PAR Laboratory Assignment

Lab 4: Parallel Task Decomposition Implementation and Analysis

Q2 2023-24, May 2024

Arnau Claramunt Soler - par1302
David García Arévalo - par1304

Contents

1 Iterative task decomposition

In our completion of the iterative parallel code, where it exploits parallelism and takes into account the dependencies/data sharing we analysed in the previous laboratory to avoid concurrency problems.

1.1 Iterative parallel code exploiting Parallelism

We did this modifications in the file: *mandel-omp-iter.c*

```
void mandel_tiled (int M[ROWS][COLS], ...)
{

  ...

  for (int y = 0; y < ROWS; y += TILE)
    for (int x = 0; x < COLS; x += TILE)
    {
        #pragma omp task
        {
            ...

            if (equal && M[y][x]==maxiter) {

                if (output2histogram)
                {
                    #pragma omp atomic
                    histogram[maxiter-1]+=(TILE*TILE);
                }
                    ...

                    if (setup_return == EXIT_SUCCESS)
                    {
                        #pragma omp critical
                        XSetForeground (display, gc, color);
                        XDrawPoint (display, win, gc, px, py);
                    }

                ...

            }
            else {

              ...

                if (output2histogram)
                {
                  #pragma omp atomic
                  histogram[M[py][px]-1]++;
                }
                    ...

                    if (setup_return == EXIT_SUCCESS)
                    {
                        #pragma omp critical
                        XSetForeground (display, gc, color);
                        XDrawPoint (display, win, gc, px, py);
                    }

                ...

            }
        }
    }
}

```

```
57
58 int main(int argc, char *argv[])
59 {
60     ...
61     #pragma omp parallel
62     #pragma omp single
63     mandel_tiled ((int (*)[width]) Hmatrix, ...);
64     ...
65 }
```

First of all we wanted a parallel code, so we had to use **#pragma omp parallel** and then a **#pragma omp single**, in the region we wanted to parallelize. See at lines 61 - 62 of the code.

Also, in the computation of the mandelbrot, each tile is a task, so we create tasks with **#pragma omp task**. This command was already specified in the original code provided.

Moreover, due to the dependencies at the color section this part must be executed serializable, so that a **#pragma omp critical** is necessary. See at lines 24 - 26, for example. It doesn't make sense to do an atomic, because internally there are more than one instructions.

Then, to avoid data sharing problems, when accessing to the histogram we have to put a **#pragma omp atomic**. You can see it at lines 17 - 19, for example.

Execution and Check the Correctness

After the execution of both codes with the respective changes using omp commands, the resultant mandelbrot images are the followings:

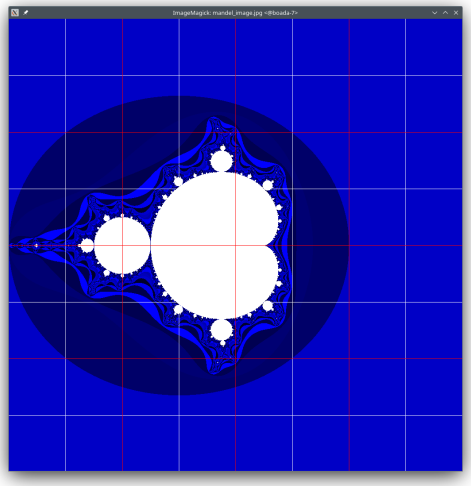


Figure 1: sequential code

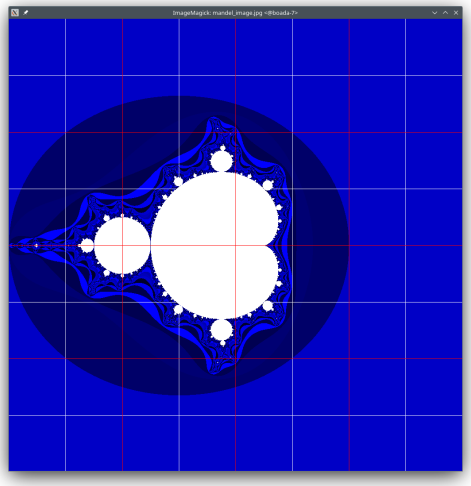


Figure 2: parallel code with omp

Comparison of histograms:

1. Execution of our code. Rename mandel_histogram.out to mandel_omp_histogram.out
2. Execution of the code they gave us. It creates the .out called mandel_histogram.out
3. Comparison between both .out with the *diff* command
4. Look for differences

```
par1304@boada-7:~/lab4$ diff mandel_histogram.out mandel_omp_histogram.out
par1304@boada-7:~/lab4$
```

Figure 3: Comparison of histograms

The diff command if the files doesn't match sends you a message. So, as there are no messages left, we observe that there are no differences between implementations.

Now, let's look at the execution times:

```
Computation of the Mandelbrot set with:
    center = (0, 0)
    size = 2
    maximum iterations = 10000

Total execution time (in seconds): 2.922629

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: mandel_image.jpg
```

Figure 4: original sequential execution

```
Computation of the Mandelbrot set with:
    center = (0, 0)
    size = 2
    maximum iterations = 10000

Total execution time (in seconds): 0.582025

Mandelbrot set: Computed
Histogram for Mandelbrot set: Computed
Writing output file to disk: mandel_image.jpg
```

Figure 5: optimized parallel execution

After comparing the original sequential implementation, with our modification exploiting parallelism we obtained a better total execution time in he .o file

In the original sequential implementation, the execution lasts 2.92 seconds whereas in our parallel implementation solving the dependencies, the execution time is 0.58 seconds.

1.2 Finer Grain implementation

This version using OpenMP we have to create finer grain tasks in the parallel strategy. Which exploits parallelism both during the check of the borders and during the full computation of the tile or fill it with the same color.

