PAR Laboratory Assignment

Lab 2: OpenMP programming model and analysis of overheads

Semester 2 - 2018

Group par2203

Bernat Gené Škrabec

Ferran Ramírez Navajón

Part1: OpenMP questionnaire

A) Basics

1.hello.c

1. How many times will you see the "Hello world!" message if the program is executed with "./1.hello"?

24, which is the number of threads that are being used.

2. Without changing the program, how to make it to print 4 times the "Hello World!" message?

Using 4 threads with the command line *OMP_NUM_THREADS=4 ./1.hello*

2.hello.c: Assuming the OMP NUM THREADS variable is set to 8 with "export OMP NUM THREADS=8"

1. Is the execution of the program correct? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier) Which data sharing clause should be added to make it correct?

No, because the id variable is shared by default and therefore is overwritten due to race conditions.

To correct it we should add the following line: *private(id);* With that we can fix the problem of repeated sequences, but we can not control their order.

2. Are the lines always printed in the same order? Could the messages appear intermixed?

They are not. The messages indeed appear intermixed. This is because we have no control over the concurrency of execution of each thread. They are running in parallel and each will finish and print on its own.

3.how_many.c: Assuming the OMP_NUM_THREADS variable is set to 8 with "export OMP NUM THREADS=8"

1. How many "Hello world ..." lines are printed on the screen?

16, because we have set 8 threads to execute the program.The first line is printed one time for thread. After that, in the code, we have set to 2 the number of threads to execute it. Third one has num_threads set to 3. The fourth printf is using 2 threads as we set before, and the last one has no defined a parallel section due to the condition if(0), and therefore is printed just once.

2. If the if(0) clause is commented in the last parallel directive, how many "Hello world ..." lines are printed on the screen?

Between 16 and 19 lines randomly, because of this clause: *num_threads(rand()%4+1)*.

4.data_sharing.c

1. Which is the value of variable x after the execution of each parallel region with different data-sharing attribute (shared, private and firstprivate)?

Shared: 8, sometimes 7

Private: 5

Firstprivate: 71

We get 8 with the *shared* part because the variable is shared and each thread increments the variable one. Sometimes we get less because of a race condition.

When we declare *private*, each thread has its own variable, which is not initialized and therefore each thread increments from zero and prints one. After the execution of each thread we get 5, the original variable.

With *firstprivate* each thread also has its own variable but they are taken from the original x from outside the parallel region. This is why each thread prints 72, but at the end we still get 71.

2. What needs to be changed/added/removed in the first directive to ensure that the value after the first parallel is always 8?

What needed to be changed was the *shared(x)* condition of the first directive, to *reduction(+:x)*, in order to avoid race conditions. That is because “reduction” clause lets the compiler ensure that the shared variable is properly updated with the partial values of each thread.

5.parallel.c

1. How many messages the program prints? Which iterations is each thread executing?

26. Each thread executes up to iteration 7, starting by their id number. This means that thread 0 will execute itreations 0 to 7, thread 1 will execute iterations 1 to 7, thread 2 2 to 7 and so on.

2. Change the for loop to ensure that its iterations are distributed among all participating threads.

This is our code:

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    int id=omp_get_thread_num();
    for (int i=id*(N/NUM_THREADS); i < (id+1)*(N/NUM_THREADS); i=i+1) {
            printf("Thread ID %d Iter %d\n",id,i);
    }
}
return 0;
```

6.datarace.c (execute several times before answering the questions)

1. Is the program always executing correctly?

No, sometimes we get an error message stating that something went wrong and showing a different value of x than usual. This is due to race conditions; the x variable is not yet updated when other threads start using it.

2. Add two alternative directives to make it correct. Explain why they make the execution correct.

The original problem arises in the line where the variable “x” is incremented. If we write right before it either “pragma omp atomic” or “pragma omp critical” we avoid the previous race conditions and thus always get a correct result.

7.barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?

Not everything, but we can affirm some things.We can assure the following:

First we get the “going to sleep for x seconds” in no particular order.

Then each thread will give the message “wakes up and enters barrier…” in order, because of the sleep time assigned to each one of them.

At last we will get the “We are all awake” message, again in no particular order. Threads do not exit the barrier in a specific order, it may change at every execution.

B) Worksharing

1.for.c

1. How many and which iterations from the loop are executed by each thread? Which kind of schedule is applied by default?

Each thread is executing 2 iterations, corresponding to ((thread’s id) * 2) and ((thread’s id) * 2 + 1). Being 2 the result of dividing the number of iterations by the number of threads.

This means the schedule applied by default is static.

2. Which directive should be added so that the first printf is executed only once by the first thread that finds it?

With “#pragma omp single” directive only one thread of the team executes the structured block.

