Universitat Oberta de Catalunya

High Performance Computing - CET1

Introduction to OpenMP

Juan José Rodríguez Aldavero

May 18, 2020

Contents

1 Questions **ii**
1.a Executing OpenMP ii
1.b Data paralellism ii
1.c OpenMP scheduling policies iii
1.d Profiling applications vii
1.e Parallelization problem with OpenMP xiv

1 Questions

1.a Executing OpenMP

1. **How many OpenMP threads you would use in the UOC cluster? Why?**
   In the cluster used for this exercise, there are 10 nodes. In each node, we check that there is a CPU socket with 4 cores, and each core supports the execution of one thread. This means that we can make parallel processes with four threads in a single node, and these should be assigned to OpenMP since using less threads would decrease the performance of the execution.

1.b Data paralellism

2. **How you would implement a program that does the sum of the different elements stored in a vector a using the reduction clause?**
   To implement parallelization of for loops in OpenMP that perform operations on a shared variable, it is necessary to consider a reduction procedure. This is because otherwise each of the threads would update the shared variable independently without taking into account the updates of the rest of the threads (data race). To avoid this, we use the reduction clause incorporated in OpenMP. This makes each thread have a local copy of the variable, and these are joined once the process is finished. The syntax is as follows:

```
#pragma omp parallel for reduction (o: x)
```

   where *o* is the operation you want to perform, and the rest of the statement is a reference to the compiler to perform the process in parallel.

   In the particular case of adding up all the elements contained in a vector a, the operation *o* would be the sum + and the implementation would be as follows

```
sum = 0;
n = 100;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < n; i++) {
  sum += a[i]
}
```

   In this case, each of the threads would have a local copy of the sum variable and at the end of the process all these local variables would be joined in the final variable.

1.c OpenMP scheduling policies

In the following cases we proceed to parallelize the mm.c and mm2.c matrix multiplication codes using OpenMP.

3. **Provide the parallel version of mm.c and mm2.c codes using the different scheduling policies.**
   When parallelizing a for-loop, it is possible to determine the type of scheduling with which it is performed, that is, how the iterations are distributed among the different threads.

   First, if we do not explicitly define the planning type OpenMP uses the default type. The syntax is

```
#pragma omp parallel for
for (...)
{ ... }
```

   On the other hand, we can explicitly define the type of scheduling using the syntax

```
#pragma omp parallel for schedule(scheduling-type, chunk-size)
for (...)
{ ... }
```

   You can choose from 5 different types of planning, which are static, dynamic, guided, auto and runtime. Each type is explained briefly below.

   - **Static**
     This type of scheduling divides the iterations into blocks of size *chunk-size* and distributes them in the different threads in a circular fashion (Round-Robin). This scheduling is efficient as long as all iterations have the same computational cost, otherwise there will be a set of threads that will take longer than the rest.
   - **Dynamic**
     This type of scheduling divides the iterations in blocks of *chunk-size* but does not distribute them in advance for each thread, but rather the threads request the blocks as they finish the execution of their previous block and as long as there are blocks available. This is efficient when the iterations have different computational costs (they are badly balanced) but has the disadvantage of having a higher computational cost (overhead).
   - **Guided**
     This scheduling is similar to the Dynamic one because each thread requests blocks of iterations as the execution of the previous block is completed. The difference is that the size of each block is not fixed in advance, but is determined from the number of pending iterations divided by the number of available threads. The parameter *chunk-size* determines the minimum size of each block.
   - **Auto**
     This category delegates the decision of the planning type to the compiler itself.
   - **Runtime**
     In this category, the scheduling type is determined by reading the control variable *run-sched-var* in the runtime itself. This can be specified via the syntax:

```
$ export OMP_SCHEDULE=scheduling-type
```

Once we have seen this, we proceed to parallelize the mm1.c and mm2.c codes. The modification is very simple since we only have to import OpenMP and define the parallelization region by calling the pragma. This allows the region to be defined as parallel when the program is compiled, as long as the computer where it is executed has this capability. Otherwise, the compiler would simply ignore the pragma and the program would run sequentially.

