PARreportlab4.pdf

Yiqi_FIB

Paralelismo

3º Grado en Ingeniería Informática

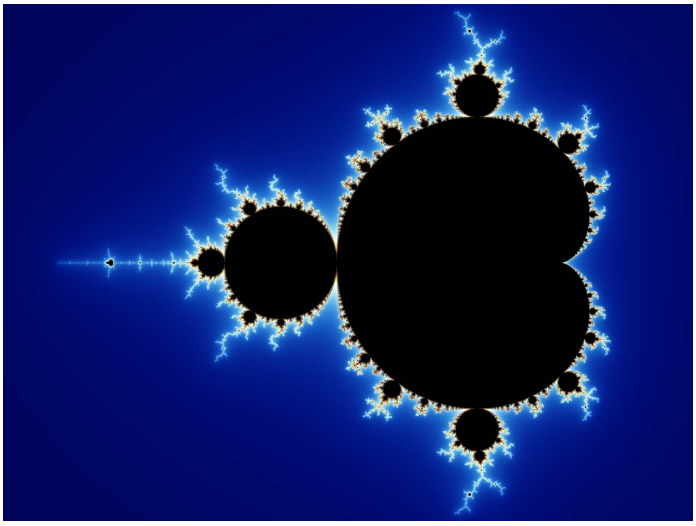
Facultad de Informática de Barcelona (Fib)
Universidad Politécnica de Catalunya

Universitat Politècnica de Catalunya

PARALLELISM

Laboratory practice nº4:

Parallel Task Decomposition Implementation and Analysis



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

WUOLAH

Index

1. Iterative task decomposition

1.1. Tile version:

Code: In order to adapt the tile version code from the previous laboratory delivery, we had to simulate the job done by the tasks tareador_start_task("mandelbrot") which created the necessary tasks and the job done by tareador_disable_object(&X11_COLOR_FAKE) which was in charge of eliminating the dependences produced by that variable when setting the foreground and drawing the point, by using OpenMP constructs.

The modifications are the underlined lines, we've added a firstprivate for the x and the to the task created at the beginning of the execution of each TILE, afterwards, to eliminate the dependences we've added an atomic construct just before the we update the histogram variable and finally, we've added a critical construct just before we set the foreground and draw the point.

