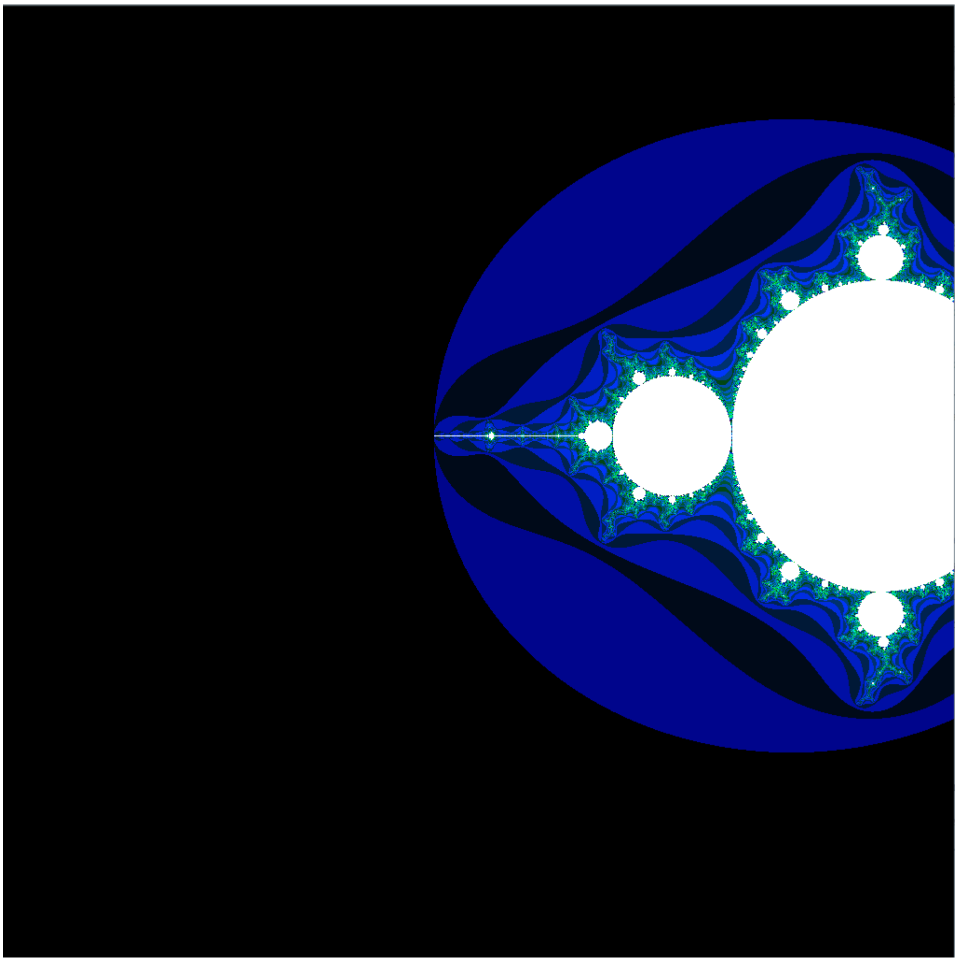
PAR Laboratory Assignment
Lab 5: Parallel Data Decomposition Implementation and Analysis



UNIVERSITAT POLITÈCNICA DE CATALUNYA
FACULTAD D'INFORMÀTICA DE BARCELONA

Nuria Lou Enseñat Balaguer (nuria.lou.ensenat)
Albert Gómez Triunfante (albert.gomez.triunfante)

Index

Introduction

In this laboratory assignment we will study three different geometric data decomposition strategies for computing the Mandelbrot set.

This report will analyse the strategies implemented: 1D block geometric data decomposition by columns, 1D block-cyclic geometric data decomposition by columns and 1D cyclic geometric data decomposition by rows. We will compare them to understand their impact on performance and scalability and determine which of the iterative strategies provides the best performance for this problem and why.

By exploring these different task decomposition approaches, this report seeks to provide a comprehensive understanding of the strengths and limitations of each parallelisation strategy.

1D Block Geometric Data Decomposition by Columns

Code

File name: mandel-omp-iter-simple-block.cpp

```cpp
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 my_id =  omp_get_thread_num();
        int howmany = omp_get_num_threads();
        int BS = COLS / howmany;
        int start = my_id  * BS;
        int end = start + BS;

        if (my_id == (howmany - 1))
            end = COLS;

        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);
                  }
              }
            }
          }
        }
    }
}
```

For this version of the code, we added the implicit tasks and its management. We also added the #pragma omp atomic and #pragma omp critical commands, to protect the variables from being modified by other tasks, as we have been doing in previous labs..

