Report

Lab 5: Parallel Data Decomposition Implementation and Analysis

Josep Antoni Martínez García
Santiago Oliver Suriñach

Index

1. Iterative task decomposition

1.1 Performance Analysis

1.1.1 - 1D Block Geometric Data Decomposition by columns

In the first place, each thread will compute BS, based on the total number of processors. Then, each CPU will iterate from its THREAD_ID*BS until completing the whole block.

Finally, we must protect access to ***histogram*** data structure and the shared use of ***X11*** system variable used to color the image, as we did in the previous lab sessions.

```
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 BS = COLS / omp_get_num_threads();
        int thread_id = omp_get_thread_num();
        int start_j = BS * thread_id;
        int end_j = start_j + BS;
        // Calcular
        for(int py = 0; py < ROWS; py++)
                for(int px = start_j; px < end_j; 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);
                                        }
                                }
                        }
                }
        }
}
```

Modelfactors

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.36	1.69	1.48	1.27	1.03	0.88	0.75	0.69	0.61	0.58	0.51
Speedup	1.00	1.40	1.59	1.86	2.30	2.68	3.15	3.45	3.85	4.09	4.59
Efficiency	1.00	0.70	0.40	0.31	0.29	0.27	0.26	0.25	0.24	0.23	0.23

Table 1: Analysis done on Fri May 24 09:44:36 AM CEST 2024, par1310

Statistics about explicit tasks in parallel fraction											
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)	2356419.55	1182560.69	605838.66	409579.37	317252.59	260209.06	221405.44	199947.71	178953.42	162556.44	152382.84
Load balancing for implicit tasks	1.0	0.7	0.41	0.33	0.31	0.3	0.3	0.3	0.3	0.29	0.3
Time in synchronization implicit tasks (average us)	0	0	0	0	0	0	0	0	0	0	0
Time in fork/join implicit tasks (average us)	20.82	0	0	0	0	0	0	0	0	0	0

Table 3: Analysis done on Fri May 24 09:44:36 AM CEST 2024, par1310

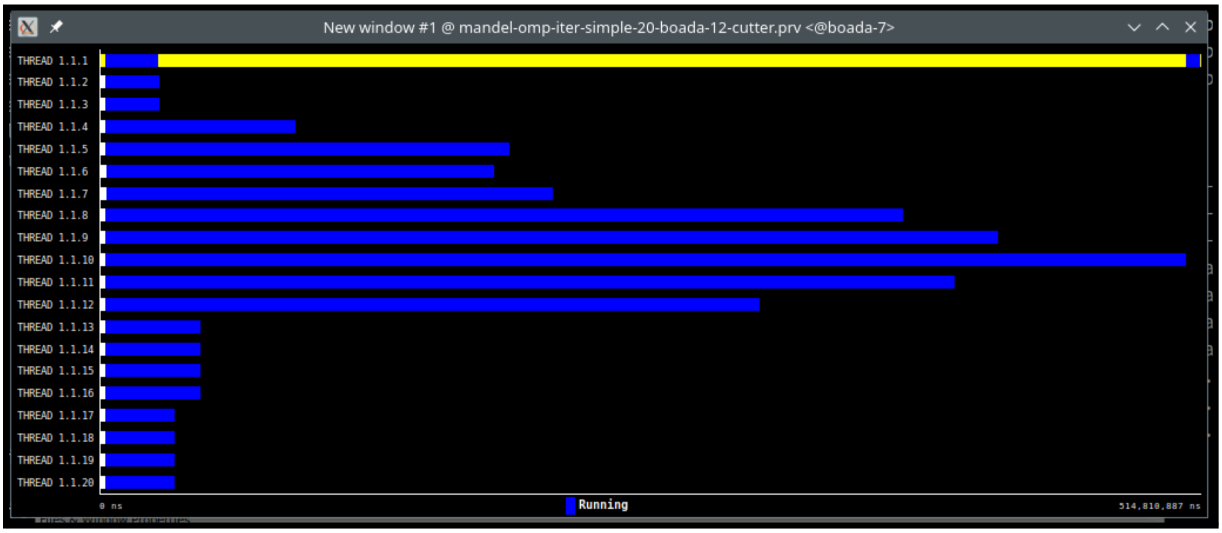
As we can see in the first image:

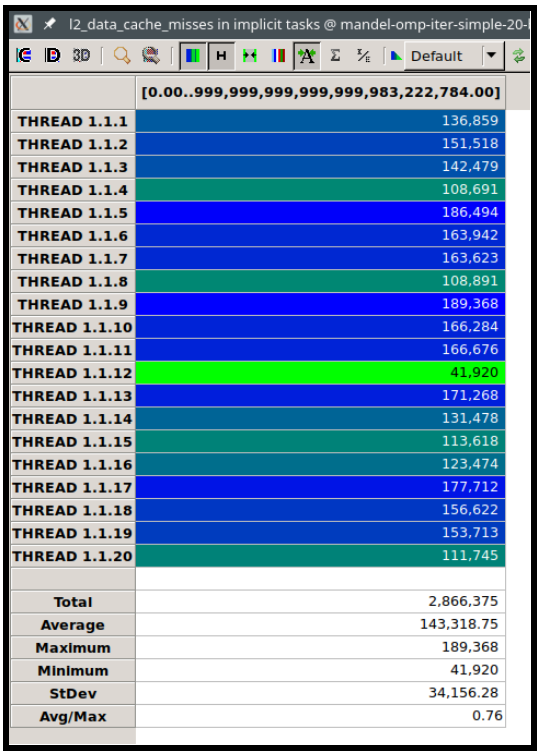
- **Elapsed time** decreases significantly as the number of processors increases.
- **Speedup** also has a positive growth, going from 1 to almost 4.6 with the total amount of CPUs.
- In terms of **efficiency** this code doesn't work properly, because after 2 processors this element is lower than the 50%.

Moving forward to the second picture:

- **Load balancing** has poor performance, because when the number of processors is increased the distribution of tasks between them is not equivalent, having values under 50% with more than 2 CPUs.

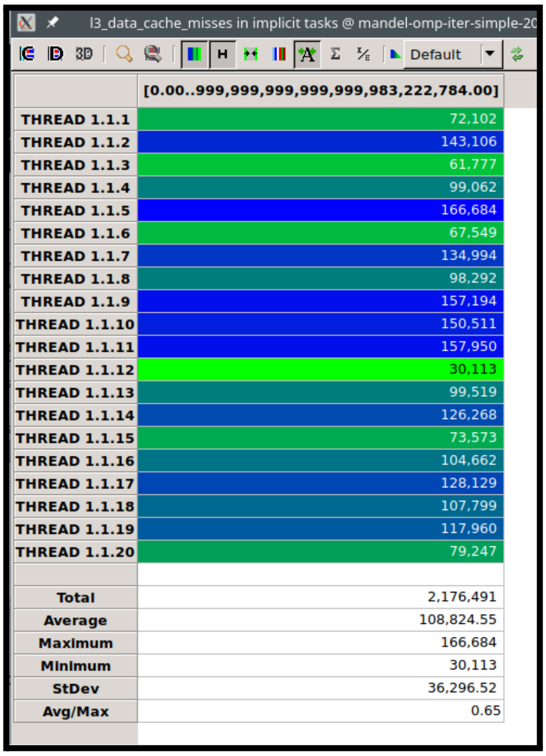
Paraver





	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	136,859
THREAD 1.1.2	151,518
THREAD 1.1.3	142,479
THREAD 1.1.4	108,691
THREAD 1.1.5	186,494
THREAD 1.1.6	163,942
THREAD 1.1.7	163,623
THREAD 1.1.8	108,891
THREAD 1.1.9	189,368
THREAD 1.1.10	166,284
THREAD 1.1.11	166,676
THREAD 1.1.12	41,920
THREAD 1.1.13	171,268
THREAD 1.1.14	131,478
THREAD 1.1.15	113,618
THREAD 1.1.16	123,474
THREAD 1.1.17	177,712
THREAD 1.1.18	156,622
THREAD 1.1.19	153,713
THREAD 1.1.20	111,745
Total	2,866,375
Average	143,318.75
Maximum	189,368
Minimum	41,920
StDev	34,156.28
Avg/Max	0.76

L2 data cache misses



	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	72,102
THREAD 1.1.2	143,106
THREAD 1.1.3	61,777
THREAD 1.1.4	99,062
THREAD 1.1.5	166,684
THREAD 1.1.6	67,549
THREAD 1.1.7	134,994
THREAD 1.1.8	98,292
THREAD 1.1.9	157,194
THREAD 1.1.10	150,511
THREAD 1.1.11	157,950
THREAD 1.1.12	30,113
THREAD 1.1.13	99,519
THREAD 1.1.14	126,268
THREAD 1.1.15	73,573
THREAD 1.1.16	104,662
THREAD 1.1.17	128,129
THREAD 1.1.18	107,799
THREAD 1.1.19	117,960
THREAD 1.1.20	79,247
Total	2,176,491
Average	108,824.55
Maximum	166,684
Minimum	30,113
StDev	36,296.52
Avg/Max	0.65