C/C++

```
void mandel_tiled (int M[ROWS][COLS], double CminR, double CminI, double CmaxR, double CmaxI, double scale_real, double scale_imag, int maxiter) {
  int equal;
  for (int y = 0; y < ROWS; y += TILE)
      for (int x = 0; x < COLS; x += TILE) {
          #pragma omp task firstprivate(x, y)
          {

                  equal = 1;

                  for (int px = x; px < x + TILE; px++)
                    {
                            M[y][px] = pixel_dwell (COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, y, scale_real, scale_imag, maxiter);
                            M[y + TILE - 1][px] = pixel_dwell (COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, y + TILE - 1, scale_real, scale_imag, maxiter);
                            equal = equal && (M[y][x] == M[y][px]);
                            equal = equal && (M[y][x] == M[y + TILE - 1][px]);
                    }
                  for (int py = y; py < y + TILE; py++)
                    {
                            M[py][x] = pixel_dwell (COLS, ROWS, CminR, CminI, CmaxR, CmaxI, x, py, scale_real, scale_imag, maxiter);
                            M[py][x + TILE - 1] = pixel_dwell (COLS, ROWS, CminR, CminI, CmaxR, CmaxI, x + TILE - 1, py, scale_real, scale_imag, maxiter);
                            equal = equal && (M[y][x] == M[py][x]);
                            equal = equal && (M[y][x] == M[py][x + TILE - 1]);
                    }
                  if (equal && M[y][x]==maxiter)
                    {
                             if (output2histogram)
                            {
                                        #pragma omp atomic
                                        histogram[maxiter-1]+=(TILE*TILE);
                            }
                             long color = (long) ((maxiter-1) * scale_color) + min_color;
                            for (int py = y; py < y + TILE; py++)
                            for (int px = x; px < x + TILE; px++)
                               {
                                        M[py][px] = M[y][x];
                                        if (output2display)
                                        {
                                        /* Scale color and display point  */
                                        if (setup_return == EXIT_SUCCESS)
                                           {
                                                    #pragma omp critical
                                                     {
                                                    XSetForeground (display, gc, color);
                                                    XDrawPoint (display, win, gc, px, py);
                                                     }
                                           }
                                        }
                             }
                    }
                  else
                    // Calcular
                    for (int py = y; py < y + TILE; py++)
                            for (int px = x; px < x + TILE; 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 there is a quite interesting decrease in the execution time without losing too much efficiency, nevertheless as we continue increasing the number of threads, we see that when we execute with 6 threads we don't obtain a big difference from the execution with 4 threads, and as we proceed increasing the number of threads, the efficiency goes down so fast, while the execution time stills almost the same. In fact, if we execute the code with 16 threads, we just improve the execution time a 3'4% (0,02 sec). Therefore, we see that this code does not have a strong scalability.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	3.03	1.53	0.89	0.85	0.67	0.59	0.58	0.57	0.57
Speedup	1.00	1.98	3.41	3.58	4.53	5.12	5.22	5.33	5.31
Efficiency	1.00	0.99	0.85	0.60	0.57	0.51	0.43	0.38	0.33

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 37% in the execution with 16 threads, as we've seen in the previous table. Moreover, we can see that the load balancing between threads is quite disappointing, due to the fact that we have very big tasks which some of them execute a huge amount of compute and others not that much. This allows us to foresee that a fragmentation of this huge task into more and smaller tasks would provide us better results in the load balance.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.99%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	100.00%	98.85%	85.18%	59.65%	56.67%	51.26%	43.52%	38.08%	33.22%
Parallelization strategy efficiency	100.00%	99.11%	85.53%	63.91%	61.18%	56.91%	48.50%	42.42%	37.32%
Load balancing	100.00%	99.16%	85.60%	63.98%	61.26%	57.01%	48.60%	42.52%	37.43%
In execution efficiency	100.00%	99.95%	99.92%	99.89%	99.86%	99.82%	99.79%	99.77%	99.71%
Scalability for computation tasks	100.00%	99.74%	99.60%	93.34%	92.62%	90.08%	89.72%	89.77%	89.01%
IPC scalability	100.00%	99.99%	99.97%	99.96%	99.95%	99.96%	99.96%	99.96%	99.95%
Instruction scalability	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Frequency scalability	100.00%	99.75%	99.63%	93.37%	92.67%	90.12%	89.75%	89.81%	89.06%

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 as commented previously. Also we seem to detect the sign of the bad results, when looking at the synchronization's overhead obtained, we see that when we pass from 2 threads to 4, the overhead time goes from 0,87% to 56% so it has increased almost by 20 times, and when we execute the code with 16 threads we see that this value has raised to 168% something we like very, very little.

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0
LB (number of explicit tasks executed)	1.0	0.91	0.57	0.51	0.38	0.32	0.27	0.23	0.2
LB (time executing explicit tasks)	1.0	0.99	0.86	0.64	0.61	0.57	0.49	0.42	0.37
Time per explicit task (average us)	47334.74	47450.89	47505.35	50674.42	50992.98	52427.53	52574.51	52505.52	52859.39
Overhead per explicit task (synch %)	0.0	0.87	16.85	56.35	63.37	75.59	106.12	135.79	168.2
Overhead per explicit task (sched %)	0.0	0.01	0.01	0.0	0.01	0.01	0.01	0.01	0.01
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

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

msi

intel INSIDE

Hay cambios de los que no te arrepientes nunca. Como este.

Business & Productivity

Si tuviéramos que definir esta línea de laptops en tres palabras, serían: ligeras, facheras y listas para el grind. Básicamente, el combo perfecto para cualquier estudiante que quiere un laptop sin luces RGB que parezca una nave espacial. Sí, MSI también piensa en la gente que tiene que hacer tareas y no solo en los gamers. Así que deja de sufrir con ese ladrillo viejo y pásate a la serie Business & Productivity. Tu espalda (y tu flow) te lo agradecerán

Procesador Intel® serie U

Paraver Analysis: If we take a look on the figure 4, in red color we can see the time when each processor is executing no task and in blue the time the processor executes a task and in figure 6, we can see that as more greener the bar is the less tasks executes that thread and contrary, the most blueish is the bar more tasks executes that thread, knowing that, we can see that there are some threads especially the number 6 and the number 10 are being executed during more than the half of the program while the number of tasks is not to high. So taking into account both images, we observe that at the beginning there is a period of time where the parallelism is exploited at its maximum but as the execution goes forward and each processor starts finishing doing its assigned task, due to the coarse granularity, some processors like the number 3, 7, 9 and 11, are still executing its tasks and if we take a look to the instantaneous paraver's image it all line up in a huge reduction of parallelism exploitation as execution goes by.

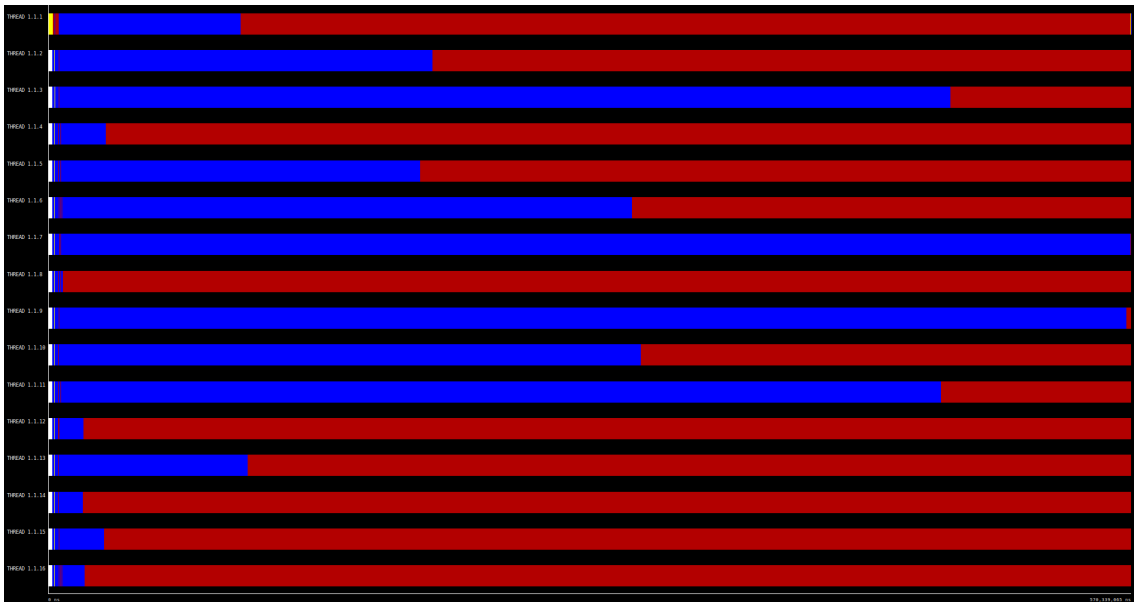


Figure 4: Image of paraver's execution trace

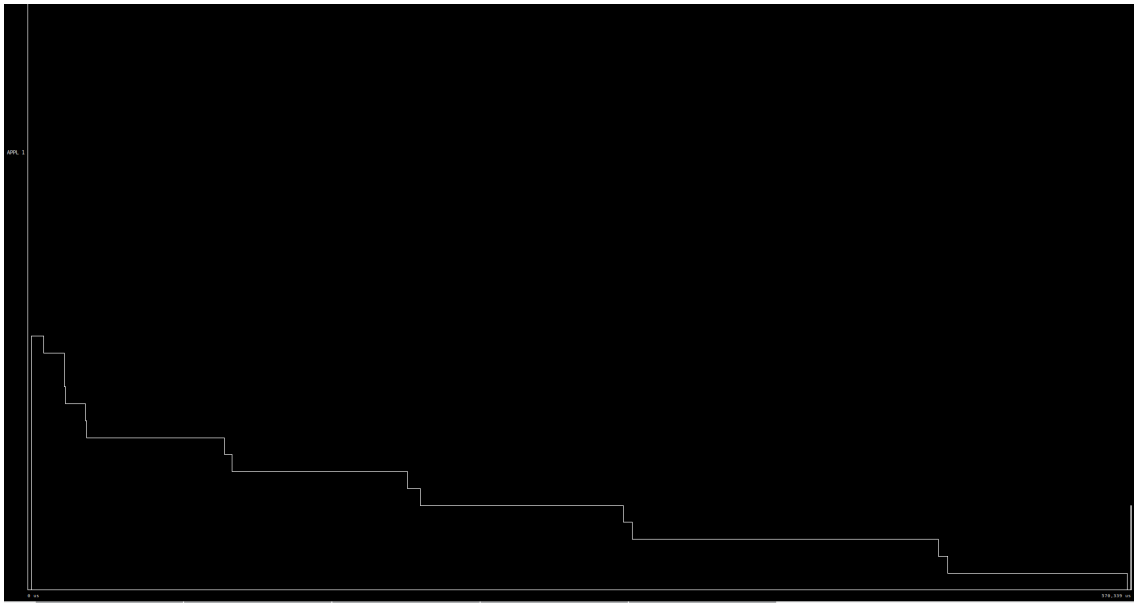


Figure 5: Image of paraver's instantaneous parallelism

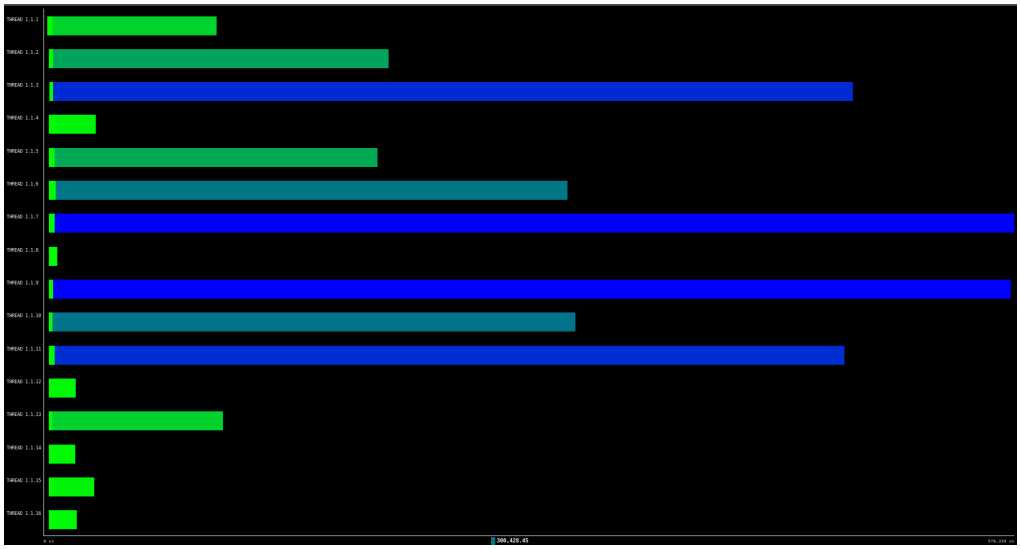


Figure 6: Image of paraver's explicit tasks executed duration

Strong Scalability: In order to analyze the scalability of the code, we can take a look at the figure 7, which shows a graphic in which we compare the threads used in the execution with the speed-up obtained in it. We can see the x = y line represented with a dashed line, which represents the ideal speed-up we could obtain when executing the code with more and more threads.
As commented before in the Modelfactors analysis, we see that we obtain very bad results, more than a lineal graphic similar to the dashed line, we obtained a logarithmic line which tends more or less to 5, that lets us see that there's a huge room for improvement.

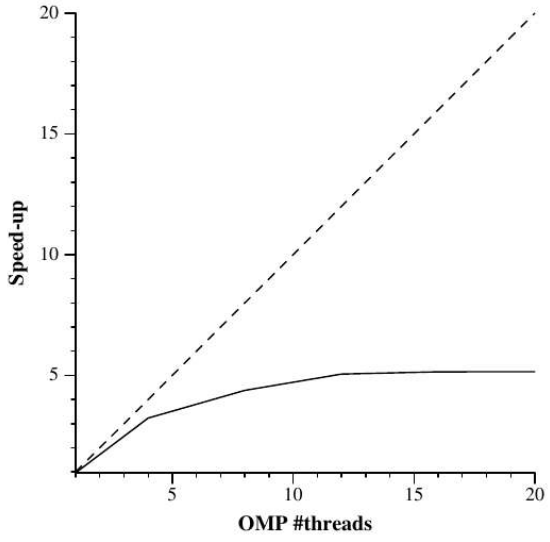


Figure 7: Image of the scalability graphic

1.2. Finer grain version:

Code: In order to adapt the finer grain version code from the previous laboratory delivery, we preserved the criticals and the atomics from the tile version, underlined in green and we've done the following modifications underlined in yellow.

We've added a task for the check of vertical borders and a task for the check of horizontal borders and as in the finer grain version of the laboratory delivery 3, we've created an equal variable for each execution, we initialize both to one and we the task by indicating that each equal must be shared in it's for loop execution. After both loops are executed, we add a barrier "taskwait" to ensure that both tasks have ended before executing the rest of the code. Next, we create a new task which will be in charge of the computation and we put to firstprivate the equal variables and the x and the y to access to the M matrix. Finally, we add two more tasks, one in the "if" execution and one for the "else" execution, both putting the x, the y and the py to firstprivate.

C/C++