1. Task for check horizontal borders
2. Task for check vertical borders
3. Task for entire if-else conditional
4. Inside if, task for each py iteration of fill
5. Inside else, task for each py iteration of computation

In the first part, the code is the same as before.

```
int main(int argc, char *argv[])
{
    ...
    #pragma omp parallel
    #pragma omp single
    mandel_tiled ((int (*)[width]) Hmatrix, ...);
    ...
}
```

Now we do the specific mentioned changes in order to obtain finer grain.

```
void mandel_tiled (int M[ROWS][COLS], ...)
{
  int equal;
  int equalh;
  int equalv;

  for (int y = 0; y < ROWS; y += TILE)
    for (int x = 0; x < COLS; x += TILE)
    {
        equal = 1;
        equalh = 1;
        equalv = 1;

        # pragma omp task shared(equalh)
        for (int px = x; px < x + TILE; px++) {
            ...
            equalh = equalh && (M[y][x] == M[y][px]);
            equalh = equalh && (M[y][x] == M[y + TILE - 1][px]);
        }
        # pragma omp task shared(equalv)
        for (int py = y; py < y + TILE; py++) {
            ...
            equalv = equalv && (M[y][x] == M[py][x]);
            equalv = equalv && (M[y][x] == M[py][x + TILE - 1]);
        }
        # pragma omp taskwait


        # pragma omp task
        if (equalh && equalv && equal && M[y][x]==maxiter) {

            if (output2histogram)
            {
               # pragma omp atomic
               histogram[maxiter-1]+=(TILE*TILE);
            }

            ...
            for (int py = y; py < y + TILE; py++)

                # pragma omp task
                for (int px = x; px < x + TILE; px++) {
                    ...
                    if (output2display)
                    {
                      ...
                        if (setup_return == EXIT_SUCCESS)
                        {
                            # pragma omp critical
                            XSetForeground (display, gc, color);
                            XDrawPoint (display, win, gc, px, py);
                        }
                    }
                }
        }

        else
            for (int py = y; py < y + TILE; py++)

                # pragma omp task
                for (int px = x; px < x + TILE; px++)
                {
                    ...
                    if (output2histogram)
                    {
                        # pragma omp atomic
                        histogram[M[py][px]-1]++;
                    }
                    if (output2display)
```

```
70                      {
71                          ...
72                          if (setup_return == EXIT_SUCCESS)
73                          {
74                              # pragma omp critical
75                              XSetForeground (display, gc, color);
76                              XDrawPoint (display, win, gc, px, py);
77                          }
78                      }
79                  }
80      }
81  }
```

Execution and Check the Correctness

After the execution of the improved version with finer grain and the original iterative version, the resulting images are the same. That's to say that both *mandel_image.jpg* and *mandel_histogram.out* have no differences between the code provided in a sequential way. The procedure to obtain this results is done following the same steps as we did in the previous: *Execution and Check the Correctness.*

1.3 Iterative Performance Analysis

In this subsection we will compare the Parallell omptimized implementation agaist the Finer Grain.

1.3.1 Modelfactors analysis

Version 1: Parallel Implementation

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	0.91	0.84	0.67	0.59	0.58	0.57	0.57
Speedup	1.00	1.98	3.34	3.59	4.53	5.12	5.22	5.33	5.33
Efficiency	1.00	0.99	0.83	0.60	0.57	0.51	0.44	0.38	0.33

Table 1: Analysis done on Fri Apr 19 09:42:47 AM CEST 2024, par1304

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%	98.89%	83.48%	59.83%	56.62%	51.23%	43.55%	38.10%	33.30%
Parallelization strategy efficiency	100.00%	98.91%	84.86%	64.02%	60.94%	56.90%	48.47%	42.44%	37.21%
Load balancing	100.00%	98.96%	84.91%	64.08%	61.01%	57.01%	48.57%	42.53%	37.31%
In execution efficiency	100.00%	99.95%	99.94%	99.91%	99.89%	99.80%	99.78%	99.78%	99.74%
Scalability for computation tasks	100.00%	99.98%	98.37%	93.46%	92.90%	90.04%	89.85%	89.77%	89.49%
IPC scalability	100.00%	99.99%	99.95%	99.96%	99.95%	99.96%	99.96%	99.95%	99.96%
Instruction scalability	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Frequency scalability	100.00%	99.99%	98.42%	93.50%	92.95%	90.07%	89.88%	89.81%	89.53%

Table 2: Analysis done on Fri Apr 19 09:42:47 AM CEST 2024, par1304

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	0.85	0.64	0.61	0.57	0.48	0.42	0.37
Time per explicit task (average us)	47346.06	47346.63	48107.88	50617.57	50890.86	52465.97	52514.83	52512.68	52583.96
Overhead per explicit task (synch %)	0.0	1.06	17.78	56.12	63.98	75.58	106.23	135.72	169.07
Overhead per explicit task (sched %)	0.0	0.01	0.01	0.01	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

Table 3: Analysis done on Fri Apr 19 09:42:47 AM CEST 2024, par1304

Version 2: Finer Grain Implementation

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	3.03	1.52	0.78	0.55	0.42	0.35	0.30	0.26	0.23	0.22	0.20
Speedup	1.00	1.99	3.89	5.49	7.15	8.61	10.09	11.70	13.09	13.86	14.83
Efficiency	1.00	0.99	0.97	0.91	0.89	0.86	0.84	0.84	0.82	0.77	0.74

