PARreportlab5.pdf

Yiqi_FIB

Paralelismo

3º Grado en Ingeniería Informática

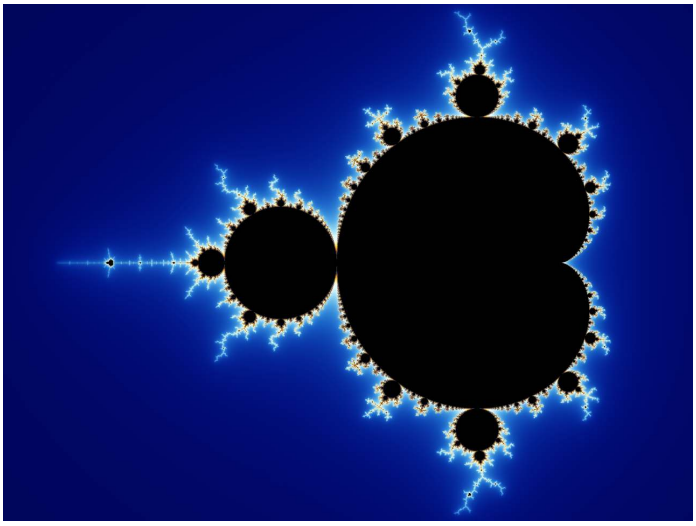
Facultad de Informática de Barcelona (Fib)
Universidad Politécnica de Catalunya

Universitat Politècnica de Catalunya

PARALLELISM

Laboratory practice nº5:

Parallel Data Decomposition Implementation and Analysis



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

WUOLAH

Index

1. Geometric Data Decomposition Strategies

1.1. 1D Block Geometric Data Decomposition by columns

Code: In order to implement the first version, we've only had to modify the mandel_simple function, we've included a parallel construct that includes all the function's body without the single construct in order to avoid creating explicit tasks and enabling every thread to compute a different part from the histogram variable by using the identifier of each thread, and forcing the loops to work taking those ids into account. We've also included the usual atomic and critical constructs used in the previous laboratory sessions.

To do the block decomposition we've used a BS(block size) variable that indicates the number of columns of each block, and a pair of variables j_start and j_end that indicate to each thread which columns will compute.

Notice that there's a special case to take into account, if the number of columns is not multiple of the number of threads the last thread (the one with the biggest identifier) will be in charge of computing the remaining columns, otherwise no thread will compute those columns.

We're aware that there's also an alternative way to treat that that issue, that is trying to balance the number of columns that each block would have, nevertheless as we know from previous laboratory sessions, the last columns (in general the ones that compute areas near the borders), will have a very low time of execution than the ones that are in the middle, therefore we've not applied this technique.

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 my_id = omp_get_thread_num();
          int howmany = omp_get_num_threads();
          int BS = COLS/howmany; // 5120
          int j_start = my_id * BS;
          int j_end = j_start + BS;

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

          for (int py = 0; py < ROWS; py++)
          for (int px = j_start; px < j_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: Here in the first table from the Modelfactors, basically we can see that as we increase the number of threads in the execution, in the three first columns we see that the efficiency goes down by a 0.3 in each column, moreover the speedup just improves by a 0.3 in each column. And from that column on, we can see that the evolution follows a linear tendency that increases with a slope much lower than the x = y ideal performance line.

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.38	1.69	1.47	1.26	1.02	0.88	0.75	0.68	0.61	0.58	0.52
Speedup	1.00	1.41	1.62	1.88	2.33	2.70	3.17	3.48	3.89	4.12	4.57
Efficiency	1.00	0.70	0.41	0.31	0.29	0.27	0.26	0.25	0.24	0.23	0.23