L3 data cache misses

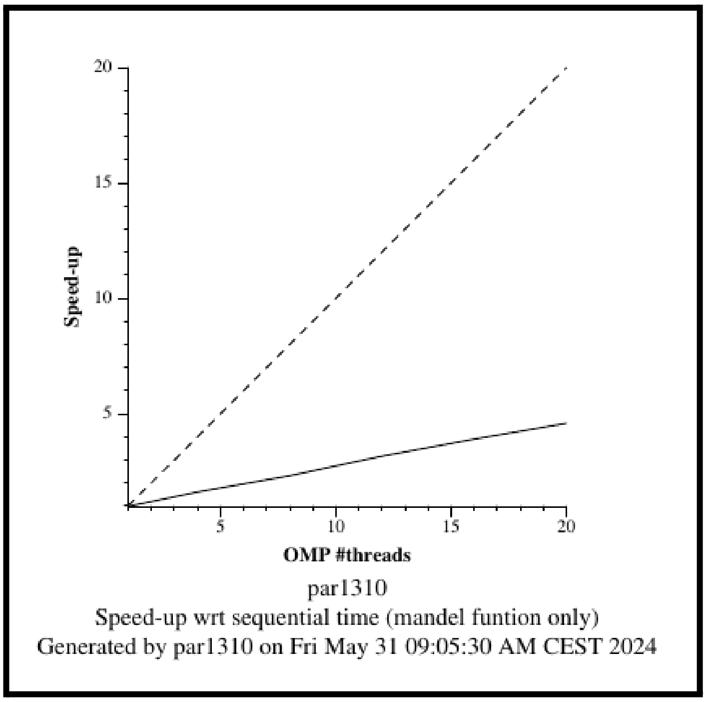
The content of the first picture shows the task distribution of each thread. As we can see the amount of execution time is not proportional among all the processors. This fact corroborates the poor load balancing values shown in Modelfactors.

The second picture shows both the L2 and L3 cache misses, the number of failed accesses is distributed unequally in both cases, and it doesn't follow any pattern.

Finally, looking at the numbers:

- **L2 Cache Misses:** The maximum number of misses are 189.368, the minimum is 41.920 and the total is 2886.375. The standard deviation has a value of 34.156, that is a high value compared with the edges.
- **L3 Cache Misses:** The maximum number is 166.684, the minimum is 30.113 and the total is 2.176. The standard deviation in this case is 36.296, also a big value.

Strong scalability



As the plot shows, the strong scalability of this strategy is not the best. This can be affirmed by the difference between the dotted and continuous lines, with the first being the perfect performance and the last being the actual execution.

1.1.2 - 1D Cyclic Geometric Data Decomposition by columns

Now, each thread is responsible for computing cols associated with its thread id, performing a cyclic assignment. For that, each thread will start at its thread id column and will jump as many threads as are available at the next iteration.
Finally, we must protect the same variables as we did before.

```
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 thread_id = omp_get_thread_num();
       int total_threads = omp_get_num_threads();
       // Calcular
       for (int py = 0; py < ROWS; py++)
       for (int px = thread_id; px < COLS; px+=total_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);
                     }
              }
              }
              }
       }
}
```

Modelfactors

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.37	1.22	0.65	0.49	0.39	0.34	0.30	0.27	0.25	0.23	0.23
Speedup	1.00	1.94	3.62	4.83	6.07	6.93	7.86	8.88	9.59	10.26	10.14
Efficiency	1.00	0.97	0.91	0.81	0.76	0.69	0.65	0.63	0.60	0.57	0.51

