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

Oriol Ramos, Saúl Vera

Spring 2023-24

INDEX

1. Geometric Data Decomposition Strategies

In this laboratory session, we will explore geometric data decomposition strategies. We have used the computation of the Mandelbrot set that is the representation of a particular set of points, in the complex domain, whose boundary generates the next two-dimensional fractal shape.

To execute the different programs, we have some parameters that are good to know before. This are:

-o to write computed image (mandel_image.jpg) and histogram (mandel_histogram.out if -h indicated) \ to disk (default no file generated)

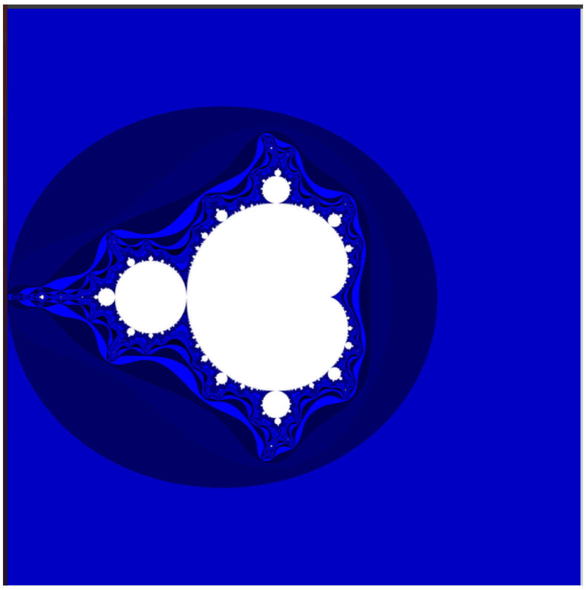
-h to produce histogram of values in computed image (default no histogram)
-d to display computed image (default no display)
-i to specify maximum number of iterations at each point (default 100)
-c to specify the center x0+iy0 of the square to compute (default origin)
-s to specify the size of the square to compute (default 2, i.e. size 4 by 4)

First of all we-re going to implement an iterative data decomposition code using a 1D Block Geometric Data Decomposition by columns. After that we will implement it using 1D Cyclic Geometric Data Decomposition by columns and finally using a 1D Cyclic Geometric Data Decomposition by rows.

To start improving the strategies we have a sequential code so we can constantly check if our codes are being correctly implemented. We just need to compare the histogram output of the sequential code and our parallel code and observe if the results are the same.
With the command:

- sbatch submit-seq.sh mandel-seq-iter -h -o -i 10000

We generated the histogram and got this result:



1.1. 1D Block Geometric Data Decomposition by columns

1.1.1. Execution and check the correctness

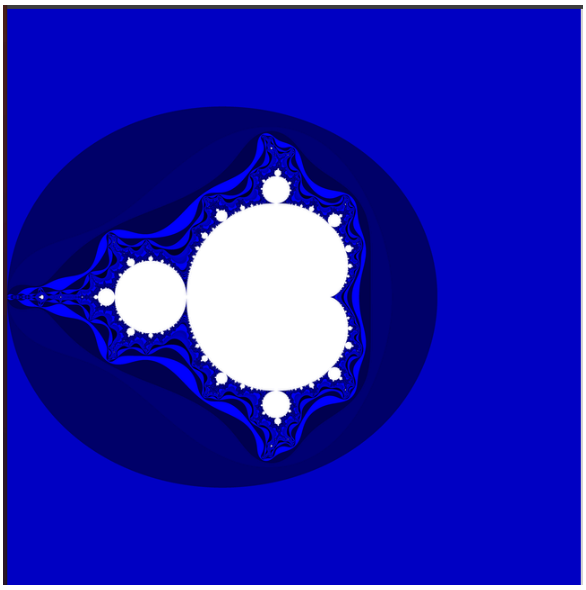
We then started to get the correct version of the block. We added #pragma omp parallel in order to make the code actually parallel. Apart from that, we added #pragma omp atomic to the parts where we calculate the histogram and #pragma omp critical when we had the dependencies of X11_COLOR_fake. This was the same procedure as in lab3 and lab4. Finally, we added different lines in the mandel_simple function in order to distribute the tasks between the different threads:

- int my_id = omp_get_thread_num();
- int howmany = omp_get_num_threads();
- int BS = COLS/howmany;
- int index_start = my_id*BS;
- int index_end = (my_id+1)*BS;

And we modified the double loop in order to make the decomposition by columns:

- for (int px = index_start; px < index_end; px++)
- for (int py = 0; py < ROWS; py++)

We proceeded to compare the outputs when generating the image on both of them and the result was identical to the image showed before:



We also used the command:

- sbatch submit-omp.sh mandel-omp-iter-simple 20

To get the execution time with 20 processors. The result is seen on the table below (elapsed time).