Figure 1: Image of the Modelfactors's execution metrics analysis

If we take a look at the second table of Modelfactors, we can observe that the parallelization strategy efficiency goes down very quickly till reaching a value of 23% in the execution with 20 threads, as we could see in the previous table. Moreover, we can see that the load balancing between threads is quite disappointing, due to the fact that we have threads that execute a lot of computation, that correspond to those blocks that execute columns in the middle of the histogram variable, since some of them execute a huge amount of compute and others not that much. This leaded us to think that may be the technique to reduce the load balance would reduce the general unbalancing, but as the blocks would still be attached to consecutive number of columns, we've concluded that the unbalancing would still be the same or a just a little bit lower, the only possible solution to improve the load balance would be a redistribution of which columns does each thread execute, letting us foresee that a cyclic strategy could be more optimal.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.76%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	70.42%	40.63%	31.43%	29.19%	27.09%	26.61%	25.00%	24.53%	23.07%	23.06%
Parallelization strategy efficiency	100.00%	70.31%	40.98%	32.54%	31.01%	29.84%	29.79%	29.05%	28.87%	28.12%	29.55%
Load balancing	100.00%	70.32%	41.00%	32.56%	31.04%	29.88%	29.83%	29.11%	28.94%	28.20%	29.67%
In execution efficiency	100.00%	99.99%	99.97%	99.94%	99.90%	99.87%	99.84%	99.79%	99.75%	99.69%	99.58%
Scalability for computation tasks	100.00%	100.15%	99.14%	96.60%	94.14%	90.80%	89.32%	86.06%	84.96%	82.05%	78.05%
IPC scalability	100.00%	99.95%	99.80%	99.73%	99.71%	99.64%	99.61%	99.44%	99.34%	99.29%	99.21%
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%	100.20%	99.34%	96.87%	94.42%	91.13%	89.68%	86.56%	85.53%	82.65%	78.69%

Figure 2: Image of the Modelfactors's efficiency metrics analysis

To complete the Modelfactors analysis, we can see that in this third table, the time that each task executes, is very high and in addition, that the overheads in synchronization or task creation is null, what helps us to strengthen the idea that an strategy that distributes better the columns that each thread executes, would give us much better results.

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)	2370787.44	1183617.17	597842.58	409019.34	314786.58	261110.21	221186.7	196761.39	174397.01	160515.71	151872.9
Load balancing for implicit tasks	1.0	0.7	0.41	0.33	0.31	0.3	0.3	0.29	0.29	0.28	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)	55.25	0	0	0	0	0	0	0	0	0	0

Figure 3: Image of the Modelfactors's statistics explicit tasks in parallel analysis

Memory Analysis:

L2 Cache: If we take a look to the table of total cache access and misses in the executions for different number of threads, we can see that they behave in the following way; as a thread is added to the execution, the total number of misses in the execution will be equal to the total number of access to the cache/ number of threads. What perfectly represents a linear behavior depending on how many threads are involved in the execution.
This analysis is caused due to the huge amount of false sharing in the execution. That leads to a lot of cache misses and a high number of invalidations that the snoopy has to send to each core.

1	52327	52327
2	809341	404670
4	1121948	210055
6	1100360	183393
8	1518760	189845
10	1862317	186231
12	2144349	178695
14	2345354	167525
16	2320515	145032

msi

Más posibilidades, más productividad, más portátil. Más de todo.

Pack portátil + monitor portátil

Si una sola pantalla te sabe a poco, MSI tiene el dúo dinámico que necesitas. Un combo portátil + monitor extra para que tu setup de trabajo te siga a todos lados como tu ex stalkeándote. Más espacio, más productividad, menos excusas.

18	2546608	141478
20	2848714	142435

Figure 4: Image total cache 2 access and misses per each execution of threads

Now if we take a look to the paravers trace, we can see much better the unbalanced produced, for instance, in the 20 threads execution the threads from number 5 to number 12, execute an enormous number of computation compared with the rest of the threads, as we've said before those threads with a biggest load could be those threads that execute blocks of columns in the middle of the histogram variable.

Moreover as more greener is the bar, the less number of misses does the thread in question execute, and as more blueish is the bar, more misses does that thread. Therefore we can see that in general all threads execute a high number of threads except the thread number 12 that performs really well.