The final code once the changes are made is

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <memory.h>
#include <malloc.h>
#include <omp.h>

int main(int argc, char **argv) {
  int SIZE = atoi(argv[1]);
  float matrixa[SIZE][SIZE], matrixb[SIZE][SIZE],
      mresult[SIZE][SIZE];
  int i,j,k;

  /* Initialize the Matrix arrays */
  for ( i=0; i<SIZE*SIZE; i++ ){
    mresult[0][i] = 0.0;
    matrixa[0][i] = matrixb[0][i] = rand()*(float)1.1;
  }

  /* Initialize the parallel region */
  #pragma omp parallel for private(i, j, k) schedule(static)

  /* Matrix-Matrix multiply */
  for (i=0;i<SIZE;i++)
    for(j=0;j<SIZE;j++)
      for(k=0;k<SIZE;k++)
        mresult[i][j]=mresult[i][j] + matrixa[i][k]*matrixb[k][j];

  exit(0);
}
```

Listing 1.1: *Parallelization of mm1.c*

In the case of mm2.c we would simply have to change the order of the *i* and *k* loops, while to execute the rest of the scheduling categories we would have to change the *scheduling-type* variable. It is important to note how we have defined the variables *i, j, k* as private, otherwise we would enter a data race and would not get any improvement in performance.

4. **Provide the performance evaluation described above. Elaborate the results obtained.** Once the mm.c and mm2.c files have been modified for parallel execution using the *static*,

dynamic and *guided* schedules for a different number of blocks, we proceed to show their performance in the following graphics. We start by looking at the evolution in mm1.c times for the sequential and parallel version.