Table 1: Analysis done on Sat Apr 27 11:47:32 AM CEST 2024, par1302

Overview of the Efficiency metrics in parallel fraction, ϕ=99.99%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	99.93%	99.40%	97.19%	91.39%	89.33%	86.10%	84.09%	83.60%	81.89%	77.06%	74.20%
Parallelization strategy efficiency	99.93%	99.52%	98.71%	97.75%	96.38%	95.28%	93.32%	92.90%	91.02%	85.97%	82.89%
Load balancing	100.00%	99.98%	99.72%	99.74%	99.29%	98.94%	97.70%	97.81%	98.64%	95.57%	93.55%
In execution efficiency	99.93%	99.54%	98.99%	98.01%	97.07%	96.31%	95.52%	94.98%	92.27%	89.96%	88.61%
Scalability for computation tasks	100.00%	99.88%	98.46%	93.50%	92.68%	90.37%	90.11%	90.00%	89.97%	89.64%	89.51%
IPC scalability	100.00%	99.88%	99.78%	99.79%	99.73%	99.81%	99.70%	99.73%	99.75%	99.69%	99.72%
Instruction scalability	100.00%	99.99%	100.01%	100.01%	100.01%	100.01%	100.01%	100.01%	100.01%	100.02%	100.03%
Frequency scalability	100.00%	100.01%	98.66%	93.68%	92.93%	90.53%	90.37%	90.24%	90.18%	89.90%	89.73%

Table 2: Analysis done on Sat Apr 27 11:47:32 AM CEST 2024, par1302

Statistics about explicit tasks in parallel fraction											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	8384.0	8384.0	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	0.37	0.27	0.22	0.17	0.15	0.13	0.12	0.11	0.1
LB (time executing explicit tasks)	1.0	1.0	0.99	1.0	0.99	0.99	0.98	0.98	0.99	0.96	0.94
Time per explicit task (average us)	360.57	361.3	366.93	386.36	389.42	399.12	399.87	399.78	399.8	400.09	399.82
Overhead per explicit task (synch %)	0.0	0.23	0.97	1.91	3.31	4.46	6.39	6.65	8.9	15.25	19.46
Overhead per explicit task (sched %)	0.07	0.23	0.26	0.27	0.25	0.23	0.28	0.28	0.27	0.3	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	64.0	64.0

Table 3: Analysis done on Sat Apr 27 11:47:32 AM CEST 2024, par1302

Comparison between Versions

Let v1 be the first parallel implemented version and v2 the one following a finer grain strategy.

Comparing the parallel implementation with the finer grain, the second one it is slightly faster regarding execution time, as the number of processor increase. The main difference is in the speedup of the first parallel implementation with 16 threads reaches 5.33, and for the finer grain code has a value of 13.09, so that the second version is much better than the first one.

Is worth mentioning that the efficiency decreases rapidly in the version 1 whereas in the second one the efficiency is not as bad as we thought it's not heavily affected by the task creation and overheads. Regarding the load balancing the main issue we find of the parallel fraction of the v1

we can affirm that it's not well balanced. This is caused inside the parallel region where it has to do the computation of the Mandelbrot, and in the white part it has to do more computation, causing an unbalance and having tiles without calculations. If tasks are big, the imbalance is more notorious. On the other hand the v2 is stable with the Load balancing, so it's a good indication that Finer Grain works way better.

Taking into account the synchronisation overhead that tells us the wasted time that tasks wait between each other, we observe that due to the reduced number of executed tasks in v1 (64 to be more concrete), compared on the high number of tasks (8300 aprox.) in v2, which is exactly the granularity that we wanted to achieve, creating more explicit tasks, but they don't have as much as the other version.

In addition, in v1 the tasks on average, take more time to be executed than v2, because they are less and have to do more work. As there are more tasks than processors, the tasks will have to wait much more time between them in order to be executed, increasing the synch. overhead mentioned before.

