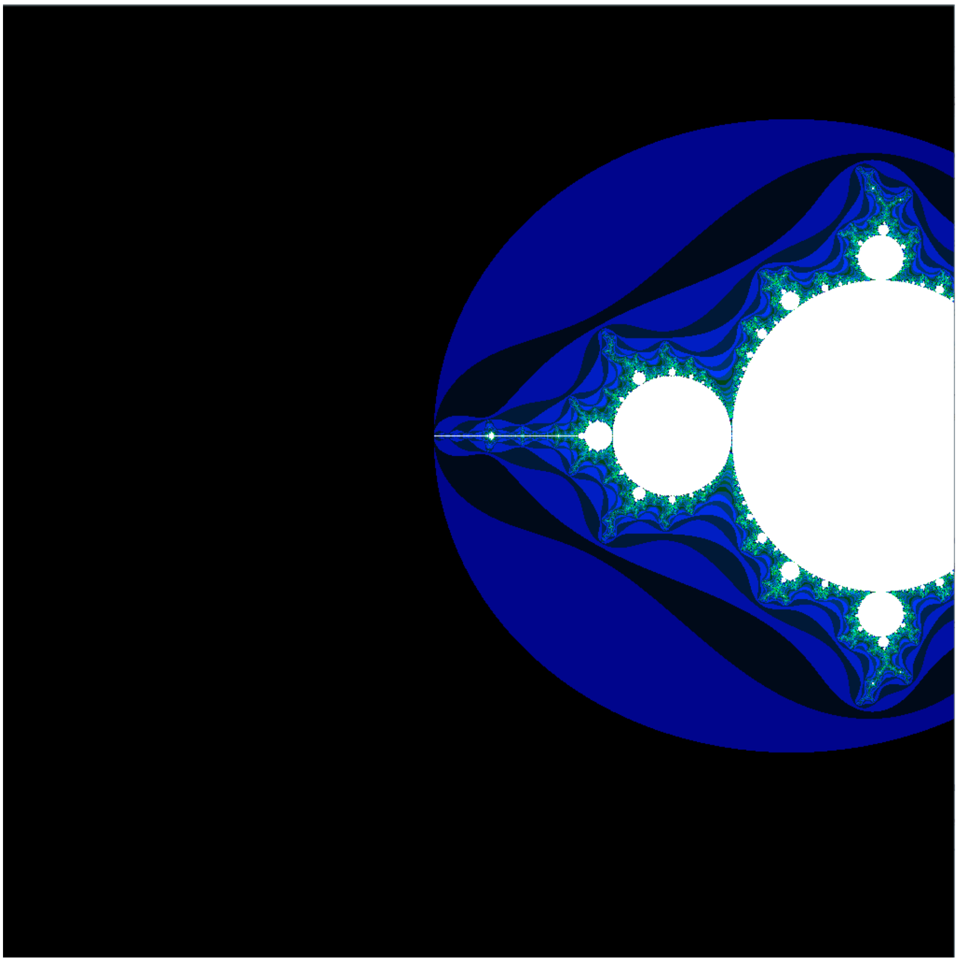
# PAR Laboratory Assignment
# Lab 4: Parallel Task Decomposition Implementation and Analysis



UNIVERSITAT POLITÈCNICA DE CATALUNYA
FACULTAD D'INFORMÀTICA DE BARCELONA

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

# Index

# Introduction

In this laboratory assignment, we aim to evaluate the strong scalability of different parallelisation strategies for computing the Mandelbrot set, as it presents an excellent case study for analysing parallel processing techniques due to its intricate fractal patterns generated through iterative computations in the complex plane.

The report focuses on comparing several strategies, both iterative and recursive, to understand their impact on performance and scalability. On the iterative side, we employ the tile and finer grain strategies, each offering distinct approaches to task decomposition, from coarser workloads to more granular subdivisions. For recursive methods, the leaf and tree strategies are analysed, which explore varying depths and hierarchies of task creation. These strategies provide insights into the trade-offs between load balancing, synchronisation overhead, and task management efficiency.

Our primary objective is to examine how these strategies scale with increasing thread counts while keeping the problem size constant. The analysis aims to determine which strategy—iterative or recursive—provides the best performance for this problem and why.

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

# Iterative task decomposition

In this section we will analyse different iterative task decompositions.