Size	**Secuential**	**Static**	**Dynamic**	**Guided**
100	0,002	0,003	0,003	0,003
300	0,357	0,098	0,104	0,102
500	1,814	0,438	0,455	0,445
1000	14,386	3,870	3,853	3,866
2000	158,128	31,181	31,168	31,166

Table 1.1: *Execution times (s) for the different scheduling options for mm1.c*

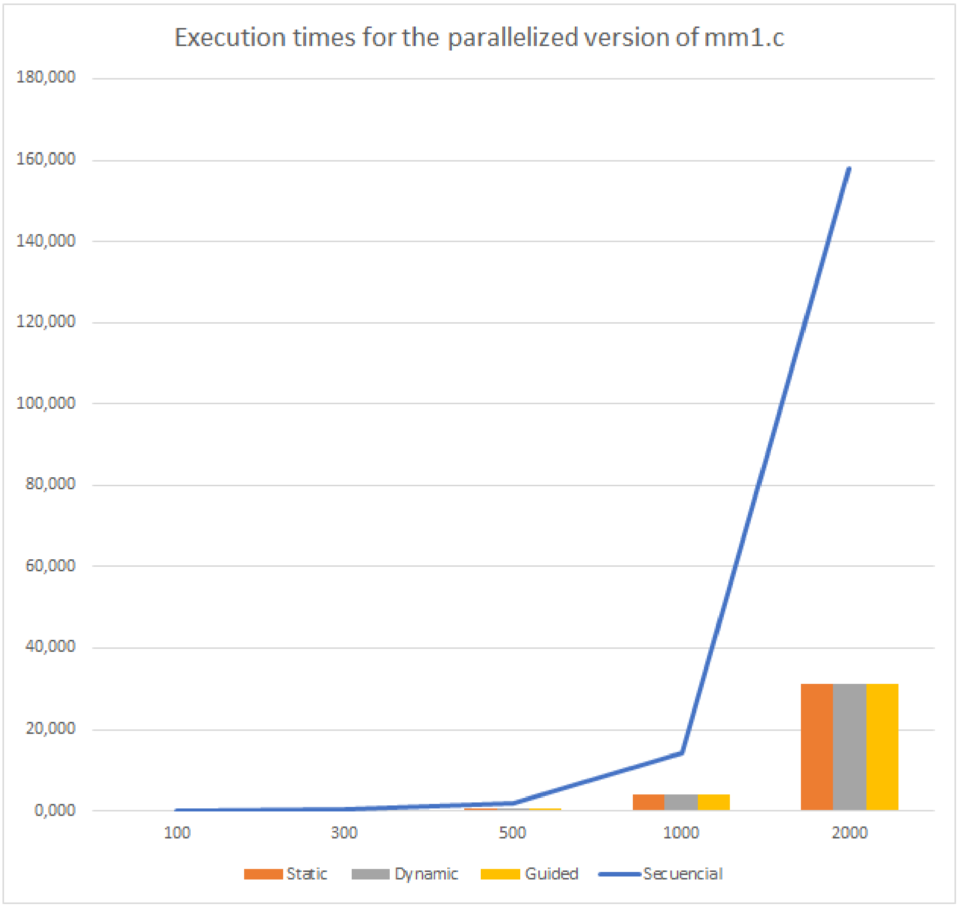

Figure 1.1: *Execution times for mm1.c*

We see that for small sizes of the matrix, the difference between parallel and sequential execution is very small, but it is amplified as the size of the data increases. This is a very typical behavior in parallelization processes. Again, if we look at the version with the loops changed in order mm2.c:

Size	Secuential	Static	Dynamic	Guided
100	0,002	0,003	0,003	0,003
300	0,392	0,154	0,131	0,143
500	1,949	0,559	0,818	0,758
1000	18,185	6,170	6,117	6,363
2000	131,136	52,045	52,132	53,137

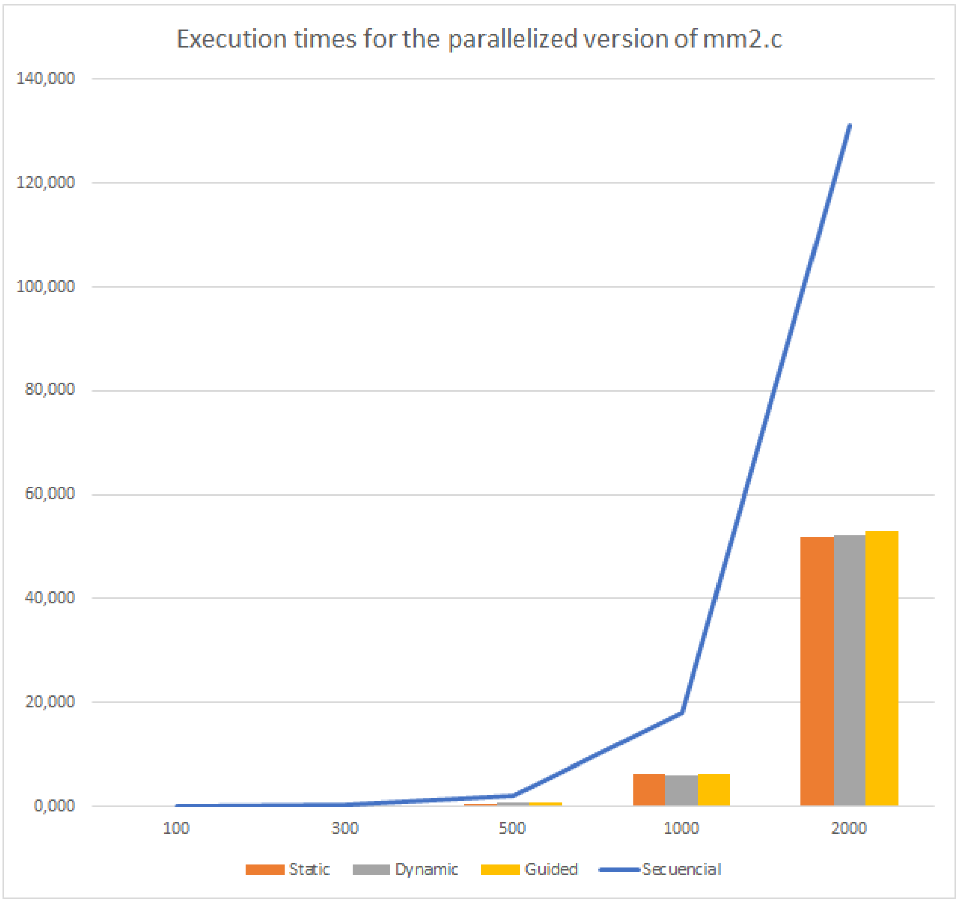

Figure 1.2: Execution times for mm2.c

In this case the behaviour is similar although the sequential version is faster and the parallel version is slower for size 2000. We compare the speeds between mm1.c and mm2.c in the following graph:

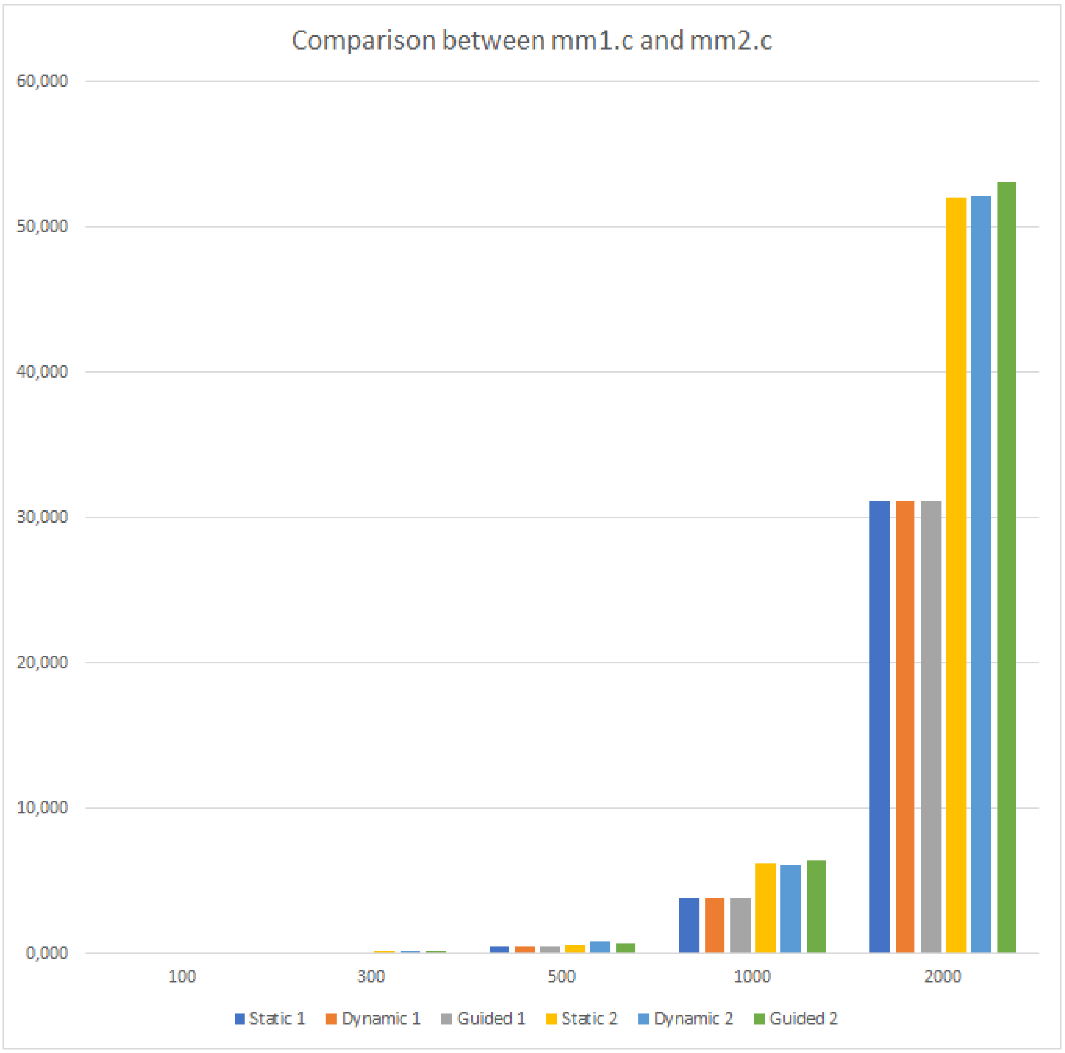

Figure 1.3: *Comparison between the execution times for each scheduling category*

We can see how the mm2.c version is slower than the original one when doing the parallelization. This is due to the fact that changing the loop orders leads to inefficiencies in memory management.

