

LAB 5: Parallel Data Decomposition Implementation and Analysis

Arnau Garcia Gonzalez

Jan Santos Bastida

**Table of contents**

[**Geometric Data Decomposition Strategies 3**](#_f0dkne10w1n)

[1D Block Geometric Data Decomposition by columns 3](#_rs8ej183492z)

[Code 3](#_r9b0nfi769iu)

[Modelfactor analysis 4](#_is6jrlyio9o9)

[Paraver analysis 5](#_aflw5jpxjk1j)

[Memory analysis 6](#_1jhkxj2lt84i)

[Strong scalability 8](#_tr5sfmlep3uf)

[1D Block-Cyclic Geometric Data Decomposition by columns 9](#_qbb7k9naxwgh)

[Code 9](#_2k9g51vquyb1)

[Block size analysis 9](#_an5icj2y7eax)

[Modelfactor analysis 11](#_7rn2o573mv8e)

[Paraver analysis 12](#_yl4q6135syaq)

[Memory analysis 13](#_ds238zsac47i)

[Strong scalability 15](#_hvoqujvgjvnu)

[1D Cyclic Geometric Data Decomposition by rows 16](#_eb5hxop06g9)

[Code 16](#_8zz0xguywyz3)

[Modelfactor analysis 17](#_12vk0sa6qx0z)

[Paraver analysis 18](#_5va480m0plun)

[Memory analysis 19](#_x1s53dq0mtzc)

[**Final results 22**](#_ei3gm88clfc1)

# [[1]](#footnote-0)

# Geometric Data Decomposition Strategies

## 1D Block Geometric Data Decomposition by columns

### Code

The file name is mandel-omp-iter-block-columns.cpp.

In order to apply the required strategy we divided the work in the columns by the number of threads, that is the BS (block size). We used the thread id to assign to each thread their respective division. To ensure a correct execution we had to use a pragma omp atomic and a pragma omp critical, protecting by doing so the variables histogram and X\_COL\_11.

void mandel\_simple(int M[ROWS][COLS], double CminR, double CminI, double CmaxR, double CmaxI, double scale\_real, double scale\_imag, int maxiter)

{

#pragma omp parallel

{

int id = omp\_get\_thread\_num();

int n\_threads = omp\_get\_num\_threads();

int BS = COLS/n\_threads;

int start = id\*BS;

int end = fmin(start+BS, COLS);

// Calcular

for (int py = 0; py < ROWS; py++) {

for (int px = start; px < end; px++) {

M[py][px] = pixel\_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, py, scale\_real, scale\_imag, maxiter);

if (output2histogram)

#pragma omp atomic

histogram[M[py][px] - 1]++;

if (output2display) {

/\* Scale color and display point \*/

long color = (long)((M[py][px] - 1) \* scale\_color) + min\_color;

if (setup\_return == EXIT\_SUCCESS) {

#pragma omp critical

{

XSetForeground(display, gc, color);

XDrawPoint(display, win, gc, px, py);

}

}

}

}

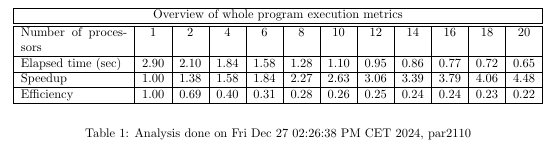
}

}

}

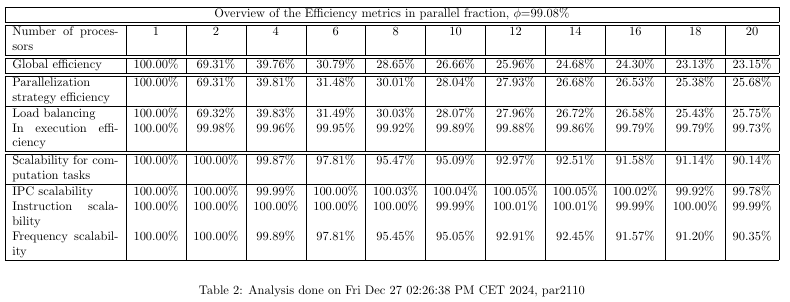


### Modelfactor analysis



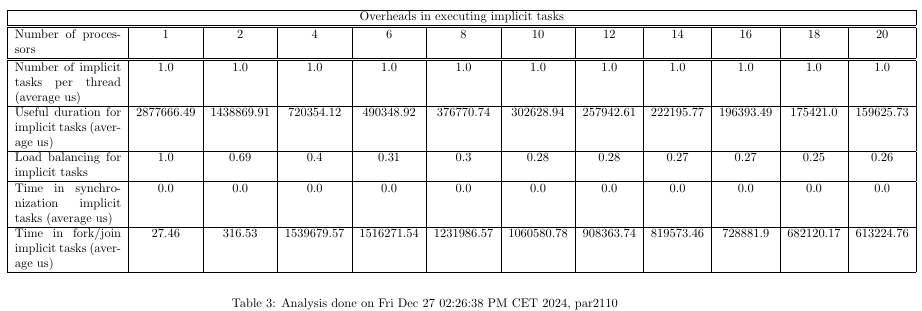
Modelfactor Table 1

By looking at this first table we can see that with this strategy we lose a lot of efficiency, for this reason it’s not a very good one since even though we obtain a speedup, with a more appropriate strategy we would get a much better one.



Modelfactor Table 2

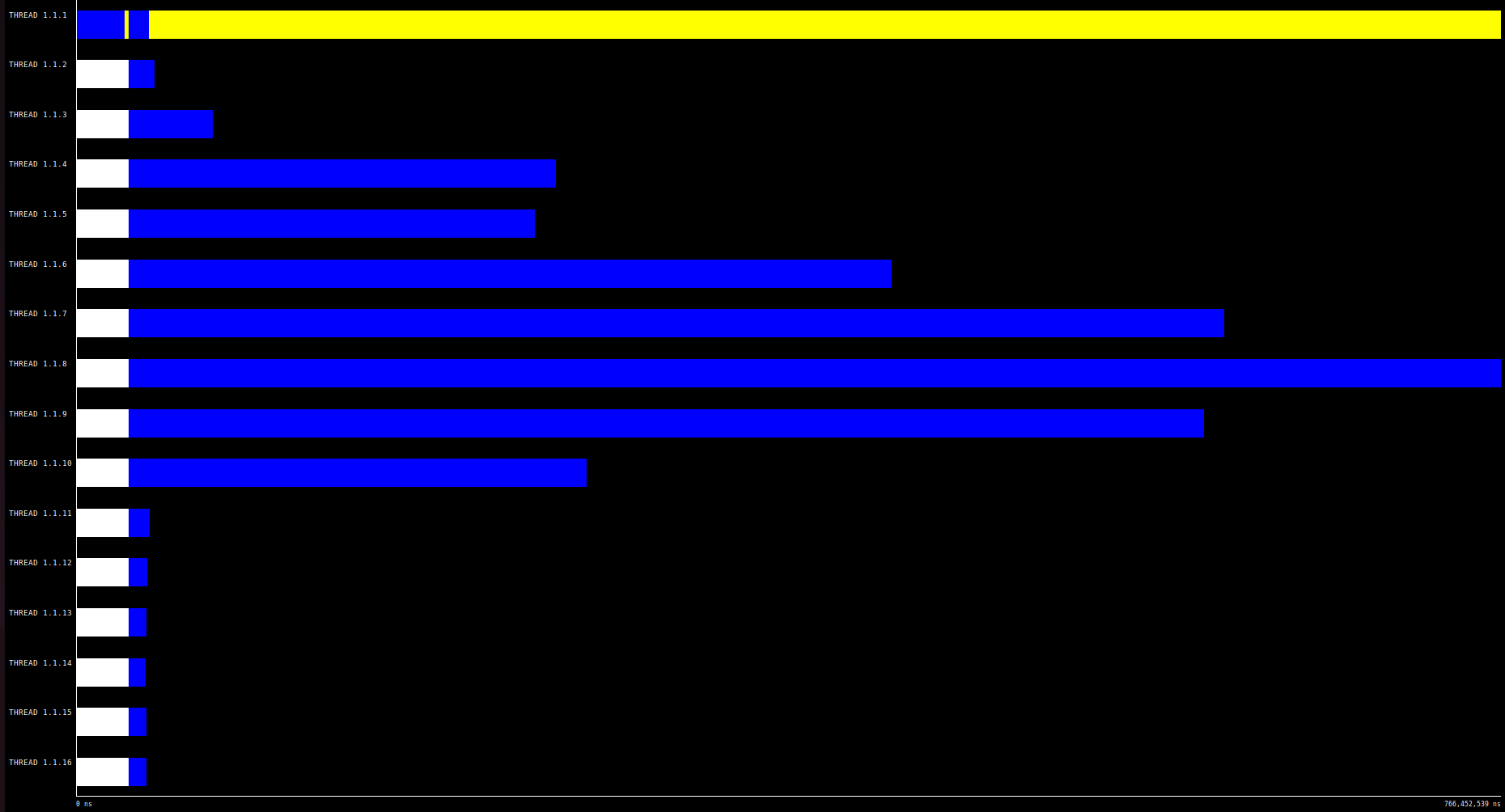
On the second Modelfactor Table we can see where the inefficiency comes from. We see that we have a load balancing problem, and that is the main reason for which the global efficiency drops.