## Tile

### Code

For the tile version we were given a sequential approach, in order to parallelise it we had to create the necessary tasks and delete the existing variables' dependencies as we studied in the previous laboratory.

We first added the `#pragma omp task firstprivate(x, y)` as well as the `#pragma omp parallel` and `#pragma omp single` directives in order to create the necessary tasks for the tile version. Then, as we previously studied that the dependencies were caused by both variables `histogram` and `&X11_COLOR_FAKE`, in order to fix this we added a `#pragma omp atomic` and `#pragma omp critical` directives.

```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) {
        // Set all matrix positions with the same value
        #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) {
              long color = (long)((M[y][x] - 1) * scale_color) + min_color;
              if (output2histogram) {
                  #pragma omp atomic
                  histogram[M[y][x] - 1]+=(TILE*TILE);
              }
              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
              // Compute
              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);
                        }
                    }
                  }
                }
        }
      }
}
```

main():

```
#pragma omp parallel
#pragma omp single
mandel_tiled((int (*)[width])Hmatrix, real_min, imag_min, real_max, imag_max, scale_real, scale_imag, maxiter);
```

## Modelfactor Analysis

These are the results from running the modelfactor analysis:

In this first table we can notice that when increasing the number of processors, the elapsed time is decreased until reaching its peak at 8 processors, where it does not matter adding more processors as the elapsed time will remain quite the same. We can notice the same behaviour in the SpeedUp row. But efficiency drops as we keep increasing the number of processors.

| Overview of whole program execution metrics | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Elapsed time (sec) | 3.09 | 1.74 | 0.92 | 0.75 | 0.70 | 0.72 | 0.71 | 0.71 | 0.71 |
| Speedup | 1.00 | 1.78 | 3.37 | 4.13 | 4.41 | 4.29 | 4.35 | 4.35 | 4.34 |
| Efficiency | 1.00 | 0.89 | 0.84 | 0.69 | 0.55 | 0.43 | 0.36 | 0.31 | 0.27 |

Table 1: Analysis done on Thu Nov 21 12:15:25 PM CET 2024, par1307

In this second table we can notice that as analysed previously, the efficiency drops until 27.12% when having 16 processors. Also, the load balancing keeps decreasing due to having coarse grained tasks.

| Overview of the Efficiency metrics in parallel fraction, $\phi$=99.99% | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Global efficiency | 100.00% | 88.82% | 84.18% | 68.81% | 55.16% | 42.88% | 36.28% | 31.10% | 27.12% |
| Parallelization strategy efficiency | 100.00% | 89.00% | 84.80% | 73.86% | 59.41% | 47.75% | 40.49% | 34.76% | 30.64% |
| Load balancing | 100.00% | 89.03% | 84.85% | 73.95% | 59.49% | 47.82% | 40.57% | 34.83% | 30.71% |
| In execution efficiency | 100.00% | 99.96% | 99.94% | 99.88% | 99.86% | 99.85% | 99.80% | 99.80% | 99.79% |
| Scalability for computation tasks | 100.00% | 99.80% | 99.27% | 93.16% | 92.84% | 89.81% | 89.61% | 89.45% | 88.49% |
| IPC scalability | 100.00% | 99.98% | 99.93% | 99.94% | 99.93% | 99.93% | 99.91% | 99.93% | 99.93% |
| Instruction scalability | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| Frequency scalability | 100.00% | 99.82% | 99.34% | 93.21% | 92.90% | 89.87% | 89.69% | 89.51% | 88.55% |

Table 2: Analysis done on Thu Nov 21 12:15:25 PM CET 2024, par1307

In the third table we see that the time per explicit task is high as expected. We can also notice that the synchronization overhead increases up to 227.11% which is not good.