1.3.2 Paraver analysis

Version 1: Parallel Implementation

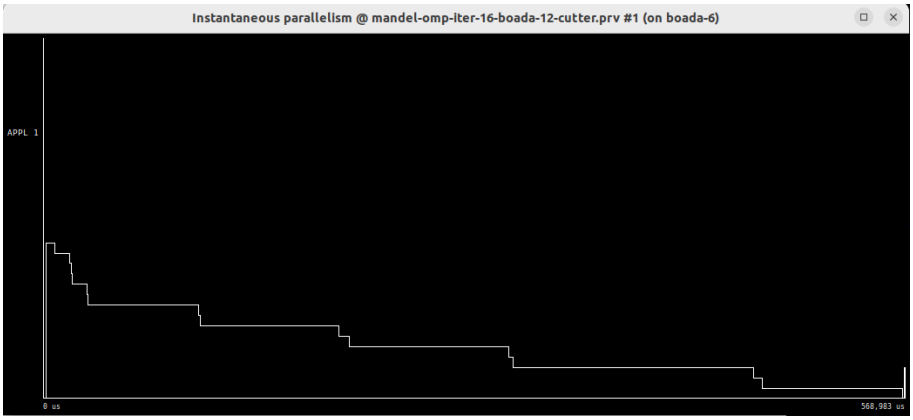


Figure 6: Useful/Instantaneous parallelism

Figure 7: OpenMP tasking/Explicit tasks function creation

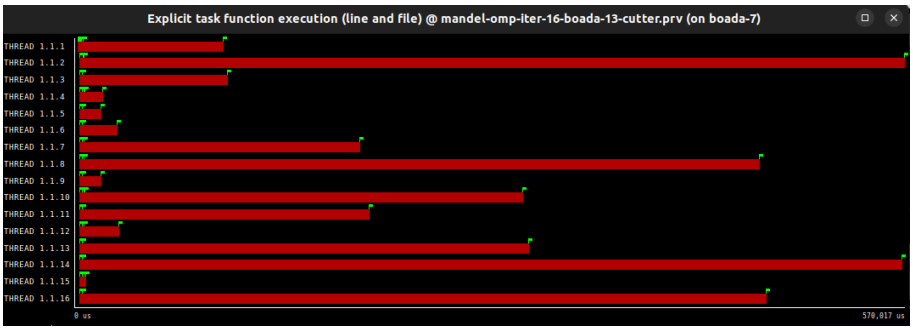


Figure 8: OpenMP tasking/Explicit tasks execution

Version 2: Finer Grain Implementation

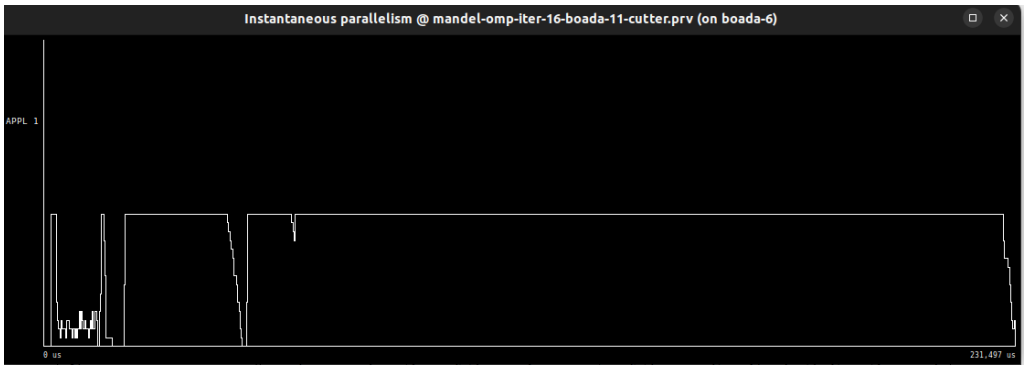


Figure 9: Useful/Instantaneous parallelism

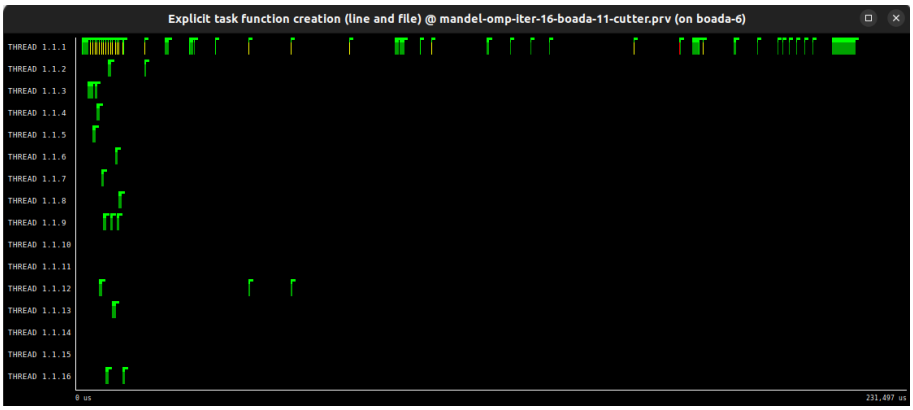


Figure 10: OpenMP tasking/Explicit tasks function creation

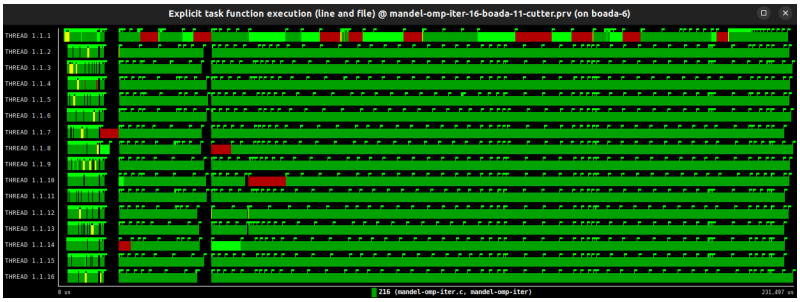


Figure 11: OpenMP tasking/Explicit tasks execution

Comparison between Versions

Referring to the parallel implementation, in the timeline of the Figure 6 of the instantaneous parallelism, the pattern starts to decrease in a slope pattern shape. And we see that parallelism efficiency decreases over time, with we can see also in the Table 2 of modelfactors (Parallelization strategy efficiency) which we refereed in the last section.

In the other instantaneous parallelism of v2 it is very constant, only being irregular in the beginning. But showing that at the end, this is a better implementation and exploits more the parallelism.

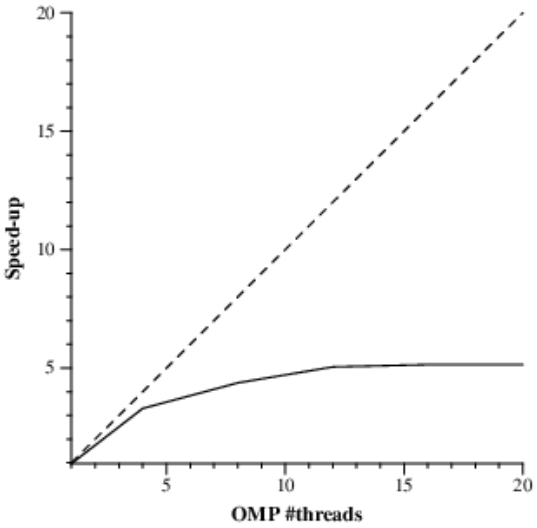
Shockingly the figure 7 of the creation of the explicit tasks, shows that only one thread creates all the tasks at the beginning. In the other case we see that at the v2, that follows a finer grain strategy there will be more tasks created, as it is shown at the figure 10, and more distributed over time and with different threads. The explicit tasks are not created all in the beginning, some are in the if or else part of the code, or with the py iterations of the fill or compute loops.

In Figure 8, we see the timeline of the execution for each explicit task. We observe that the times are very different depending on the thread. Going further, and surprisingly, all executed tasks appear in a red colour, meaning that tasks are waiting for its execution, increasing the synchronisation overhead. Also, they are not equally distributed, this is due to the tasks, that have to do the computation of the white part of the Mandelbrot.

Finally, for the explicit tasks execution of the v2, they are more equally distributed. As there is more green, which is associated with less overhead and they are executed at all the threads. Resulting in a better implementation, even though it has much more tasks than v1.