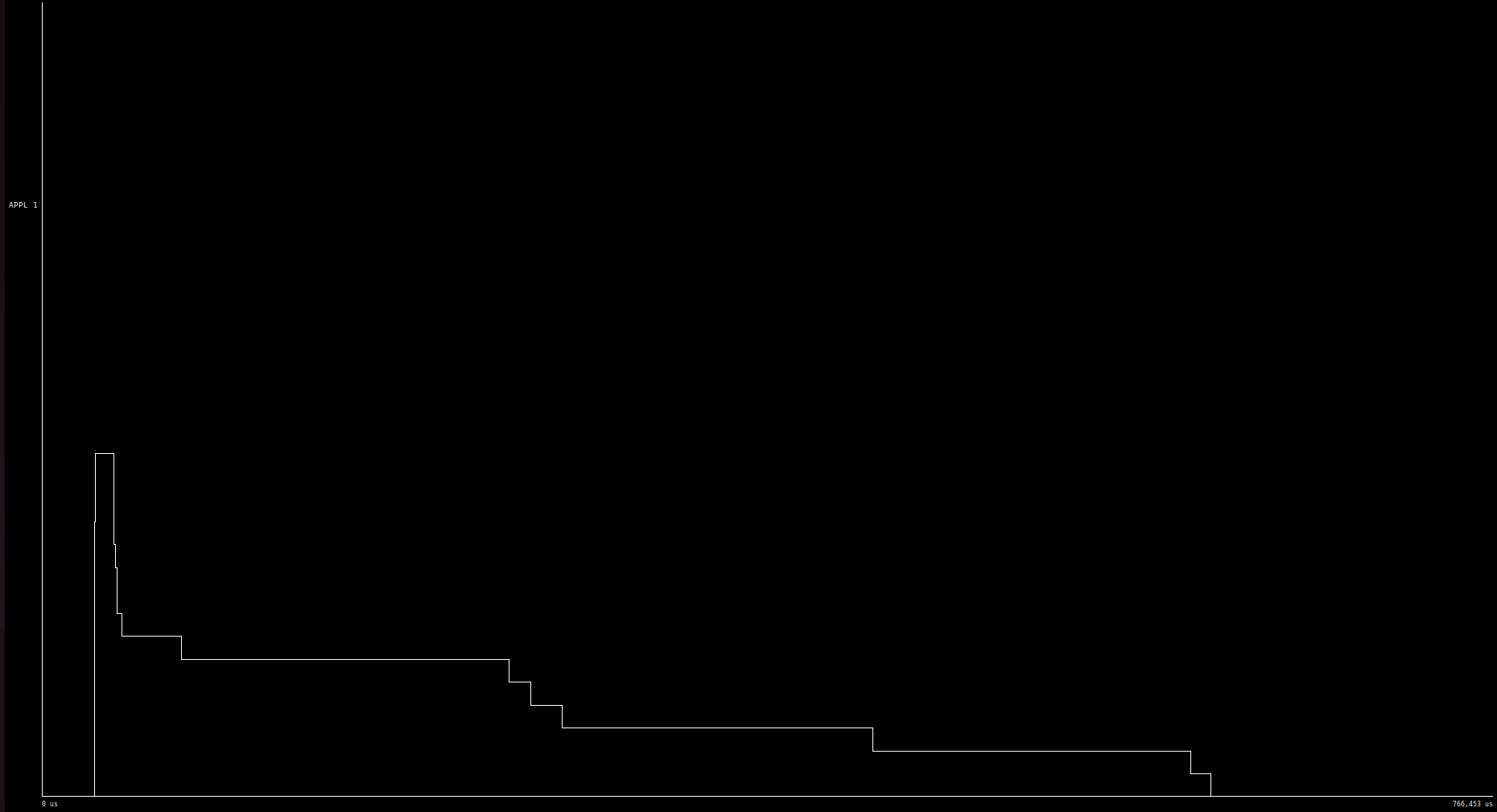
Modelfactor Table 3

Finally on the last table we can again see the load unbalance, by looking at load balancing and by looking at the useful duration for tasks. However we can see as well that despite not having problems on synchronization we waste time on the fork/join of the implicit tasks.

