers. They both use the OpenMP standard to take on a parallel approach with multiple threads. In 01_array_sum.c the array that contains the addends for the summation is initialized by the master thread, while in 04_touch_by_all.c the array is initialized in parallel by all threads in a parallel region. The first case is called "touch by one" policy and the second case is called "touch by all" policy. The second policy allows for multiple threads to "own"(i.e to store) the array in their cache memory, while in the first policy only the master thread "owns" the array in its cache. We will see how the "touch by all policy" speeds up the execution time by favouring the first cache hits for each processor during the calculations on the array. In other words each processor already has a portion of the array in the cache, so the access will be faster. The analysis of the programs can start by a strong scalability test.

2.1 Strong Scalability Test

We report the elapsed times versus the number of cores and also the speedups versus the number of cores for both programs:

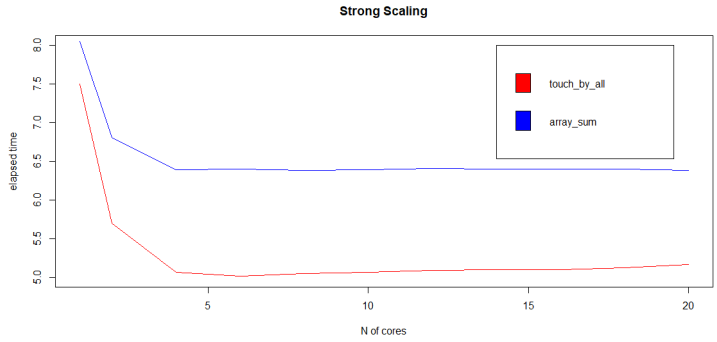

Figure 1: Elapsed times for $N = 10^9$

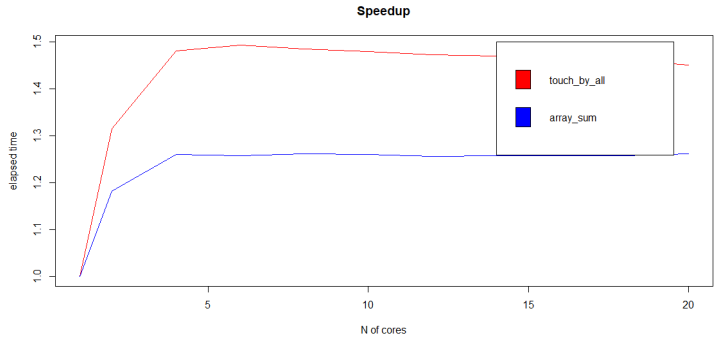

Figure 2: Speedup for $N = 10^9$

We see that both codes scale and that the code that implements the touch-by-all policy is faster. The faster access to the cache memory saves time during the process, making touch_by_all.c a faster program.

2.2 Parallel Overhead

We will proceed our comparison by calculating the parallel overhead of the two programs. We can use the formula $e(n,p) = \frac{\frac{1}{S_p(n,t)} - \frac{1}{p}}{1 - \frac{1}{p}}$ to estimate the parallel overhead. A plot with the measures we have would return:

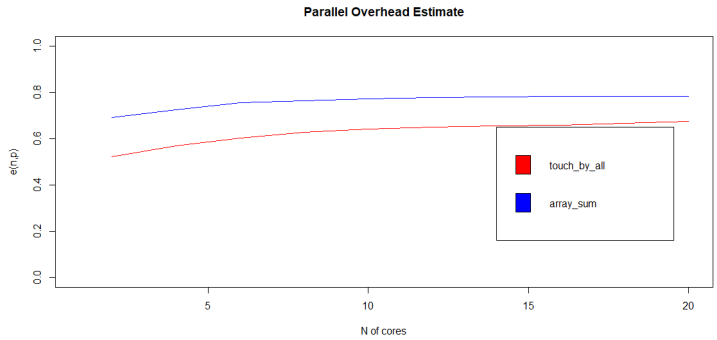

Figure 3: Overhead estimation for $N = 10^9$

The touch_by_all code once again is better than the array_sum code, since it has a lower parallel overhead. But both lines are almost constant, showing that both codes have an almost constant parallel overhead while we increment the number of processors.