```
void mandel_tiled(int M[ROWS][COLS], double CminR, double CminI, double CmaxR,double CmaxI, double scale_real, double scale_imag, int maxiter){
   int equal1, equal2;
   for (int y = 0; y < ROWS; y += TILE)
     for (int x = 0; x < COLS; x += TILE) {
              equal1 = 1;
              equal2 = 1;
              #pragma omp task shared(equal1)
              for (int px = x; px < x + TILE; px++)
              {
              M[y][px] = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, y, scale_real, scale_imag, maxiter);
              M[y + TILE - 1][px] = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, y + TILE - 1, scale_real, scale_imag, maxiter);
              equal1 = equal1 && (M[y][x] == M[y][px]);
              equal1 = equal1 && (M[y][x] == M[y + TILE - 1][px]);
              }
              #pragma omp task shared(equal2)
              for (int py = y; py < y + TILE; py++)
              {
              M[py][x] = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, x, py, scale_real, scale_imag, maxiter);
              M[py][x + TILE - 1] = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, x + TILE - 1, py, scale_real, scale_imag, maxiter);
              equal2 = equal2 && (M[y][x] == M[py][x]);
              equal2 = equal2 && (M[y][x] == M[py][x + TILE - 1]);
              }
              #pragma omp taskwait

              #pragma omp task firstprivate(equal1, equal2, x, y)
              {
              if (equal1 && equal2 && M[y][x] == maxiter)
              {
                    if (output2histogram)
                    {
                          #pragma omp atomic
                          histogram[maxiter - 1] += (TILE * TILE);
                    }
                    long color = (long)((maxiter - 1) * scale_color) + min_color;
                    for (int py = y; py < y + TILE; py++)
                    {
                       #pragma omp task firstprivate(x, y, py)
                       for (int px = x; px < x + TILE; px++)
                       {
                            M[py][px] = M[y][x];
                            if (output2display)
                            {
                            /* Scale color and display point  */
                            if (setup_return == EXIT_SUCCESS)
                            {
                                        #pragma omp critical
                                        {
                                        XSetForeground(display, gc, color);
                                        XDrawPoint(display, win, gc, px, py);
                                        }
                            }
                            }
                     }
                    }
              }
              else
              {
                          // Calcular
                 for (int py = y; py < y + TILE; py++) {
                      #pragma omp task firstprivate(x, y, py)
                      for (int px = x; px < x + TILE; 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:

If we look at Figure 8, we can observe that the elapsed time of this code, once applied the Finer Grain Strategy, can be reduced by more than a 1000% as the number of processors increases. At the beginning, we can see very large jumps of time as we add processors but as soon as we use 6, this number is not so big. We also have to point out that the program seems to have a very good scalability.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	3.03	1.53	0.78	0.55	0.42	0.35	0.30	0.26	0.23
Speedup	1.00	1.99	3.89	5.48	7.22	8.59	10.20	11.73	13.24
Efficiency	1.00	0.99	0.97	0.91	0.90	0.86	0.85	0.84	0.83

Figure 8: Image of the Modelfactors's execution metrics analysis

In this second table, more information is provided but in general no metric decreases for more than a 10% in exception of Global efficiency, which if we compare it with the result obtained in the tile version before, we see a huge increase of almost 50%, obtaining a final result of 82%. We observe that Parallelization strategy efficiency is quite high even in the 16 number of processors case, this is thanks to the good Load balancing that the code has because of the creation of tasks with a finer granularity so consequently we avoid that a processor executes a huge more bigger amount of iterations than others.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.99%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.93%	99.31%	97.16%	91.33%	90.19%	85.89%	85.00%	83.81%	82.75%
Parallelization strategy efficiency	99.93%	99.50%	98.56%	97.59%	96.58%	95.03%	94.18%	93.00%	91.92%
Load balancing	100.00%	100.00%	99.59%	99.45%	99.31%	97.62%	99.14%	97.69%	98.52%
In execution efficiency	99.93%	99.50%	98.97%	98.13%	97.26%	97.35%	95.00%	95.19%	93.30%
Scalability for computation tasks	100.00%	99.81%	98.58%	93.59%	93.38%	90.38%	90.25%	90.12%	90.03%
IPC scalability	100.00%	99.86%	99.78%	99.77%	99.77%	99.77%	99.78%	99.73%	99.78%
Instruction scalability	100.00%	99.98%	100.00%	100.00%	100.03%	100.01%	100.00%	100.01%	100.02%
Frequency scalability	100.00%	99.98%	98.80%	93.80%	93.57%	90.57%	90.45%	90.35%	90.21%

Figure 9: Image of the Modelfactors's efficiency metrics analysis

The final table reveals a notable difference in the synchronization overhead between the two versions. The percentage of synchronization overhead per explicit task is significantly lower in the new version compared to the Tile version. As previously explained, this is because we now create a larger number of tasks (8384, to be exact), which leads to a negligible increase in scheduling overhead but a substantial reduction in synchronization overhead. Consequently, the time per explicit task is significantly lower in the new version compared to the Tile version.

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	8384.0	8384.0	8384.0	8384.0	8384.0	8384.0	8384.0	8384.0	8384.0
LB (number of explicit tasks executed)	1.0	0.76	0.38	0.27	0.22	0.17	0.15	0.13	0.12
LB (time executing explicit tasks)	1.0	1.0	0.99	0.99	0.99	0.97	0.99	0.98	0.98
Time per explicit task (average us)	360.83	361.74	366.82	386.21	386.84	399.37	399.46	399.6	399.52
Overhead per explicit task (synch %)	0.0	0.24	1.09	2.03	3.1	4.73	5.5	6.66	7.93
Overhead per explicit task (sched %)	0.07	0.25	0.31	0.28	0.25	0.25	0.26	0.26	0.27
Number of taskwait/taskgroup (total)	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0

Figure 10: Image of the Modelfactors's statistics explicit tasks in parallel analysis

Paraver Analysis: Now in this version, if we take a look on the figure 4 and the figure 6 we observe that at the beginning there is a period of time where the parallelism is not being exploited at its maximum since now we have a starting period of scheduling and synchronization due to the new tasks created, therefore there is some time where most of the processors does not execute any task, but after it, we see that all processors execute tasks in almost all the execution of the program, except the first processor which seems to be the one in charge to schedule most of the synchronization. All that is thanks to the finer granularity of the tasks that leads to a better distribution of tasks between processors, deriving in a huge parallelism exploitation.

As observation we can perfectly see in the figure 4 in red intervals and in the figure 6 as a diminution of the instantaneous parallelism is that both show how the taskwait constructs affect to the execution, since we can foresee that those red intervals in the middle of the code seem to represent the moment in which the tasks of checking horizontal and vertical borders are being finished and therefore they have to wait ones to the others before going to the computation part.

