UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

LAB 5: Parallel Data Decomposition Implementation and Analysis

Arnau Garcia Gonzalez
Jan Santos Bastida

Table of contents

[1] Note: The analysis of each image is done underneath the same.

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.

```
C/C++
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

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.90	2.10	1.84	1.58	1.28	1.10	0.95	0.86	0.77	0.72	0.65
Speedup	1.00	1.38	1.58	1.84	2.27	2.63	3.06	3.39	3.79	4.06	4.48
Efficiency	1.00	0.69	0.40	0.31	0.28	0.26	0.25	0.24	0.24	0.23	0.22

Table 1: Analysis done on Fri Dec 27 02:26:38 PM CET 2024, par2110

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.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.08%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	69.31%	39.76%	30.79%	28.65%	26.66%	25.96%	24.68%	24.30%	23.13%	23.15%
Parallelization strategy efficiency	100.00%	69.31%	39.81%	31.48%	30.01%	28.04%	27.93%	26.68%	26.53%	25.38%	25.68%
Load balancing	100.00%	69.32%	39.83%	31.49%	30.03%	28.07%	27.96%	26.72%	26.58%	25.43%	25.75%
In execution efficiency	100.00%	99.98%	99.96%	99.95%	99.92%	99.89%	99.88%	99.86%	99.79%	99.79%	99.73%
Scalability for computation tasks	100.00%	100.00%	99.87%	97.81%	95.47%	95.09%	92.97%	92.51%	91.58%	91.14%	90.14%
IPC scalability	100.00%	100.00%	99.99%	100.00%	100.03%	100.04%	100.05%	100.05%	100.02%	99.92%	99.78%