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)	2.89	2.10	1.83	1.58	1.28	1.11	0.95	0.86	0.76	0.72	0.65
Speedup	1.00	1.38	1.58	1.83	2.26	2.62	3.05	3.37	3.79	4.02	4.48
Efficiency	1.00	0.69	0.39	0.30	0.28	0.26	0.25	0.24	0.24	0.22	0.22

Table 1: Analysis done on Thu Dec 5 01:33:31 PM CET 2024, par1307

Overview of the Efficiency metrics in parallel fraction, ϕ=99.10%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	69.20%	39.64%	30.66%	28.56%	26.59%	25.85%	24.55%	24.19%	23.00%	23.06%
Parallelization strategy efficiency	100.00%	69.24%	39.73%	31.41%	29.84%	28.28%	27.83%	26.60%	26.43%	25.31%	25.63%
Load balancing	100.00%	69.25%	39.74%	31.43%	29.86%	28.32%	27.87%	26.64%	26.49%	25.37%	25.70%
In execution efficiency	100.00%	99.99%	99.97%	99.95%	99.93%	99.88%	99.86%	99.84%	99.78%	99.76%	99.73%
Scalability for computation tasks	100.00%	99.94%	99.78%	97.62%	95.73%	93.99%	92.88%	92.32%	91.52%	90.86%	89.96%
IPC scalability	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	99.99%	99.98%	99.99%	99.78%	99.61%
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%	99.94%	99.78%	97.62%	95.74%	94.00%	92.89%	92.34%	91.53%	91.07%	90.31%

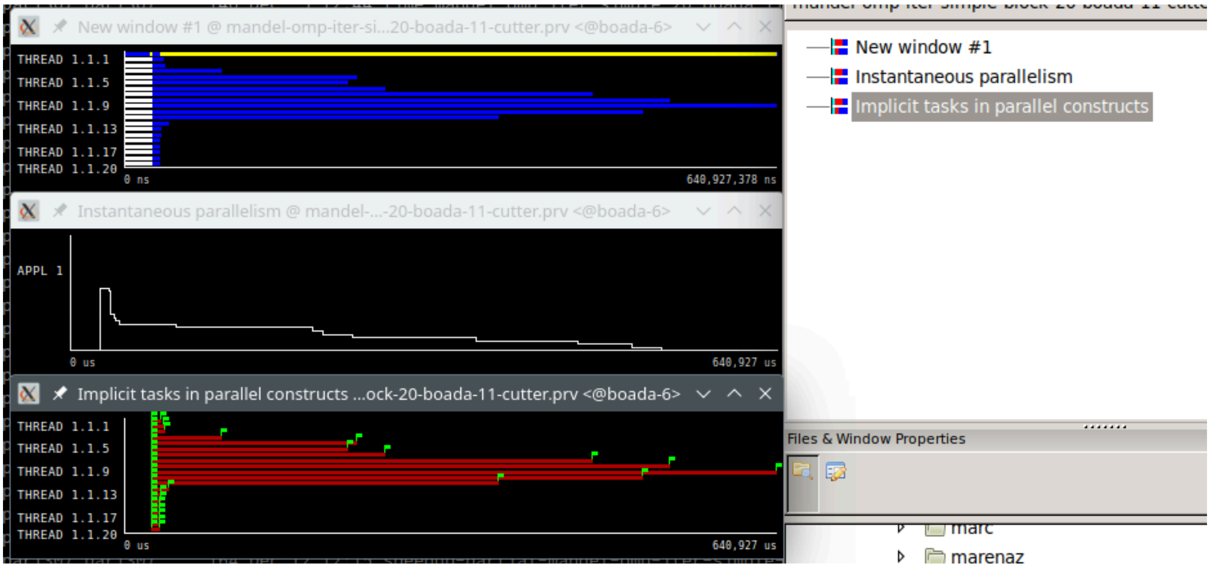
Table 2: Analysis done on Thu Dec 5 01:33:31 PM CET 2024, par1307

From the Modelfactor Analysis we can observe that both the speedup and the elapsed time improvement follows a much less than ideal improvement with respect to the x=y line, which is the optimal standard that we're aiming towards.