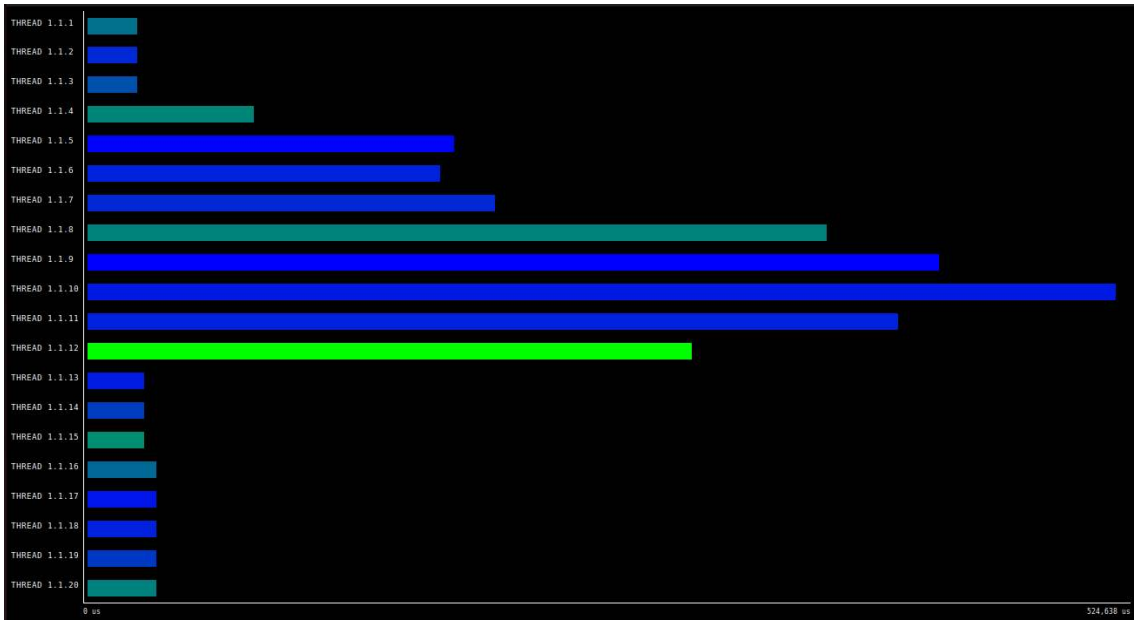


Figure 5: Image total cache 2 access in length and misses in color

In that image we can see some statistics of the number of misses in cache L2, we can see that the thread that produces more cache misses does around 180.000 misses, and the one that does less misses does less than a quarter of misses, a little bit more than 40.000 misses.

Moreover, we can see that the value of the standard deviation is really high compared to the total number of misses, that lets us foresee that the variation of the variable expected about its mean is really high.

Total	2,837,613
Average	141,880.65
Maximum	182,972
Minimum	40,726
StDev	34,155.24
Avg/Max	0.78

Figure 6: Image of some statistics about cache 2 misses

L3 Cache: If we take a look at the table of total cache access and misses in the executions for different number of threads in L3 cache, we see that the total number of access is lower than the access in L2 cache as it was expected, but the performance is still the same.

1	2554	2554
2	552063	276031
4	840222	210055
6	870796	145132
8	1222843	152855
10	1530192	153019
12	1842433	153536
14	2030170	145012
16	1874213	117138
18	1976345	109796
20	2161970	108098

Figure 7: Image total cache 3 access and misses per each execution of threads

If we look at the Paraver trace we can see that the threads in general intensified it's color, the ones that were a little bit green, and the same with the blue ones.