1.1.2. Performance Analysis

The second part of every code consisted in analyzing the performance of the strategy. We used the modelfactor tables, paraver analysis, cache misses analysis and the strong scalability graphic.

We're going to start with the modelfactor tables:

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.51	1.77	1.53	1.31	1.09	0.92	0.79	0.72	0.64	0.59	0.54
Speedup	1.00	1.42	1.64	1.91	2.31	2.74	3.20	3.51	3.95	4.23	4.62
Efficiency	1.00	0.71	0.41	0.32	0.29	0.27	0.27	0.25	0.25	0.23	0.23

Table 1: Analysis done on Tue May 14 11:13:44 AM CEST 2024, par1116

Overview of the Efficiency metrics in parallel fraction, ϕ=99.82%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	71.09%	41.09%	31.95%	28.96%	27.57%	26.84%	25.26%	24.89%	23.71%	23.35%
Parallelization strategy efficiency	100.00%	71.48%	43.20%	34.92%	33.50%	32.90%	32.96%	32.53%	32.72%	33.00%	33.57%
Load balancing	100.00%	71.49%	43.22%	34.95%	33.54%	32.93%	33.02%	32.61%	32.81%	33.09%	33.70%
In execution efficiency	100.00%	99.99%	99.94%	99.94%	99.90%	99.89%	99.82%	99.75%	99.72%	99.73%	99.61%
Scalability for computation tasks	100.00%	99.45%	95.13%	91.49%	86.45%	83.81%	81.45%	77.67%	76.08%	71.84%	69.57%
IPC scalability	100.00%	99.53%	97.30%	95.96%	94.98%	93.98%	93.93%	93.91%	93.71%	94.39%	92.02%
Instruction scalability	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.02%	100.02%	100.00%	100.02%	100.00%
Frequency scalability	100.00%	99.92%	97.77%	95.33%	91.01%	89.18%	86.71%	82.69%	81.18%	76.09%	75.61%