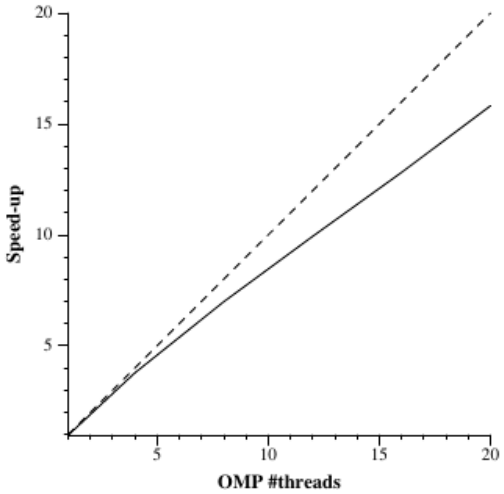
1.3.3 Strong Scalability

Version 1: Parallel Implementation



par1304
Speed-up wrt sequential time (mandel funtion only)
Generated by par1304 on Mon Apr 22 05:32:04 PM CEST 2024

Version 2: Finer Grain Implementation



par1302
Speed-up wrt sequential time (mandel funtion only)
Generated by par1302 on Fri Apr 26 09:49:04 AM CEST 2024

Comparison between Versions

In this part, we will discuss the strong scalability. An initial observation would be that we can observe in a graphical way the behaviour of the speedup mentioned when comparing the modelfactor tables, depending on the number of threads.

The most important behaviour of the programs, would be a constant lineal growth, like the dashed line in the diagonal. This would indicate that it escalates perfectly with more threads, making worth to run the code with more of them (in the case we don't increase the size of the problem). But in reality, the first version, don't achieve a constant growth, resulting in a stagnation after 10 threads.

Having said that, the pattern it follows, it's caused, by a worst performing implementation we and seen in previous sections, and the parallel efficiency is not maintained, causing it to not continue improving the speedup as the threads increase, causing this logarithmic pattern shape.

Furthermore, in the first plot of v1, there isn't load balancing, we could improve it if we reduce the size of the TILE for example (down to 16). This would imply a better strong scalability. Is it worth mentioning that the bigger the TILES, the more pixelated the resulting image will appear.

For the second version, as seen in the modelfactors, the parallel efficiency was nearly constant and now, increasing the number of threads, the speed-up is nearly the optimal. This is the desirable behaviour we want when we try to exploit parallelism, and we personally think that is a good implementation.

Overall, the curve plotted at the v1 is much more noticeable than in the v2. And in the v2, after 10 threads, it don't remains constant (horizontal) as the v1 does. To conclude, the v2 compared with v1 is much better, and it can also scale better.

2 Recursive task decomposition

2.1 Recursive Leaf parallel code exploiting parallelism

We did this modifications in the file: *mandel-omp-rec.c*

```
void mandel_tiled (int M[ROWS][COLS], ...)
{
    ...

    if (equal && M[y][x]==maxiter) {
        #pragma omp task
        if (output2histogram)
        {
            #pragma omp atomic
            histogram[maxiter-1]+=(TILE*TILE);
        }
            ...

            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 firstprivate(y,x)
          for (int py=y; py<y+NRows; py++)
            for (int px=x; px<x+NCols; px++) {
                ...

                if (output2histogram)
                {
                  #pragma omp atomic
                  histogram[M[py][px]-1]++;
                }
                    ...

                    if (setup_return == EXIT_SUCCESS)
                    {
                        #pragma omp critical
                        XSetForeground (display, gc, color);
                        XDrawPoint (display, win, gc, px, py);
                    }

                ...
            }
        }
    }


}


int main(int argc, char *argv[])
{
    ...
    #pragma omp parallel
    #pragma omp single
    mandel_tiled ((int (*)[width]) Hmatrix, ...);
    ...
}
```

Execution and Check the Correctness

We observe that applying the respective changes to the Leaf Recursive version, both *mandel_image.png* and *mandel_histogram.out* are identical to the ones in the sequential recursive code provided.

```
par1304@boada-6:~/lab4$ diff mandel_image.jpg mandel_image_rec_leaf.jpg
par1304@boada-6:~/lab4$ diff mandel_histogram.out mandel_histogram_rec_leaf.out
par1304@boada-6:~/lab4$
```

Figure 12: Comparison of histograms

2.2 Recursive Tree task decomposition

Comparing the code with the previous version, we have removed the tasks at the base cases. Now, a task is created for every *mandel_tiled* recursive call, exploiting parallelism as a Tree-recursive version.

```
1
2       // same code here ...
3
4       if (NRows > TILE)
5       {
6           # pragma omp task
7           mandel_tiled_rec(M, NRows/2, ...);
8           # pragma omp task
9           mandel_tiled_rec(M, NRows/2,  ...);
10          # pragma omp task
11          mandel_tiled_rec(M, NRows/2, ...);
12          # pragma omp task
13          mandel_tiled_rec(M, NRows/2, ...);
14      }
15      else
16      {
17          # pragma omp task
18          mandel_tiled_rec(M, NRows, ...);
19          # pragma omp task
20          mandel_tiled_rec(M, NRows, ...);
21      }
```

Execution and Check the Correctness

Like we have done with the iterative versions, let's compare the image and histogram, to make sure the new implementation is correct.

```
par1302@boada-6:~/lab4$ diff outForCompareOrigSeqRec/mandel_ outForCompareTree/mandel_image.jpg
mandel_histogram.out  mandel_image.jpg
par1302@boada-6:~/lab4$ diff outForCompareOrigSeqRec/mandel_histogram.out outForCompareTree/mandel_histogram.out
par1302@boada-6:~/lab4$
```

After the comparison, there are no differences, so the implementation is correct. Now we can proceed looking at the results.