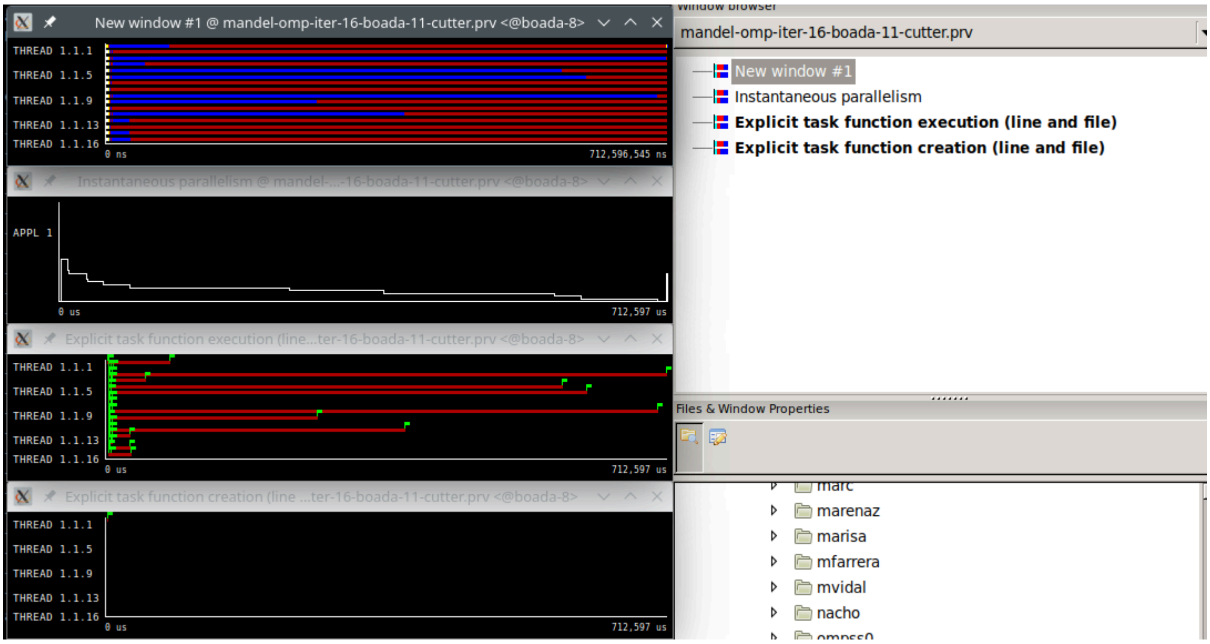
| Statistics about explicit tasks in parallel fraction | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Number of explicit tasks executed (total) | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 |
| LB (number of explicit tasks executed) | 1.0 | 0.94 | 0.64 | 0.43 | 0.47 | 0.38 | 0.31 | 0.27 | 0.24 |
| LB (time executing explicit tasks) | 1.0 | 0.89 | 0.85 | 0.74 | 0.59 | 0.48 | 0.41 | 0.35 | 0.31 |
| Time per explicit task (average us) | 48281.47 | 48368.7 | 48620.97 | 51792.18 | 51937.49 | 53632.85 | 53720.96 | 53739.44 | 54209.95 |
| Overhead per explicit task (synch %) | 0.0 | 12.33 | 17.87 | 35.28 | 68.18 | 109.4 | 146.93 | 187.93 | 227.11 |
| Overhead per explicit task (sched %) | 0.0 | 0.01 | 0.01 | 0.0 | 0.0 | 0.0 | 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 Thu Nov 21 12:15:25 PM CET 2024, par1307

## Paraver Analysis

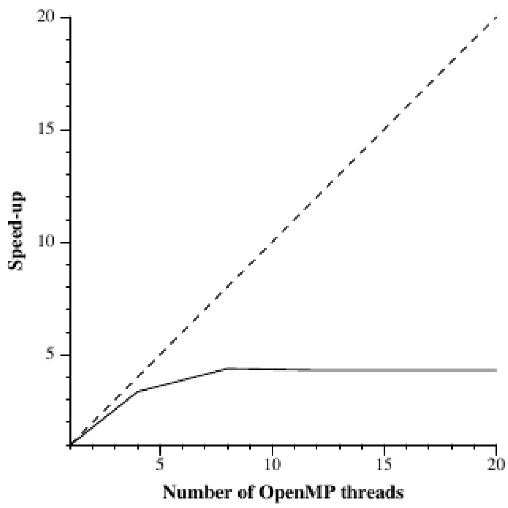
In the following picture we can see different paraver windows: execution trace, instantaneous parallelism and the explicit task function execution and creation.

Analysing the different images we can notice that, as mentioned before, parallelism is exploited at the beginning, but there is a point, when the processors finish their tasks the parallelism drops instantaneously.