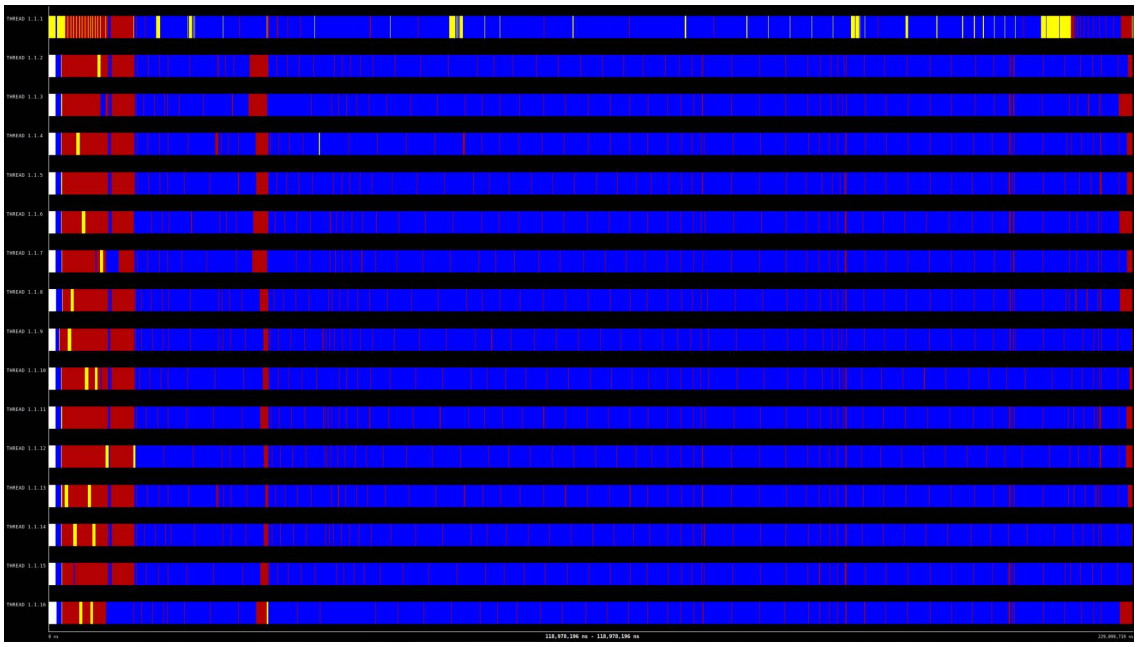


Figure 11: Image of paraver's execution trace

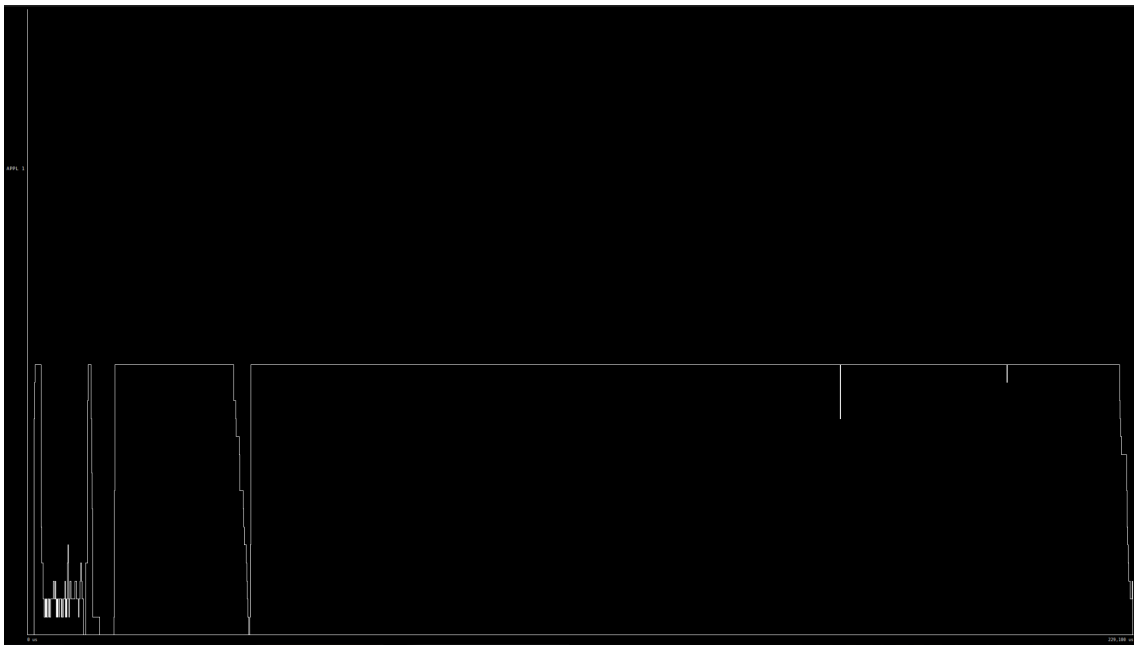


Figure 12: Image of paraver's instantaneous parallelism

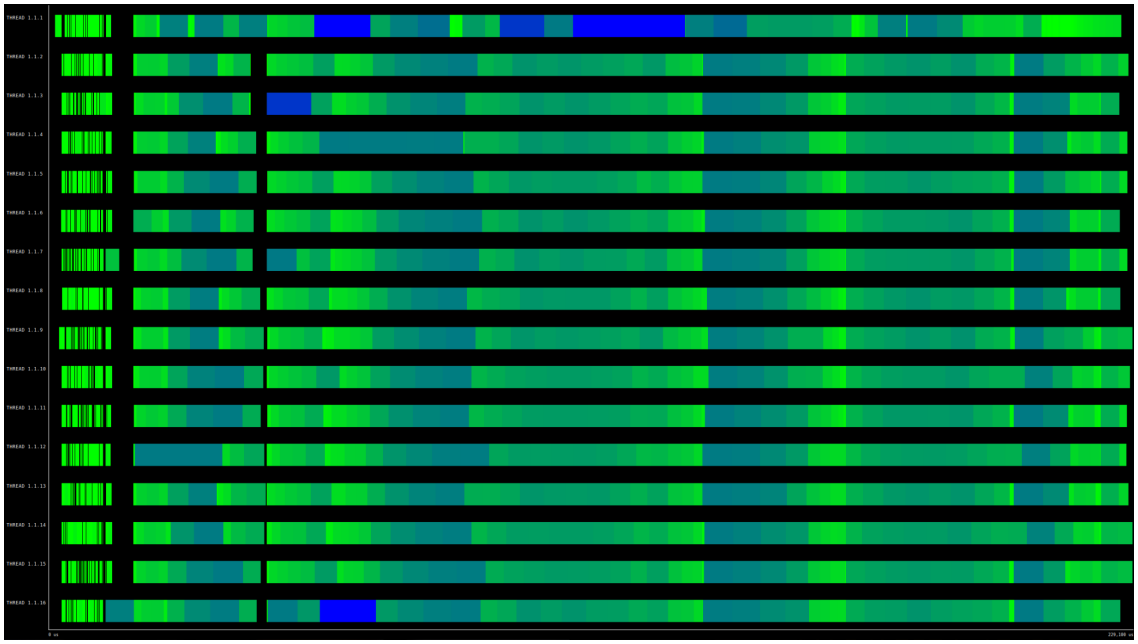


Figure 13: Image of paraver's explicit tasks executed duration

Strong scalability: In this version, in order to analyze the scalability of the code, we can also take a look at the figure 14, which shows a graphic in which we compare the threads used in the execution with the speed-up obtained in it.
As commented before in the Modelfactors analysis, we see that we obtain more interesting results, we see now the line of the function that represents de scalability of the program is now much more fitted with the dashed line, we see that when is executed with 20 threads, the slope of the function starts to decrease a little bit compared with the x = y function, but nevertheless, as said before, we've achieved a huge improvement in parallelising the code.

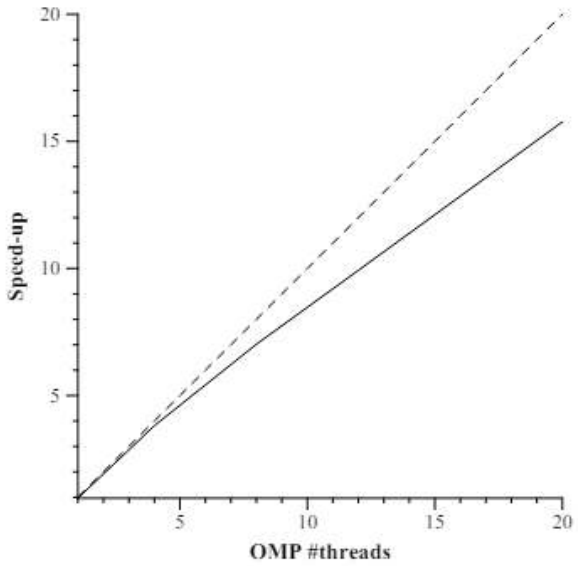


Figure 14: Image of the scalability graphic

2. Recursive task decomposition

2.1. Leaf version:

Code: In order to adapt the leaf strategy from the lab3 with OpenMP constructs, we've added the parallel and single construct before the call to the mandel_tile function, moreover apart from the task added by default in before the checking of the horizontal and vertical borders and the atomic and critical constructs from the previous laboratory session, we've added an explicit task construct in the base case of the execution of the function.