2.3 Recursive Performance Analysis

2.3.1 Modelfactors analysis

Version 1: Leaf Implementation

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	2.01	1.50	1.54	1.61	1.63	1.66	1.66	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	1.21	1.21
Efficiency	1.00	0.67	0.33	0.21	0.15	0.12	0.10	0.09	0.08	0.07	0.06

Table 1: Analysis done on Sat Apr 27 10:57:45 AM CEST 2024, par1302

Overview of the Efficiency metrics in parallel fraction, ϕ=99.98%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	99.79%	66.83%	32.50%	20.81%	15.35%	12.04%	10.05%	8.62%	7.54%	6.70%	6.03%
Parallelization strategy efficiency	99.79%	67.17%	33.56%	22.47%	16.82%	13.52%	11.28%	9.68%	8.49%	7.57%	6.86%
Load balancing	100.00%	68.09%	34.07%	22.79%	17.05%	13.68%	11.43%	9.80%	8.60%	7.67%	6.94%
In execution efficiency	99.79%	98.66%	98.50%	98.63%	98.64%	98.77%	98.72%	98.82%	98.79%	98.73%	98.83%
Scalability for computation tasks	100.00%	99.49%	96.86%	92.59%	91.22%	89.09%	89.07%	88.99%	88.71%	88.48%	87.87%
IPC scalability	100.00%	99.30%	98.69%	98.59%	98.50%	98.32%	98.45%	98.45%	98.43%	98.43%	98.38%
Instruction scalability	100.00%	100.14%	100.25%	100.24%	100.25%	100.23%	100.23%	100.22%	100.23%	100.23%	100.22%
Frequency scalability	100.00%	100.05%	97.90%	93.69%	92.37%	90.40%	90.26%	90.18%	89.91%	89.69%	89.11%

Table 2: Analysis done on Sat Apr 27 10:57:45 AM CEST 2024, par1302

Statistics about explicit tasks in parallel fraction											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	15244.0	15244.0	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	0.7	0.93	0.94	0.86	0.86	0.82	0.93	0.93	0.9	0.91
LB (time executing explicit tasks)	1.0	0.5	0.96	0.81	0.83	0.67	0.76	0.76	0.73	0.66	0.67
Time per explicit task (average us)	34.91	35.33	36.03	37.97	38.18	39.2	39.21	39.14	39.18	39.22	39.2
Overhead per explicit task (synch %)	0.0	179.07	741.55	1285.59	1861.77	2403.46	2954.62	3515.18	4068.12	4619.18	5175.4
Overhead per explicit task (sched %)	0.79	3.66	4.15	3.66	3.69	3.24	3.38	3.07	3.14	3.24	2.93
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Sat Apr 27 10:57:45 AM CEST 2024, par1302

Version 2: Tree Implementation

Overview of whole program execution metrics											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	1.99	1.01	0.51	0.38	0.29	0.25	0.22	0.19	0.18	0.16	0.15
Speedup	1.00	1.98	3.90	5.25	6.89	8.09	9.24	10.37	11.33	12.38	12.87
Efficiency	1.00	0.99	0.98	0.87	0.86	0.81	0.77	0.74	0.71	0.69	0.64

Table 1: Analysis done on Fri May 3 08:53:03 AM CEST 2024, par1302

Overview of the Efficiency metrics in parallel fraction, ϕ=99.98%											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	99.73%	98.75%	97.36%	87.25%	85.98%	80.75%	76.84%	74.00%	70.71%	68.70%	64.29%
Parallelization strategy efficiency	99.73%	98.72%	98.22%	93.68%	92.37%	89.81%	85.73%	82.74%	79.38%	77.50%	72.71%
Load balancing	100.00%	99.82%	98.85%	95.74%	94.54%	90.72%	88.17%	84.75%	80.68%	80.09%	74.81%
In execution efficiency	99.73%	98.90%	99.36%	97.85%	97.70%	98.99%	97.24%	97.63%	98.38%	96.76%	97.19%
Scalability for computation tasks	100.00%	100.04%	99.12%	93.14%	93.08%	89.91%	89.63%	89.44%	89.09%	88.65%	88.42%
IPC scalability	100.00%	99.81%	99.74%	99.19%	99.25%	99.13%	99.00%	99.06%	98.92%	98.93%	98.88%
Instruction scalability	100.00%	100.36%	100.37%	100.36%	100.36%	100.36%	100.36%	100.36%	100.36%	100.36%	100.36%
Frequency scalability	100.00%	99.87%	99.01%	93.57%	93.44%	90.37%	90.20%	89.96%	89.73%	89.29%	89.11%

Table 2: Analysis done on Fri May 3 08:53:03 AM CEST 2024, par1302

Statistics about explicit tasks in parallel fraction											
Number of processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	20324.0	20324.0	20324.0	20324.0	20324.0	20324.0	20324.0	20324.0	20324.0	20324.0	20324.0
LB (number of explicit tasks executed)	1.0	0.77	0.57	0.47	0.5	0.39	0.57	0.44	0.61	0.43	0.59
LB (time executing explicit tasks)	1.0	1.0	0.99	0.97	0.96	0.93	0.92	0.9	0.87	0.87	0.84
Time per explicit task (average us)	97.34	97.95	98.9	106.65	106.81	111.56	113.6	115.44	117.82	118.76	122.35
Overhead per explicit task (synch %)	0.0	0.94	1.29	4.69	5.93	7.8	10.93	12.88	15.5	16.9	21.06
Overhead per explicit task (sched %)	0.27	0.32	0.39	1.73	1.91	2.91	4.48	6.05	7.76	8.51	11.11
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Fri May 3 08:53:03 AM CEST 2024, par1302

Comparison between Versions

Let also v1 be the first recursive implemented version (Leaf strategy) and v2 (Tree-recursive strategy) the other implemented version.

Comparing the leaf-recursive with the tree-recursive implementations, the second one it is faster regarding execution time, as the number of processor increase. The main difference is in the speedup of the first implementation with 20 threads reaches 1.21, and for the tree code has a value of 11.33, so that the second version is much better than the first one.

The Efficiency line of the table of v1 clearly indicates that it isn't an improvement if you add more processors. But in the tree strategy this efficiency is somehow maintained to 0.64 with 20 processors.