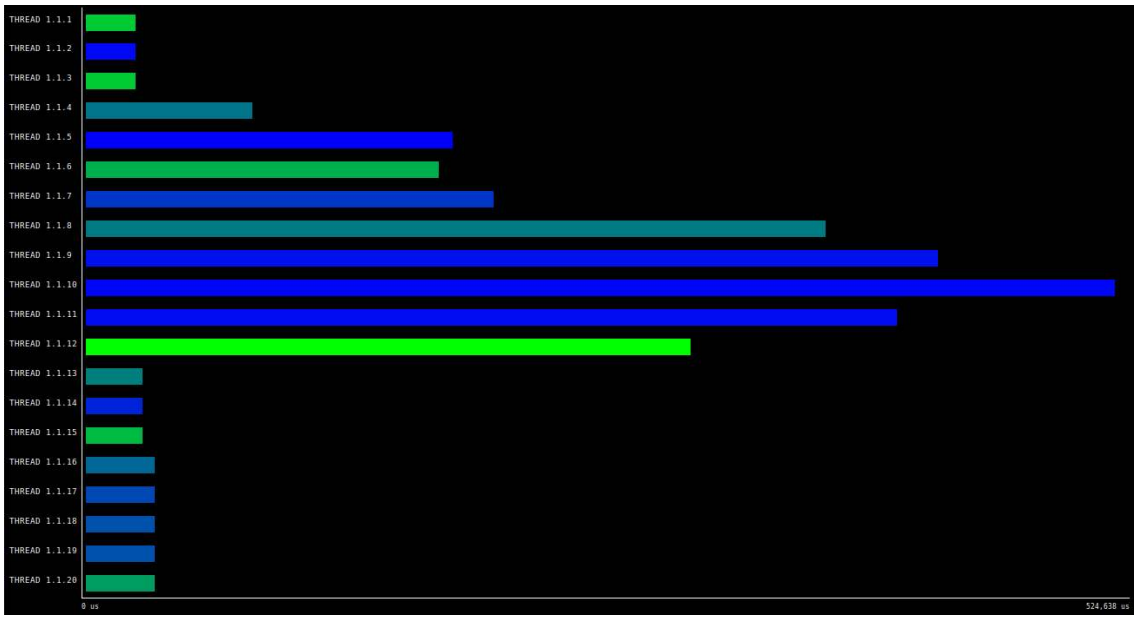


Figure 8: Image total cache 3 access in length and misses in color

In the statistics table we can see that the standard deviation is higher than in L2 cache, which strengthens the idea commented when analyzing the trace.

Total	2,158,719
Average	107,935.95
Maximum	158,743
Minimum	30,729
StDev	37,705.31
Avg/Max	0.68

Figure 9: Image of some statistics about cache 3 misses

Strong Scalability: Finally we can see with the strong scalability graphic, that all the numbers of the tables we've seen before fit perfectly. We can see how badly this strategy performs, the performance is not worse than the leaf strategy used in the previous laboratory, but is one of the worse strategies we've implemented. As commented previously and after all the analysis of this version we can strongly affirm that a redistribution of which columns must each core execute would increase the performance of the execution.