```c
int main()
{
    int i;

    omp_set_num_threads(8);
    #pragma omp parallel
    {
        #pragma omp single
        printf("Going to distribute iterations in first loop ...\n");
        #pragma omp for
        for (i=0; i < N; i++) {
                int id=omp_get_thread_num();
                printf("(%d) gets iteration %d\n",id,i);
        }
    }

    return 0;
}
```

2.schedule.c

1. Which iterations of the loops are executed by each thread for each schedule kind?

Static:

Each thread executes the proportional number of iterations in order. This means that thread 0 executes the first 4 iterations, thread 1 the next 4 and thread 2 the last 4.

Static, 2:

Each thread executes 2 iterations, in order, and then the next thread executes the next 2. When we run out of threads but still have iterations the first thread again executes the correspondent 2 iterations. Therefore thread (0) will execute iterations 0,1,6,7; (1) -> 2,3,8,9; (2) -> 4,5,10,11.

Dynamic, 2:

It varies between executions, but we can assure the following: Each thread will executed in order chunks of 2 iterations. This means that any thread can execute a given number of pairs (i, i+1) iterations, where 0<=i<=n and i is even.

Guided, 2:

It also varies between executions, but has the following structure: One thread will be assigned 7 iterations, then another one 3 and another on 2. This is because the guided schedule will reduce the size of the chunks as they are assigned but they will always be at least of size N.

3.nowait.c

1. How does the sequence of printf change if the nowait clause is removed from the first for directive?

After removing the nowait clause we get the sequence of all of loop1 messages first, and after that, all the messages of loop2, without any order between them.

2. If the nowait clause is removed in the second for directive, will you observe any difference?

No, because each thread which is going to the second loop does not have to wait other threads to execute the loop task because there is no code to execute after the second loop.

4.collapse.c

1. Which iterations of the loop are executed by each thread when the collapse clause is used?

Thread 0 gets the first 4 iterations, that is from 0,0 to 0,3. Then each thread gets the following 3 iterations. That means that thread 1 will get from 0,4 to 1,1 and so on.

2. Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?.

No, it is not correct. We can add the clause **ordered** to make it correct.

5.ordered.c

1. Can you explain the order in which printf appear?

We cannot determine the order of the “Before ordered” as they appear randomly. Nevertheless, we know the sequence of the “Inside ordered” messages will appear in increasing order due to the “ordered” clause used before the corresponding printf.

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the ordered part of the loop body?

By changing the schedule chunk size to two.

6.doacross.c

1. In which order are the ”Outside” and ”Inside” messages printed?

The “Outside” message is printed in no particular order. The “inside” message is printed in order of i. This is caused by the #pragma omp ordered depend (sink: i-2).

2. In which order are the iterations in the second loop nest executed?

We start with 1 1, being the only one with no active dependencies. Once this iteration is executed, 1 2 and 2 1 can be executed (in the sequence defined by the inner loop). This two iterations at the same time let two iterations each be executed. As the iteration 1 2 was the first of the second round to be executed, this means that it’s “children” (1 3 and 2 2) will be executed before the “children” of 2 1. We could make a comparison with a queue; each time an iteration is completed, the iterations that depended of it are added to the queue, with their respective order given by the inner loop.

3. What would happen if you remove the invocation of sleep(1). Execute several times to answer in the general case.

Then the order of execution of the different iterations is no longer defined like we said and is only determined by how fast each thread is. This means that we can no longer say in what order the iterations are going to be executed.

C) Tasks

1.serial.c

1. Is the code printing what you expect? Is it executing in parallel?

The computation of the fibonacci numbers is correct, but it is done only by the thread 0, i.e. there is no parallelism.

2.parallel.c

1. Is the code printing what you expect? What is wrong with it?

No, the numbers computed do not correspond to the fibonacci sequence.

2. Which directive should be added to make its execution correct?

If we add *#pragma omp single* after the parallel directive we avoid having threads influence each other’s work and therefore get the correct result.

3. What would happen if the firstprivate clause is removed from the task directive? And if the first private clause is ALSO removed from the parallel directive? Why are they Redundant?

If we remove only the firstprivate clause in the task directive, nothing happens, the code still works as expected. If we remove the other one, the code works just as fine as well.

Their redundancy is explained by the fact that the variables of the data-sharing tasks are, by default, already first-private.

4. Why does the program break when variable p is not firstprivate to the task?

Because if the variable p is not declared as firstprivate all threads are accessing it at the same time so finally some tread will try to get to some element of p, which will be already accessed by other thread and that will become a break in the program

5. Why the firstprivate clause was not needed in 1.serial.c?

Because in that case we only had a thread as a consequence we do not have any problem with other thread accesses to the same variables.

3.taskloop.c

1. Execute the program several times and make sure you are able to explain when each thread in the threads team is actually contributing to the execution of work (tasks) generated in the Taskloop.

We start with 4 threads, of which only one will create the tasks T1 and T2, due to the #pragma omp single. The remaining threads will execute those tasks; T1 and T2. The latter, T2, creates two additional tasks, T3 and T4. Finally, the thread which created the two original tasks “wakes up” and executes the last tasks as well.

Part II: Parallelization overheads

1. Which is the order of magnitude for the overhead associated with a parallel region (fork and join) in OpenMP? Is it constant? Reason the answer based on the results reported by the pi_omp_parallel.c code.

The overhead varies between a tenth of a microsecond and five microseconds, and it is not constant. This is because it depends on a number of things, such as the reservation and initialization of dynamic memory (which can vary) and the number of threads. In fact, we can observe that the more threads we have, the longer the overhead is.