Table 2: Analysis done on Tue May 14 11:13:44 AM CEST 2024, par1116

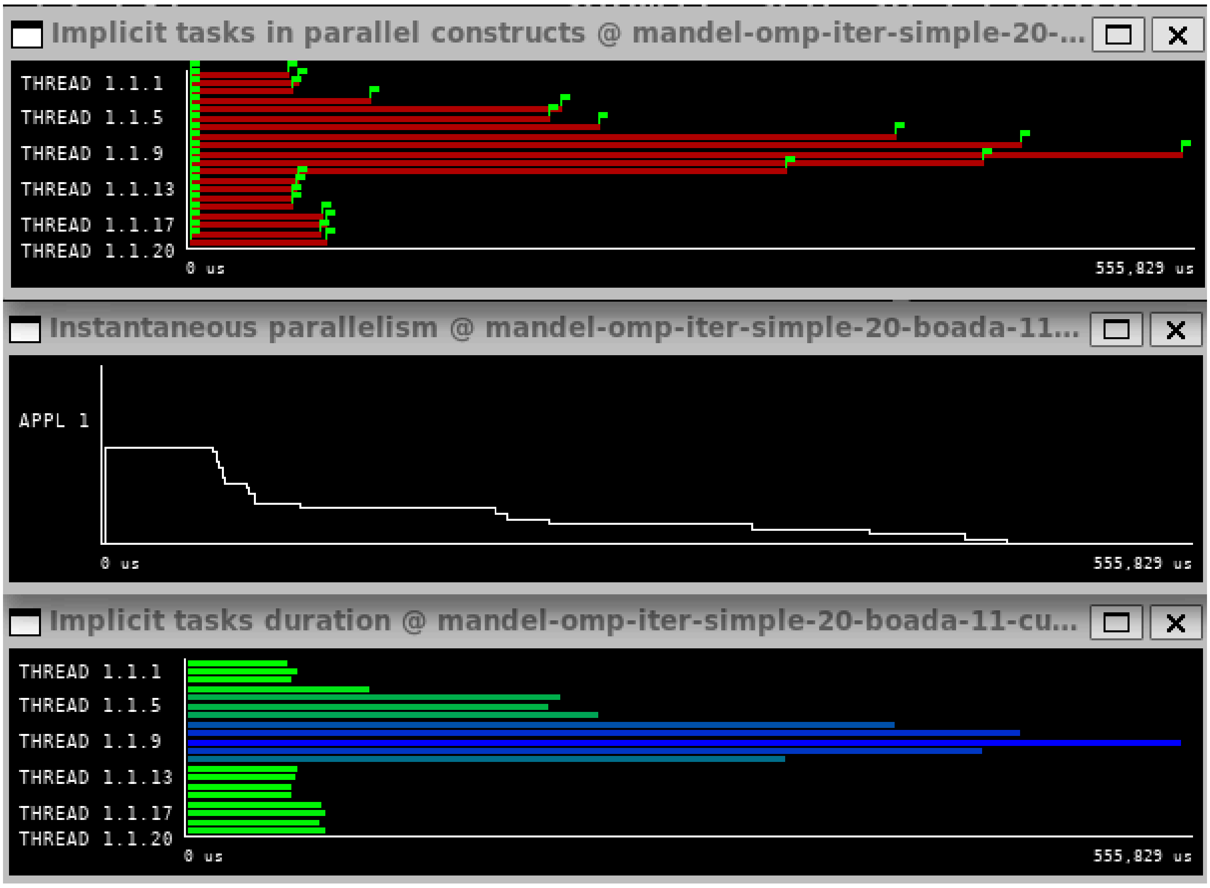
```
\hline
Number of processors & 1 & 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & 18 & 20 \\
\hline
\hline
Number of implicit tasks per thread (average us)    &            1.0 &            1.0 &            1.0 &
1.0 &            1.0 &            1.0 &            1.0 &            1.0 &            1.0 &            1.0 &
      1.0 \\
\hline
Useful duration for implicit tasks (average us)     &      2506513.54 &      1260192.12 &       658717.74 &       456621
.92 &       362422.55 &       299085.48 &        256439.5 &       230508.87 &        205920.5 &       193846.48 &
 180138.7 \\
\hline
Load balancing for implicit tasks                &            1.0 &           0.71 &           0.43 &           0.35
 &           0.34 &           0.33 &           0.33 &           0.33 &           0.33 &           0.33 &
  0.34 \\
\hline
Time in synchronization implicit tasks (average us) &            0 &            0 &            0 &
  0 &            0 &            0 &            0 &            0 &            0 &            0 &
      0 \\
\hline
Time in fork/join implicit tasks (average us)       &          22.05 &            0 &            0 &
  0 &            0 &            0 &            0 &            0 &            0 &            0 &
      0 \\
\hline
========================================================================================================================
========================================================================================================================
====== \\
```

Making an overall analysis of the modelfactor tables, we can see that in the first table the principal problem is the efficiency. In the second one we see that there are several problems, global efficiency, parallelization and load balancing are the principal ones. There are other problems such as scalability for computation tasks and frequency scalability, but they are not as bad as the other ones.

We had to get the third table with cat table3.tex because it did not charge well when executing the tables.

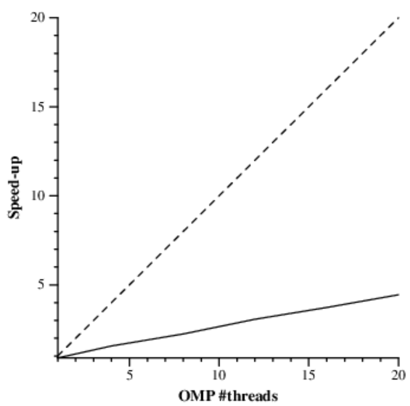
Then we made the execution of l2 and l3 cache misses and we included that into the table.