Instruction scalability	100.00%	100.00%	100.00%	100.00%	100.00%	99.99%	100.01%	100.01%	99.99%	100.00%	99.99%
Frequency scalability	100.00%	100.00%	99.89%	97.81%	95.45%	95.05%	92.91%	92.45%	91.57%	91.20%	90.35%

Table 2: Analysis done on Fri Dec 27 02:26:38 PM CET 2024, par2110

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.

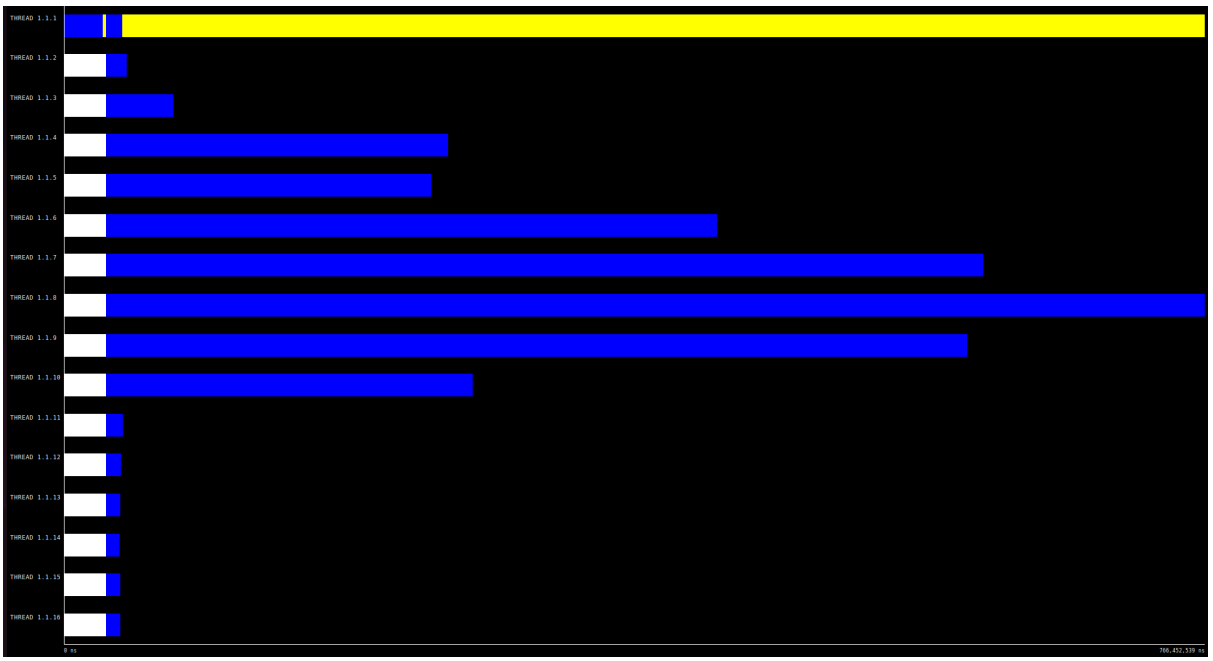
Overheads in executing implicit tasks											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Number of implicit tasks per thread (average us)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Useful duration for implicit tasks (average us)	2877666.49	1438869.91	720354.12	490348.92	376770.74	302628.94	257942.61	222195.77	196393.49	175421.0	159625.73
Load balancing for implicit tasks	1.0	0.69	0.4	0.31	0.3	0.28	0.28	0.27	0.27	0.25	0.26
Time in synchronization implicit tasks (average us)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Time in fork/join implicit tasks (average us)	27.46	316.53	1539679.57	1516271.54	1231986.57	1060580.78	908363.74	819573.46	728881.9	682120.17	613224.76

Table 3: Analysis done on Fri Dec 27 02:26:38 PM CET 2024, par2110

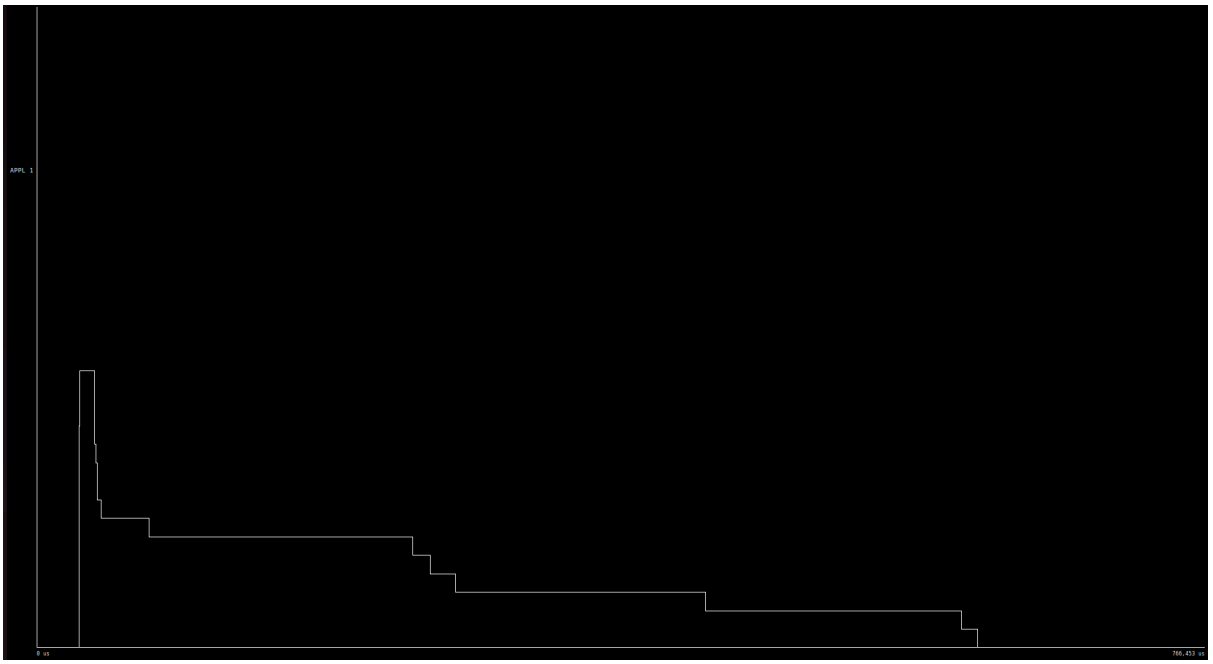
Modelfactor Table 3

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.

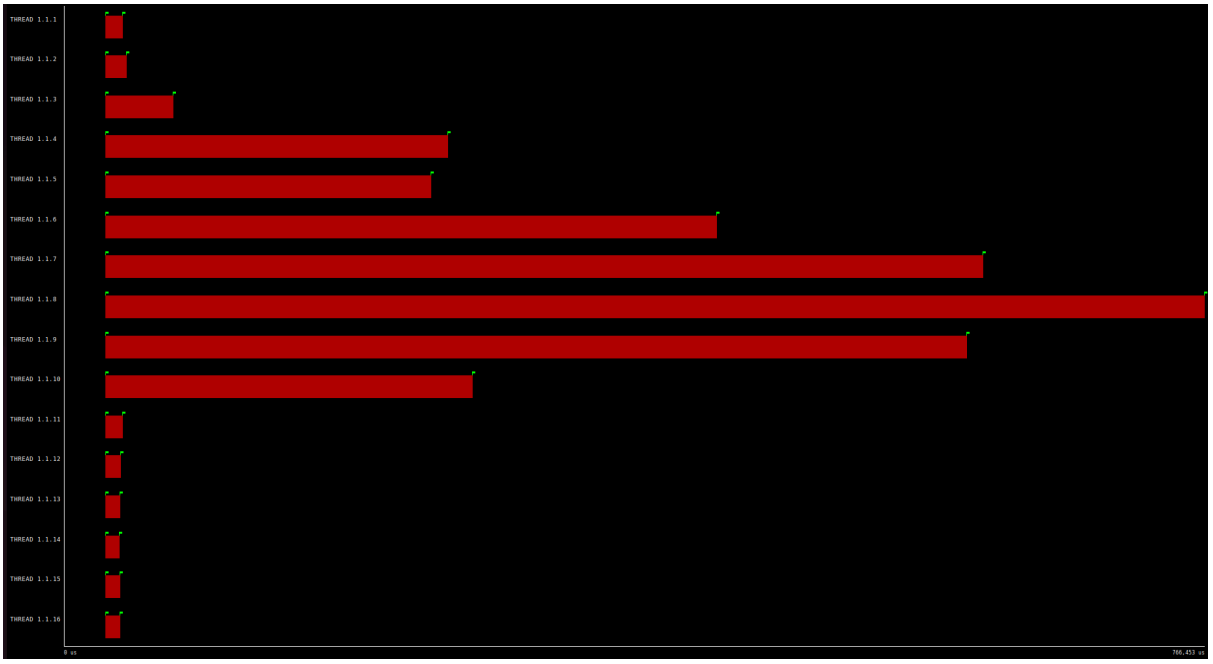
Paraver analysis



Paraver execution 16 threads



Paraver hint Instantaneous parallelism



Paraver hint Implicit tasks in parallel constructs