1.d Profiling applications

We proceed with the exercise using the PAPI tool to make measurements of the internal counters of the cluster. To do this, we start by compiling and executing the code provided *counter.c* and *counter2.c* for 1000×1000 arrays and studying the execution time, the number of floating point operations and instructions. The results are as follows:

	flops1.c	**flops2.c**	**Difference**
Time (s)	11,18	13,06	14,38%
MFLOPs	1794,04	1538,07	16,64%
Instructions ($\times 10^6$)	20009,13	20033,83	0,12%

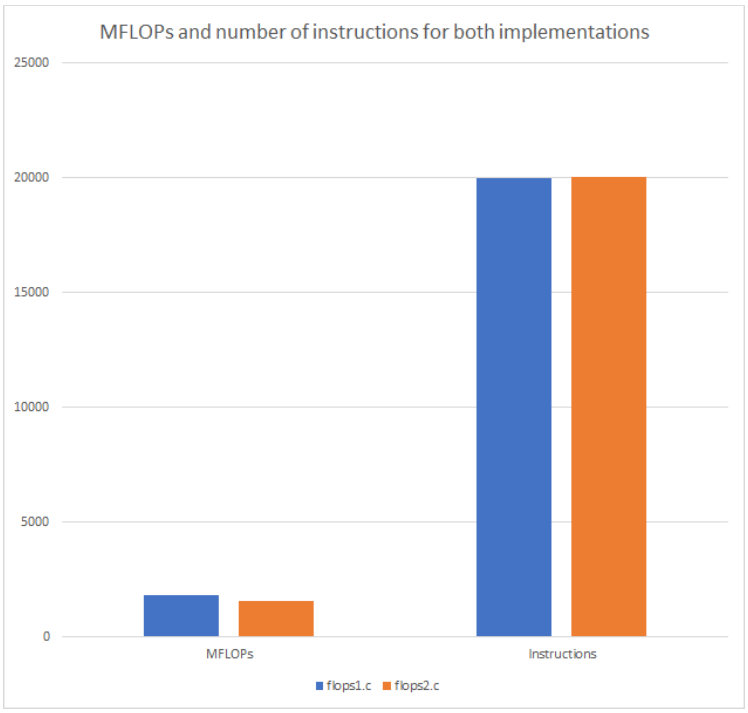

Figure 1.4: MFLOPs and number of instructions for the secuential implementation of flops.c

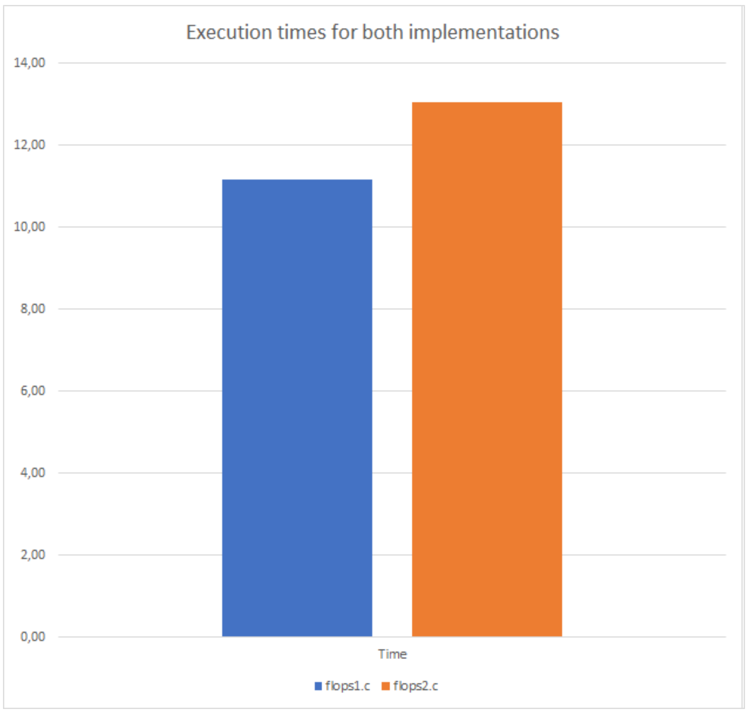

Figure 1.5: Execution times for the secuential implementation of flops.c

We see how the original version has a significantly higher number of MFLOPs and instructions than the version with the changed order loops. In addition, the execution time is shorter as we already saw in the previous version.

5. **Why do you think it is this difference in execution time and MFLOPs?**
   The differences found in runtime and MFLOPs are due to the in-memory optimization of each of the two implementations. In particular, the change of the loop of the variable *i* and the variable *k* has important consequences in this sense, as we explained in the previous version.
6. **What hardware counters would you use to study the differences between the two implementations? Why?**
   Executing the code