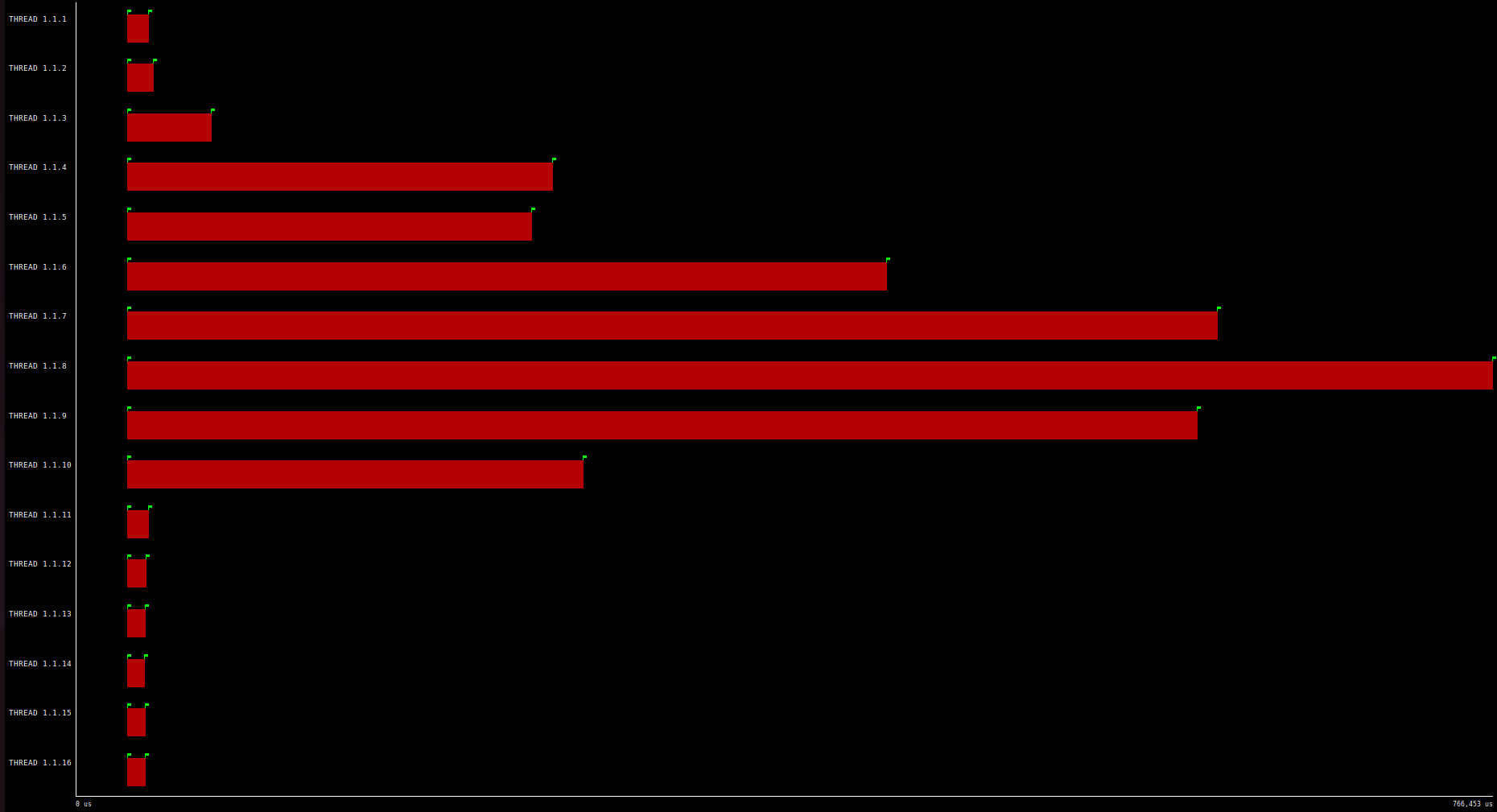
### Paraver analysis



Paraver execution 16 threads



Paraver hint Instantaneous parallelism



Paraver hint Implicit tasks in parallel constructs

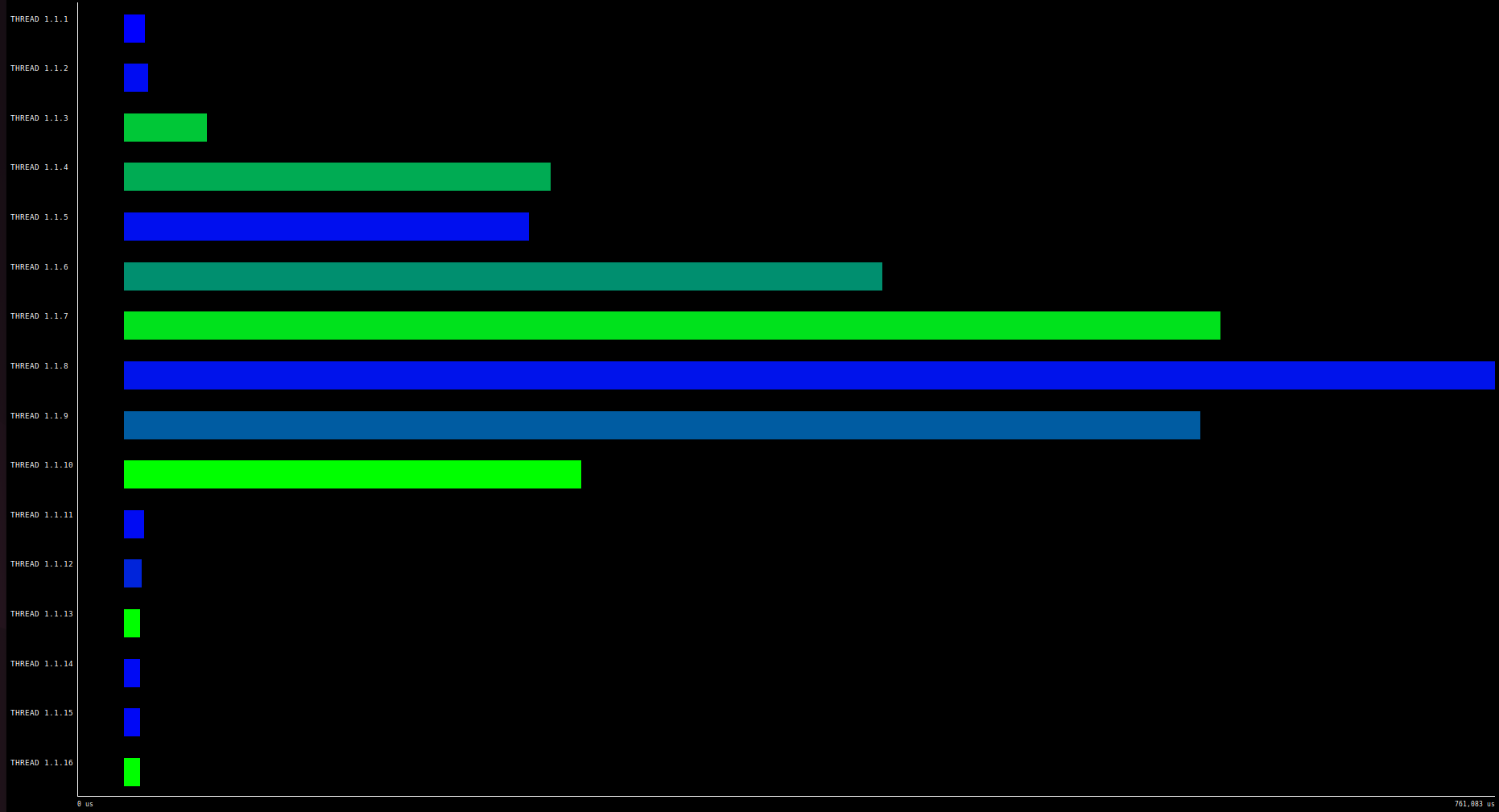
On the paraver graphs we can confirm that this strategy is not good since the parallelism through the execution drops sharply (we can see that on the instantaneous parallelism graph). On the other ones we can confirm that the work is very unbalanced between the threads.

### Memory analysis

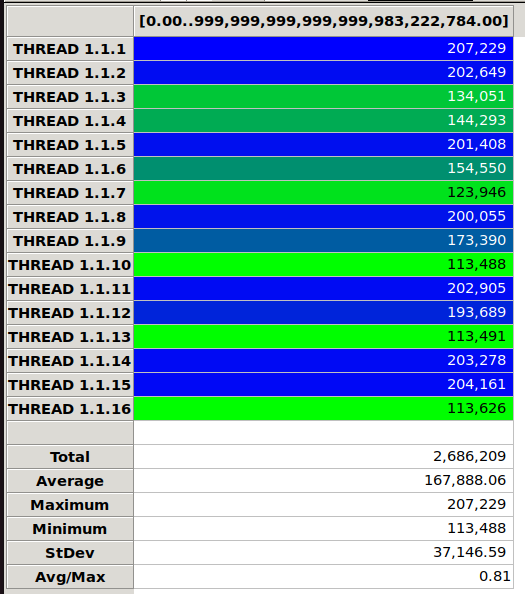
| **Number of threads** | **L2 cache accesses** | **L2 cache misses** |
| --- | --- | --- |
| **1** | 1642085 | 1642085 |
| **2** | 1764131 | 882065 |
| **4** | 1981612 | 495403 |
| **6** | 2186449 | 364408 |
| **8** | 2333196 | 291649 |
| **10** | 2258804 | 225880 |
| **12** | 2711165 | 225930 |
| **14** | 2825750 | 201839 |
| **16** | 2696027 | 168501 |
| **18** | 2866290 | 159238 |
| **20** | 2997336 | 149866 |

Table of L2 accesses/misses

In this first table we can see the relation between the number of threads, the number of cache accesses and the number of misses per thread. As we can see, the more threads we have, the more accesses to the cache we do. However, since we have more threads as well, in the end the number of misses per thread is lower.