```
par2203@boada-1:~/lab2/overheads$ cat pi_omp_parallel_1_24.txt
All overheads expressed in microseconds
Nthr    Overhead        Overhead per thread
2       1.8499          0.9250
3       1.7596          0.5865
4       2.2284          0.5571
5       2.7312          0.5462
6       2.8150          0.4692
7       2.7619          0.3946
8       3.1469          0.3934
9       3.3766          0.3752
10      3.3764          0.3376
11      3.6310          0.3301
12      3.4193          0.2849
13      3.8733          0.2979
14      4.0839          0.2917
15      4.1308          0.2754
16      4.5566          0.2848
17      4.5499          0.2676
18      4.1641          0.2313
19      4.7614          0.2506
20      4.3259          0.2163
21      4.7958          0.2284
22      4.9969          0.2271
23      5.2523          0.2284
24      5.3231          0.2218
```

2. Which is the order of magnitude for the overhead associated with the creation of a task and its synchronization at taskwait in OpenMP? Is it constant? Reason the answer based on the results reported by the pi_omp_tasks.c code.

In this case the order of magnitude for the overhead per task is a tenth of a microsecond and is indeed constant. Therefore, the overhead depends on the number of tasks, and in fact is directly proportional to it. This makes sense because each time we add a task, we need to do some work to add it to the list of remaining tasks.

```
par2203@boada-1:~/lab2/overheads$ cat pi_omp_tasks_10_1.txt
All overheads expressed in microseconds
Ntasks  Overhead per task
2       0.1256
4       0.1182
6       0.1166
8       0.1150
10      0.1224
12      0.1254
14      0.1234
16      0.1226
18      0.1240
20      0.1229
22      0.1221
24      0.1213
26      0.1206
28      0.1195
30      0.1198
32      0.1194
34      0.1198
36      0.1195
38      0.1191
40      0.1190
42      0.1186
44      0.1185
46      0.1183
48      0.1182
50      0.1185
52      0.1183
54      0.1182
56      0.1175
58      0.1177
60      0.1176
62      0.1175
64      0.1174
```

3. Based on the results reported by the pi_omp_taskloop.c code, If you have to generate tasks out of a loop, what seems to be better: to use task or taskloop? Try to reason the answer.

In this version of the code, the function “difference” calculates the difference between the execution time using task or taskloop. The results printed in the column named “overhead per task” is this difference. We can infer the following from the results:

The result is positive with a small number of tasks, and it decreases as we add more tasks. This means that with a small number of tasks is better to use taskloop (as we said, the result is the difference between task and taskloop, being positive means the execution time of the latter is better).

As we go on adding tasks, we pass a certain threshold (in this case 18 tasks) where the result becomes negative. Therefore we can conclude that for a greater number of tasks, it is better to use task instead of taskloop.