2.3 Performance evaluation

Now lets compare some valuable metrics for the two codes' executions:

Metric	measures					mean	
CPU cycles	array_sum	372889698	143797361	671828181	373809554	390581199	-
	touch_by_all	602487792	16412868	15030736	15901403	162458200	=
							228122999
Cache misses	array_sum	9459	11492	49514	9484	19987.25	-
	touch_by_all	10356	10256	8417	12081	10277.5	=
							9709.75
Elapsed time	array_sum	0,014486573	0,007287606	0,024476034	0,014344770	0.01514875	-
	touch_by_all	0,021872540	0,002893806	0,002676134	0,002677902	0.007530096	=
							0.00761865

Even if the single performances, measured with Perf, are quite similar in their randomness, we see that the touch_by_all code outperforms in average the array_sum code in all the metrics. The most important metric is the cache misses, which decreases if a portion of the array is saved in each processor, which is the "touch by all" policy case.

2.4 Conclusion

The touch-by-all policy gives us a more performant code in terms of memory access and speed. Once again we have found an optimization approach that exploits the cache's speed of access to speed up the execution time of the code.

3 Exercise 1

We need to rewrite the code mpi_pi.c by means of OpenMP directives. We had to solve all the problems we encountered in lesson, like avoiding race conditions and false sharing. In the end we produced the simple code openmp_pi.c that exploits thread parallelization to compute an estimate of pi based on a montecarlo geometrical approach.

3.1 Strong and Weak Scalability

We show the plots for the strong and weak scalability:

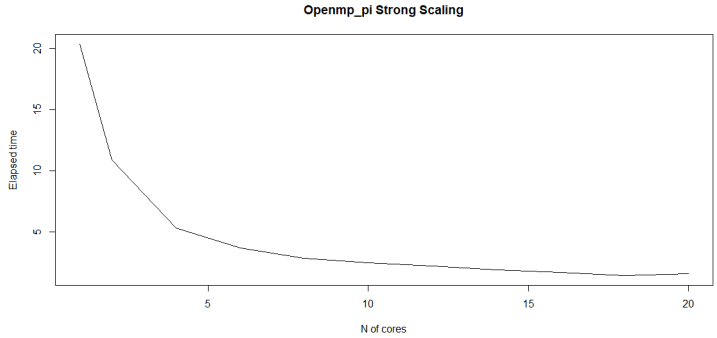

Figure 4: Strong scaling test for $N = 10^9$

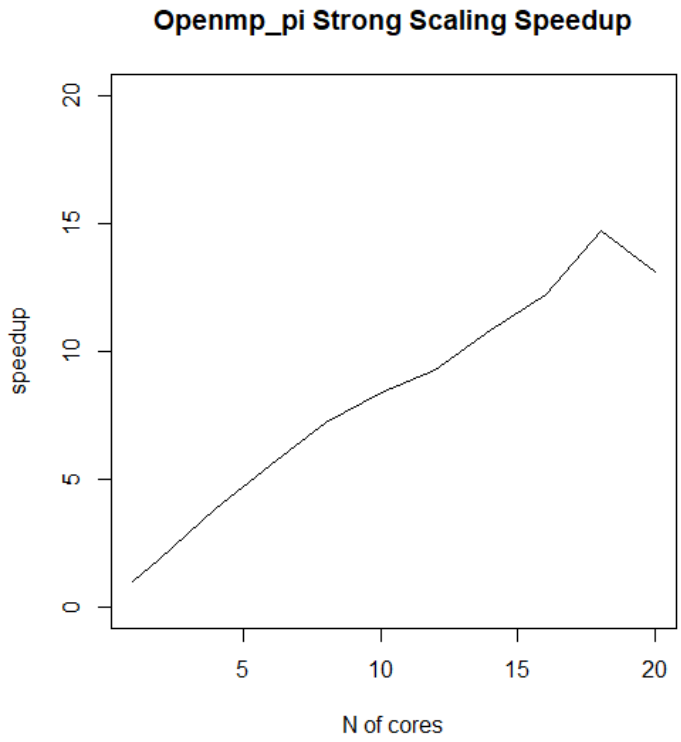

Figure 5: Speedup for $N = 10^9$

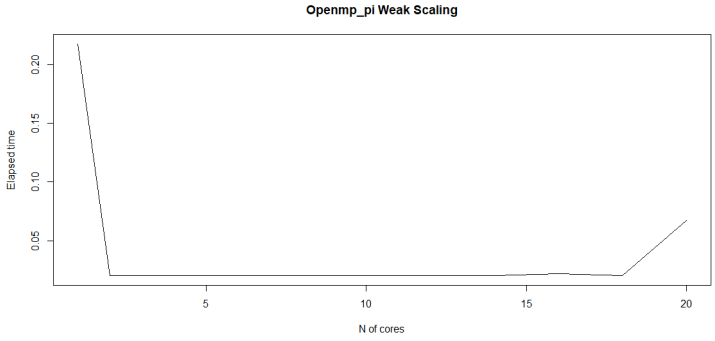

Figure 6: Weak Scaling test for $N = 10^4 * P$

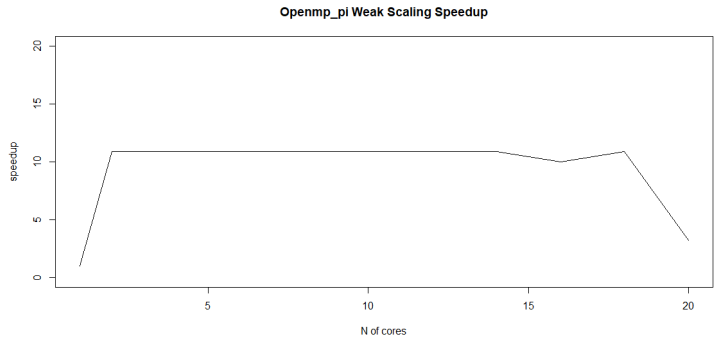

Figure 7: Weak Scaling speedup for $N = 10^4 * P$

In the strong scalability setting the code scales very well, the speedup is almost linear. The weak scalability test also shows that the program handles well problems of increasing size through an increase of threads.

3.2 Parallel overhead

Now we estimate the parallel overhead, using again the formula to calulate $e(n, p)$ in the strong scaling context:

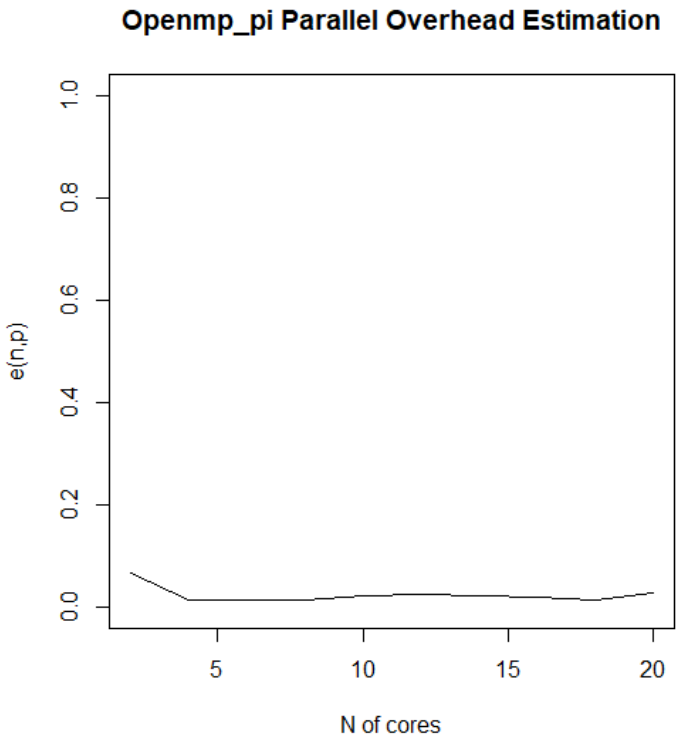

Figure 8: Overhead estimate for $N = 10^4 * P$

We see that the overhead estimation remains almost constant, showing that the program's parallel overhead doesn't increase over the number of processors.