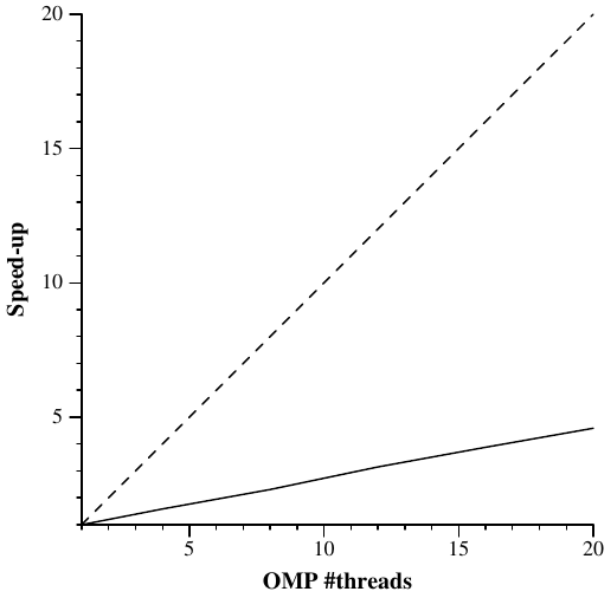


Figure 10: Image of the scalability graphic

1.2. 1D Cyclic Geometric Data Decomposition by columns

Code: On that case, in order to do the cyclic data decomposition, we've also included the parallel, the atomic and the critical constructs, also we've erased the BS variable since we just want it to be cyclic, and the j_end variable since here we need to iterate the inner group by increasing the iterative variable by the number of threads in each iteration.

```C/C++
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 j_start = my_id;

            for (int py = 0; py < ROWS; py++)
            for (int px = j_start; px < COLS; px += howmany)
            {
                        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: In this version we can see that the speedup increases almost to a value of 10 in the execution with 20 threads, we can also see that now the performance seems not to fit to a line with a positive slope, we can see that from the execution with 14 threads to the execution with 20 threads we just gain 0.03 seconds in the execution time, moreover the speedup value just increases by 1.

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	1.22	0.65	0.49	0.39	0.34	0.30	0.27	0.24	0.24	0.24
Speedup	1.00	1.94	3.62	4.84	6.08	6.91	7.89	8.83	9.70	9.79	9.92
Efficiency	1.00	0.97	0.90	0.81	0.76	0.69	0.66	0.63	0.61	0.54	0.50

Figure 11: Image of the Modelfactors's execution metrics analysis

If we observe this second table of the modelfactor, we can see that the parallelization strategy efficiency has increased significantly, in the previous strategy we've reached a value around 30% and in this strategy we conserve a value of 99% in the execution with 20 threads. That's thanks to the huge reduction of unbalancing with the redistribution of columns executed by each thread thanks to the cyclic strategy implementation. Nevertheless we still see that the efficiency despite increasing from 23% to 50%, it's still a low value.

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%	97.23%	91.00%	81.51%	76.95%	70.05%	66.85%	64.31%	61.91%	55.55%	50.78%
Parallelization strategy efficiency	100.00%	99.97%	99.94%	99.86%	99.79%	99.70%	99.63%	99.43%	99.40%	99.01%	99.07%
Load balancing	100.00%	99.99%	99.98%	99.99%	99.99%	99.99%	99.99%	99.98%	99.99%	99.97%	99.97%
In execution efficiency	100.00%	99.98%	99.96%	99.87%	99.80%	99.72%	99.64%	99.45%	99.41%	99.04%	99.10%
Scalability for computation tasks	100.00%	97.26%	91.06%	81.62%	77.12%	70.25%	67.10%	64.68%	62.29%	56.11%	51.26%
IPC scalability	100.00%	99.21%	95.98%	91.55%	87.06%	81.94%	77.95%	74.76%	71.60%	68.89%	67.68%
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%	98.03%	94.88%	89.16%	88.58%	85.74%	86.09%	86.52%	87.01%	81.46%	75.75%

Figure 12: Image of the Modelfactors's efficiency metrics analysis

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)	2355202.85	1210807.58	646587.41	480907.53	381760.13	335240.46	292482.64	260094.23	236326.45	233207.49	229727.58
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)	54.59	0	0	0	0	0	0	0	0	0	0

Figure 13: Image of the Modelfactors's statistics explicit tasks in parallel analysis

Memory Analysis:

L2 Cache: We can see that this version of the program is in memory, worse than the "1D Block Geometric Data Decomposition by columns". This is because we modified the code and now each thread access to non-contiguous memory locations because of "px += howmany" and this can get worse because of the frequent cache line replacements.