```
par2203@boada-1:~/lab2/overheads$ cat pi_omp_taskloop_10_1.txt
All overheads expressed in microseconds
Ntasks  Overhead per task
2       0.0282
4       0.0107
6       0.0058
8       0.0019
10      0.0103
12      0.0072
14      0.0054
16      0.0038
18      0.0002
20      -0.0009
22      -0.0017
24      -0.0024
26      -0.0030
28      -0.0030
30      -0.0038
32      -0.0033
34      -0.0045
36      -0.0056
38      -0.0058
40      -0.0060
42      -0.0062
44      -0.0059
46      -0.0068
48      -0.0068
50      -0.0073
52      -0.0077
54      -0.0076
56      -0.0082
58      -0.0082
60      -0.0083
62      -0.0080
64      -0.0071
```

4. Which is the order of magnitude for the overhead associated with the execution of critical regions in OpenMP? How is this overhead decomposed? How and why does the overhead associated with critical increase with the number of processors? Identify at least three reasons that justify the observed performance degradation. Base your answers on the execution times reported by the pi_omp.c and pi_omp_critical.c programs and their Paraver execution traces.

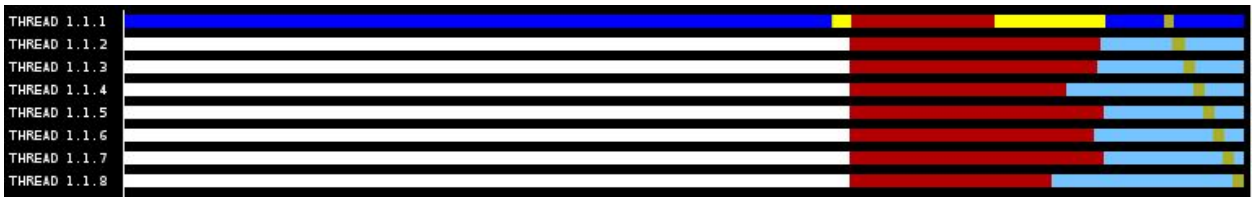
The results we obtained are the following:

	One thread	**Eight threads**
pi_omp_critical	1,794936 s	46,324014 s
pi_omp	0.790860 s	0.135238 s

The first version of the code, critical, defines too many critical regions; one for each iteration, but when executed with one thread it does not have a massive impact because there is no need for synchronization between threads, only defining the critical regions. When executed with 8 threads however, we see an enormous degradation of performance caused by all the synchronization between all threads an the excessive amount of critical regions. In comparison, the second version of the code, which defines only as many critical region as threads, performs much better.

If we look at the timelines generated by paraver, which compare the execution of pi_omp_critical when using either 1 or 8 threads we can confirm the previous hypothesis. The red regions define the time lost in synchronization. When using 8 threads, we see that the time spent in synchronization is severely increased. Also, given that no two threads can execute the same critical region, the other threads spent no time doing any actual work. They stay idle. The conclusion is that we should be careful with how many critical regions we define at the same time and that this number should not exceed the number of threads we will be using.

Paraver trace with one thread and 100.000 iterations



Paraver trace with eight threads and 100.000 iterations

5. Which is the order of magnitude for the overhead associated with the execution of atomic memory accesses in OpenMP? How and why does the overhead associated with atomic increase with the number of processors? Reason the answers based on the execution times reported by the pi_omp.c and pi_omp_atomic.c programs.

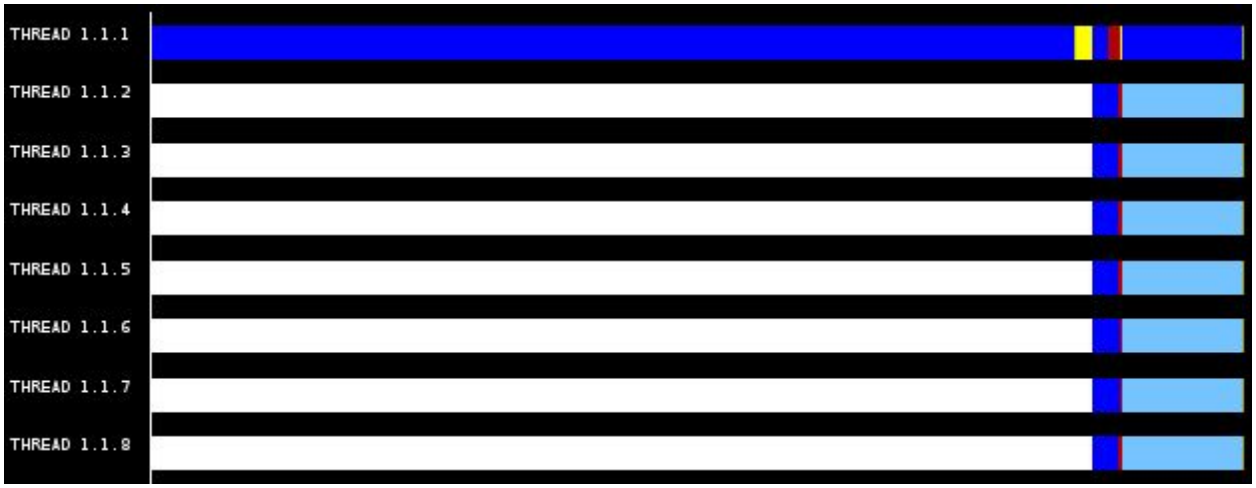