```
/export/apps/papi/bin/papi_avail
```

   we can see all the counters available in the software. Given the characteristics of the problem, we decided to use the cache reading counters for both types of cache (data and instruction) and for the three levels L1 L2 L3. These indicators are as follows: *PAPI_L2_DCR, PAPI_L3_DCR, PAPI_L1_ICR, PAPI_L2_ICR , PAPI_L3_ICR* as the read indicator for the data cache L1 is not available.

   We use these counters because we can get information about the in-memory structure of the code based on the amount of information they read from the cache.
7. **Provide the code where you use hardware counters to quantify the differences between the two examples. Provide the results obtained.**
   First, we proceed to execute the original code with the counters *PAPI_TOT_INS* and *PAPI_FP_OPS* and compare the results with a graph. The result is:

	counters1.c	**counters2.c**	**Difference**
MFLOPs	2003,11	2005,59	0,124%
Instructions ($\times 10^6$)	33073,99	33073,99	0,001%

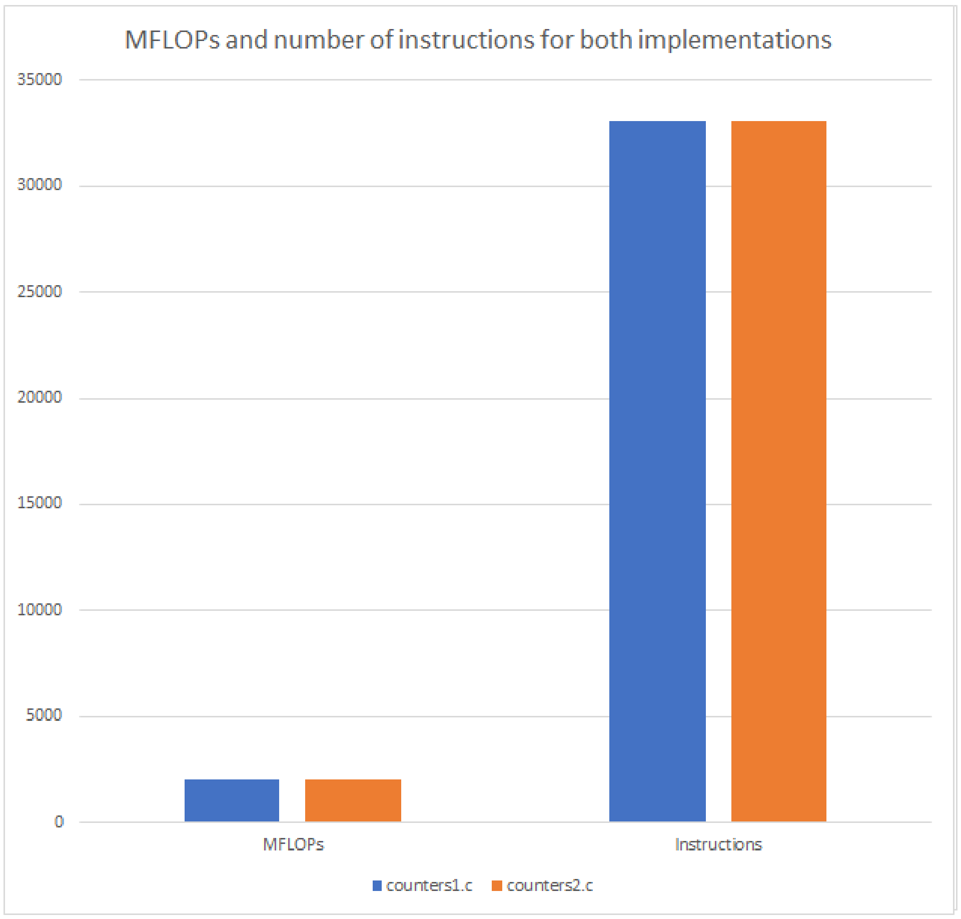

Figure 1.6: *MFLOPs and number of instructions for the secuential implementation of counters.c*

To implement these changes, it is enough to modify the code provided by introducing the 5 counters in the *events* array. The result is the following:

Reading operations ($\times 10^6$)	**counters1.c**	**counters2.c**	**Difference**
PAPI_L2_DCR	1873,09	3002,36	37,61%
PAPI_L3_DCR	73,63	295,09	75,05%
PAPI_L1_ICR	10117,99	10145,46	0,27%
PAPI_L2_ICR	0,01	0,01	4,14%
PAPI_L3_ICR	0,01	0,01	33,86%

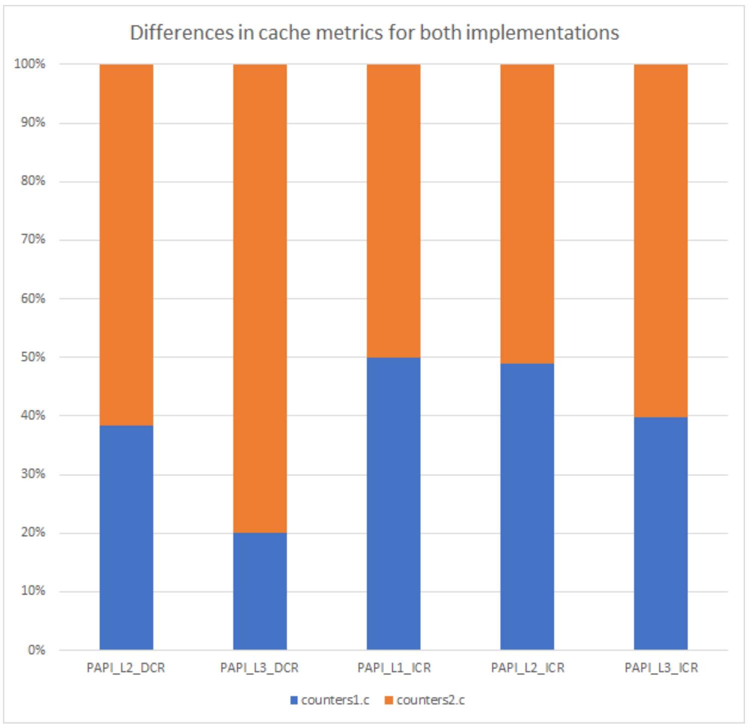

Figure 1.7: Comparison between the secuential implementations of the counters.c code