C/C++

```
void mandel_tiled_rec(int M[ROWS][COLS], int NRows, int NCols, int start_fil, int start_col, double CminR, double CminI, double CmaxR, double CmaxI,double scale_real, double
scale_imag, int maxiter)
{
  int equal;
  int x, y;

  equal = 1;
  y = start_fil;
  x = start_col;
              for (int px=x; px<x+NCols; px++) {
              M[y][px]       = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, y, scale_real, scale_imag, maxiter);
              M[y+NRows-1][px] = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, y+NRows-1, scale_real, scale_imag, maxiter);
                           equal = equal && (M[y][x] == M[y][px]);
              equal = equal && (M[y][x] == M[y+NRows-1][px]);
  }
  for (int py=y+1; py<y+NRows-1; py++){
              M[py][x]       = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, x, py, scale_real, scale_imag, maxiter);
              M[py][x+NCols-1] = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, x+NCols-1, py, scale_real, scale_imag, maxiter);
              equal = equal && (M[y][x] == M[py][x]);
              equal = equal && (M[y][x] == M[py][x+NCols-1]);
  }
  if (equal && M[y][x]==maxiter)
  {
              #pragma omp task
              {
     if (output2histogram)
              {
              #pragma omp atomic
                histogram[M[y][x]-1]+=(NRows*NCols);
              }
     long color = (long) ((M[y][x]-1) * scale_color) + min_color;
              for (int py=y; py<y+NRows; py++)
              for (int px=x; px<x+NCols; px++)
              {
              M[py][px] = M[y][x];
                 if (output2display)
                              {
                              if (setup_return == EXIT_SUCCESS)
                                 {
              #pragma omp critical
              {
                                          XSetForeground (display, gc, color);
                                          XDrawPoint (display, win, gc, px, py);
                                          }
              }
                              }
              }
              }
  }
  else {
              if (NCols <= TILE)
              {
              #pragma omp task
              for (int py=y; py<y+NRows; py++)
              for (int px=x; px<x+NCols; 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);
                              }
              }
              }
              }
              }
```

```
                else
                {
                if (NRows > TILE)
                {
                mandel_tiled_rec(M, NRows/2, NCols/2, start_fil, start_col, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
                mandel_tiled_rec(M, NRows/2, NCols/2, start_fil, start_col+NCols/2, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
                mandel_tiled_rec(M, NRows/2, NCols/2, start_fil+NRows/2, start_col, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
                mandel_tiled_rec(M, NRows/2, NCols/2, start_fil+NRows/2, start_col+NCols/2, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
                }
                else
                {
                mandel_tiled_rec(M, NRows, NCols/2, start_fil, start_col, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
                mandel_tiled_rec(M, NRows, NCols/2, start_fil, start_col+NCols/2, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
                }

                }
    }
  }
```

Modelfactor Analysis: In that first case of the recursive implementation, we can see by looking at the figure 15, the speedup just increases when increasing the number of threads from 1 to 2, passing from 1 to 1.34, after this point every raise of threads, just leads to a worse speedup result, far away from increasing the speedup's value, we obtain an even lower value, arriving to a value of 1.21 that is maintained from 10 threads to 16. On the other hand we can see that the efficiency has a horrible performance since it reduces by 33% in the first 2 steps and after, it slowly goes down till arriving at a value of 8% with 16 threads. Overall it seems that the fact of adding threads, instead of helping to exploit the parallelism, penalizes.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	2.01	1.50	1.54	1.61	1.63	1.66	1.66	1.66	1.66
Speedup	1.00	1.34	1.30	1.25	1.23	1.21	1.21	1.21	1.21
Efficiency	1.00	0.67	0.33	0.21	0.15	0.12	0.10	0.09	0.08

Figure 15: Image of the Modelfactors's execution metrics analysis

If we take a look at the figure 16 we can reassure the speedup values seen above, moreover, if we take a look at the load balance we obtain even worse results than in the first iterative version. We observe that the load balancing it performs in the same way as the parallelization strategy efficiency, initially it's reduced by a third part and as we add more threads to the execution we arrive at a value of 8.6%. That lets us foresee that the main contributor to the disappointing results could be a bad load balance between tasks.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.99%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.80%	66.81%	32.46%	20.81%	15.36%	12.05%	10.07%	8.61%	7.53%
Parallelization strategy efficiency	99.80%	67.21%	33.48%	22.46%	16.85%	13.52%	11.29%	9.69%	8.49%
Load balancing	100.00%	68.09%	34.04%	22.79%	17.07%	13.69%	11.42%	9.81%	8.60%
In execution efficiency	99.80%	98.72%	98.36%	98.59%	98.70%	98.78%	98.82%	98.77%	98.68%
Scalability for computation tasks	100.00%	99.39%	96.95%	92.63%	91.19%	89.11%	89.16%	88.82%	88.71%
IPC scalability	100.00%	99.28%	98.73%	98.62%	98.52%	98.34%	98.54%	98.40%	98.40%
Instruction scalability	100.00%	100.13%	100.26%	100.25%	100.24%	100.24%	100.23%	100.23%	100.24%
Frequency scalability	100.00%	99.98%	97.94%	93.69%	92.35%	90.40%	90.28%	90.06%	89.93%

Figure 16: Image of the Modelfactors's efficiency metrics analysis

To conclude with the modelfactors analysis we can see in figure 17 that there is an extremely high number of tasks created: 15244, which each lasts just a mean of 25us, that shows that probably by reducing the number of tasks or grouping them, the efficiency would increase significantly. Furthermore we can also take a look at the synchronization overhead per task and we can see that it increases by around 300% per thread added to the execution (600% between columns), while the scheduling overhead remains constant no matter how many threads are involved in the execution.

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	15244.0	15244.0	15244.0	15244.0	15244.0	15244.0	15244.0	15244.0	15244.0
LB (number of explicit tasks executed)	1.0	0.71	0.96	0.91	0.9	0.85	0.93	0.87	0.84
LB (time executing explicit tasks)	1.0	0.5	0.92	0.82	0.82	0.74	0.71	0.64	0.61
Time per explicit task (average us)	34.91	35.33	35.95	37.96	38.26	39.22	39.14	39.17	39.2
Overhead per explicit task (synch %)	0.0	179.01	744.49	1285.94	1854.82	2401.28	2954.13	3515.48	4069.56
Overhead per explicit task (sched %)	0.75	3.49	4.55	3.81	3.46	3.21	3.08	3.21	3.42
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 17: Image of the Modelfactors's statistics explicit tasks in parallel analysis

Paraver Analysis: Thanks to paraver, we can see the horrible exploitation of parallelism that achieves this version, if we observe the paraver trace for the bigger amount of threads, we can see that the big majority of the time of the execution, the threads do not execute any task, in red color, except the thread number one, which seems to be the one in charge of almost the execution of all tasks. Moreover, we can see that thread number one spends a huge amount of time synchronizing all the tasks, in yellow color. If we take a look at the instantaneous parallelism we see that averagely the exploitation is 0 but in some moments of the execution when some tasks match and execute simultaneously we see some ups and downs. Finally if we look at the paraver's explicit tasks executed duration, we can see more in detail how bad affects the high number of tasks created since we see how much time do threads waste in doing nothing.