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)	2867837.67	1434757.89	718562.25	489629.51	374467.76	305108.91	257310.33	221886.06	195856.31	175353.15	159403.06
Load balancing for implicit tasks	1.0	0.69	0.4	0.31	0.3	0.28	0.28	0.27	0.26	0.25	0.26
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)	25.12	0	0	0	0	0	0	0	0	0	0

Table 3: Analysis done on Thu Dec 5 01:33:31 PM CET 2024, par1307

Paraver analysis



In this execution of wxparaver over this particular parallel implementation, we can see all the tasks created and their behaviour throughout their execution. The graphs look pretty unbalanced, therefore there is a big Load Unbalance or task creation overhead problem causing a bottleneck that can be seen by the naked eye on these graphs.

Memory analysis

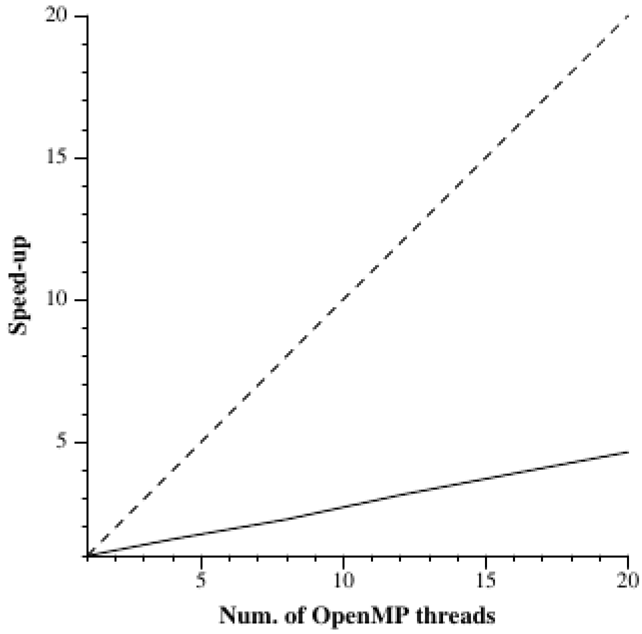
L2 MISSES		
Num threads	**Overall number of cache misses**	**Avg number of misses per thread**
1	1642228	1642228
2	1764416	882208
4	1971540	492885
6	2186016	364336
8	2332868	291608
10	2285110	228511
12	2725749	227145
14	2827944	201996
16	2697909	168619
18	2863210	159067

20	2997797	149889

What can we extract from this correlation? Basically, we can see that the average number of misses per cache is equal to the division of the total number of misses divided by the number of tasks. This represents a fully linear behaviour.

	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	206,235
THREAD 1.1.2	202,126
THREAD 1.1.3	134,079
THREAD 1.1.4	144,083
THREAD 1.1.5	201,206
THREAD 1.1.6	154,584
THREAD 1.1.7	124,130
THREAD 1.1.8	201,334
THREAD 1.1.9	173,949
THREAD 1.1.10	113,481
THREAD 1.1.11	201,831
THREAD 1.1.12	193,664
THREAD 1.1.13	113,546
THREAD 1.1.14	202,821
THREAD 1.1.15	204,047
THREAD 1.1.16	113,933
Total	2,685,049
Average	167,815.56
Maximum	206,235
Minimum	113,481
StDev	36,978.23
Avg/Max	0.81

Strong scalability analysis



par1307
Speed-up wrt sequential time (mandel funtion only)
Generated by par1307 on Thu Dec 12 12:14:56 PM CET 2024

In this graph we can clearly see that the graph, though linear, is far from the optimal x=y shape. This makes us expect that the following approaches will be better.

1D Block-Cyclic Geometric Data Decomposition by columns

Code

File name: mandel-omp-iter-simple-block-cyclic.cpp

```
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 my_id = omp_get_thread_num();
        int how_many = omp_get_num_threads();
        int start = my_id;
      // Calcular
      for (int py = 0; py < ROWS; py++) {
        for (int px = start; px < COLS; px+=how_many) {
          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);
                }

                }
          }
        }
      }
    }
}
```

For this version of the code, we added the implicit tasks and its management. We also added the #pragma omp atomic and #pragma omp critical commands, to protect the variables from being modified by other tasks. Furthermore, we added the logic in the appropiate for loop so that tasks get assignated blocks of "how_many" columns.