Table 1: Analysis done on Fri May 31 09:47:24 AM CEST 2024, par1310

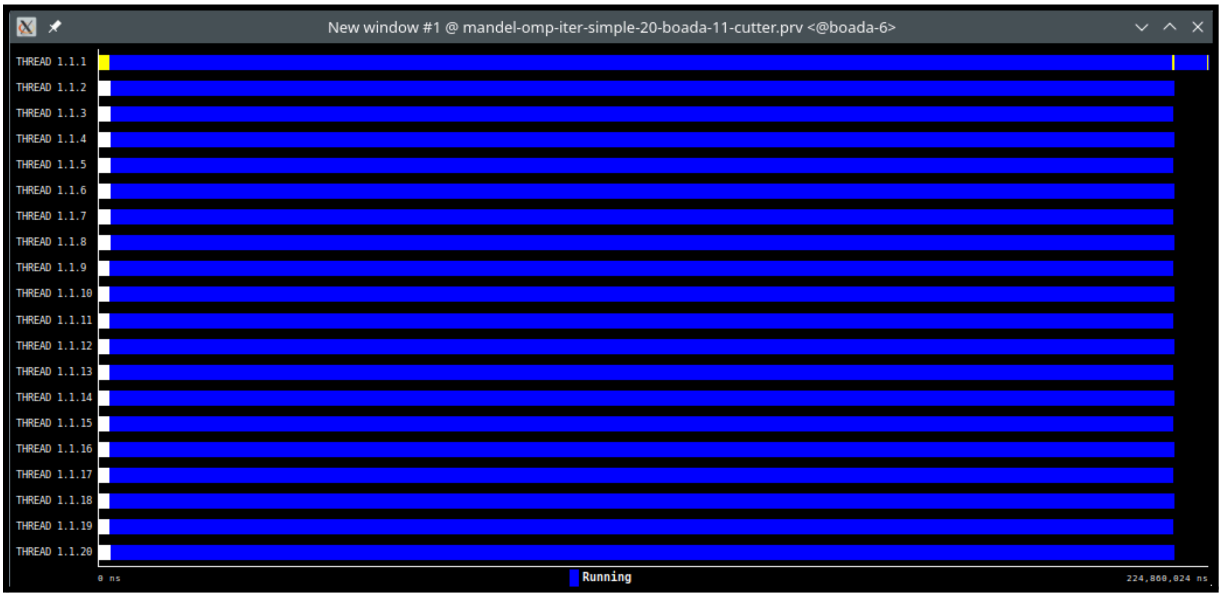
Statistics about explicit tasks in parallel fraction											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Number of implicit tasks per thread (average us)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Useful duration for implicit tasks (average us)	2358934.9	1213455.06	645175.4	481879.05	381582.11	333871.66	292581.02	258662.52	238516.91	221795.25	224308.55
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
Time in fork/join implicit tasks (average us)	54.4	0	0	0	0	0	0	0	0	0	0

Table 3: Analysis done on Fri May 31 09:47:24 AM CEST 2024, par1310

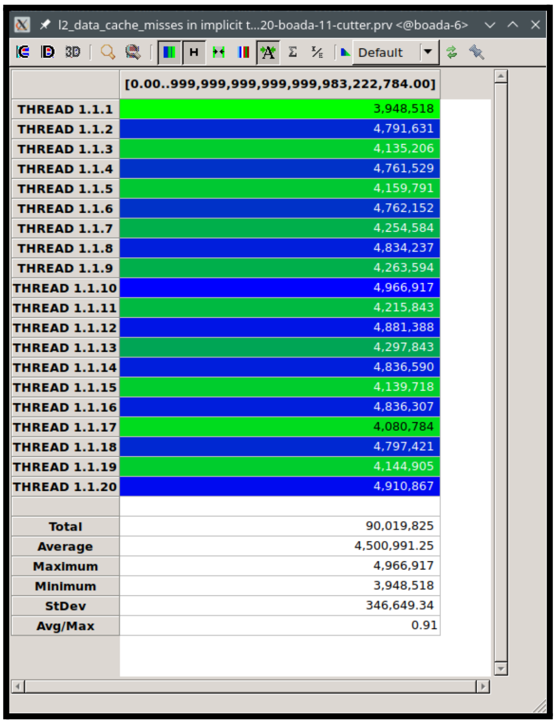
As always, the most significant values to analyze are:

- **Elapsed time:** as we can see, we get improvement in execution time up to 8-10 processors.
- **Speedup:** this value increases regularly as the number of threads increases to finally obtain a 10.14 with 20 processors, which is close to two times higher than the Block strategy.
- **Efficiency:** this time, the metric increases in general but 50% for 20 threads is still an improvable value.
- **Load balancing:** now the load balancing is 1.0 and we achieve an optimal LB, as the Paraver plot will show later.