After that we're going to include the paraver analysis:



We can see the problems mentioned in the tables reflected here, for example de parallelism in the instantaneous parallelism, or load balancing seeing the bad balance between the tasks.

Finally, we're going to include the strong scalability graphic:



par1116
Speed-up wrt sequential time (mandel funtion only)
Generated by par1116 on Thu May 23 09:37:01 AM CEST 2024

The curve is bad, it is so far away from the ideal one. That is being caused by the huge amount of problems that this implementation of the block is giving.

In summary, the block strategy causes a lot of problems due to the difficulty of balancing the task and parallelizing the code.

1.2. 1D Cyclic Geometric Data Decomposition by columns

1.2.1. Execution and check the correctness

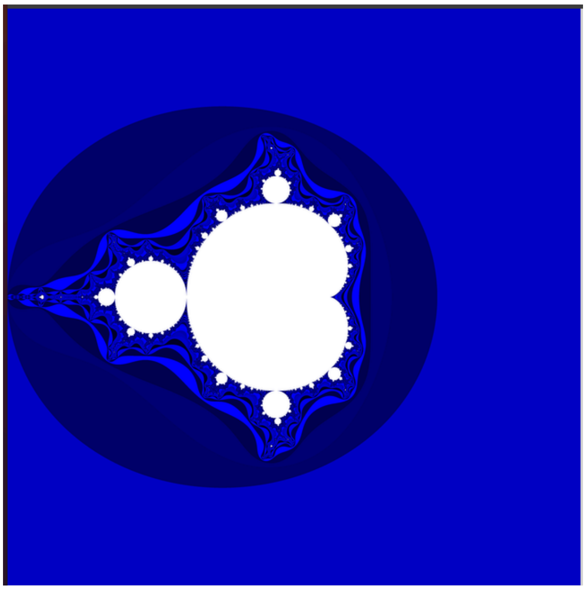
To do the cyclic version we just made changes to the double loop and auxiliary variables:

- int my_id = omp_get_thread_num();
- int howmany = omp_get_num_threads();

And we modified the double loop in order to make the decomposition by columns:

- for (int py = 0; py < ROWS; py++)
- for (int px = my_id; px < COLS; px += howmany)

We proceeded to compare the outputs when generating the image on both of them and the result was identical to the image showed before:



We also used the command:

- sbatch submit-omp.sh mandel-omp-iter-simple 20

To get the execution time with 20 processors. The result is seen on the table below (elapsed time).

1.2.2. Performance Analysis

The second part of every code consisted in analyzing the performance of the strategy. We used the modelfactor tables, paraver analysis, cache misses analysis and the strong scalability graphic.

We're going to start with the modelfactor tables:

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.34	1.21	0.64	0.49	0.39	0.33	0.30	0.27	0.25	0.23	0.24
Speedup	1.00	1.93	3.64	4.80	6.07	7.00	7.84	8.80	9.53	10.13	9.67
Efficiency	1.00	0.97	0.91	0.80	0.76	0.70	0.65	0.63	0.60	0.56	0.48

Table 1: Analysis done on Wed May 29 06:35:04 PM CEST 2024, par1116

Overview of the Efficiency metrics in parallel fraction, ϕ=99.74%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	96.99%	91.93%	80.97%	77.04%	71.29%	66.81%	64.47%	61.24%	57.95%	49.75%
Parallelization strategy efficiency	100.00%	99.97%	99.93%	99.83%	99.76%	99.68%	99.51%	99.43%	99.40%	99.14%	99.07%
Load balancing	100.00%	99.99%	99.98%	99.99%	99.99%	99.99%	99.97%	99.98%	99.99%	99.96%	99.97%
In execution efficiency	100.00%	99.98%	99.95%	99.84%	99.76%	99.69%	99.54%	99.44%	99.41%	99.17%	99.10%
Scalability for computation tasks	100.00%	97.02%	91.99%	81.10%	77.22%	71.52%	67.14%	64.84%	61.61%	58.45%	50.22%
IPC scalability	100.00%	99.19%	96.45%	91.32%	86.36%	82.20%	77.87%	74.77%	71.33%	68.54%	67.17%
Instruction scalability	100.00%	100.00%	100.00%	100.00%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%
Frequency scalability	100.00%	97.82%	95.38%	88.82%	89.42%	87.02%	86.23%	86.74%	86.39%	85.29%	74.77%