Regarding the load balancing main issue we find of the parallel fraction of the v1 we can affirm that it's incredibly not well balanced. Starting at 1.0, decreases in an insignificant 7%. On the other hand, the v2 is acceptable with the Load balancing (LB), it starts to be less balanced very slowly as processors increase, but overall the v2 is way better.

Taking into account the synchronisation overhead that tells us the wasted time that tasks wait between each other, we observe that in v1 the number is incredibly high as processors increase. Instead, for the tree-recursive implementation, there is much less percentage of synchronisation overhead.

Another aspect to comment, is that in the Tree strategy, it creates a lot of tasks, and we could do a better implementation controling the depth the tasks stop being created. So as we learned in theory we think is a great idea to add a CUTOFF parameter, and also in_final to control task creation. And causing less overhead cased by task creation.

Seeing all this results we can affirm that the leaf strategy has a poor performance and should be improved or not be used. If someone wants to improve v1, a good idea could be taken into consideration the load balancing issue, which we saw it's one of the main problems.

2.3.2 Paraver analysis

Version 1: Recursive Leaf Implementation

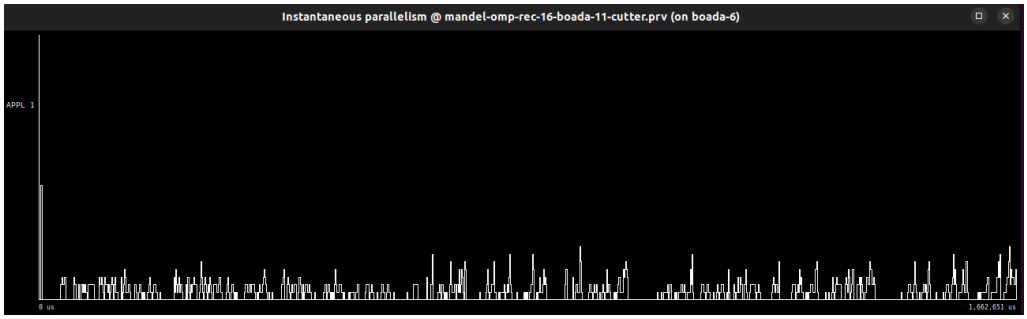


Figure 13: Useful/Instantaneous parallelism

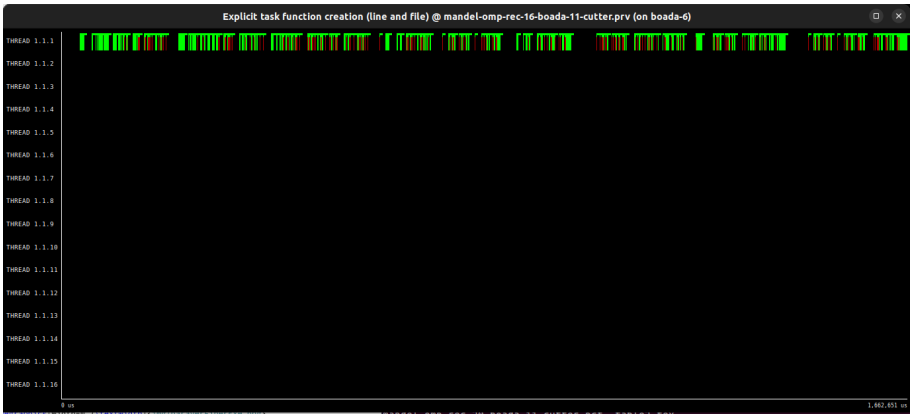


Figure 14: OpenMP tasking/Explicit tasks function creation

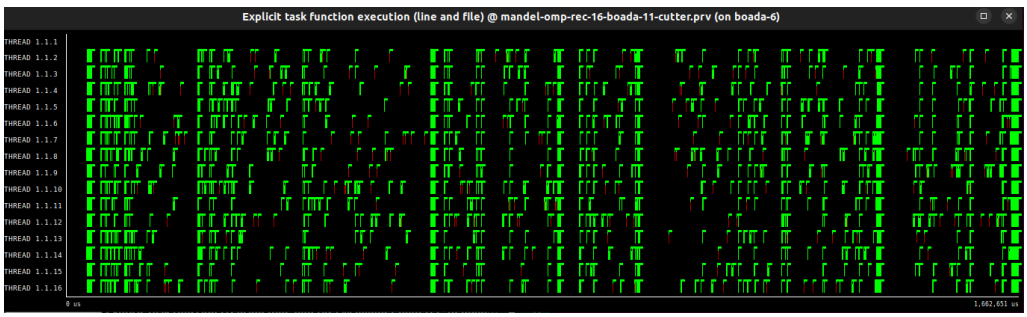


Figure 15: OpenMP tasking/Explicit tasks execution

Version 2: Recursive Tree Implementation

Figure 16: Useful/Instantaneous parallelism

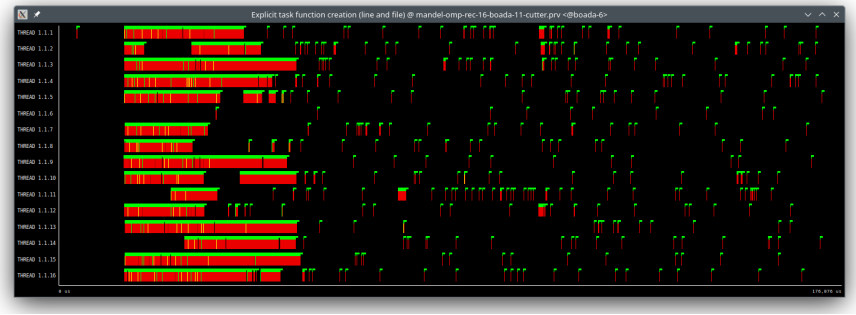


Figure 17: OpenMP tasking/Explicit tasks function creation

Figure 18: OpenMP tasking/Explicit tasks execution

Comparison between Versions

Referring to the leaf implementation, in the timeline of the Figure 13 about the instantaneous parallelism, the pattern tends to a very small number. It is advising us that this version is not exploiting as much parallelism as it could explode.