Paraver

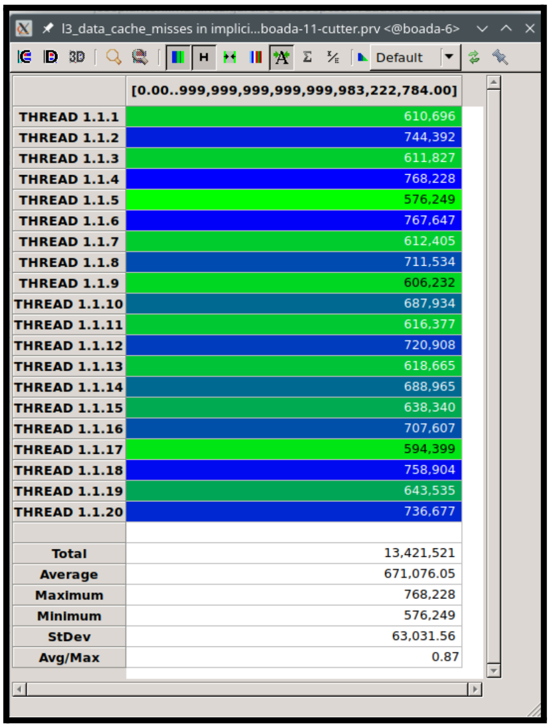


As we can see, every thread gets a matrix block and therefore we obtain a perfect task assignment between processors. It supports the obtained values with Modelfactors table as every CPU gets 1 task.



l2_data_cache_misses in implicit t...20-boada-11-cutter.prv <@boada-6>

	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	3,948,518
THREAD 1.1.2	4,791,631
THREAD 1.1.3	4,135,206
THREAD 1.1.4	4,761,529
THREAD 1.1.5	4,159,791
THREAD 1.1.6	4,762,152
THREAD 1.1.7	4,254,584
THREAD 1.1.8	4,834,237
THREAD 1.1.9	4,263,594
THREAD 1.1.10	4,966,917
THREAD 1.1.11	4,215,843
THREAD 1.1.12	4,881,388
THREAD 1.1.13	4,297,843
THREAD 1.1.14	4,836,590
THREAD 1.1.15	4,139,718
THREAD 1.1.16	4,836,307
THREAD 1.1.17	4,080,784
THREAD 1.1.18	4,797,421
THREAD 1.1.19	4,144,905
THREAD 1.1.20	4,910,867
Total	90,019,825
Average	4,500,991.25
Maximum	4,966,917
Minimum	3,948,518
StDev	346,649.34
Avg/Max	0.91



l3_data_cache_misses in implici...boada-11-cutter.prv <@boada-6>

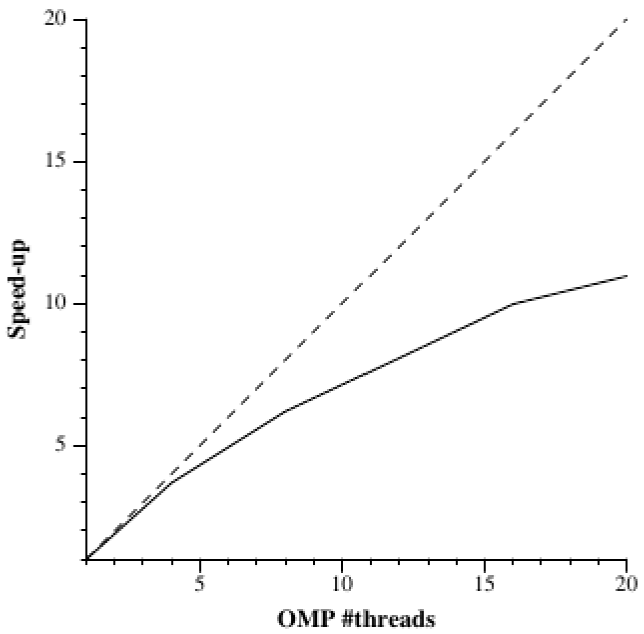
	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	610,696
THREAD 1.1.2	744,392
THREAD 1.1.3	611,827
THREAD 1.1.4	768,228
THREAD 1.1.5	576,249
THREAD 1.1.6	767,647
THREAD 1.1.7	612,405
THREAD 1.1.8	711,534
THREAD 1.1.9	606,232
THREAD 1.1.10	687,934
THREAD 1.1.11	616,377
THREAD 1.1.12	720,908
THREAD 1.1.13	618,665
THREAD 1.1.14	688,965
THREAD 1.1.15	638,340
THREAD 1.1.16	707,607
THREAD 1.1.17	594,399
THREAD 1.1.18	758,904
THREAD 1.1.19	643,535
THREAD 1.1.20	736,677
Total	13,421,521
Average	671,076.05
Maximum	768,228
Minimum	576,249
StDev	63,031.56
Avg/Max	0.87