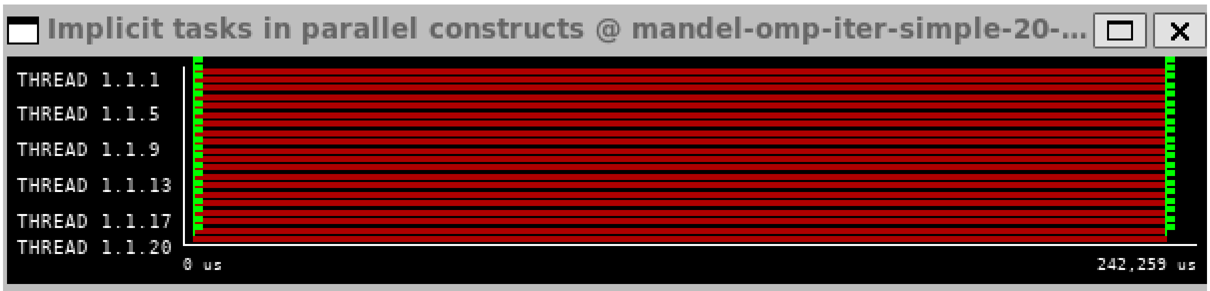
Table 2: Analysis done on Wed May 29 06:35:04 PM CEST 2024, par1116

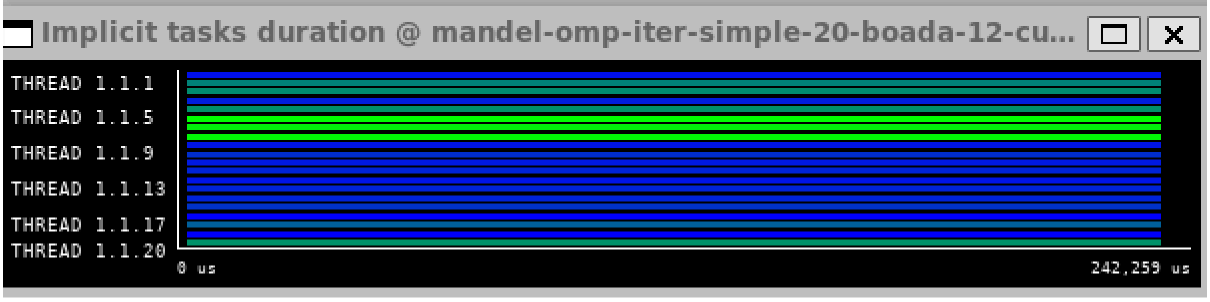
```
Number of processors & 1 & 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & 18 & 20 \\
\hline
\hline
Number of implicit tasks per thread (average us)    &            1.0 &            1.0 &            1.0 &
1.0 &            1.0 &            1.0 &            1.0 &            1.0 &            1.0 &            1.0 &
      1.0 \\
\hline
Useful duration for implicit tasks (average us)     &      2337138.59 &      1204439.68 &       635141.04 &       480273
.36 &       378307.59 &       326774.59 &        290083.0 &        257455.99 &       237077.42 &       222124.67 &
232692.75 \\
\hline
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 \\
\hline
Time in synchronization implicit tasks (average us) &              0 &              0 &              0 &
  0 &              0 &              0 &              0 &              0 &              0 &              0 &
        0 \\
\hline
Time in fork/join implicit tasks (average us)       &          23.55 &              0 &              0 &
  0 &              0 &              0 &              0 &              0 &              0 &              0 &
        0 \\
\hline
========================================================================================================================
========================================================================================================================
====== \\
```

Seeing the modelfactor tables, we can see that in the first table the principal problem is still the efficiency, but it has improved compared to the block strategy. The speedup is also better, so it has a better parallelization strategy, as we can see in the second and third tables. In the second one we see that the global efficiency is still bad and the second problem now happens to be scalability for computation tasks. There are other secondary problems such as IPC and frequency scalability. On the other hand the parallelization is now correct.

Then we made the execution of l2 and l3 cache misses and we included that into the table.