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)	2.89	1.46	0.77	0.54	0.41	0.35	0.30	0.26	0.24	0.21	0.20
Speedup	1.00	1.98	3.74	5.37	6.98	8.21	9.74	10.98	11.94	13.47	14.17
Efficiency	1.00	0.99	0.94	0.90	0.87	0.82	0.81	0.78	0.75	0.75	0.71

Table 1: Analysis done on Thu Dec 12 12:45:43 PM CET 2024, par1307

Overview of the Efficiency metrics in parallel fraction, ϕ=99.07%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	99.69%	96.01%	92.55%	91.75%	87.30%	86.81%	85.65%	82.81%	83.52%	79.47%
Parallelization strategy efficiency	100.00%	99.93%	98.26%	99.46%	99.44%	98.65%	98.81%	98.70%	98.29%	98.14%	95.13%
Load balancing	100.00%	99.96%	98.34%	99.59%	99.64%	98.94%	99.19%	99.23%	99.00%	98.98%	96.06%
In execution efficiency	100.00%	99.97%	99.92%	99.87%	99.80%	99.71%	99.62%	99.46%	99.29%	99.15%	99.04%
Scalability for computation tasks	100.00%	99.76%	97.71%	93.06%	92.26%	88.49%	87.86%	86.78%	84.25%	85.10%	83.53%
IPC scalability	100.00%	99.86%	99.67%	99.37%	98.71%	97.92%	97.28%	96.24%	93.61%	94.66%	93.17%
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%	99.90%	98.03%	93.65%	93.47%	90.37%	90.32%	90.17%	90.01%	89.91%	89.66%

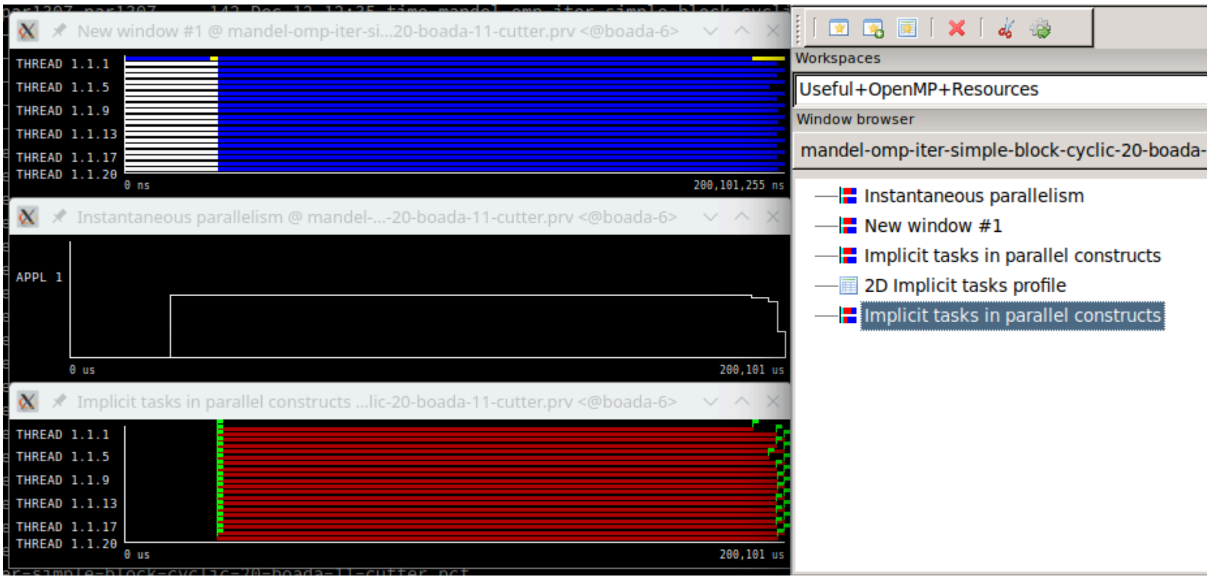
Table 2: Analysis done on Thu Dec 12 12:45:43 PM CET 2024, par1307

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)	2858827.68	1432852.27	731486.61	512027.4	387324.1	323073.56	271165.77	235310.01	212072.35	186635.9	171126.03
Load balancing for implicit tasks	1.0	1.0	0.98	1.0	1.0	0.99	0.99	0.99	0.99	0.99	0.96
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)	30.89	0	0	0	0	0	0	0	0	0	0

Table 3: Analysis done on Thu Dec 12 12:45:43 PM CET 2024, par1307

From the Modelfactor Analysis we can observe that this implementation is better than the previous one. The shape is now not going to be linear, but much closer to the ideal. Regarding the speedup and efficiency, we can see that in a small number of threads they are really close to ideal, but as we increment the number of threads the distance between the real and the ideal value increases.