In the first execution, using only one thread, we see that there is almost no overhead, the CPU is running almost a 100% of the time. We executed with 100.000.000 iterations, but the total time spent with synchronization is only 4,357 ns, so we could say that there is almost no overhead for defining atomic regions when executing with only one thread.

With the second execution however, when using eight threads, we see that there is some time spent in synchronization. We executed with 100.000.000 iterations. The total time spent in synchronization is 470,478,680 ns so the order of magnitude for defining atomic regions when using various threads is measured in **nanoseconds**.

Comparing the execution time f pi_omp_atomic with pi_omp we can see that independently of the number of threads, the execution time is always better in the second one. However, when using only one thread is very similar. In the other hand, when using eight threads, we observe a great degradation in the first version of the code.

	One thread	**Eight threads**
pi_omp_atomic	1,790,258,618 ns	6,845,589,885 ns
pi_omp	1,057,973,393 ns	364,641,959 ns

Paraver trace of *pi_omp_atomic* execution with one thread and 100.000.000 iterations



Paraver trace of *pi_omp_atomic* execution with eight threads and 100.000.000 iterations

6. In the presence of false sharing (as it happens in pi_omp_sumvector.c), which is the additional average time for each individual access to memory that you observe? What is causing this increase in the memory access time? Reason the answers based on the execution times reported by the pi_omp_sumvector.c and pi_omp_padding.c programs. Explain how padding is done in pi_omp_padding.c.

In our execution of the code “pi_omp_sumvector” we put as a input 100.000.000 iterations and 8 threads. That means that there are 100.000.000 individual accesses to memory. Comparing it’s execution with the execution of the code “padding”, which solves the problem causing this increase in memory access time we can determine the additional average time for each individual access to memory. To do so, we used the tools provided by paraver which present the data in an organized an intelligible way.

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	769,975,526 ns	-	24,066,551 ns	12,615 ns	2,473 ns
THREAD 1.1.2	493,193,345 ns	227,906,061 ns	-	12,967 ns	-
THREAD 1.1.3	492,463,529 ns	227,907,596 ns	-	8,998 ns	-
THREAD 1.1.4	545,845,627 ns	227,906,904 ns	-	6,518 ns	-
THREAD 1.1.5	537,709,489 ns	227,904,651 ns	-	6,118 ns	-
THREAD 1.1.6	537,812,692 ns	227,793,700 ns	-	6,258 ns	-
THREAD 1.1.7	415,528,484 ns	227,806,870 ns	-	6,063 ns	-
THREAD 1.1.8	542,847,956 ns	227,907,369 ns	-	5,950 ns	-
THREAD 1.1.9	-	794,056,790 ns	-	375 ns	-
THREAD 1.1.10	-	794,051,765 ns	-	5,400 ns	-
THREAD 1.1.11	-	794,056,810 ns	-	355 ns	-
THREAD 1.1.12	-	794,056,957 ns	-	208 ns	-
THREAD 1.1.13	-	794,056,903 ns	-	262 ns	-
THREAD 1.1.14	-	794,056,905 ns	-	260 ns	-
THREAD 1.1.15	-	794,054,973 ns	-	2,192 ns	-
THREAD 1.1.16	-	794,056,922 ns	-	243 ns	-