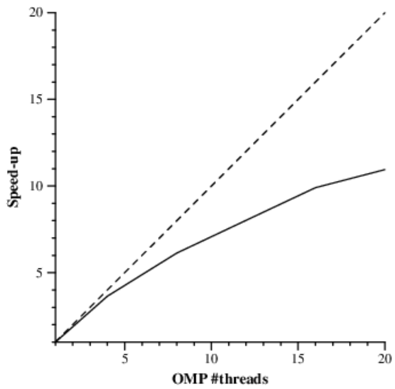
After that we're going to include the paraver analysis:





We can clearly see that the instantaneous parallelization is now correct and the other improvements in the implicit tasks duration and implicit tasks.

Finally, we're going to include the strong scalability graphic:



par1116
Speed-up wrt sequential time (mandel funtion only)
Generated by par1116 on Wed May 29 06:49:32 PM CEST 2024

The curve clearly shows the improvements mentioned, the code is better parallelized but it is still far from the ideal one.

1.3. 1D Cyclic Geometric Data Decomposition by rows

1.3.1. Execution and check the correctness

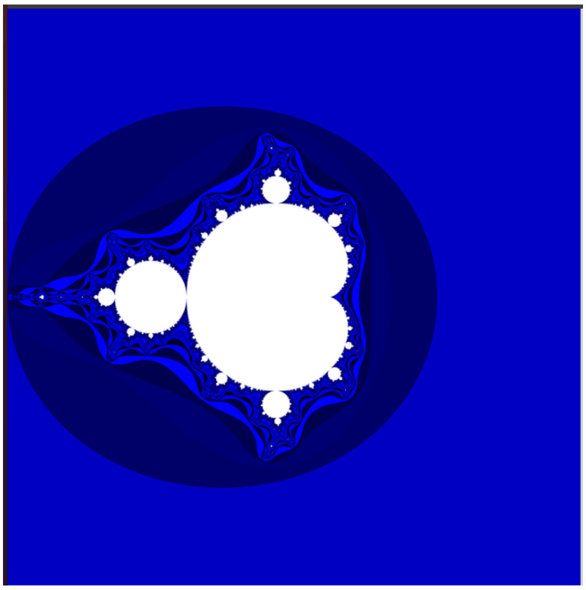
To do the Cyclic Geometric Data Decomposition by rows we have done the same as before but changing the order in the loops, as this.

-for (int py = id; py < ROWS; py += num_threads)
-for (int px = 0; px < COLS; px++)

And also we have declared these two variables.

-int id = omp_get_thread_num();
- int num_threads = omp_get_num_threads();

In order to check the correctness of the code we have compared both images, as we can see, the result is the same.



We also used the command to see the elapsed time:

- sbatch submit-omp.sh mandel-omp-iter-simple 20

To get the execution time with 20 processors. The result is seen on the table below (elapsed time)

1.3.2. Performance Analysis

Now we are going to do the performance analysis of the code. In order to do that we have user modelfactor tables, paraver analysis, cache misses analysis and the strong scalability graphic.

In mandelbrot tables this is the output:

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	1.19	0.61	0.43	0.33	0.27	0.23	0.20	0.17	0.16	0.14
Speedup	1.00	1.99	3.87	5.54	7.21	8.76	10.39	12.04	13.58	15.19	16.75
Efficiency	1.00	0.99	0.97	0.92	0.90	0.88	0.87	0.86	0.85	0.84	0.84