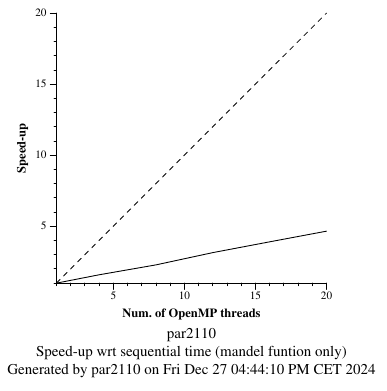
Paraver of L2 cache accesses/misses in color



Statistics about L2 accesses/misses

On the previous figures we can see that not all threads make the same number of accesses to the cache. The ones marked in blue access more than the others, on the statistics we can see the specific number of accesses.

### Strong scalability



Strong scalability graph

The strong scalability graph shows us that again this strategy is not good at all. The speedup we obtain is linear, however is very far from the ideal speedup.

## 1D Block-Cyclic Geometric Data Decomposition by columns

### Code

The file name is mandel-omp-iter-block-cyclic-columns.cpp.

In this version we don’t make one thread work for the whole block of BS size, what we do in this version is start each thread on the iteration of their id. Then we increase the next iteration by the number of threads. By doing so we will compute all the columns but we will not have the blocks of consecutive iterations.

void mandel\_simple(int M[ROWS][COLS], double CminR, double CminI, double CmaxR, double CmaxI, double scale\_real, double scale\_imag, int maxiter)

{

#pragma omp parallel

{

int id = omp\_get\_thread\_num();

int n\_threads = omp\_get\_num\_threads();

int BS = COLS/n\_threads;

int start = id;

// Calcular

for (int py = 0; py < ROWS; py++) {

for (int px = start; px < COLS; px+=n\_threads) {

M[py][px] = pixel\_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, py, scale\_real, scale\_imag, maxiter);

if (output2histogram)

#pragma omp atomic

histogram[M[py][px] - 1]++;

if (output2display) {

/\* Scale color and display point \*/

long color = (long)((M[py][px] - 1) \* scale\_color) + min\_color;

if (setup\_return == EXIT\_SUCCESS) {

#pragma omp critical

{

XSetForeground(display, gc, color);

XDrawPoint(display, win, gc, px, py);

}

}

}

}

}

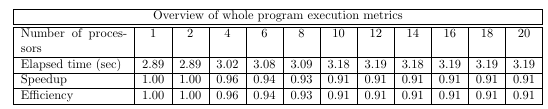
}

}

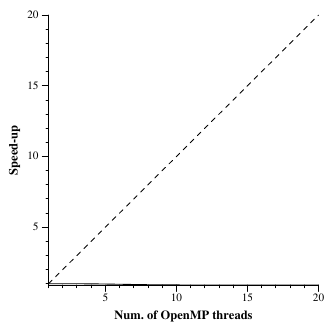
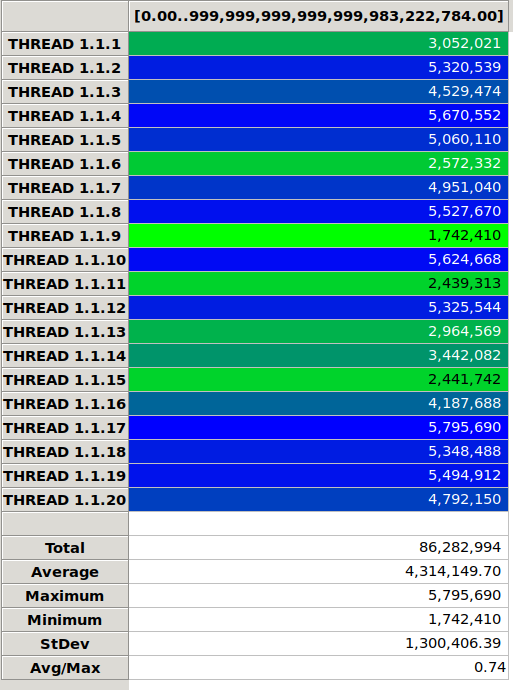


### Block size analysis

We tried different block-cyclic sizes, in order to find the best ones, in the end we decided to choose the n\_threads size. As an example the next figures on this section are from the version of size 1. And we can see that it is much worse than the version we chose. Even though work is balanced. This is because threads do work that has already been done by another thread.



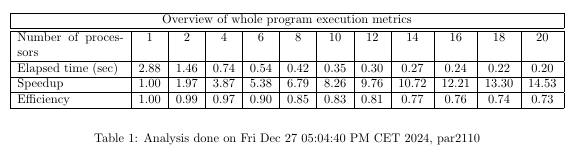




FIgures of Block size 1

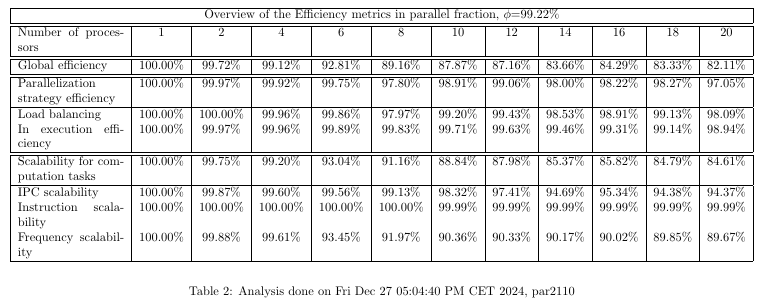
### 

### Modelfactor analysis

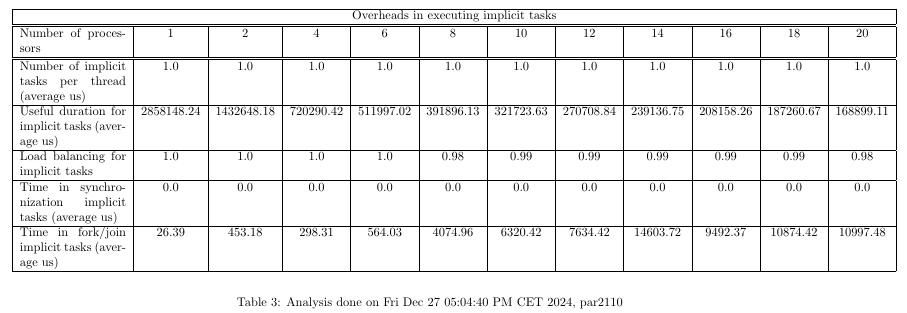


Modelfactor Table 1

On this first table we can clearly see that the efficiency is much better than on the previous one. It does not drop so drastically thus the speedup we obtain at the end is much better.