L2 data cache misses **L3 data cache misses**

This time, as we are performing a cyclic strategy by columns, every thread makes a very large number of misses, even higher than the block strategy. The main reason is that the first CPU can read its column from the cache but the next one needs the next column, not present at the cache.
Other important values to take into account when comparing them when switching to the Cyclic Geometric Data Decomposition by **rows**:

- **Average [L2-L3]:** 4500,991.25 - 671,076.05
- **Standard deviation [L2-L3]:** 346,649.34 - 63,031.56

Strong scalability



par1310
Speed-up wrt sequential time (mandel funtion only)
Generated by par1310 on Fri May 31 09:53:23 AM CEST 2024

Compared to the previous strong scalability plot of Block Geometric Data Decomposition by columns, we get an improvement. This is due to the fact that now the computation of the blocks is assigned better. With a Block strategy, a thread can have a very low workload if in its block no computation is needed, and therefore another processor is very busy. Now, with the cyclic column assignment, the workload to each thread is distributed.

1.1.3 - 1D Cyclic Geometric Data Decomposition by rows

Referring to the code, a little change is made: every thread does the computation by rows. The same approach of the column strategy is used so the unique change is that now every thread computes its ID row.

```
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 thread_id = omp_get_thread_num();
        int total_threads = omp_get_num_threads();
        // Calcular
        for (int py = thread_id; py < ROWS; py+=total_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);
                        }
                }
                }
                }
        }
}
```

Modelfactors

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.36	1.19	0.61	0.43	0.33	0.27	0.23	0.20	0.17	0.16	0.14
Speedup	1.00	1.98	3.85	5.49	7.17	8.69	10.33	11.96	13.49	14.99	16.49
Efficiency	1.00	0.99	0.96	0.91	0.90	0.87	0.86	0.85	0.84	0.83	0.82

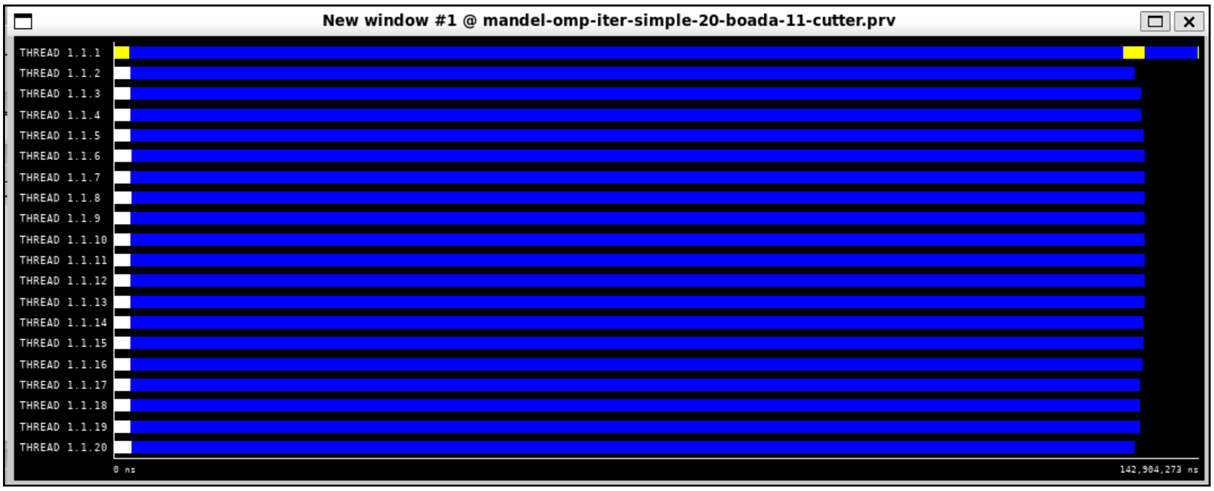
Table 1: Analysis done on Tue Jun 4 03:23:18 PM CEST 2024, par1310

Statistics about explicit tasks in parallel fraction											
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)	2351049.68	1183162.75	599081.94	421603.92	319070.94	262214.87	219231.53	188142.6	165568.7	147526.97	133058.85
Load balancing for implicit tasks	1.0	1.0	0.99	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
Time in fork/join implicit tasks (average us)	24.48	0	0	0	0	0	0	0	0	0	0