Paraver analysis



This paraver analysis shows a much better distribution of tasks. Free of load Unbalance, the tasks perform at almost their beast for the entire time.

Memory analysis

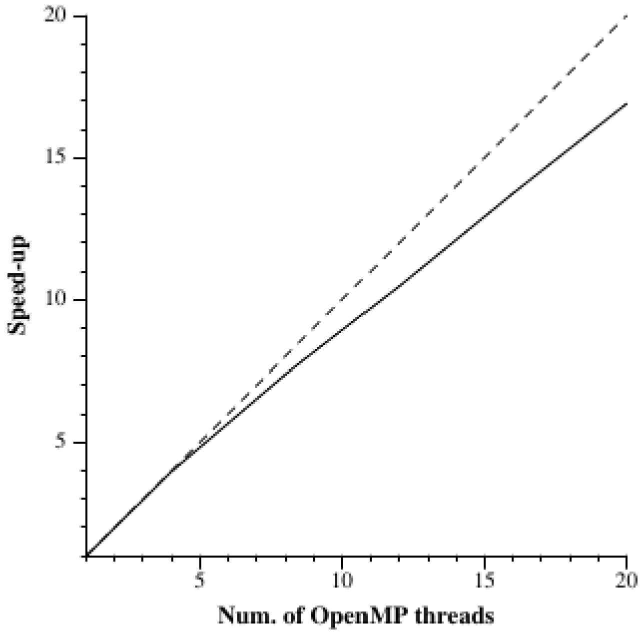
L2 MISSES		
Num threads	**Overall number of cache misses**	**Avg number of misses per thread**
1	1642105	1642105
2	3314533	1657266
4	7275511	1818877
6	11077248	1846208
8	16413780	2051722
10	20360807	2036080
12	25791694	2149307
14	29173807	2083843
16	33310525	2081907
18	38068795	2114933

20	44014784	2200739

	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	1,640,115
THREAD 1.1.2	1,994,632
THREAD 1.1.3	2,216,515
THREAD 1.1.4	1,994,013
THREAD 1.1.5	2,305,060
THREAD 1.1.6	1,955,175
THREAD 1.1.7	2,293,632
THREAD 1.1.8	2,002,307
THREAD 1.1.9	2,321,275
THREAD 1.1.10	1,985,578
THREAD 1.1.11	2,176,638
THREAD 1.1.12	1,919,753
THREAD 1.1.13	2,235,511
THREAD 1.1.14	2,037,975
THREAD 1.1.15	2,213,077
THREAD 1.1.16	2,010,100
Total	33,301,356
Average	2,081,334.75
Maximum	2,321,275
Minimum	1,640,115
StDev	175,959.63
Avg/Max	0.90

What can we extract from this correlation? Basically, we can see that the average number of misses per cache is equal to the division of the total number of misses divided by the number of tasks. This represents a fully linear behaviour.

Strong scalability analysis



par1307
Speed-up wrt sequential time (mandel funtion only)
Generated by par1307 on Thu Dec 12 12:50:25 PM CET 2024

For this decomposition we can see that the graph looks almost like a x=y optimal scenario. However, there's still room for improvement, as we'll see in the following and last strategy.

1D Cyclic Geometric Data Decomposition by rows

Code

File name: mandel-omp-iter-simple-cyclic.cpp

```
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 my_id = omp_get_thread_num();
        int how_many = omp_get_num_threads();
        int start = my_id;
      // Calcular
      for (int py = start; py < ROWS; py+=how_many) {
        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);
                }

                }
          }
        }
      }
    }
}
```

In order to implement the 1D cyclic geometric decomposition by rows we created the implicit tasks by adding the #pragma omp parallel directive to the mandel_simple function's body. Then we created three different variables: my_id, how_many and start, which will make sure that the loops will be executed by rows. As in the other implementations, we also added the directives #pragma omp atomic and #pragma omp critical in order to avoid dataraces problems.