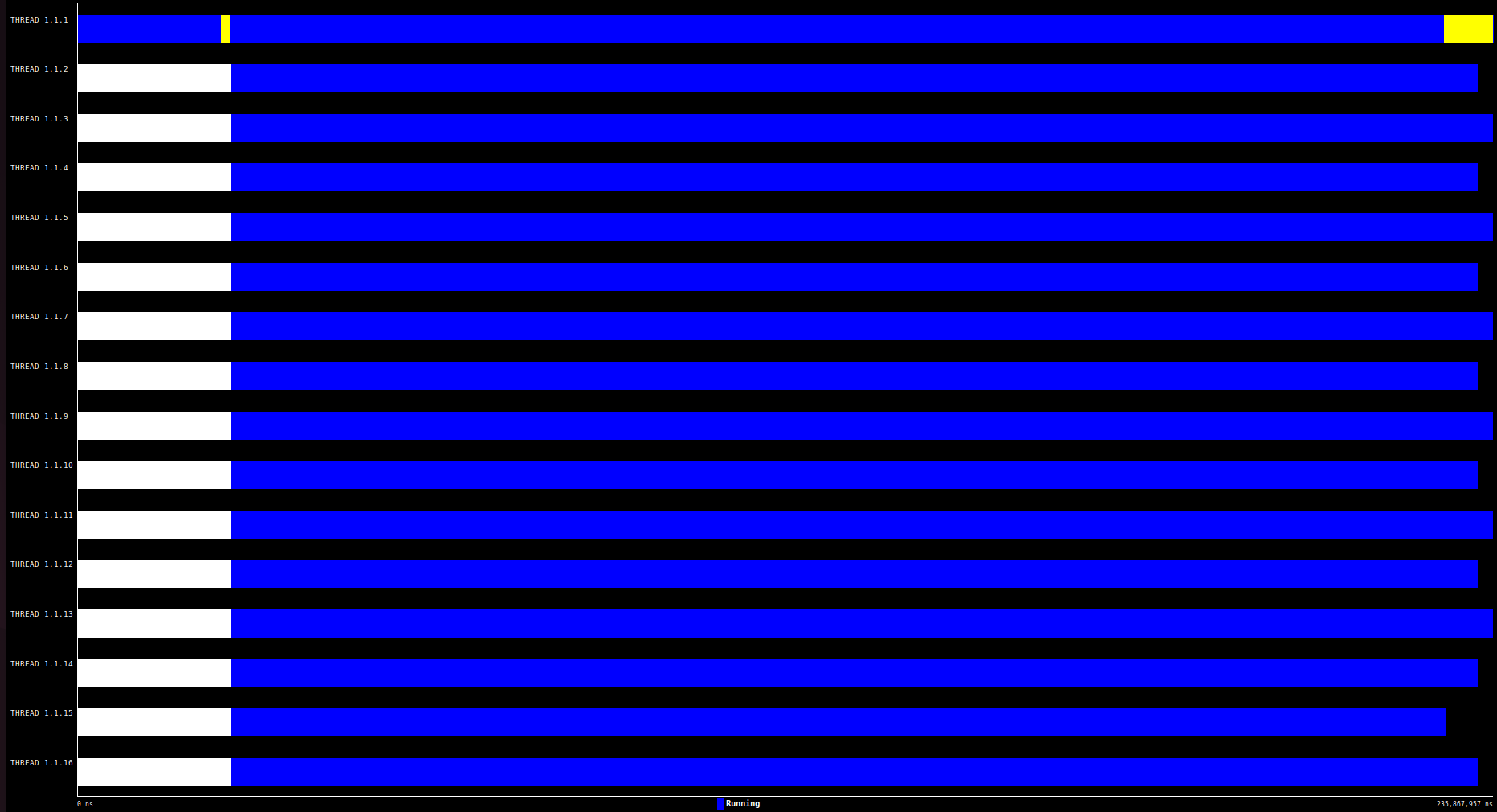
Modelfactor Table 2



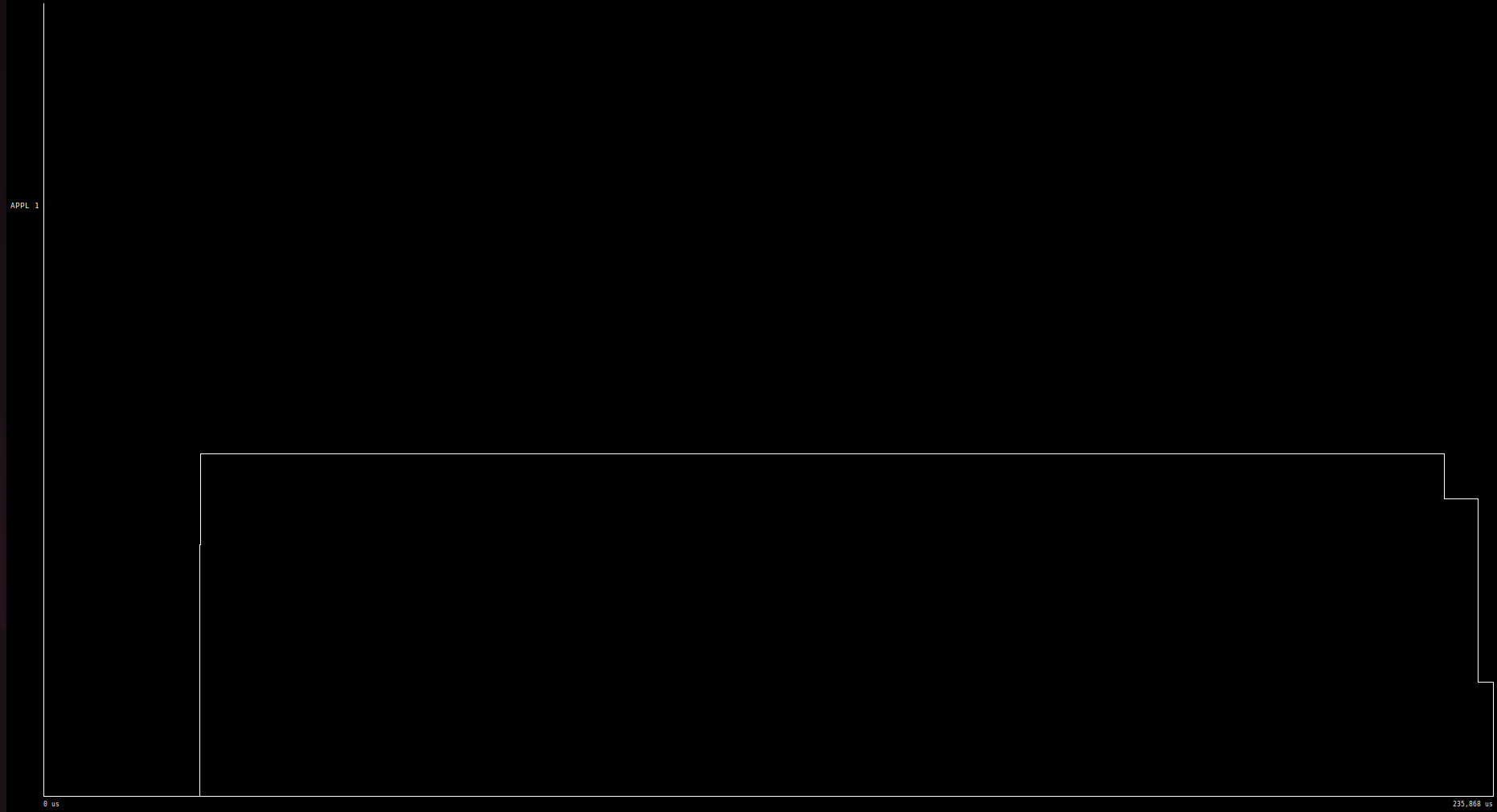
Modelfactor Table 3

On the other tables we see that the work is much better balanced, almost perfect. Now the main problem is in scalability. So although we have reduced the unbalance by a lot we can still improve the efficiency, after having analyzed the memory (in the next section) we can deduce that this problem in scalability is caused by the big amount of cache misses.

### Paraver analysis



Paraver execution 16 threads



Paraver hint Instantaneous parallelism



Paraver hint Implicit tasks in parallel constructs

The paraver graphs show us the same as we saw on the modelfactor analysis. The improved load balance that causes the instantaneous and effective parallelism to be much better.