1	17805	17805
2	7325560	3662780
4	13858881	3464720
6	27017230	4502871
8	37923131	4740391
10	50167705	5016770
12	60202638	5016886
14	67723825	4837416
16	74173370	4635835
18	81575828	4531990
20	81509954	4075497

Figure 14: Image total cache 2 access and misses per each execution of threads

A good thing about this version is that compared to the previous one, we have a more consistent number of access to L2 cache. We have to also remark that there are 3 threads with a very good performance and don't have a lot of misses.

Figure 15: Image total cache 2 access in length and misses in color

If we take a look at the table below, we can see that the number of misses is about 1 for every 2 access, which is very high but at least we can see a consistent performance between all threads as the maximum and minimum are close.

Total	81,500,477
Average	4,075,023.85
Maximum	4,156,167
Minimum	3,942,237
StDev	62,465.97
Avg/Max	0.98

Figure 16: Image of some statistics about cache 2 misses

L3 Cache: Here in L3 cache we have less accesses and misses than L2 cache. Compared to the previous version where both L2 and L3 caches had more or less the same number of access and misses, here there's a reduction but the quantity is still very high.

1	2223	2223
2	6658036	3329018
4	4682167	1170541
6	10711994	1785332

8	12111361	1513920
10	13245912	1324591
12	14019585	1168298
14	15954511	1139607
16	17368112	1085507
18	19233713	1068539
20	20427568	1021378

Figure 17: Image total cache 3 access and misses per each execution of threads

We can't add much more information, as the behavior is the same where all threads have the same amount of access but we can point out that the majority of threads don't have misses in this L3 cache.

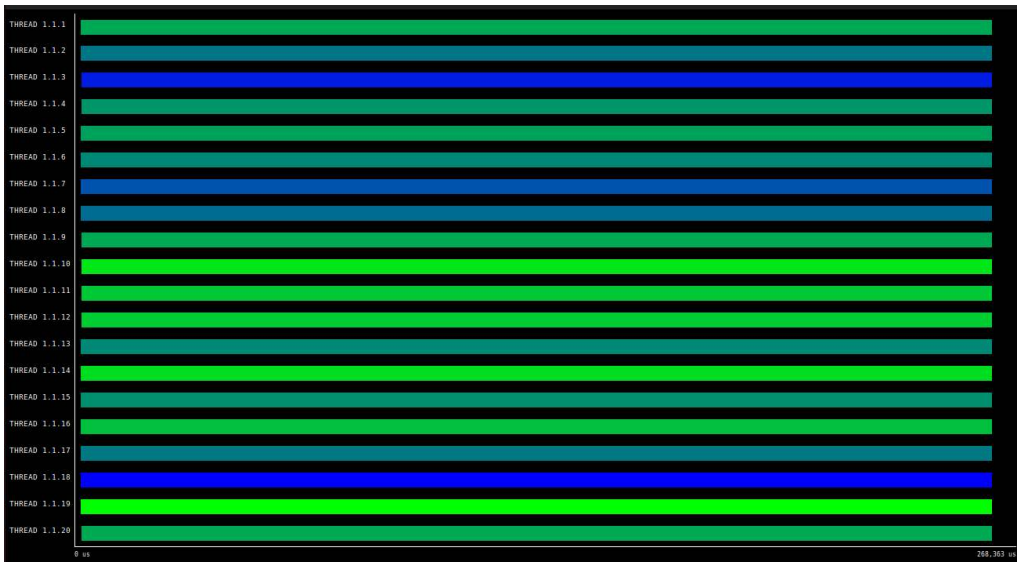


Figure 18: Image total cache 3 access in length and misses in color

To end, we can reinforce what we've told before looking at the total and average rows, where the number is 4 times lower than in the L2 cache, but as the number decreased linearly, the Avg/Max number is similar to the previous one and we still have 1 cache miss for every cache access.

Total	20,426,156
Average	1,021,307.80
Maximum	1,070,226
Minimum	987,226
StDev	20,233.56
Avg/Max	0.95

Figure 19: Image of some statistics about cache 3 misses

Strong Scalability:
Looking at the image, we can see that the program has a good scalability until there are 4 threads involved as it starts to decrease until it reaches again 15 threads and speed-up is almost null adapting a kind of logarithmic tendency, which tends to a speedup value of 10. This conclusion can also be obtained from images above. We have to remark that this speed-up lowering is largely due to the problems of cache misses but is better than the previous version because work is better distributed.

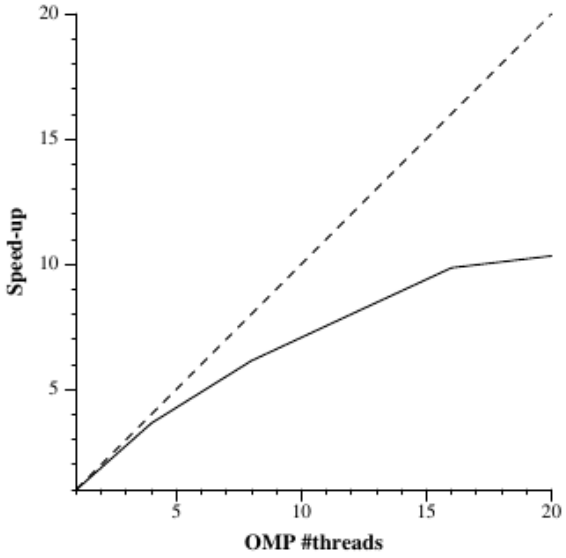


Figure 20: Image of the scalability graphic

1.3. 1D Cyclic Geometric Data Decomposition by rows

Code:
For the 1D Cyclic Geometric Data Decomposition by rows, we didn't have to include in the main function the #pragma omp parallel nor the #pragma omp single, we did it directly inside the function because the code will be executed by all processors considering their thread number. As we want it to be executed by rows, we will change the first forloop parameters instead the second one by starting the py variable at i_start and increasing this every iteration with the number of threads that are in total. We need to also include atomic and critical to avoid Data Problems.