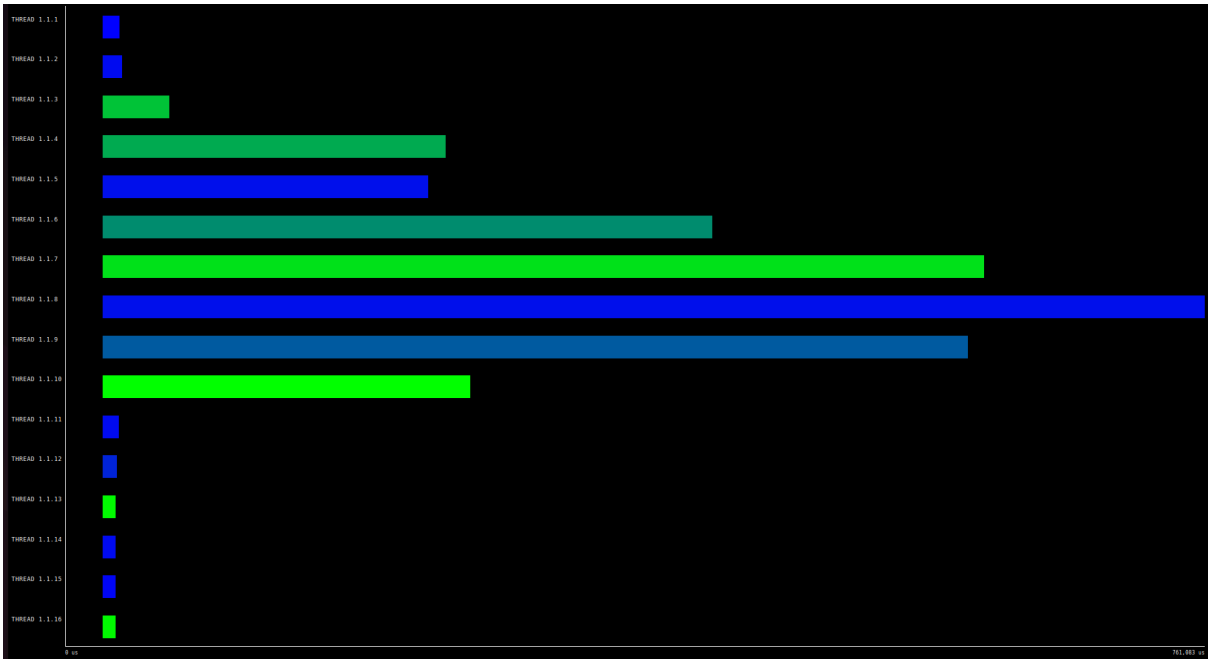
On the paraver graphs we can confirm that this strategy is not good since the parallelism through the execution drops sharply (we can see that on the instantaneous parallelism graph). On the other ones we can confirm that the work is very unbalanced between the threads.

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.



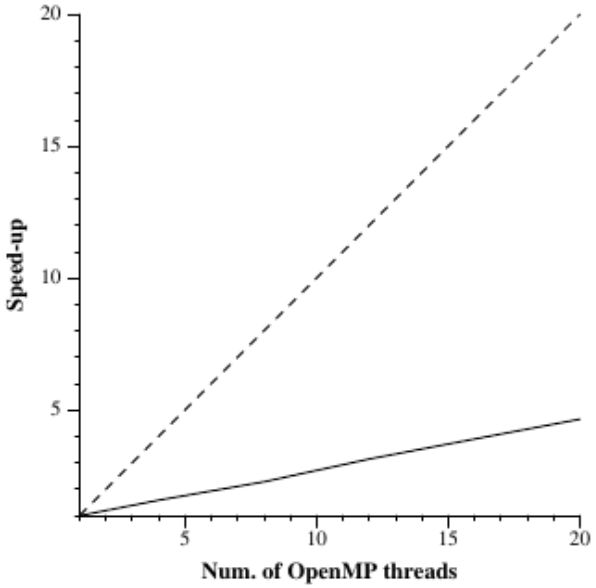
Paraver of L2 cache accesses/misses in color

	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	207,229
THREAD 1.1.2	202,649
THREAD 1.1.3	134,051
THREAD 1.1.4	144,293
THREAD 1.1.5	201,408
THREAD 1.1.6	154,550
THREAD 1.1.7	123,946
THREAD 1.1.8	200,055
THREAD 1.1.9	173,390
THREAD 1.1.10	113,488
THREAD 1.1.11	202,905
THREAD 1.1.12	193,689
THREAD 1.1.13	113,491
THREAD 1.1.14	203,278
THREAD 1.1.15	204,161
THREAD 1.1.16	113,626
Total	2,686,209
Average	167,888.06
Maximum	207,229
Minimum	113,488
StDev	37,146.59
Avg/Max	0.81

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



par2110
Speed-up wrt sequential time (mandel funtion only)
Generated by par2110 on Fri Dec 27 04:44:10 PM CET 2024

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.

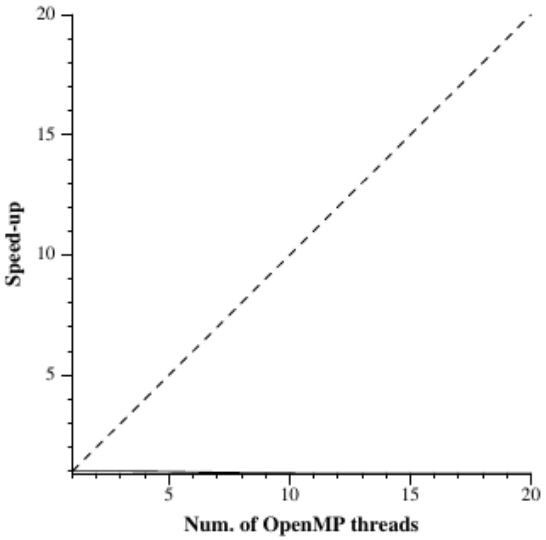
```
C/C++
void mandel_simple(int M[ROWS][COLS], double CminR, double CminI, double CmaxR, double
CmaxI, double scale_real, double scale_imag, int maxiter)
{
        #pragma omp parallel
        {