### Memory analysis

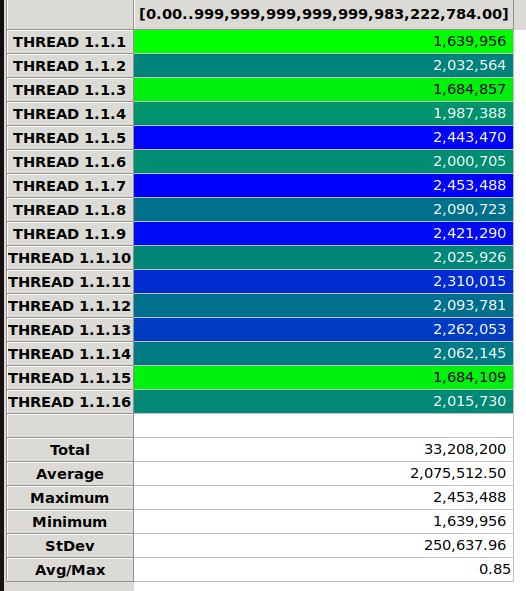
| **Number of threads** | **L2 cache accesses** | **L2 cache misses** |
| --- | --- | --- |
| **1** | 1642369 | 1642369 |
| **2** | 3286372 | 1643186 |
| **4** | 6628953 | 1657238 |
| **6** | 11195032 | 1865838 |
| **8** | 16386148 | 2048268 |
| **10** | 19187093 | 1918709 |
| **12** | 25516346 | 2126362 |
| **14** | 30386693 | 2170478 |
| **16** | 33217324 | 2076082 |
| **18** | 39046487 | 2169249 |
| **20** | 43770154 | 2188507 |

Table of L2 accesses/misses

On this version we have the same relation between number of threads and the total number of accesses, however we can see that the number of misses per thread is not linear like on the previous strategy. The number of misses is higher than the previous case, this gives us a clue about what the problem of this strategy is (probably how the matrix is stored, by rows). This is what we will fix on the next strategy, working by rows instead of columns will reduce the number of misses.



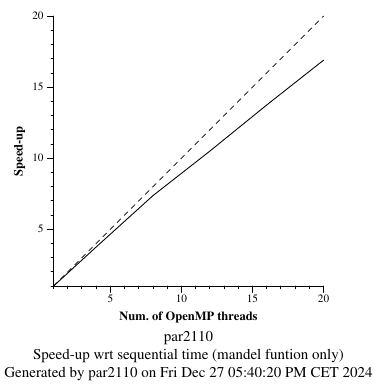
Paraver of L2 cache accesses/misses in color



Statistics about L2 accesses/misses

By looking at the other two images we obtain similar conclusions, being the most important the high number of misses, caused by the way the matrix is stored and the way we access it, causing more misses.

### Strong scalability



Strong scalability graph

As we can see in terms of parallelism this option is better than the previous one, however we can see that the more threads we add, the further we are from the ideal parallelism.

## 1D Cyclic Geometric Data Decomposition by rows

### Code

The file name is mandel-omp-iter-block-cyclic-rows.cpp.

As the previous version, we don’t make one thread work for the whole block of BS size. Instead, we start each thread on the iteration of their own id. We also increment the external for by the n\_threads, to avoid doing too much work on the same thread. Each thread will do all the columns of the rows it modifies, but not all rows (the opposite of the previous code).

void mandel\_simple(int M[ROWS][COLS], double CminR, double CminI, double CmaxR, double CmaxI, double scale\_real, double scale\_imag, int maxiter)

{

#pragma omp parallel

{

int id = omp\_get\_thread\_num();

int n\_threads = omp\_get\_num\_threads();

int BS = COLS/n\_threads;

int start = id;

// Calcular

for (int py = start; py < ROWS; py+=n\_threads) {

for (int px = 0; px < COLS; px++) {

M[py][px] = pixel\_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, py, scale\_real, scale\_imag, maxiter);

if (output2histogram)

#pragma omp atomic

histogram[M[py][px] - 1]++;

if (output2display) {

/\* Scale color and display point \*/

long color = (long)((M[py][px] - 1) \* scale\_color) + min\_color;

if (setup\_return == EXIT\_SUCCESS) {

#pragma omp critical

{

XSetForeground(display, gc, color);

XDrawPoint(display, win, gc, px, py);