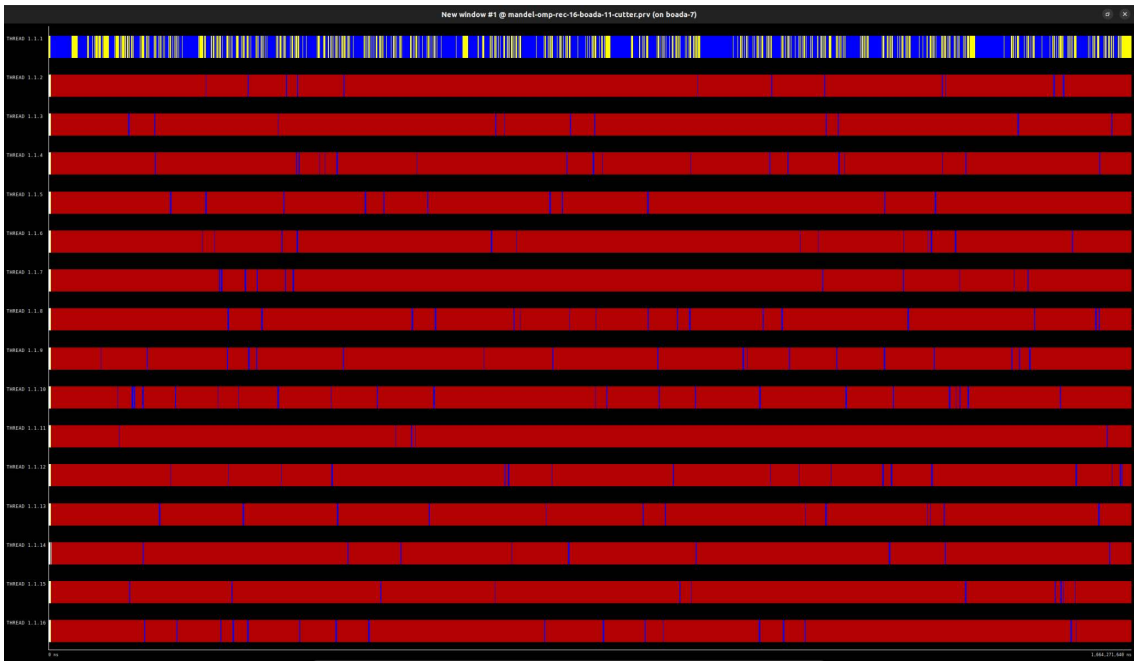


Figure 18: Image of paraver's execution trace

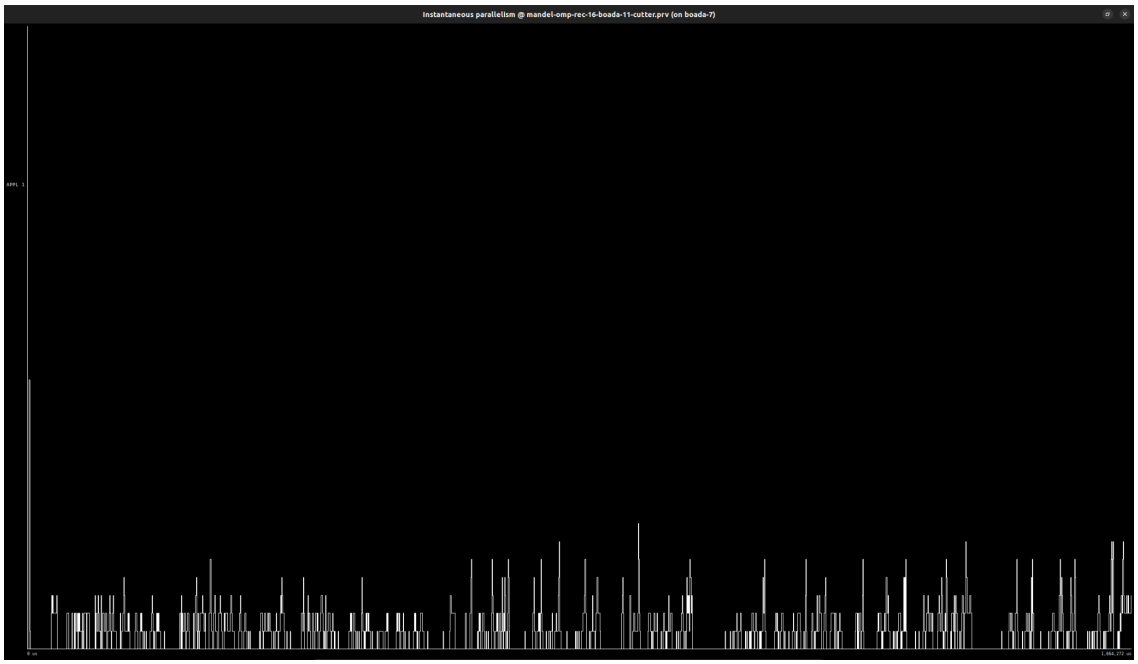


Figure 19: Image of paraver's instantaneous parallelism

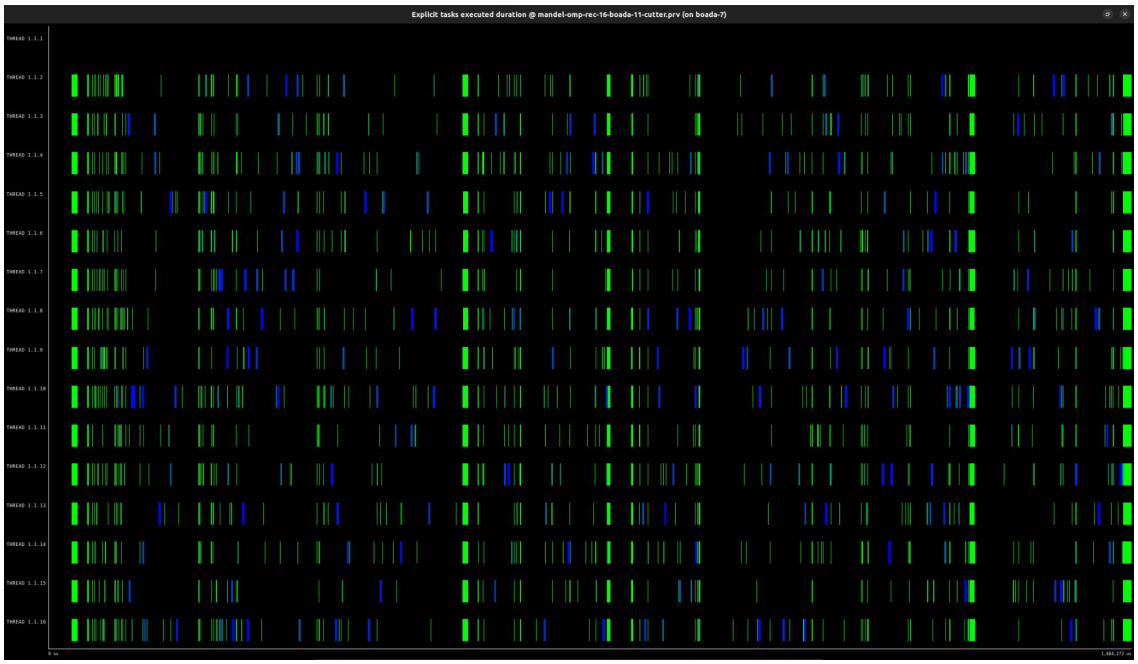


Figure 20: Image of paraver's explicit tasks executed duration

Strong scalability: As we expected after the modelfactor and paraver analysis, the scalability leaves much to be desired, we see that far away from obtaining a slope near the dashed line, we obtain a horrible result since as explained more in detail in the model factors analysis, as we increase the number of threads, the speedup goes down and down.
We can easily conclude that the leaf strategy is the worst of the four versions implemented in that laboratory practice.

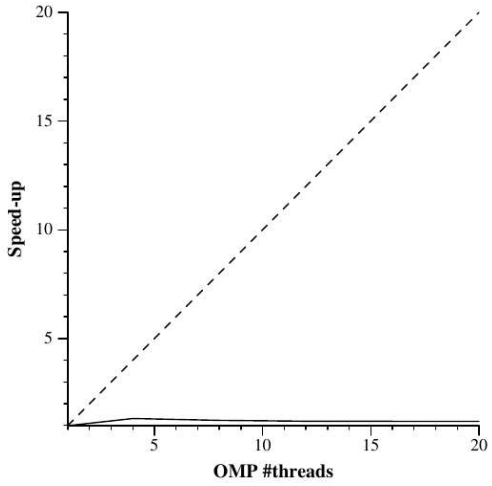


Figure 21: Image of the scalability graphic

2.2. Tree version:

Code: In order to implement the tree recursive version, we've added the pragma omp parallel and single in the main before calling the mandel_tiled_rec function. We also needed to use pragma omp atomic and critical in those lines of codes (in orange color) where variables would be edited in the same moment due to the parallelization. In the end, with all these done, we just needed to put a pragma omp task (in yellow color) in those lines of codes where we put tareador_ON in lab3 that are in the recursive calls and not in the base case.