In the other instantaneous parallelism of v2 it is very constant, only being irregular in the beginning. But showing that at the end, this is a better implementation and exploits more the parallelism.

The leaf strategy creates it's tasks in the base case iteratively, this is what we learned in the theory classes, and it is present in the figure 14, when the explicit tasks are created, they are all done by the same thread and sequential. Over time, they are created one by one. In the other case we see that at the v2, there will be more tasks created, as it is shown at the figure 17, and more distributed over time and with different threads. The number of creations is higher in the beginning and then it decreases.

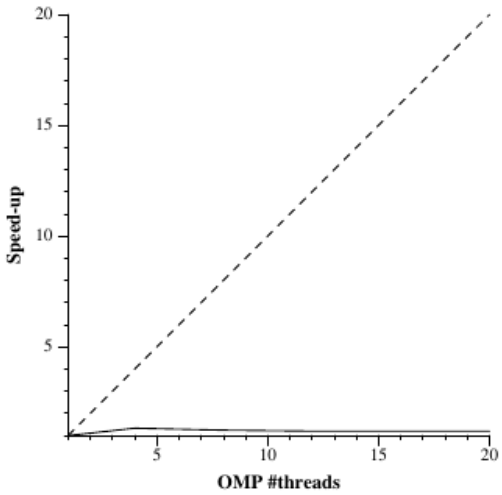
In Figure 15, we see the timeline of the execution for each explicit task. We observe that as we have a lot of tasks due to the leaf-recursive version, we can't differ between each other in terms of length.

In the tree version (figure 18) First of all there is a lot of computation on each task. It is due to the fact that it calculates all the perimeter, and the algorithm works better in this version, rather than using the Leaf. It seems that is embarrassingly parallel, and it's executing in all the tasks that can, exploiting parallelism, and the tasks take longer as we say in modelfactors.

Finally, for the explicit tasks execution of the v2, they are more equally distributed. As there is more grain, which is associated with less overhead and they are executed at all the threads. Resulting in a better implementation, even though it has much more tasks than v1. It's directly related that in all the recursive calls it's creating tasks, not in the base case at the end.

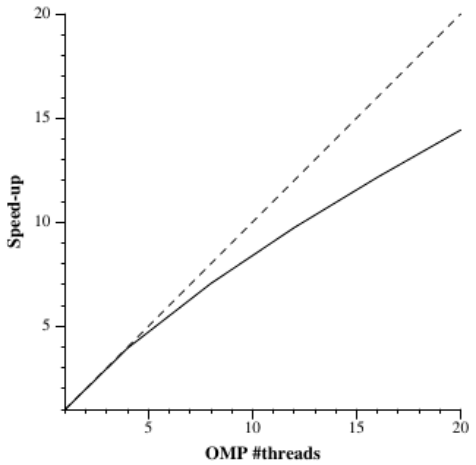
2.3.3 Strong Scalability

Version 1: Parallel Implementation



par1304
Speed-up wrt sequential time (mandel funtion only)
Generated by par1304 on Sat Apr 27 12:14:04 PM CEST 2024

Version 2: Finer Grain Implementation



par1302
Speed-up wrt sequential time (mandel funtion only)
Generated by par1302 on Fri May 3 08:35:32 AM CEST 2024

Comparison between Versions

In this part, we will discuss the strong scalability. We can observe in a graphical way the behaviour of the speedup mentioned when comparing the modelfactor tables, depending on the number of threads.

The most important behaviour of the programs, would be a constant lineal growth, like the dashed line in the diagonal. This would indicate that it escalates perfectly with more threads, making worth to run the code with more of them (in the case we don't increase the size of the problem).

Having said that, the pattern that follows the v1, it's caused, by a worst performing implementation we and seen in previous sections, and the parallel efficiency is not maintained, causing it to not continue improving the speedup as the threads increase, causing this weird shape. As we have said, this is produced by the sequential execution of tasks at leafs.

For the second version, as seen in the modelfactors, the parallel efficiency was nearly constant and now, increasing the number of threads, the speed-up is nearly the optimal. This is the desirable behaviour we want when we try to exploit parallelism, and we personally think that is a good implementation. The line of the plot is very similar to the finer grain implementation, and in the beggining it's close to the diagonal.

If we had to choose one implementation, this one is a good idea, and also can be improved with the CUTOFF method.

Overall, the curve plotted at the v1 is incredibly different than in the v2. To conclude, the v2 compared with v1 is much better, and it can also scale better.

3 Elapsed execution times for each of the versions

	Number of threads					
Version	**1**	**4**	**8**	**12**	**16**	**20**
Iterative: Tile (provided, but should be completed)	3.03	0.89	0.67	0.58	0.57	0.57
Iterative: Finer grain	3.03	0.78	0.42	0.30	0.23	0.20
Recursive: Leaf (provided, but should be completed)	2.01	1.54	1.63	1.66	1.66	1.66
Recursive: Tree	1.99	0.51	0.29	0.22	0.18	0.15

Table 1: Summary of the elapsed execution times for each of the versions, obtained from the output files after the execution of submit-strong-omp.sh script

This values are obtained in the modelfactors, and as we discussed the Tree strategy, is the best strategy amongs the other, and effectively calculates the Mandelbrot Set.

Deliberable:
The zip of the deliberable in Atenea contains the the 4 codes, we modified. This are the names:

1. Iterative code exploiting Parallelism: *mandel-omp-iter-v1.c*
2. Iterative code with Finer Grain: *mandel-omp-iter-finer.c*
3. Recursive Leaf parallel code: *mandel-omp-rec-leaf.c*
4. Recursive Tree parallel code: *mandel-omp-rec-tree.c*