        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.

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.89	2.89	3.02	3.08	3.09	3.18	3.19	3.18	3.19	3.19	3.19
Speedup	1.00	1.00	0.96	0.94	0.93	0.91	0.91	0.91	0.91	0.91	0.91
Efficiency	1.00	1.00	0.96	0.94	0.93	0.91	0.91	0.91	0.91	0.91	0.91

	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	3,052,021
THREAD 1.1.2	5,320,539
THREAD 1.1.3	4,529,474
THREAD 1.1.4	5,670,552
THREAD 1.1.5	5,060,110
THREAD 1.1.6	2,572,332
THREAD 1.1.7	4,951,040
THREAD 1.1.8	5,527,670
THREAD 1.1.9	1,742,410
THREAD 1.1.10	5,624,668
THREAD 1.1.11	2,439,313
THREAD 1.1.12	5,325,544
THREAD 1.1.13	2,964,569
THREAD 1.1.14	3,442,082
THREAD 1.1.15	2,441,742
THREAD 1.1.16	4,187,688
THREAD 1.1.17	5,795,690
THREAD 1.1.18	5,348,488
THREAD 1.1.19	5,494,912
THREAD 1.1.20	4,792,150
Total	86,282,994
Average	4,314,149.70
Maximum	5,795,690
Minimum	1,742,410
StDev	1,300,406.39
Avg/Max	0.74



FIgures of Block size 1

Modelfactor analysis

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.88	1.46	0.74	0.54	0.42	0.35	0.30	0.27	0.24	0.22	0.20
Speedup	1.00	1.97	3.87	5.38	6.79	8.26	9.76	10.72	12.21	13.30	14.53
Efficiency	1.00	0.99	0.97	0.90	0.85	0.83	0.81	0.77	0.76	0.74	0.73

Table 1: Analysis done on Fri Dec 27 05:04:40 PM CET 2024, par2110

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.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.22%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	99.72%	99.12%	92.81%	89.16%	87.87%	87.16%	83.66%	84.29%	83.33%	82.11%
Parallelization strategy efficiency	100.00%	99.97%	99.92%	99.75%	97.80%	98.91%	99.06%	98.00%	98.22%	98.27%	97.05%
Load balancing	100.00%	100.00%	99.96%	99.86%	97.97%	99.20%	99.43%	98.53%	98.91%	99.13%	98.09%
In execution efficiency	100.00%	99.97%	99.96%	99.89%	99.83%	99.71%	99.63%	99.46%	99.31%	99.14%	98.94%
Scalability for computation tasks	100.00%	99.75%	99.20%	93.04%	91.16%	88.84%	87.98%	85.37%	85.82%	84.79%	84.61%
IPC scalability	100.00%	99.87%	99.60%	99.56%	99.13%	98.32%	97.41%	94.69%	95.34%	94.38%	94.37%
Instruction scalability	100.00%	100.00%	100.00%	100.00%	100.00%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%
Frequency scalability	100.00%	99.88%	99.61%	93.45%	91.97%	90.36%	90.33%	90.17%	90.02%	89.85%	89.67%