Table 1: Analysis done on Tue May 28 10:18:39 AM CEST 2024, par2222

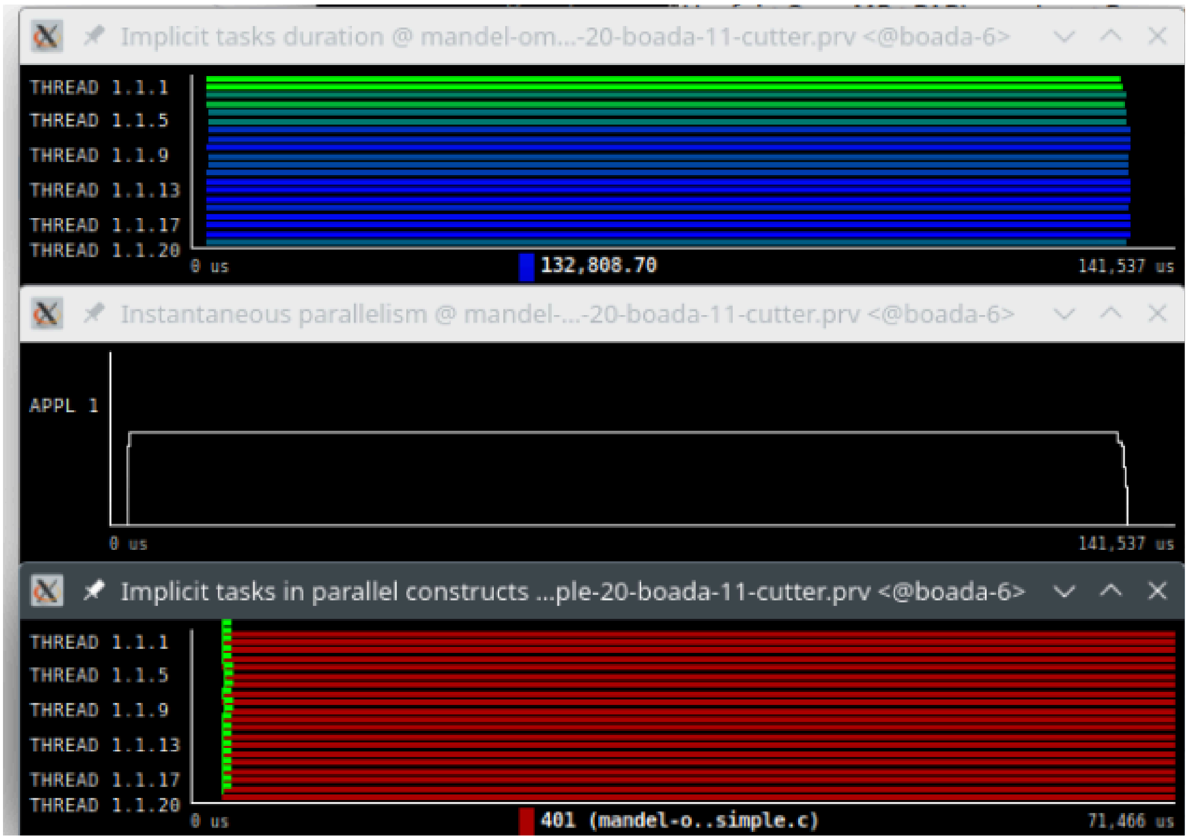
Overview of the Efficiency metrics in parallel fraction, ϕ=99.69%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	99.78%	97.65%	93.48%	91.85%	89.60%	89.02%	88.91%	88.02%	88.12%	87.57%
Parallelization strategy efficiency	100.00%	99.97%	98.96%	99.71%	98.95%	99.47%	99.22%	98.89%	98.91%	98.43%	98.15%
Load balancing	100.00%	99.99%	99.00%	99.86%	99.23%	99.91%	99.82%	99.69%	99.73%	99.63%	99.69%
In execution efficiency	100.00%	99.98%	99.96%	99.85%	99.72%	99.56%	99.40%	99.20%	99.18%	98.80%	98.46%
Scalability for computation tasks	100.00%	99.81%	98.67%	93.75%	92.82%	90.08%	89.72%	89.90%	88.99%	89.53%	89.22%
IPC scalability	100.00%	99.87%	99.91%	99.93%	99.91%	99.89%	99.88%	99.90%	99.88%	99.91%	99.89%
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%	99.94%	98.76%	93.82%	92.91%	90.18%	89.82%	89.99%	89.10%	89.61%	89.32%