We see that case 1 has a lower number of cache readings than case 2. This is because it is more optimized and therefore the calculation requires fewer steps in memory.

8. **Provide parallel versions (OpenMP) and the results obtained using PAPI for the two examples.**
We repeat the previous problem but this time paralleling the operation with OpenMP. We use the same counters as in the previous sequential case. The code used is the following:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <memory.h>
#include <malloc.h>
#include <papi.h>
#include <omp.h>

#define SIZE 1000

int main(int argc, char **argv) {

float matrixa[SIZE][SIZE], matrixb[SIZE][SIZE], mresult[SIZE][SIZE];
int i,j,k;
int events[2] = {PAPI_TOT_INS, PAPI_FP_OPS}, ret;
long long values[2];

if (PAPI_num_counters() < 2) {
```

```
fprintf(stderr, "No hardware counters here, or PAPI not
    supported.\n");
exit(1);
}

if ((ret = PAPI_start_counters(events, 2)) != PAPI_OK) {
fprintf(stderr, "PAPI failed to start counters: %s\n",
    PAPI_strerror(ret));
exit(1);
}

/* Initialize the Matrix arrays */
for ( i=0; i<SIZE*SIZE; i++ ){
mresult[0][i] = 0.0;
matrixa[0][i] = matrixb[0][i] = rand()*(float)1.1; }

/* Initialize parallel region using default scheduling */
#pragma omp parallel for private(i, j, k) schedule(static)

/* Matrix-Matrix multiply */
for (i=0;i<SIZE;i++)
for(j=0;j<SIZE;j++)
for(k=0;k<SIZE;k++)
mresult[i][j]=mresult[i][j] + matrixa[i][k]*matrixb[k][j];

if ((ret = PAPI_read_counters(values, 2)) != PAPI_OK) {
fprintf(stderr, "PAPI failed to read counters: %s\n",
    PAPI_strerror(ret));
exit(1);
}

printf("A = %lld\n",values[0]);
printf("B %lld\n", values[1]);

exit(0);
}
```