THREAD 1.1.17	-	794,056,972 ns	-	193 ns	-
THREAD 1.1.18	-	794,056,893 ns	-	272 ns	-
THREAD 1.1.19	-	794,055,080 ns	-	2,085 ns	-
THREAD 1.1.20	-	794,056,925 ns	-	240 ns	-
THREAD 1.1.21	-	794,056,975 ns	-	190 ns	-
THREAD 1.1.22	-	794,056,912 ns	-	253 ns	-
THREAD 1.1.23	-	794,056,978 ns	-	187 ns	-
THREAD 1.1.24	-	794,055,003 ns	-	2,162 ns	-
Total	4,335,376,648 ns	14,300,032,914 ns	24,066,551 ns	80,364 ns	2,473 ns
Average	541,922,081 ns	621,740,561.48 ns	24,066,551 ns	3,348.50 ns	2,473 ns
Maximum	769,975,526 ns	794,056,978 ns	24,066,551 ns	12,967 ns	2,473 ns
Minimum	415,528,484 ns	227,793,700 ns	24,066,551 ns	187 ns	2,473 ns
StDev	95,430,103.80 ns	260,516,812.57 ns	0 ns	3,948.53 ns	0 ns
Avg/Max	0.70	0.78	1	0.26	1

Paraver analysis of *pi_omp_sumvector* execution with eight threads and 100.000.000 iterations

	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	372,441,315 ns	-	88,606,409 ns	13,030 ns	2,553 ns
THREAD 1.1.2	142,865,237 ns	252,745,771 ns	-	11,531 ns	-
THREAD 1.1.3	152,596,093 ns	252,823,064 ns	-	8,103 ns	-
THREAD 1.1.4	187,046,017 ns	252,821,824 ns	-	5,723 ns	-
THREAD 1.1.5	137,075,554 ns	252,825,829 ns	-	5,352 ns	-
THREAD 1.1.6	162,786,494 ns	252,747,421 ns	-	5,520 ns	-
THREAD 1.1.7	109,153,738 ns	252,727,275 ns	-	5,405 ns	-
THREAD 1.1.8	130,617,932 ns	256,668,814 ns	-	5,502 ns	-
THREAD 1.1.9	-	461,062,989 ns	-	318 ns	-
THREAD 1.1.10	-	461,055,602 ns	-	7,705 ns	-
THREAD 1.1.11	-	461,062,970 ns	-	337 ns	-
THREAD 1.1.12	-	461,063,089 ns	-	218 ns	-
THREAD 1.1.13	-	461,063,034 ns	-	273 ns	-
THREAD 1.1.14	-	461,063,037 ns	-	270 ns	-
THREAD 1.1.15	-	461,061,074 ns	-	2,233 ns	-
THREAD 1.1.16	-	461,063,105 ns	-	202 ns	-
THREAD 1.1.17	-	461,063,115 ns	-	192 ns	-
THREAD 1.1.18	-	461,063,042 ns	-	265 ns	-
THREAD 1.1.19	-	461,061,330 ns	-	1,977 ns	-
THREAD 1.1.20	-	461,063,110 ns	-	197 ns	-
THREAD 1.1.21	-	461,063,114 ns	-	193 ns	-
THREAD 1.1.22	-	461,063,117 ns	-	190 ns	-
THREAD 1.1.23	-	461,063,040 ns	-	267 ns	-
THREAD 1.1.24	-	461,061,239 ns	-	2,068 ns	-
Total	1,394,582,380 ns	9,150,356,005 ns	88,606,409 ns	77,071 ns	2,553 ns
Average	174,322,797.50 ns	397,841,565.43 ns	88,606,409 ns	3,211.29 ns	2,553 ns
Maximum	372,441,315 ns	461,063,117 ns	88,606,409 ns	13,030 ns	2,553 ns
Minimum	109,153,738 ns	252,727,275 ns	88,606,409 ns	190 ns	2,553 ns
StDev	77,903,802.96 ns	95,583,639.35 ns	0 ns	3,794.40 ns	0 ns
Avg/Max	0.47	0.86	1	0.25	1

Paraver analysis of *pi_omp_padding* execution with eight threads and 100.000.000 iterations

We can calculate the additional time dedicated to accessing memory in the following way:

794,056,978 ns - 461,063,117 ns = 332,993,861 ns.

This means that the average time spent in accessing memory for each individual access, considering we realized 100,000,000 iterations is 3,3 ns.

The cause for such difference is the way we access cache memory. The code in “pi_omp_padding” solves the original issue by transforming the *vectorsum* array into an array of arrays of size proportional to the cache size and therefore avoids a false sharing situation.