Table 2: Analysis done on Fri Dec 27 05:04:40 PM CET 2024, par2110

Modelfactor Table 2

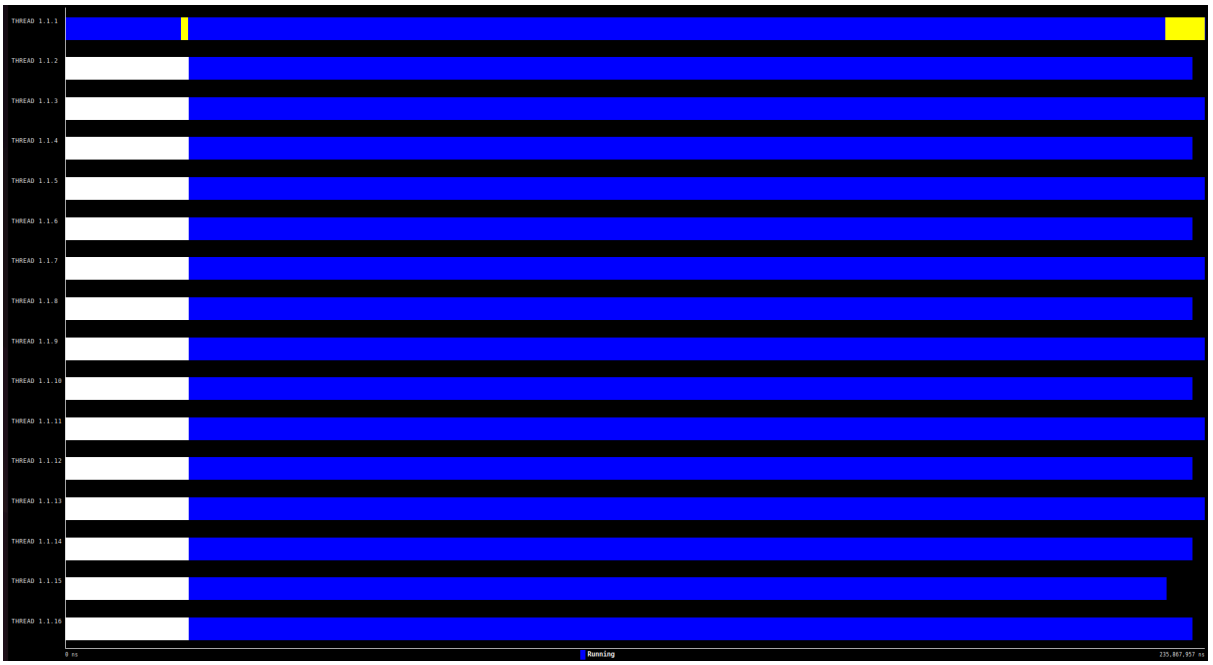
Overheads in executing implicit tasks											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Number of implicit tasks per thread (average us)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Useful duration for implicit tasks (average us)	2858148.24	1432648.18	720290.42	511997.02	391896.13	321723.63	270708.84	239136.75	208158.26	187260.67	168899.11
Load balancing for implicit tasks	1.0	1.0	1.0	1.0	0.98	0.99	0.99	0.99	0.99	0.99	0.98
Time in synchronization implicit tasks (average us)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Time in fork/join implicit tasks (average us)	26.39	453.18	298.31	564.03	4074.96	6320.42	7634.42	14603.72	9492.37	10874.42	10997.48

Table 3: Analysis done on Fri Dec 27 05:04:40 PM CET 2024, par2110

Modelfactor Table 3

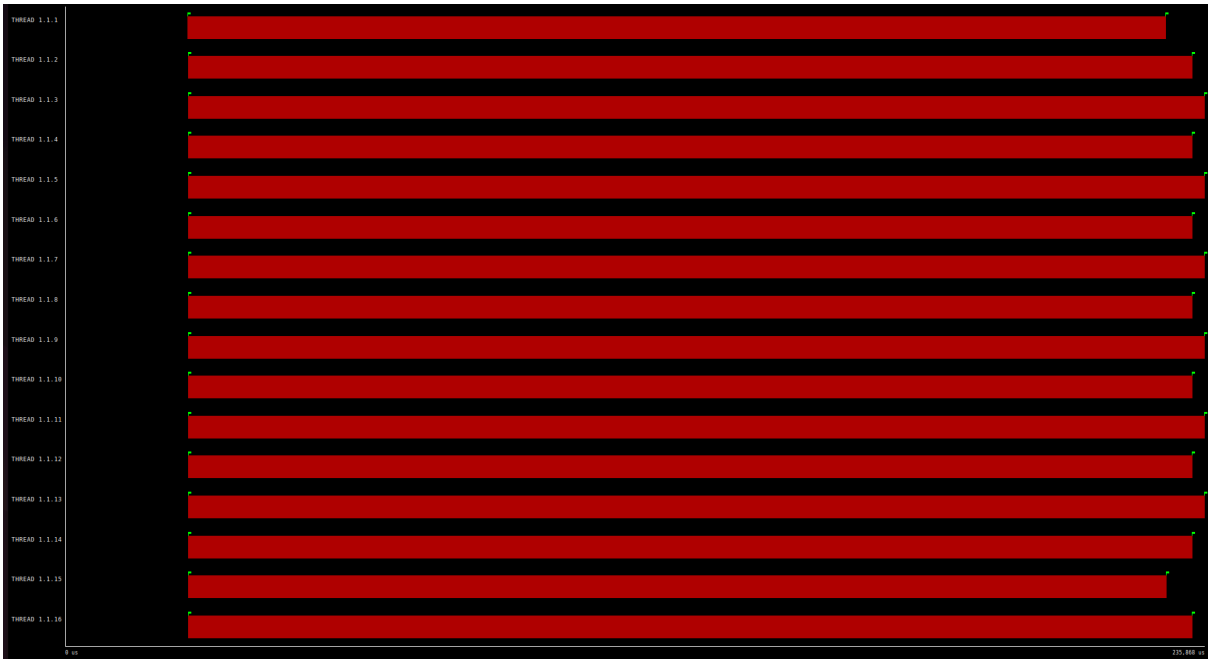
On the other tables we see that the work is much better balanced, almost perfect. Now the main problem is in scalability. So although we have reduced the unbalance by a lot we can still improve the efficiency, after having analyzed the memory (in the next section) we can deduce that this problem in scalability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