The results are as follows:

Operations ($\times 10^6$)	**Parallel counters1.c**	**Parallel counters2.c**	**Difference**
MFLOPs	502,78	503,18	0,08%
Instructions	10597,09	10692,44	0,89%
PAPI_L2_DCR	476,75	841,72	43,36%
PAPI_L3_DCR	18,75	121,71	84,60%
PAPI_L1_ICR	3301,44	3310,77	0,28%
PAPI_L2_ICR	0,01	0,01	5,16%
PAPI_L3_ICR	0,01	0,01	0,44%

We can see how there is a big difference in the loading of L2 and L3 data cache between both implementations, which could be the source of the inefficiency.

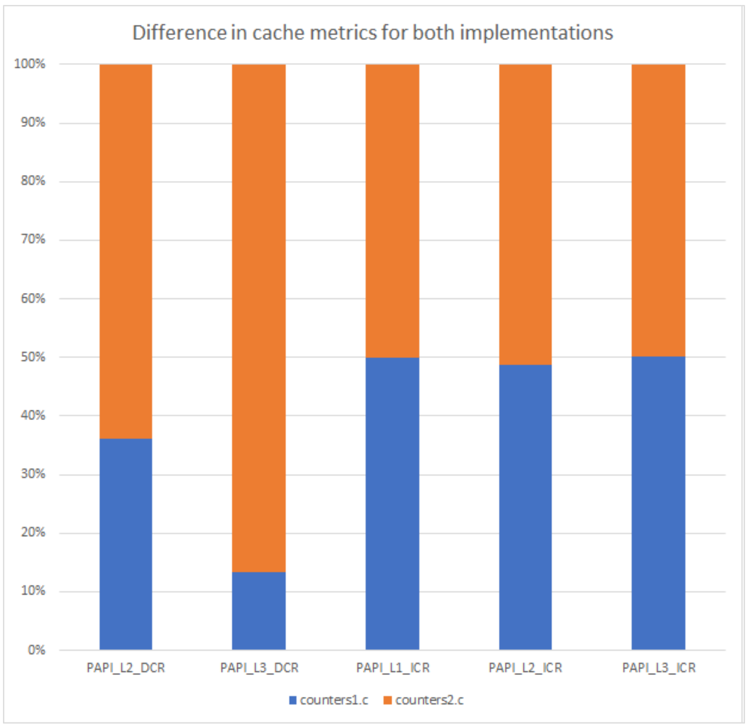

Figure 1.8: Comparison between the parallel implementations of the counters.c code