3.3 Comparison with mpi_pi.c

Let's make some comparisons between the openMP version of the file and the MPI version. As a first comparison we can plot the execution times of the two programs given a growing problem size and a fixed number of processors. Given 20 processors ,we run mpi_pi.c on all processors, and then we run omp_pi.c on 20 threads, one per processor. The plot is:

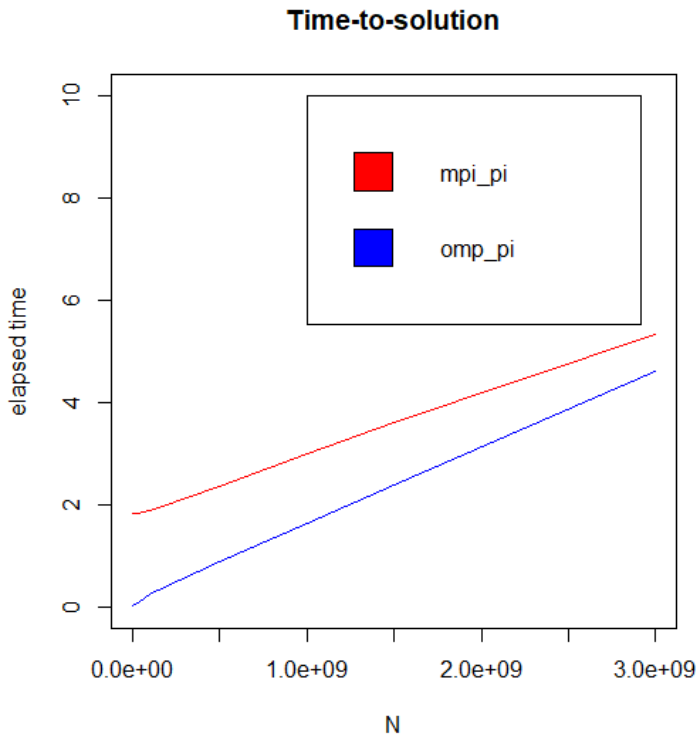

Figure 9: Elapsed time for P =20

We see that omp_pi.c is faster than mpi_pi.c, even thought the fact that the message passing in mpi_pi.c does some buffering(latency hiding) and thought the fact that there is an "atomic"' section in omp_pi.c that introduces a serialization that slows down the program. Now it's time to test the performance of mpi_pi.c on a single node and on multiple nodes. We started with 1 node with 20 cores and we finished with 10 cores having 2 nodes each. We see if splitting processors on multiple nodes gives a slower execution time due to the network latency.

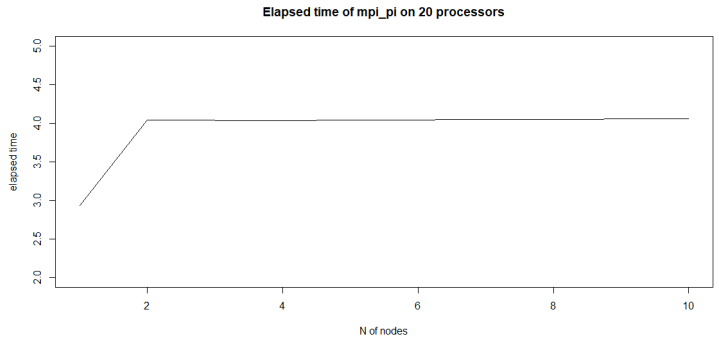

Figure 10: Overhead for $N = 10^4 * P$

We see that the separation among nodes slows down the execution time in respect with the case of just one node. The openmp_pi.c program running on 20 threads would instead have an elapsed time of 1.56 seconds, which is faster. The inter-core communication is faster than inter-node communication. We can then compare the portions of the codes that have the same function. There are two portions of interest: the initialization and the identification of the points that belong inside the circle and the consequent partial sum of such points, and then there is the full summation of those partial sums. These two portions are critical in the two codes. The first portion is parallelized in both codes, so it could be seen as a parallelization comparison. The execution time for the first portion is the following:

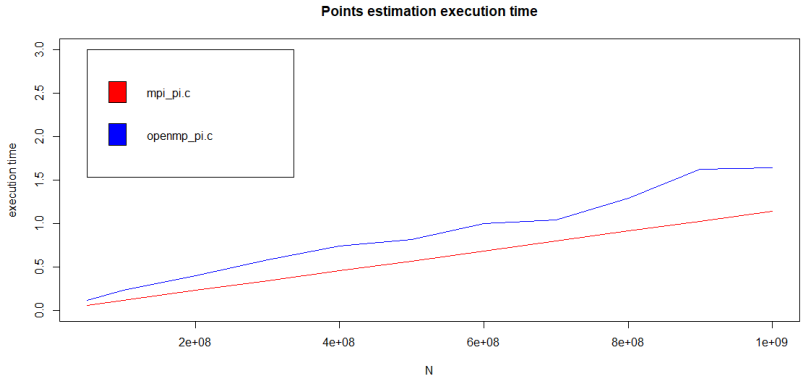

Figure 11: Execution time for a growing N

We see that, in contrast with the other graph, that openmp_pi.c has a slower execution time. This points out the fact that mpi_pi.c has a higher overhead than openmp_pi.c that slows down the total execution time. Now we see that

the second portion of the code has two similar implementations in mpi_pi.c and openmp_pi.c. In mpi_pi.c the total sum of the results from each processor is carried out thanks to message passing. The master node fetches the partial sums from each slave processor. In openmp_pi.c the total sum is carried out thanks to a serialization due to the parallel atomic directive. In both cases we have the sequential participation of each parallel process to the resulting summation. We see that this portion of the code has an execution time given by:

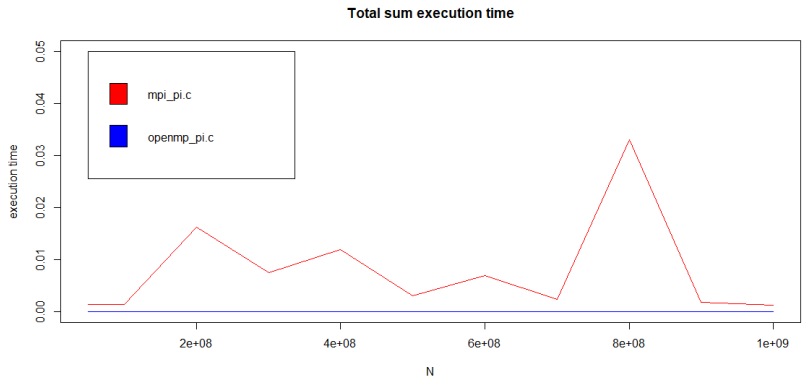

Figure 12: Execution time for a growing N

We see that the execution of the second portion of the code is way faster in the OpenMP version of the code. Since in both codes there is still a sequential access from the parallel threads/processors we can clearly identify the cause of this difference in the overhead introduced by the send and receive functions.
We plot here the sum of the times of the two portions of the code:

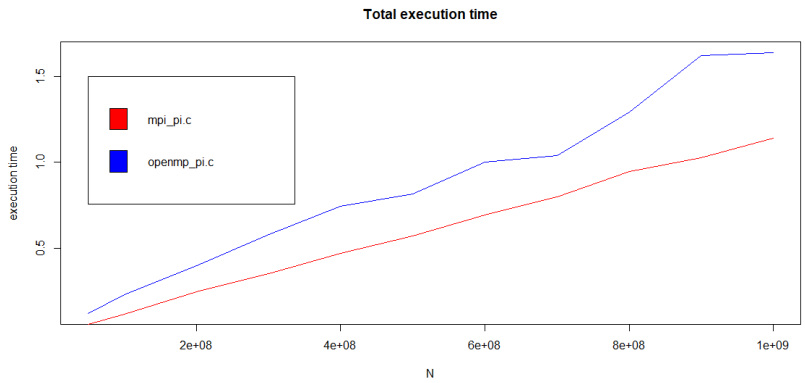

Figure 13: Sum of the execution times for a growing N

Now we have the elapsed time for both programs, and we have the walltime for both programs. We can estimate the parallel overhead of both codes by subtracting the walltime from the execution time:

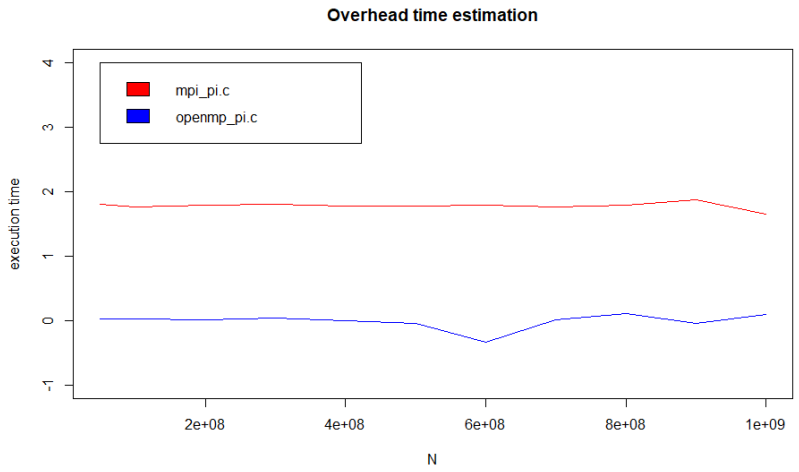

Figure 14: Overhead times estimation for a growing N

Here we see again that mpi_pi.c has a higher overhead than openmp_pi.c. We also see that openmp_pi.c's overhead is again constant.In the previous plot the overhead has been calculated on the parallel speedup of the strong scaling test, while here we are instead increasing the problem size. We see that there is some noise in the measurement and that the graph of openmp_pi.c's overhead estimation is mostly negative. This is due to the truncation error on the value taken from /usr/bin/time because of the small difference between elapsed time and walltime (that can be seen by outputting all the variables containing the results of /usr/bin/time and omp_get_Wtime() and on the graph below).

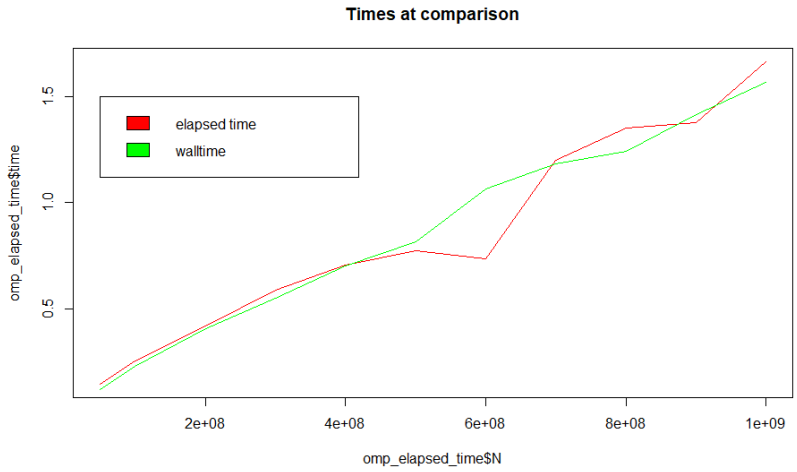

Figure 15: execution times comparison for a growing N

We can still conclude that the parallel overhead of openmp_pi.c is relatively small and also constant, much like the e(n,t) estimate. The second estimate we did in Figure 14 is useful to show that mpi_pi.c has a bigger parallel overhead than openmp_pi.c.

3.4 Strong scalability comparison

We can obtain the same results in a strong scalability test. We plot the elapsed time and the walltime for both codes. The distance between the two graphs represents the overhead estimate:

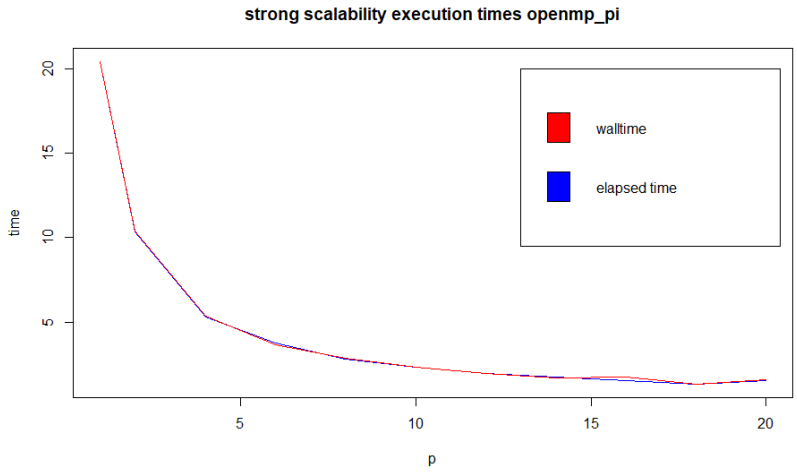

Figure 16: execution times comparison for $N = 1000000000$

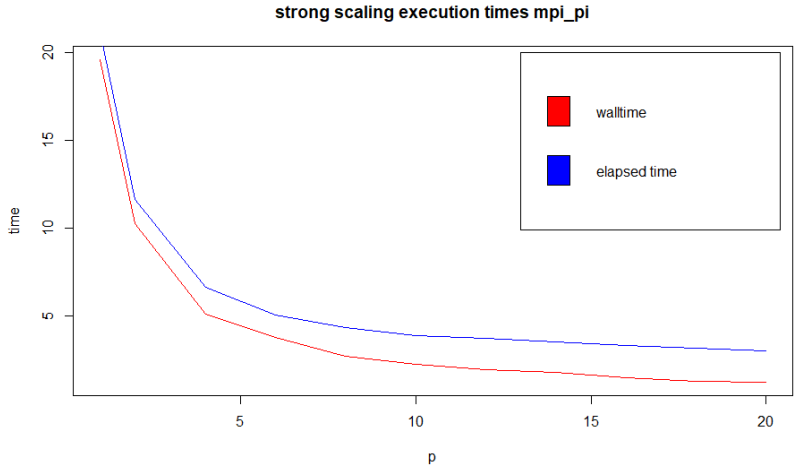

Figure 17: execution times comparison for $N = 1000000000$

In the second plot there is a larger difference between walltime and elapsed time, showing once again that mpi_pi_pi.c has a bigger overhead than openmp_pi.c. We plot now both the previous graphs on the same graph:

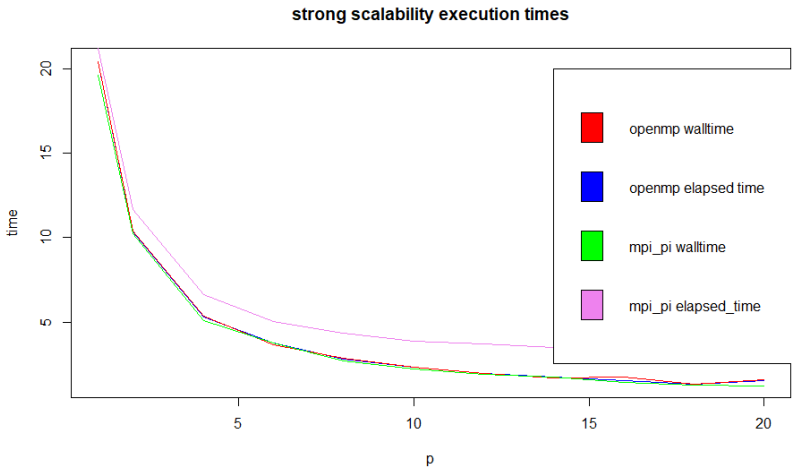

Figure 18: execution times comparison for $N = 1000000000$

Here once again we see that the two walltimes are comparable, while mpi_pi.c's walltime is a little faster than openmp_pi.c's walltime(as we saw in the previous figure 13), and that mpi_pi.c's elapsed time execution is longer.

3.5 Conclusion

In conclusion we can say that the MPI version of the code has a faster execution of the script, but its parallel overhead makes the code slower than the OpenMP version, as we have seen by comparing the walltimes with the elapsed times.
We have seen that the overhead estimation of the MPI version has a flat and non-negligible graph with respect to the OpenMP version of the code, where the overhead is flat and negligible. Figure 14 proves our point.