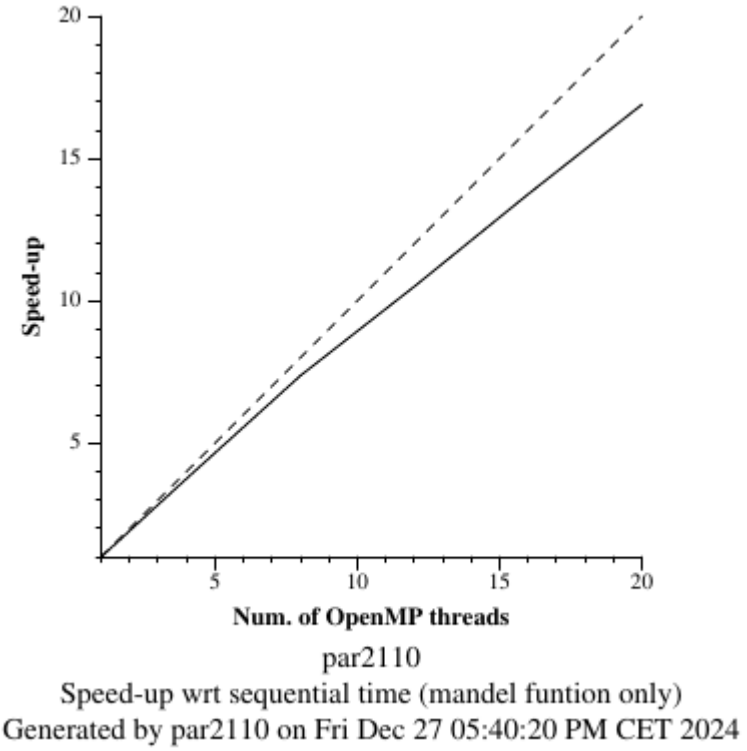
Paraver of L2 cache accesses/misses in color

	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	1,639,956
THREAD 1.1.2	2,032,564
THREAD 1.1.3	1,684,857
THREAD 1.1.4	1,987,388
THREAD 1.1.5	2,443,470
THREAD 1.1.6	2,000,705
THREAD 1.1.7	2,453,488
THREAD 1.1.8	2,090,723
THREAD 1.1.9	2,421,290
THREAD 1.1.10	2,025,926
THREAD 1.1.11	2,310,015
THREAD 1.1.12	2,093,781
THREAD 1.1.13	2,262,053
THREAD 1.1.14	2,062,145
THREAD 1.1.15	1,684,109
THREAD 1.1.16	2,015,730
Total	33,208,200
Average	2,075,512.50
Maximum	2,453,488
Minimum	1,639,956
StDev	250,637.96
Avg/Max	0.85