Table 2: Analysis done on Tue May 28 10:18:39 AM CEST 2024, par2222

```
\begin{table}[h]
\begin{center}
\begin{tabular}{|l|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\multicolumn{12}{|c|}{Statistics about explicit tasks in parallel fraction} \\
\hline
\hline
Number of processors & 1 & 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & 18 & 20 \\
\hline
\hline
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 \\
\hline
Useful duration for implicit tasks (average us)     &      2363106.25 &      1183806.82 &      598720.78 &      420121.65 &      318223.94 &      262333.11 &      219499.15 &      187756.46 &
 165962.52 &       146640.97 &       132436.36 \\
\hline
Load balancing for implicit tasks                  &            1.0 &            1.0 &            0.99 &            1.0 &            0.99 &            1.0 &            1.0 &            1.0 &
    1.0 &            1.0 &            1.0 \\
\hline
Time in synchronization implicit tasks (average us) &              0 &              0 &              0 &              0 &              0 &              0 &              0 &              0 &
         0 &              0 &              0 \\
\hline
Time in fork/join implicit tasks (average us)       &          57.25 &              0 &              0 &              0 &              0 &              0 &              0 &              0 &
         0 &              0 &              0 \\
\hline
==================================================================================================================================================================================
=========================================== \\
\hline
 \\
```

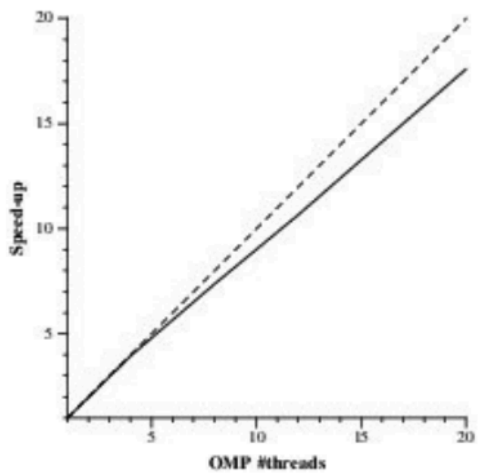
As we can see the speed up in this code is much better than the others and now the problem with the efficiency seems to have disappeared. Also we can see that scalability is much better than before. So we can see that this code seems to work much better than the others and is more parallel.

Now we will see the paraver analysis.



As we can see, the instantaneous parallelization keeps implicit tasks duration and implicit tasks is the same.

Finally we have included the scalability graphic.



par2222
Speed-up wrt sequential time (mandel funtion only)
Generated by par2222 on Sat Jun 1 01:54:15 PM CEST 2024

We can see that this curve is the best and is almost perfect. The improvements seen in the tables and paraver analysis are being also represented here by the huge improvements in the scalability curve.

2. Table

	Number of threads (elapsed)					
Version	1	4	8	12	16	20
1D Block Geometric Data Decomposition by columns	3.240252	1.989764	1.499153	1.177716	1.012080	0.983315
1D Cyclic Geometric Data Decomposition by columns	3.118981	1.688568	1.141864	0.987589	0.964619	0.941827
1D Cyclic Geometric Data Decomposition by rows	2.461413	1.060215	0.764886	0.714026	0.675852	0.652769
	Number of threads (L2 Cache Misses per thread)					
1D Block Geometric Data Decomposition by columns	26488426 26488426	26535658 6633914	26665842 3333230	26759170 2229930	26863697 1678981	27038594 1351929
1D Cyclic Geometric Data Decomposition by columns	19021 19021	16890248 4222562	41451250 5181406	64306995 5358916	79619190 4976199	92112366 4605618
1D Cyclic Geometric Data Decomposition by rows	14383 14383	231240 223481	489048 61131	269508 22459	267522 16720	464081 23204
	Number of threads (L3 Cache Misses per thread)					
1D Block Geometric Data Decomposition by columns	12332155 12332155	18762065 4690516	19792013 2474001	19795732 1649644	20332504 1270781	22269238 1113461
1D Cyclic Geometric Data Decomposition by columns	4006 4006	7654643 1913660	8531474 1066434	9153195 762766	9799729 612483	11868836 593441
1D Cyclic Geometric Data Decomposition by rows	2629 2629	150102 37525	143009 17876	115145 9595	125507 7844	118404 5920

In summary, to comment on the results of the table, we can clearly see the improvements of time, so as the same way the improvements in parallelism during the process of changing the strategies (3rd is the best one). In terms of cache hits and misses. Cache hits and misses improve due to better data locality and more efficient memory access patterns, reducing latency and increasing overall performance with optimized threading and data decomposition strategies. So, the conclusion is that Cyclic Decomposition by rows is the best strategy, as we have confirmed with the model factor tables, the paraver analysis, the cache analysis and the strong scalability curve.