}

}

}

}

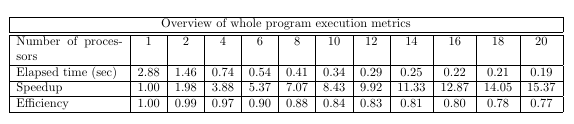
}

}

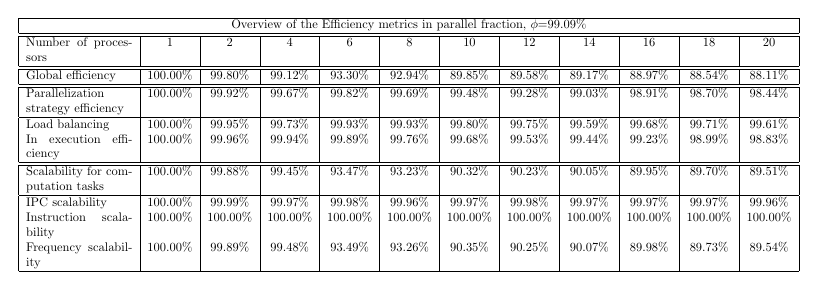
}



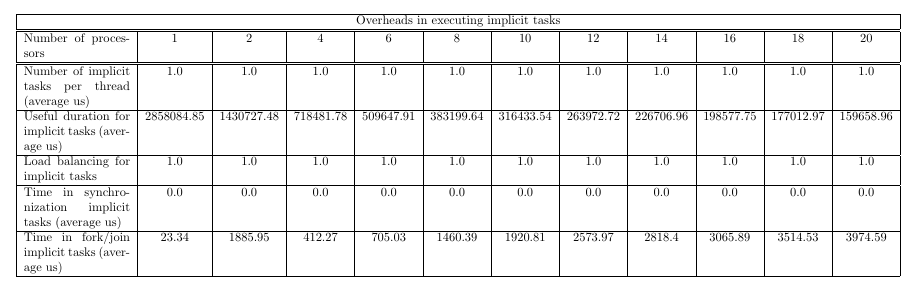
### Modelfactor analysis



Modelfactor Table 1



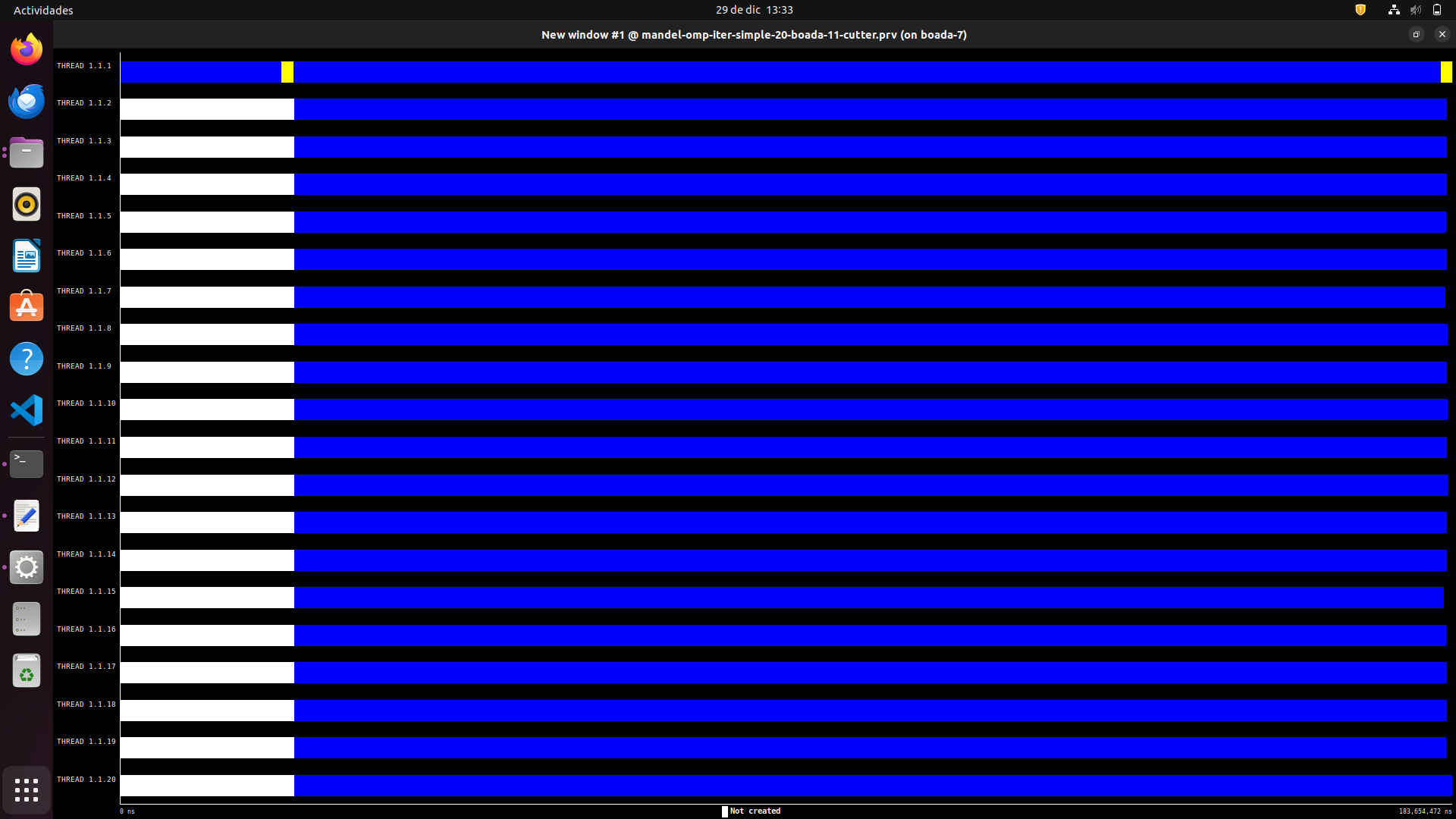
Modelfactor Table 2



Modelfactor Table3

With the modelfactor tables, we can see that with this implementation we obtain a high value for the speed-up, achieving 15,37 with 20 processors. The efficiency is also good, which we can prove by watching the second table also. We have to consider though, that with more processors, this value will continue falling. The load balancing is also very high, which indicates that the work’s division for each processor is well distributed. Apart from the well distributed work, we can see that each of the processors works almost all the time, they are not inactive.