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

           int i_start = my_id;

           for (int py = i_start; py < ROWS; py += howmany)
           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:

As we can observe in the following table, we can see that the efficiency of this program is quite high, compared to previous ones where the 20 cores had a score of 0.20 or 0.50 and this version doesn't fall below 0.80. This version is so good that elapsed time almost halves for every double number of processors and this also applies to speedup, where it doubles.

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.38	1.20	0.60	0.43	0.33	0.27	0.23	0.20	0.17	0.16	0.14
Speedup	1.00	1.99	3.93	5.53	7.31	8.79	10.49	12.07	13.74	15.18	16.75
Efficiency	1.00	1.00	0.98	0.92	0.91	0.88	0.87	0.86	0.86	0.84	0.84

Figure 21: Image of the Modelfactors's execution metrics analysis

In this second table we have information related to efficiency in parallel fractions of the program. As said before, numbers are really good and this indicates that we have a very good scalability even though we have to know that as we increase the number of processors, these scores will be decreasing.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.75%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	99.76%	99.03%	93.24%	92.82%	89.60%	89.48%	88.80%	88.69%	87.81%	87.48%
Parallelization strategy efficiency	100.00%	99.97%	99.51%	99.70%	99.54%	99.37%	99.38%	98.84%	98.81%	98.60%	98.32%
Load balancing	100.00%	99.99%	99.59%	99.90%	99.81%	99.90%	99.88%	99.69%	99.64%	99.73%	99.66%
In execution efficiency	100.00%	99.98%	99.91%	99.80%	99.73%	99.47%	99.50%	99.15%	99.17%	98.87%	98.65%
Scalability for computation tasks	100.00%	99.79%	99.52%	93.52%	93.25%	90.16%	90.04%	89.85%	89.77%	89.06%	88.98%
IPC scalability	100.00%	99.87%	99.91%	99.90%	99.90%	99.89%	99.89%	99.91%	99.91%	99.89%	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%	99.92%	99.61%	93.61%	93.34%	90.26%	90.14%	89.93%	89.85%	89.16%	89.08%

Figure 22: Image of the Modelfactors's efficiency metrics analysis

Concluding modelfactor analysis, we got this table about explicit tasks where at first sight there's just a slight augment of average time in fork/join implicit tasks and in the useful duration for implicit tasks, but this difference is so small that we can ignore it. Apart from that, there are no differences and the results are good.

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)	2372642.73	1188803.25	596002.23	422852.23	318053.86	263146.51	219586.96	188625.74	165197.18	148000.99	133331.5
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)	59.74	0	0	0	0	0	0	0	0	0	0

Figure 23: Image of the Modelfactors's statistics explicit tasks in parallel analysis

Memory Analysis:

L2 Cache: As commented in previous versions, here we have the overall number of cache misses and the average number of misses per thread. The average number of misses behaves equal, where on average they all have the same amount of misses. We can also realize that the amount of cache misses is lower than both previous versions.

1	59831	59831
2	242932	121466
4	957912	239478
6	555415	92569
8	766833	95854
10	751295	75129

12	611507	50958
14	554428	39602
16	548714	34294
18	471860	26214
20	657929	32896

Figure 24: Image total cache 2 access and misses per each execution of threads