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)	2.89	1.46	0.74	0.54	0.40	0.34	0.29	0.25	0.22	0.21	0.19
Speedup	1.00	1.98	3.88	5.38	7.13	8.44	10.01	11.40	12.94	14.04	15.41
Efficiency	1.00	0.99	0.97	0.90	0.89	0.84	0.83	0.81	0.81	0.78	0.77

Table 1: Analysis done on Thu Dec 12 01:02:38 PM CET 2024, par1307

In this first table we can see that this version is the most efficient one out of all the ones we have studied in this lab, where the biggest improvement is seen when comparing it to the 1D block geometric data decomposition by columns.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.10%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	99.90%	99.38%	93.48%	93.21%	89.94%	89.72%	89.11%	89.04%	88.45%	87.89%
Parallelization strategy efficiency	100.00%	99.97%	99.81%	99.81%	99.69%	99.47%	99.36%	98.87%	98.93%	98.52%	98.18%
Load balancing	100.00%	99.99%	99.85%	99.92%	99.93%	99.82%	99.80%	99.56%	99.73%	99.60%	99.22%
In execution efficiency	100.00%	99.97%	99.96%	99.89%	99.76%	99.66%	99.56%	99.30%	99.19%	98.91%	98.95%
Scalability for computation tasks	100.00%	99.94%	99.57%	93.65%	93.50%	90.42%	90.30%	90.13%	90.00%	89.78%	89.52%
IPC scalability	100.00%	99.99%	99.97%	99.97%	99.97%	99.97%	99.98%	99.97%	99.98%	99.97%	99.94%
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%	99.95%	99.60%	93.68%	93.53%	90.44%	90.32%	90.16%	90.03%	89.80%	89.57%

Table 2: Analysis done on Thu Dec 12 01:02:38 PM CET 2024, par1307

In this second table we can see the efficiency in parallel fractions, where we can notice that there is a high scalability as well as a very good load balancing, meaning that the parallelisation strategy is very efficient.

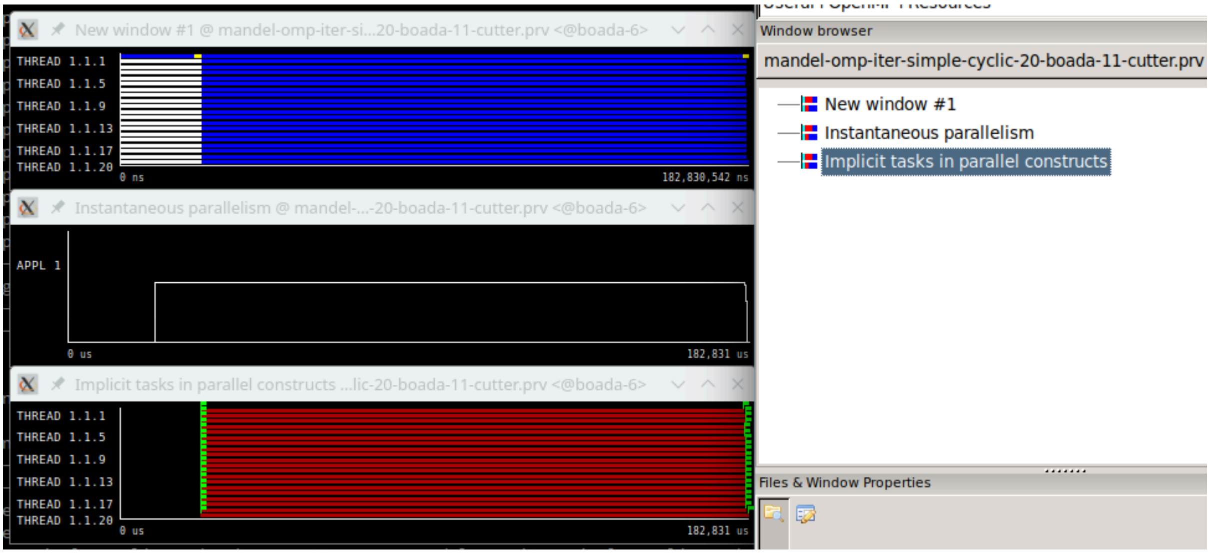
Statistics about explicit tasks in parallel fraction											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Number of implicit tasks per thread (average us)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Useful duration for implicit tasks (average us)	2859967.85	1430905.64	718092.78	508964.87	382344.91	316312.2	263934.98	226646.31	198600.05	176980.85	159747.61
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	0.99
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)	30.62	0	0	0	0	0	0	0	0	0	0

Table 3: Analysis done on Thu Dec 12 01:02:38 PM CET 2024, par1307

In the third and last table we can see the statistics about explicit tasks, where we see an even load balancing among the different number of processors used.