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).

```
C/C++
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

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.88	1.46	0.74	0.54	0.41	0.34	0.29	0.25	0.22	0.21	0.19
Speedup	1.00	1.98	3.88	5.37	7.07	8.43	9.92	11.33	12.87	14.05	15.37
Efficiency	1.00	0.99	0.97	0.90	0.88	0.84	0.83	0.81	0.80	0.78	0.77

Modelfactor Table 1

Overview of the Efficiency metrics in parallel fraction, ϕ=99.09%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	99.80%	99.12%	93.30%	92.94%	89.85%	89.58%	89.17%	88.97%	88.54%	88.11%
Parallelization strategy efficiency	100.00%	99.92%	99.67%	99.82%	99.69%	99.48%	99.28%	99.03%	98.91%	98.70%	98.44%
Load balancing	100.00%	99.95%	99.73%	99.93%	99.93%	99.80%	99.75%	99.59%	99.68%	99.71%	99.61%
In execution efficiency	100.00%	99.96%	99.94%	99.89%	99.76%	99.68%	99.53%	99.44%	99.23%	98.99%	98.83%
Scalability for computation tasks	100.00%	99.88%	99.45%	93.47%	93.23%	90.32%	90.23%	90.05%	89.95%	89.70%	89.51%
IPC scalability	100.00%	99.99%	99.97%	99.98%	99.96%	99.97%	99.98%	99.97%	99.97%	99.97%	99.96%
Instruction scalability	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Frequency scalability	100.00%	99.89%	99.48%	93.49%	93.26%	90.35%	90.25%	90.07%	89.98%	89.73%	89.54%

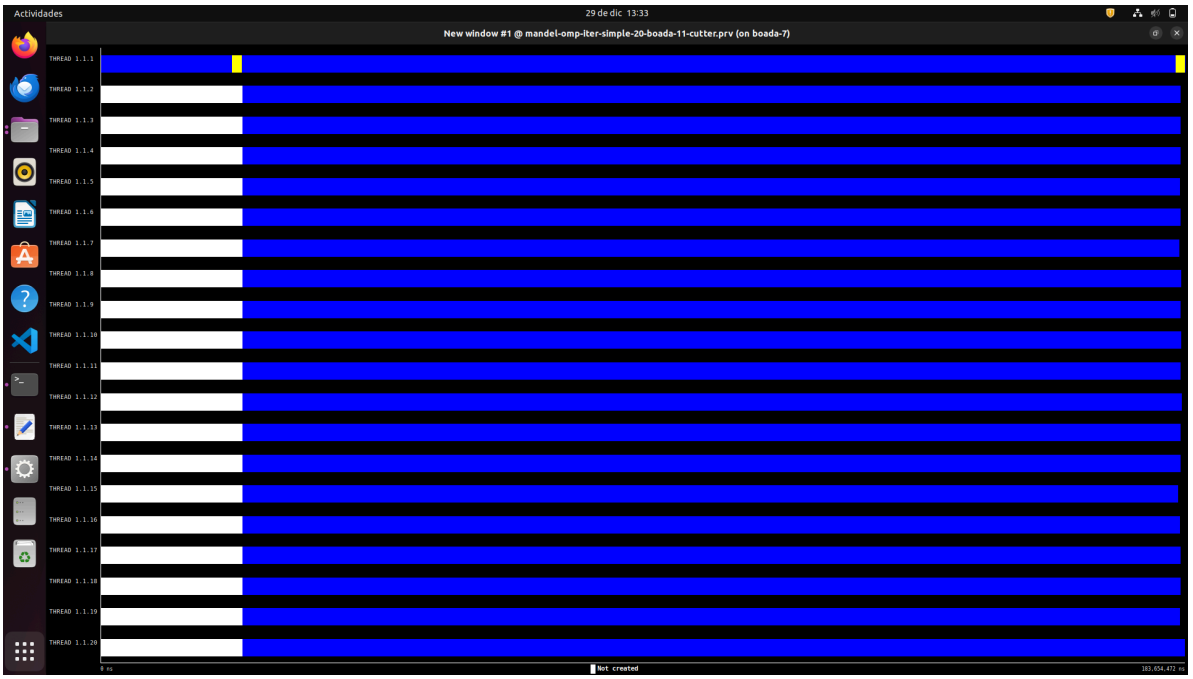
Modelfactor Table 2

Overheads in executing implicit tasks											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Number of implicit tasks per thread (average us)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Useful duration for implicit tasks (average us)	2858084.85	1430727.48	718481.78	509647.91	383199.64	316433.54	263972.72	226706.96	198577.75	177012.97	159658.96