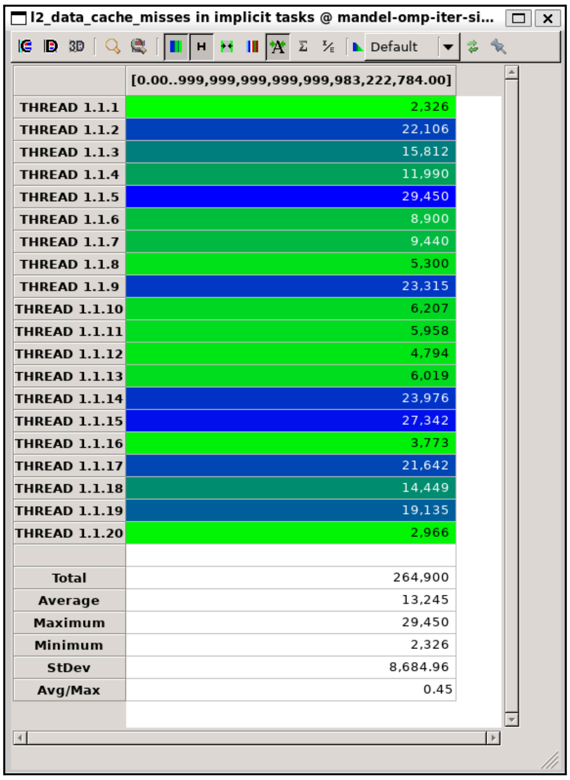
Table 3: Analysis done on Tue Jun 4 03:23:18 PM CEST 2024, par1310

As we can see, we obtain an improvement in all commented values previously. But now the efficiency is fixed and gets a very little penalty when the number of threads increases. Also, the speedup is even higher reaching 160.49%. In essence, the same pattern is found as the previous approach but reaching higher values.

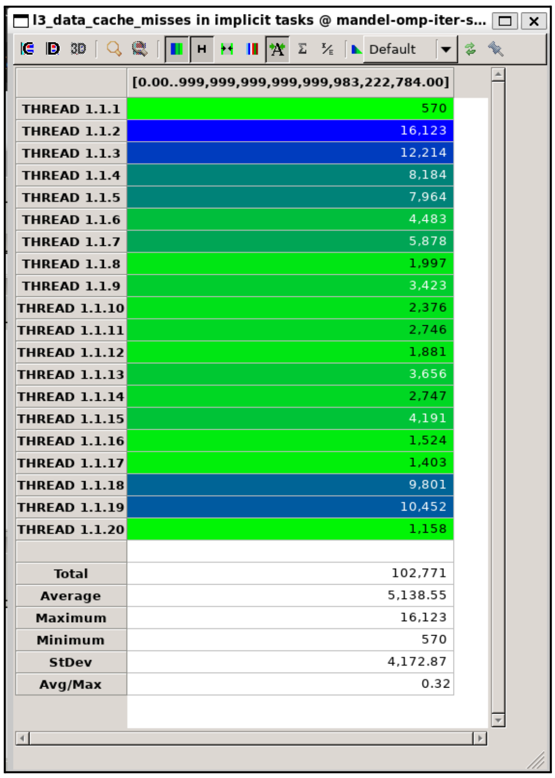
Paraver



As we can see, the task assignment is the same because each thread gets a row instead of a column, so the load balancing per thread continues being the same.


l2_data_cache_misses in implicit tasks @ mandel-omp-iter-si...

	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	2,326
THREAD 1.1.2	22,106
THREAD 1.1.3	15,812
THREAD 1.1.4	11,990
THREAD 1.1.5	29,450
THREAD 1.1.6	8,900
THREAD 1.1.7	9,440
THREAD 1.1.8	5,300
THREAD 1.1.9	23,315
THREAD 1.1.10	6,207
THREAD 1.1.11	5,958
THREAD 1.1.12	4,794
THREAD 1.1.13	6,019
THREAD 1.1.14	23,976
THREAD 1.1.15	27,342
THREAD 1.1.16	3,773
THREAD 1.1.17	21,642
THREAD 1.1.18	14,449
THREAD 1.1.19	19,135
THREAD 1.1.20	2,966
Total	264,900
Average	13,245
Maximum	29,450
Minimum	2,326
StDev	8,684.96
Avg/Max	0.45


l3_data_cache_misses in implicit tasks @ mandel-omp-iter-s...