In the following graphs we briefly compare the measurements between the sequential case and the parallel case in order to better observe the differences:

	Secuential counters1.c	**Parallel counters1_p.c**	**Difference**
MFLOPs	2003,11	502,78	75%
Instructions	33073,99	10597,09	68%
PAPI_L2_DCR	1873,09	476,75	75%
PAPI_L3_DCR	73,63	18,75	75%
PAPI_L1_ICR	10117,99	3301,44	67%
PAPI_L2_ICR	0,01	0,01	44%
PAPI_L3_ICR	0,01	0,01	18%

Table 1.2: Comparison between the parallel implementations of the counters.c code

	Secuential counters2_s.c	**Parallel counters2_p.c**	**Difference**
MFLOPs	2005,59	503,18	75%
Instructions	33073,99	10692,44	68%
PAPI_L2_DCR	3002,36	841,72	72%
PAPI_L3_DCR	295,09	121,71	59%
PAPI_L1_ICR	10145,46	3310,77	67%
PAPI_L2_ICR	0,01	0,01	46%
PAPI_L3_ICR	0,01	0,01	23%

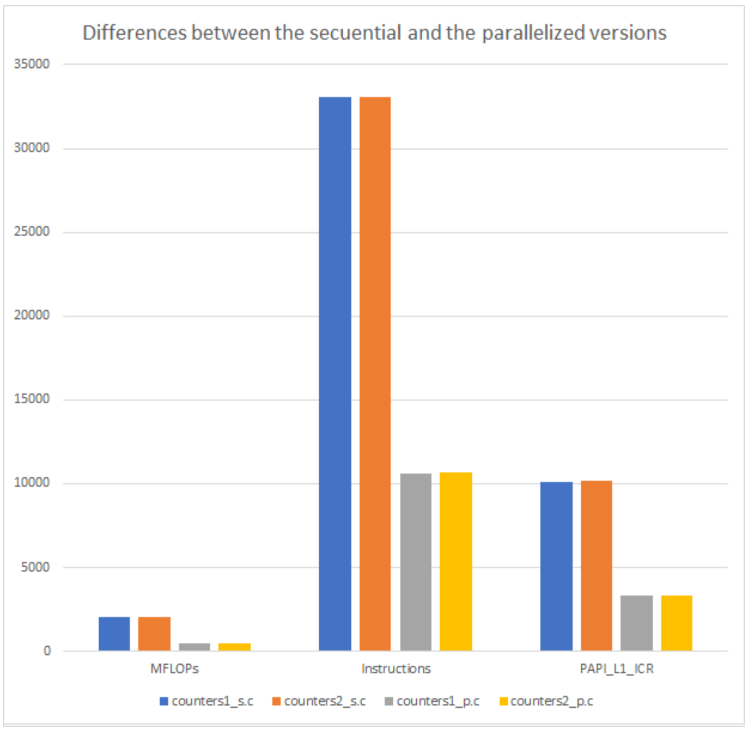

Figure 1.9: Comparison between the secuential and parallel implementations of the counters.c code

We see how the sequential case presents a greater amount of instructions and readings from the instruction cache, close to proportional to the number of threads used (4). This is because the execution load is shared between the 4 threads.

1.e Parallelization problem with OpenMP

9. **Provide a parallel implementation of p2.c using OpenMP.** We proceed to parallelize the code by providing for the application of the procedures seen throughout the exercise. As the variable *sum* is updated iteratively, we see that it has to be expressed as a reduction. At the same time, it is necessary to make the variables *x* and *y* private as each thread must have a copy.

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

int main(int argc, char **argv) {

  double input=100000;
  double output;
  int i;
  double m;
  double sum;
  double x;

  m = 1.0 / ( double ) ( 2 * input );

  sum = 0.0;
```

```
    #pragma omp parallel for private(i, x) reduction(+: sum)
    for ( i = 1; i <= (int)input; i++ ) {
      x = m * ( double ) ( 2 * i - 1 );
      sum += 1.5 / ( 1.0 + x * x );
    }

    output = 4.0 * sum / ( double ) ( input );

    return(0);
  }
```