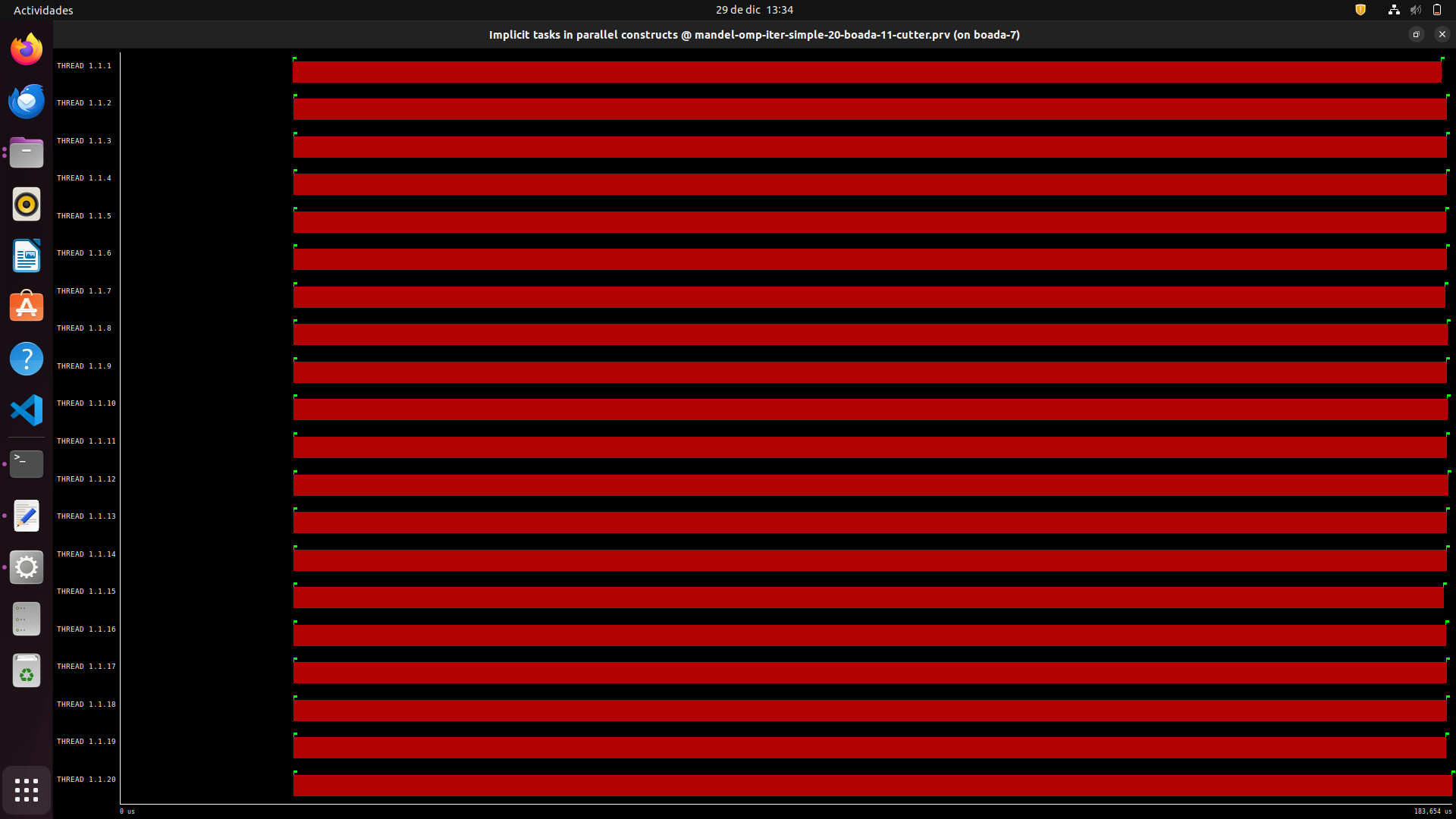
### Paraver analysis



Paraver execution



Paraver Instantaneous parallelism



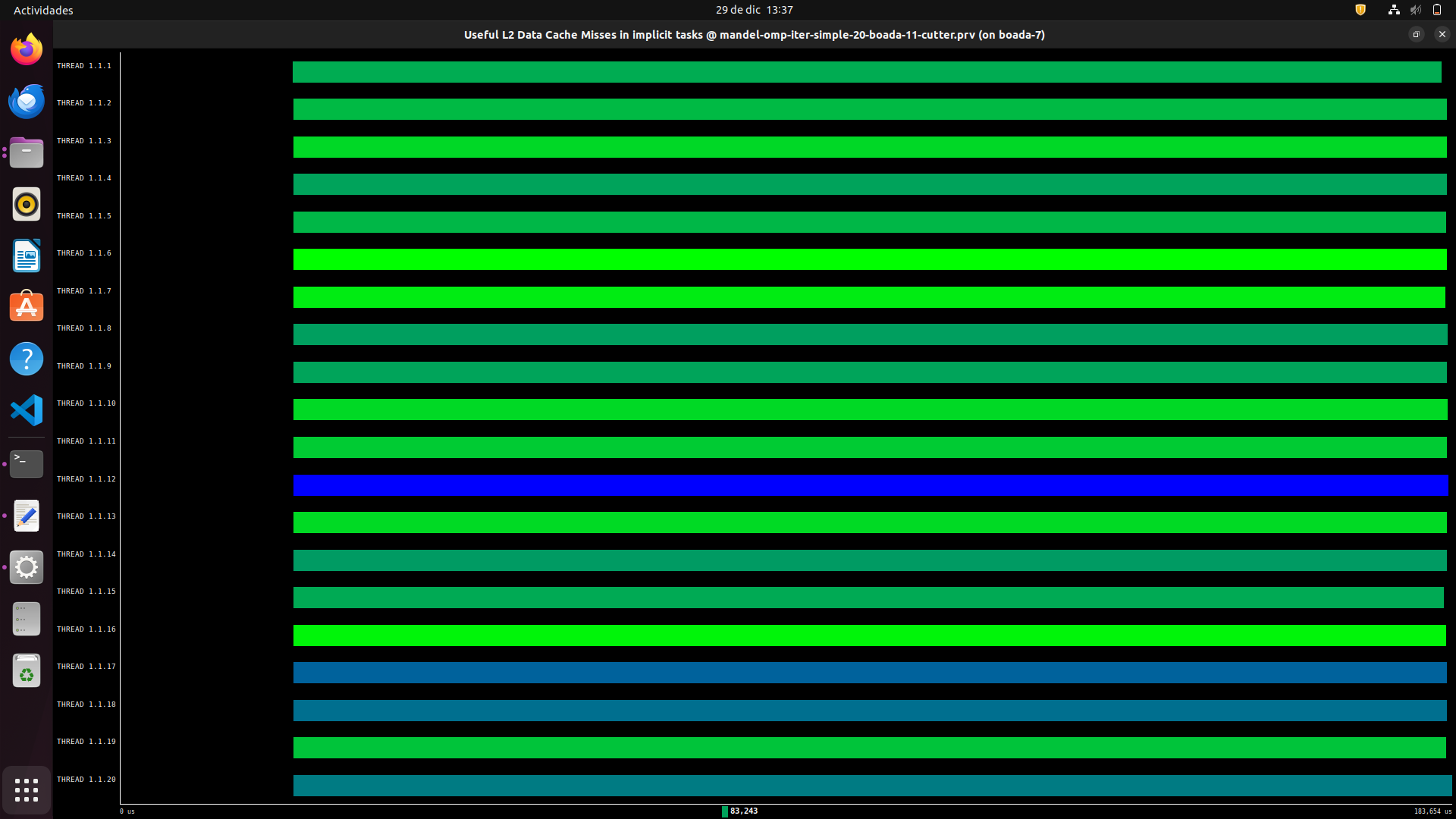
Paraver Implicit tasks in parallel constructs

As commented with the modelfactor tables, we can see with the graphics that the threads work almost the majority of the time once they are assigned to do the work. The parallelism is very straight during all the execution, which can be seen with the instantaneous parallelism.

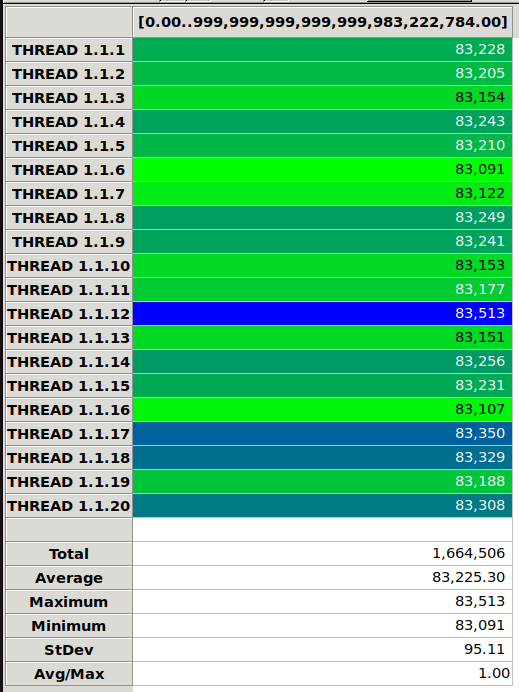
### Memory analysis

| **Number of threads** | **L2 cache accesses** | **L2 cache misses** |
| --- | --- | --- |
| **1** | 1642420 | 1642420 |
| **2** | 1646518 | 823259 |
| **4** | 1649726 | 412431 |
| **6** | 1652606 | 275434 |
| **8** | 1654672 | 206834 |
| **10** | 1659500 | 165950 |
| **12** | 1663134 | 138594 |
| **14** | 1665285 | 118948 |
| **16** | 1662696 | 103918 |
| **18** | 1670126 | 92784 |
| **20** | 1674415 | 83720 |

Table of L2 accesses/misses



Paraver of L2 cache accesses/misses in color

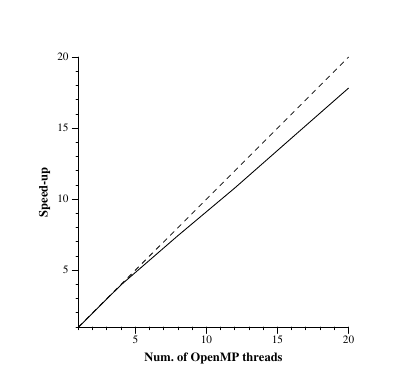


Statistics about L2 accesses/misses

With the first table, we can see that misses fall as the number of threads increases. We can also see that these misses are less than the previous implementations, especially as the number of threads increases. With the second graph, we can appreciate that almost all the threads have few misses compared with the total access of each thread (almost all of them have a green color, which means less misses).

The third graph combines the information of the other ones. We see that the maximum number of misses and the minimum is very similar. This improvement is thanks to travessing the matrix by rows instead of columns. This shows us the importance of knowing how data is stored in memory and in cache.

Strong scalability



Strong scalability graph

The obtained result is very similar to the ideal one. At the beginning it is almost perfect, but as the number of threads increases, the speed-up obtained gets away from the ideal one. Despite this deviation, the result is very good.

# Final results

As we can see in terms of time version 2 and 3 are similar, however if we analyze deeper we can see that the last strategy is much more efficient in terms of access to the cache since we have less misses. This is very important and makes this strategy much better because it makes the problem more scalable, and makes that even though the parallelism is efficient in both cases with less misses it is faster.

|  | Number of threads (elapsed time (s) ) | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Version** | 1 | 4 | 8 | 12 | 16 | 20 |
| 1D Block Geometric Data Decomposition by columns | 2,90 | 1,84 | 1,28 | 0,95 | 0,77 | 0,65 |
| 1D Block-Cyclic Geometric Data Decomposition by columns | 2,88 | 0,74 | 0,42 | 0,30 | 0,24 | 0,20 |
| 1D Cyclic Geometric Data Decomposition by rows | 2.88 | 0.74 | 0.41 | 0.29 | 0.22 | 0.19 |
|  | Number of threads (L2 Cache Misses per thread) | | | | | |
| 1D Block Geometric Data Decomposition by columns | 1642085 | 495403 | 291649 | 225930 | 168501 | 149866 |
| 1D Block-Cyclic Geometric Data Decomposition by columns | 1642369 | 1657238 | 2048268 | 2126362 | 2076082 | 2188507 |
| 1D Cyclic Geometric Data Decomposition by rows | 1642420 | 412431 | 206834 | 138594 | 103918 | 83720 |

1. Note: The analysis of each image is done underneath the same. [↑](#footnote-ref-0)