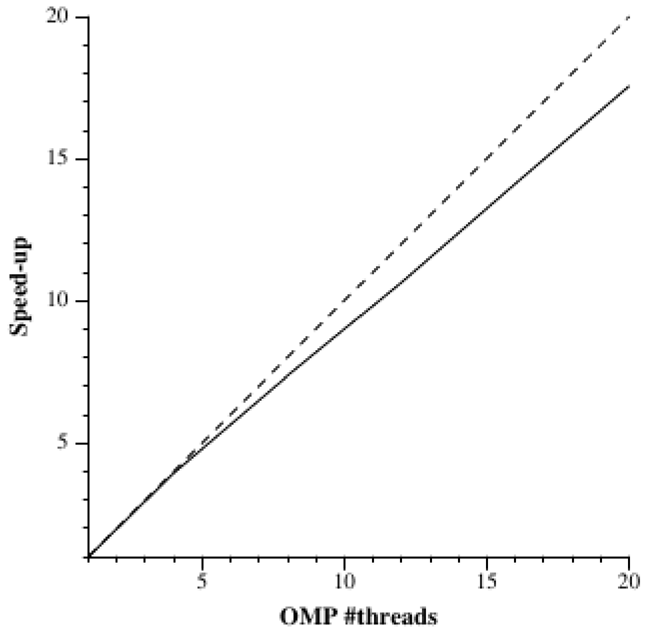
	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	570
THREAD 1.1.2	16,123
THREAD 1.1.3	12,214
THREAD 1.1.4	8,184
THREAD 1.1.5	7,964
THREAD 1.1.6	4,483
THREAD 1.1.7	5,878
THREAD 1.1.8	1,997
THREAD 1.1.9	3,423
THREAD 1.1.10	2,376
THREAD 1.1.11	2,746
THREAD 1.1.12	1,881
THREAD 1.1.13	3,656
THREAD 1.1.14	2,747
THREAD 1.1.15	4,191
THREAD 1.1.16	1,524
THREAD 1.1.17	1,403
THREAD 1.1.18	9,801
THREAD 1.1.19	10,452
THREAD 1.1.20	1,158
Total	102,771
Average	5,138.55
Maximum	16,123
Minimum	570
StDev	4,172.87
Avg/Max	0.32

L2 data cache misses **L3 data cache misses**

As we know the matrix is distributed by rows, so now the number of misses decreases significantly because each thread has the row to compute already in the cache.

The number of total misses in both caches is ridiculously little compared to the columns approach.

Strong scalability



par1310
Speed-up wrt sequential time (mandel funtion only)
Generated by par1310 on Tue Jun 4 03:33:37 PM CEST 2024

With this approach, we obtain an excellent strong scalability very close to the ideal behavior. As we mentioned before, the matrix is located in the cache by rows, so memory accesses receive a very low penalty and that explains this good performance.

2.0 - Comparison table

	Number of Threads					
Version	**1**	**4**	**8**	**12**	**16**	**20**
1D Block Geometric Data Decomposition by columns	2.36s	1.48s	1.03s	0.75s	0.61s	0.51s
1D Cyclic Geometric Data Decomposition by columns	2.37s	0.65s	0.39s	0.3s	0.25s	0.23s
1D Cyclic Geometric Data Decomposition by rows	2.36s	0.61s	0.33s	0.23s	0.16s	0.17s
	Number of threads (L2 Cache Misses per thread)					
Version	**1**	**4**	**8**	**12**	**16**	**20**
1D Block Geometric Data Decomposition by columns	18500	301698	184374	171936	143602	141847
1D Cyclic Geometric Data Decomposition by columns	19653	3520754	4936703	5428068	4980965	4501454
1D Cyclic Geometric Data Decomposition by rows	31632	33355	30826	33997	17306	13744
	Number of threads (L3 Cache Misses per thread)					
Version	**1**	**4**	**8**	**12**	**16**	**20**
1D Block Geometric Data Decomposition by columns	3839	268644	160354	147750	117656	109270
1D Cyclic Geometric Data Decomposition by columns	4475	1062434	1045274	832500	648732	671218
1D Cyclic Geometric Data Decomposition by rows	2594	18580	10120	10525	6564	5345

This table shows the results of the execution of our solution to the strategies proposed in the statement.

Looking at the first part, the execution time for only using one thread is the same for all the techniques. As the number of processors increases, we notice that the performance of both cyclic implementations is way better than the block distribution. This could be explained by the fact that some blocks may have more tiles with edges of different colors, causing a greater computational cost.

Moving forward to cache misses, we can notice that solutions divided by columns are worse than the rows strategy. This happens because the matrix is distributed by rows among the processors, so using this splitting in the code also decreases a lot of the cache misses.

In conclusion, the best strategy is “1D Cyclic Geometric Data Decomposition by Rows," which shows the lowest values both in terms of execution time and cache misses.