By taking all of these results into account, we can expect an implementation very close to the ideal.

Paraver analysis



By looking at the paraver windows we see a good distribution of tasks, without load unbalancing. Following our assumptions, we can expect an optimal parallelisation from this strategy.

Memory analysis

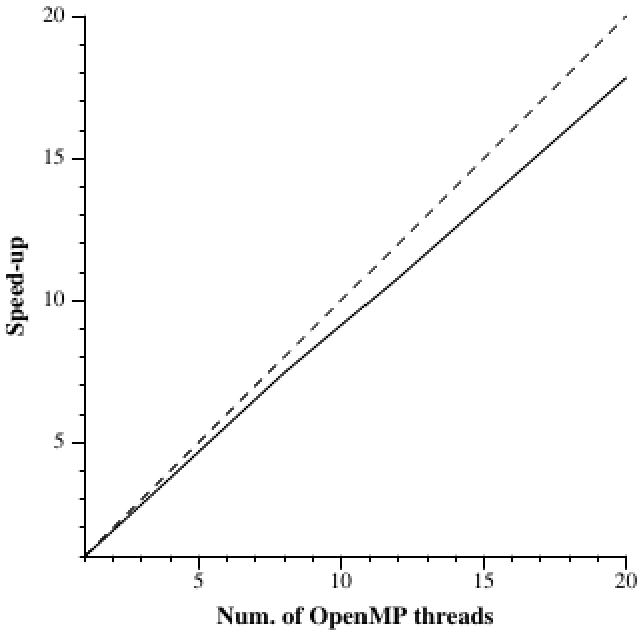
L2 MISSES		
Num threads	**Overall number of cache misses**	**Avg number of misses per thread**
1	1642296	1642296
2	1646966	823483
4	1649961	412490
6	1652808	275468
8	1654994	206874
10	1659407	165940
12	1661450	138454
14	1666256	119018
16	1662728	103920
18	1670890	92827

20	1674185	83709

	[0.00..999,999,999,999,999,983,222,784.00]
THREAD 1.1.1	103,513
THREAD 1.1.2	103,378
THREAD 1.1.3	103,373
THREAD 1.1.4	103,406
THREAD 1.1.5	103,348
THREAD 1.1.6	103,603
THREAD 1.1.7	103,407
THREAD 1.1.8	103,315
THREAD 1.1.9	103,260
THREAD 1.1.10	103,201
THREAD 1.1.11	103,417
THREAD 1.1.12	103,479
THREAD 1.1.13	103,297
THREAD 1.1.14	103,351
THREAD 1.1.15	103,218
THREAD 1.1.16	103,557
Total	1,654,123
Average	103,382.69
Maximum	103,603
Minimum	103,201
StDev	110.94
Avg/Max	1.00

We see a linear behaviour as the total number of misses/number of threads = average number of misses. This is the same as saying that the number of tasks created is directly related to the number of misses.

Strong scalability analysis



par1307
Speed-up wrt sequential time (mandel funtion only)
Generated by par1307 on Thu Dec 12 01:12:36 PM CET 2024

As we have mentioned during the analysis of this implementation, we can see that its graph is the closest to the perfect linear tendency meaning that it is the best one out of the three strategies seen in this report.

Summary of the elapsed execution times

	Number of threads (elapsed)					
Version	**1**	**4**	**8**	**12**	**16**	**20**
1D Block Geometric Data Decomposition by columns	2.867647	1.806438	1.254591	0.910142	0.732974	0.614575
1D Block-Cyclic Geometric Data Decomposition by columns	2.856705	0.719533	0.387473	0.272767	0.207882	0.169048
1D Cyclic Geometric Data Decomposition by rows	2.857920	0.761692	0.383473	0.264496	0.199432	0.160141
	Number of threads (L2 Cache Misses per thread)					
1D Block Geometric Data Decomposition by columns	1642228	492885	291608	227145	168619	149889
1D Block-Cyclic Geometric Data Decomposition by columns	1642105	1818877	2051722	2149307	2081907	2200739
1D Cyclic Geometric Data Decomposition by rows	1642296	412490	206874	138454	103920	83709

Table: Summary of the elapsed execution times for each of the versions, obtained from the output files after the execution of submit-strong-omp.sh script. Average L2 cache misses per thread are obtained from the memory analysis script execution.