C/C++

```
void mandel_tiled_rec(int M[ROWS][COLS], int NRows, int NCols, int start_fil, int start_col, double CminR, double CminI, double CmaxR, double CmaxI,double scale_real, double
scale_imag, int maxiter)
{
  int equal;
  int x, y;

  equal = 1;
  y = start_fil;
  x = start_col;
    for (int px=x; px<x+NCols; px++) {
        M[y][px]      = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, y, scale_real, scale_imag, maxiter);
        M[y+NRows-1][px] = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, y+NRows-1, scale_real, scale_imag, maxiter);
              equal = equal && (M[y][x] == M[y][px]);
                  equal = equal && (M[y][x] == M[y+NRows-1][px]);
  }
  for (int py=y+1; py<y+NRows-1; py++){
        M[py][x]      = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, x, py, scale_real, scale_imag, maxiter);
        M[py][x+NCols-1] = pixel_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, x+NCols-1, py, scale_real, scale_imag, maxiter);
                  equal = equal && (M[y][x] == M[py][x]);
                  equal = equal && (M[y][x] == M[py][x+NCols-1]);
  }
  if (equal && M[y][x]==maxiter)
  {

    // Rellenar, son todos iguales
```

msi

intel INSIDE

Hay cambios de los que no te arrepientes nunca. Como este.

Business & Productivity

Si tuviéramos que definir esta línea de laptops en tres palabras, serían: ligeras, facheras y listas para el grind. Básicamente, el combo perfecto para cualquier estudiante que quiere un laptop sin luces RGB que parezca una nave espacial. Sí, MSI también piensa en la gente que tiene que hacer tareas y no solo en los gamers. Así que deja de sufrir con ese ladrillo viejo y pásate a la serie Business & Productivity. Tu espalda (y tu flow) te lo agradecerán

Procesador Intel® serie U

```
#pragma omp task
{
          if (output2histogram)
 {
    #pragma omp atomic
                        histogram[M[y][x]-1]+=(NRows*NCols);
 }
          long color = (long) ((M[y][x]-1) * scale_color) + min_color;
 for (int py=y; py<y+NRows; py++)
   for (int px=x; px<x+NCols; px++)
   {
      M[py][px] = M[y][x];
                          if (output2display)
                            {
                              if (setup_return == EXIT_SUCCESS)
                                        {
          #pragma omp critical
          {
                                    XSetForeground (display, gc, color);
                                    XDrawPoint (display, win, gc, px, py);
                                    }
        }
                              }
   }
 }
}
else {
  if (NCols <= TILE)
  {
          // Calcular
       for (int py=y; py<y+NRows; py++)
         for (int px=x; px<x+NCols; 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);
                          }
               }
             }
        }
  }
  else
  {
    if (NRows > TILE)
    {
      #pragma omp task
       mandel_tiled_rec(M, NRows/2, NCols/2, start_fil, start_col, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
      #pragma omp task
       mandel_tiled_rec(M, NRows/2, NCols/2, start_fil, start_col+NCols/2, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
      #pragma omp task
       mandel_tiled_rec(M, NRows/2, NCols/2, start_fil+NRows/2, start_col, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
      #pragma omp task
       mandel_tiled_rec(M, NRows/2, NCols/2, start_fil+NRows/2, start_col+NCols/2, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
    }
    else
    {
      #pragma omp task
       mandel_tiled_rec(M, NRows, NCols/2, start_fil, start_col, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
      #pragma omp task
       mandel_tiled_rec(M, NRows, NCols/2, start_fil, start_col+NCols/2, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
    }

  }
 }
}
```

Modelfactor Analysis: Compared to the leaf strategy that we implemented before, this version is much better in all characteristics as we have a higher speedup as we approach 16 cores, a lower time of elapse and a higher efficiency. We can observe that in the first 4 cores, speedup doubles for each core and the elapsed time is reduced by half.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	2.00	1.01	0.51	0.37	0.30	0.25	0.22	0.19	0.18
Speedup	1.00	1.98	3.89	5.35	6.72	7.94	9.23	10.30	11.30
Efficiency	1.00	0.99	0.97	0.89	0.84	0.79	0.77	0.74	0.71

Table 1: Analysis done on Fri Apr 26 09:56:14 AM CEST 2024, par1313

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

Here in table 2 we can see that speedup values are coherent and that this version of the program scales quite well, not arriving to 70% of efficiency nor scalability in any case. We have to comment that the load balancing numbers that we obtain are very good but they are not as good as the iterative finer grain version as that version had a higher granularity compared to this one.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.99%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.72%	98.94%	97.06%	88.95%	83.82%	79.28%	76.77%	73.48%	70.58%
Parallelization strategy efficiency	99.72%	98.61%	98.17%	95.17%	90.83%	88.21%	85.68%	82.10%	79.11%
Load balancing	100.00%	99.66%	98.80%	97.03%	92.83%	90.20%	86.98%	86.54%	82.07%
In execution efficiency	99.72%	98.95%	99.36%	98.08%	97.85%	97.79%	98.50%	94.88%	96.40%
Scalability for computation tasks	100.00%	100.33%	98.87%	93.47%	92.28%	89.88%	89.60%	89.50%	89.22%
IPC scalability	100.00%	99.84%	99.76%	99.37%	99.03%	99.04%	99.02%	99.01%	98.98%
Instruction scalability	100.00%	100.36%	100.37%	100.37%	100.36%	100.36%	100.36%	100.36%	100.36%
Frequency scalability	100.00%	100.13%	98.74%	93.72%	92.84%	90.42%	90.17%	90.07%	89.81%

Table 2: Analysis done on Fri Apr 26 09:56:14 AM CEST 2024, par1313

Figure 23: Image of the Modelfactors's efficiency metrics analysis

In this last table we have explicit tasks data and at first sight we can say that the recursive tree version is the one with the highest amount of explicit tasks executed. This conducts us to have a high but not that high percentage of overhead per explicit task in both synchronization and scheduling. As expected, the time of execution of these tasks is quite low.

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	20545.0	20545.0	20545.0	20545.0	20545.0	20545.0	20545.0	20545.0	20545.0
LB (number of explicit tasks executed)	1.0	0.78	0.55	0.58	0.56	0.6	0.58	0.5	0.61
LB (time executing explicit tasks)	1.0	1.0	0.99	0.98	0.95	0.94	0.91	0.92	0.89
Time per explicit task (average us)	96.36	96.73	98.19	104.66	107.65	111.43	112.97	113.78	116.7
Overhead per explicit task (synch %)	0.0	1.03	1.38	3.63	6.73	8.68	10.49	14.29	15.57
Overhead per explicit task (sched %)	0.28	0.34	0.38	1.22	2.8	3.71	4.98	5.64	7.94
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Fri Apr 26 09:56:14 AM CEST 2024, par1313

Figure 24: Image of the Modelfactors's statistics explicit tasks in parallel analysis

Paraver Analysis: With paraver providing us with charts, we can see that in the beginning of the execution there's the main core which schedules all tasks, after that scheduling the threads number 2, 6, 12 and 14 are the ones assigned to the first 4 explicit tasks created for the first 4 branches of the tree. Afterwards, we see that as every recursive call creates 4 tasks, we have all threads in charge of

some tasks since each one of these 4 initial threads creates 4 tasks more, 4*4 = 16 which is the total number of threads of the execution, and finally having a long synchronization time before executing all tasks that are created, nevertheless we see that we have to pay a quite big amount of time for synchronization and scheduling overheads, in yellow color, which it's perfectly represented in the figure 26 where we see a period where there is an increase of parallelism with ups and downs till arrive to the maximum exploitation level.