Having a look at the following image, we can imagine why L2 cache access decreases so much. All threads have the same amount of cache access and half of them perform worse than the other half (more green is less misses and more blue is more misses).

Figure 25: Image total cache 2 access in length and misses in color

Now we have the table of statistics which in combination with the 2 images above, we have further information that reinforces what we've sensed. The total number of misses is very low and is reflected on the average with its standard deviation.

Total	648,228
Average	32,411.40
Maximum	47,434
Minimum	18,518
StDev	8,161.11
Avg/Max	0.68

Figure 26: Image of some statistics about cache 2 misses

L3 Cache: Next, we have the same images but in this case for L3 cache. With just a first view, we can see that L3 cache has way less access compared to L2 cache but this happens always so it's not very surprising.

1	2115	2115
2	127686	63843
4	189520	47380
6	139277	23212
8	160281	20035
10	121347	12134
12	127569	10630
14	121120	8651
16	123942	7746
18	128790	7155
20	124332	6216

Figure 27: Image total cache 3 access and misses per each execution of threads

In this second L3 cache image we can see that the behavior is the same as the L2 cache so there's not an anomaly. All threads have the same number of access and we can also see that there are more threads that have a green color so that's very positive.

Figure 28: Image total cache 3 access in length and misses in color

Finally we have again the statistics table where we can see that the Avg/Max is reduced even more, having less than 1 cache miss for every 2 access. This data is so good as this means that the code is very optimized and it is not even necessary to make use of L3 cache.

Total	120,588
Average	6,029.40
Maximum	13,436
Minimum	817
StDev	4,151.85
Avg/Max	0.45

Figure 29: Image of some statistics about cache 3 misses

Strong Scalability:

Finishing the analysis of this version of the program, we have the scalability graphic that reinforces the tables seen before where the program has an almost linear tendency of scalability. This version is almost perfect because it is the one that approaches most to the perfect linear tendency.

msi

Más posibilidades, más productividad, más portátil. Más de todo.

Pack portátil + monitor portátil

Si una sola pantalla te sabe a poco, MSI tiene el dúo dinámico que necesitas. Un combo portátil + monitor extra para que tu setup de trabajo te siga a todos lados como tu ex stalkeándote. Más espacio, más productividad, menos excusas.

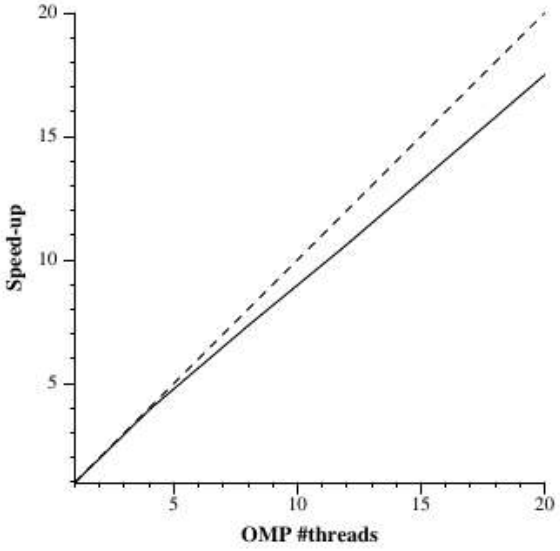


Figure 30: Image of the scalability graphic

2. Summary of the elapsed execution times

	Number of Threads (elapsed)					
Version	1	2	4	8	12	16
1D Block Geometric Data Decomposition by columns	2.38	1.69	1.47	1.02	0.75	0.61
1D Cyclic Geometric Data Decomposition by columns	2.36	1.22	0.65	0.39	0.30	0.24
1D Cyclic Geometric Data Decomposition by rows	2.38	1.20	0.60	0.33	0.23	0.17
	Number of threads (L2 Cache Misses per thread)					
1D Block Geometric Data Decomposition by columns	52327	404670	280487	189845	178695	145032
1D Cyclic Geometric Data Decomposition by columns	17805	3662780	3464720	4740391	5016886	4635835
1D Cyclic Geometric Data Decomposition by rows	59831	121466	239478	95854	50958	34294

	Number of threads (L3 Cache Misses per thread)					
1D Block Geometric Data Decomposition by columns	2554	276031	210055	152855	153536	117138
1D Cyclic Geometric Data Decomposition by columns	2223	3329018	1170541	1513920	1168298	1085507
1D Cyclic Geometric Data Decomposition by rows	2115	63843	47380	20035	10630	7746