Load balancing for implicit tasks	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Time in synchronization implicit tasks (average us)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Time in fork/join implicit tasks (average us)	23.34	1885.95	412.27	705.03	1460.39	1920.81	2573.97	2818.4	3065.89	3514.53	3974.59

Modelfactor Table3

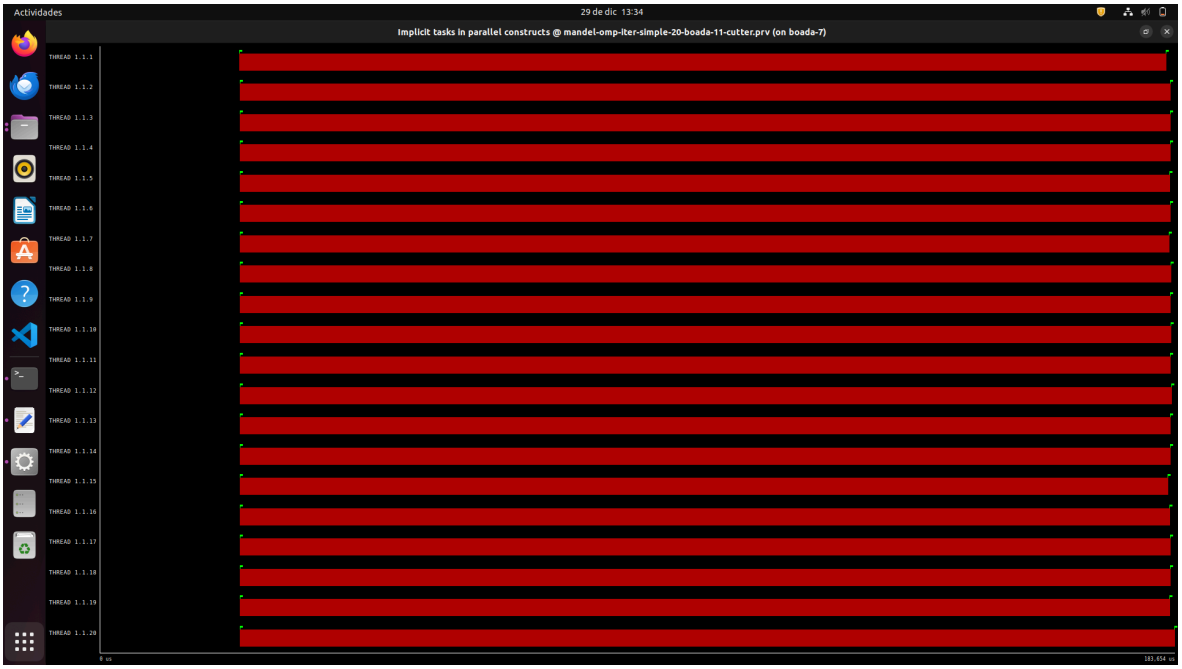
With the modelfactor tables, we can see that with this implementation we obtain a high value for the speed-up, achieving 15,37 with 20 processors. The efficiency is also good, which we can prove by watching the second table also. We have to consider though, that with more processors, this value will 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



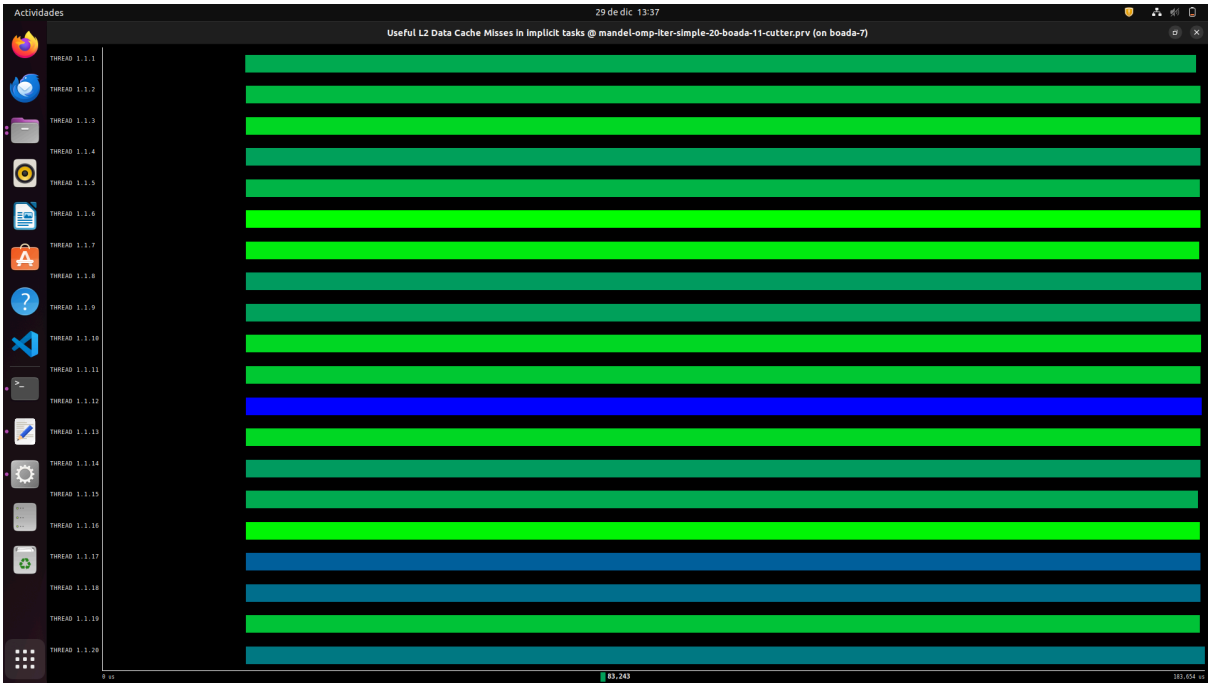
Paraver Implicit tasks in parallel constructs

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

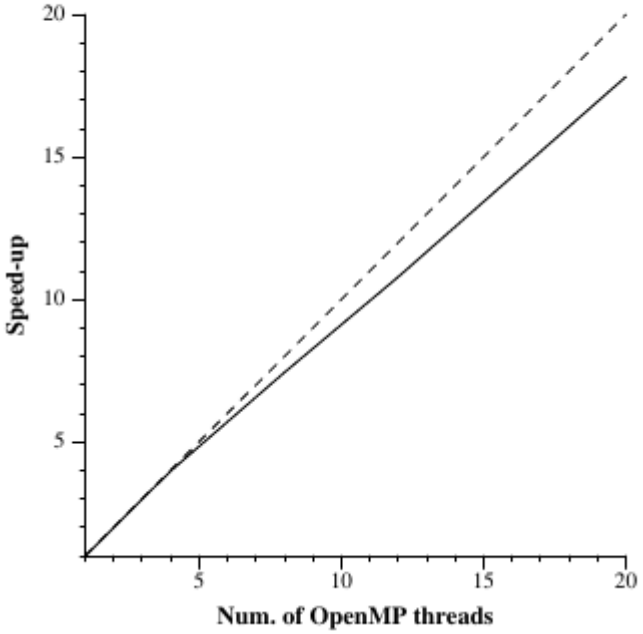
	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	83,228
THREAD 1.1.2	83,205
THREAD 1.1.3	83,154
THREAD 1.1.4	83,243
THREAD 1.1.5	83,210
THREAD 1.1.6	83,091
THREAD 1.1.7	83,122
THREAD 1.1.8	83,249
THREAD 1.1.9	83,241
THREAD 1.1.10	83,153
THREAD 1.1.11	83,177
THREAD 1.1.12	83,513
THREAD 1.1.13	83,151
THREAD 1.1.14	83,256
THREAD 1.1.15	83,231
THREAD 1.1.16	83,107
THREAD 1.1.17	83,350
THREAD 1.1.18	83,329
THREAD 1.1.19	83,188
THREAD 1.1.20	83,308
Total	1,664,506
Average	83,225.30
Maximum	83,513
Minimum	83,091
StDev	95.11
Avg/Max	1.00

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