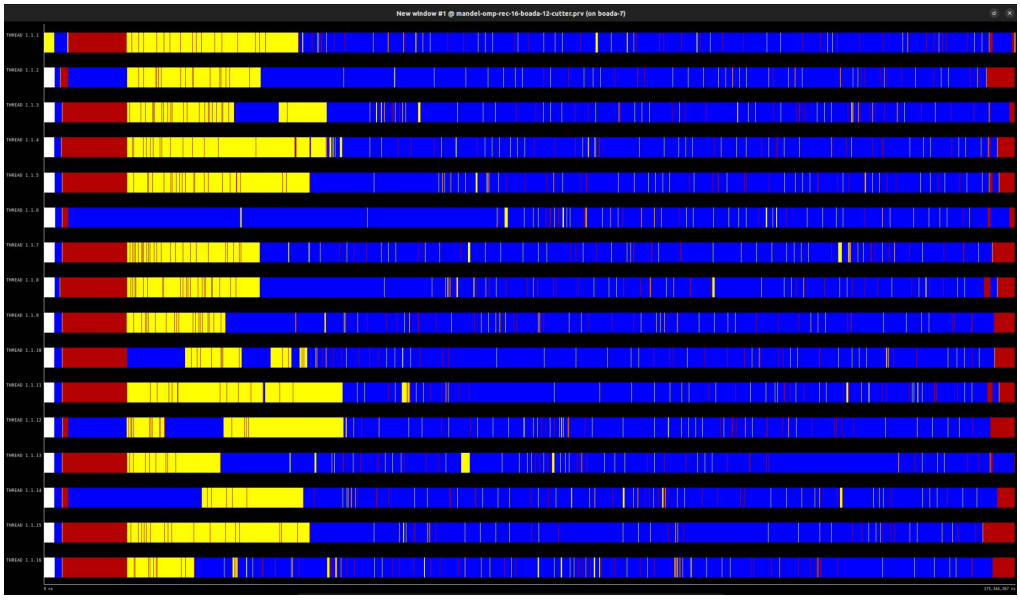


Figure 25: Image of paraver's execution trace

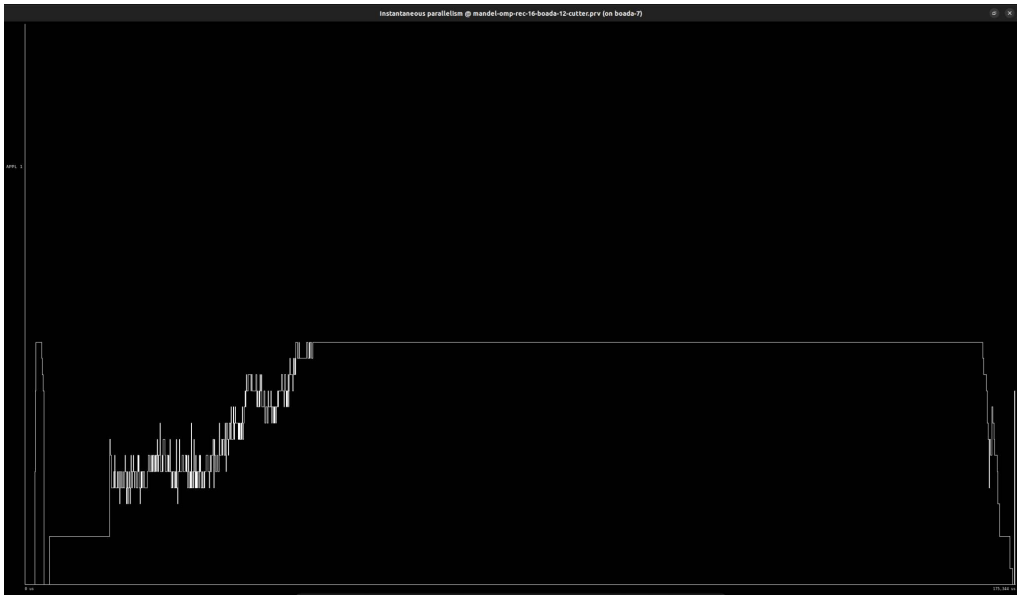


Figure 26: Image of paraver's instantaneous parallelism

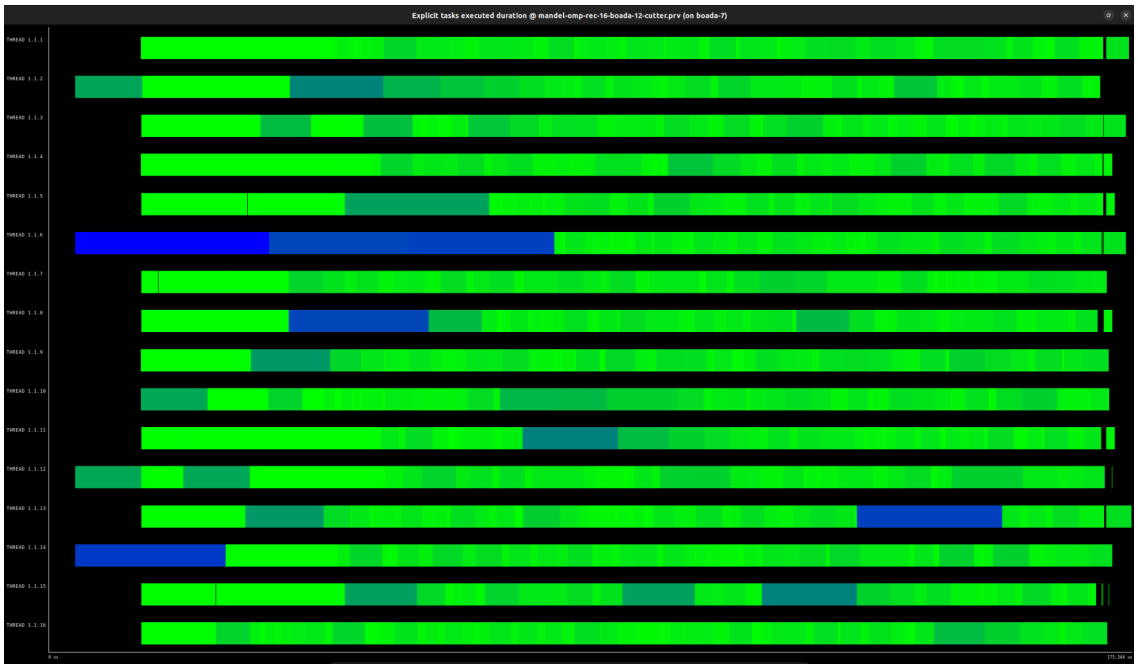


Figure 27: Image of paraver's explicit tasks executed duration

Strong scalability: The scalability in tree version increases linearly until reaching 5 threads because then it starts separating from that linearity up to a 25/30% until the end. This value is quite good, being almost the same as the iterative finer grain version.

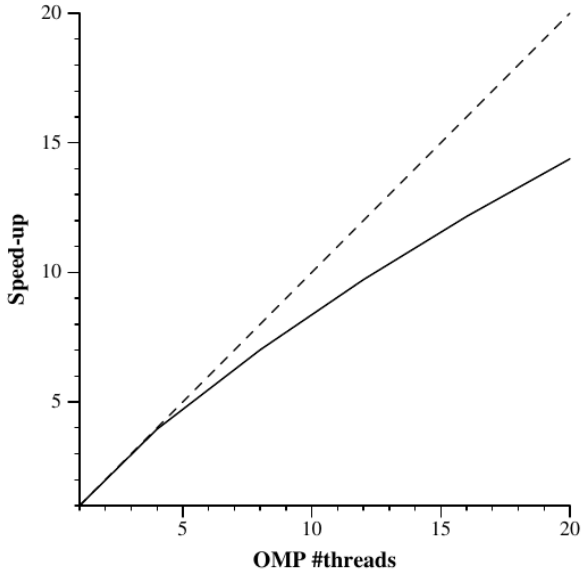


Figure 28: Image of the scalability graphic

3 Summary of the elapsed execution times

	Number of Threads					
Version	1	2	4	8	12	16
Iterative: Tile	3.03	1.53	0.89	0.67	0.58	0.57
Iterative: Finer grain	3.03	1.53	0.78	0.42	0.30	0.23
Recursive: Leaf	2.01	1.50	1.54	1.63	1.66	1.66
Recursive: Tree	2.00	1.01	0.51	0.30	0.22	0.18