## Strong Scalability

This graphics shows the speed-up against the number of OpenMP threads, we notice that when using 1 to 4 threads it is close to the ideal speed-up, but as we keep increasing the threads, it starts getting constant, tending to a speed-up of 5.




par1307
Speed-up wrt sequential time (mandel funtion only)
Generated by par1307 on Thu Nov 21 12:31:31 PM CET 2024

| | **Number of threads** | | | | | |
|---|---|---|---|---|---|---|
| **Iterative: Tile** | **1** | **4** | **8** | **12** | **16** | **20** |
| **Elapsed time (ns)** | 3.089155 | 0.910804 | 0.698836 | 0.709182 | 0.709561 | 0.709295 |

*Table 4: Elapsed time in (ns) when computing the iterative tile version.*

# Finer

## Code

For the finer grain implementation we added some modifications to the tile approach based on the analysis done in the previous laboratory assignment:

We divided the `equal` variable in `equal1` and `equal2`, then in order to create the tasks, we added a `#pragma omp task shared(equal1)` and `#pragma omp task shared(equal2)` directives for each loop (vertical and horizontal borders) together with a `#pragma omp taskwait` to avoid data races. Finally, we will find some `#pragma omp task firstprivate(...)` directives which will create the finer grain tasks, one for each computation loop.

```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 = 0, equal2 = 0;

    for (int y = 0; y < ROWS; y += TILE)
      for (int x = 0; x < COLS; x += TILE) {
            equal1 = 1;
            equal2 = 1;
            #pragma omp task firstprivate (x,y)
                  {

            #pragma omp task shared(equal1)
            // Set all matrix positions with the same value
                // 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);
                    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[M[y][x] - 1]+=(TILE*TILE);
                  }

                  long color = (long)((M[y][x] - 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 {
                  // Compute
                  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

These are the results from running the modelfactor analysis:

In this first table we can notice that with the finer grain strategy, the elapsed time keeps decreasing when increasing the number of processors. Same behaviour can be seen in the speedup row.

| Overview of whole program execution metrics | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Elapsed time (sec) | 3.11 | 1.57 | 0.81 | 0.57 | 0.43 | 0.36 | 0.31 | 0.27 | 0.24 |
| Speedup | 1.00 | 1.98 | 3.84 | 5.48 | 7.16 | 8.56 | 10.15 | 11.67 | 13.04 |
| Efficiency | 1.00 | 0.99 | 0.96 | 0.91 | 0.90 | 0.86 | 0.85 | 0.83 | 0.82 |

Table 1: Analysis done on Thu Nov 21 01:44:46 PM CET 2024, par1307

In the second table we see that the efficiency keeps quite constant which means that it has a great scalability. In this case, the load balancing drops a bit when increasing the number of processors but it stills keep above 95%.

| Overview of the Efficiency metrics in parallel fraction, $\phi$=99.99% | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Global efficiency | 99.92% | 98.85% | 96.05% | 91.36% | 89.53% | 85.66% | 84.62% | 83.42% | 81.57% |
| Parallelization strategy efficiency | 99.92% | 99.06% | 98.22% | 97.64% | 96.18% | 95.16% | 94.31% | 92.96% | 91.35% |
| Load balancing | 100.00% | 99.91% | 99.62% | 99.41% | 99.07% | 98.35% | 97.99% | 96.25% | 95.65% |
| In execution efficiency | 99.92% | 99.15% | 98.59% | 98.22% | 97.08% | 96.76% | 96.24% | 96.59% | 95.50% |
| Scalability for computation tasks | 100.00% | 99.79% | 97.79% | 93.56% | 93.08% | 90.01% | 89.73% | 89.73% | 89.30% |
| IPC scalability | 100.00% | 99.85% | 99.87% | 99.91% | 99.80% | 99.83% | 99.83% | 99.80% | 99.81% |
| Instruction scalability | 100.00% | 100.10% | 100.10% | 100.10% | 100.10% | 100.10% | 100.09% | 100.10% | 100.09% |
| Frequency scalability | 100.00% | 99.84% | 97.83% | 93.56% | 93.18% | 90.07% | 89.80% | 89.83% | 89.38% |

Table 2: Analysis done on Thu Nov 21 01:44:46 PM CET 2024, par1307

In the third table we see that the time per explicit task increases a bit together with the processors and the synchronization overhead just increases up to 4.84%.

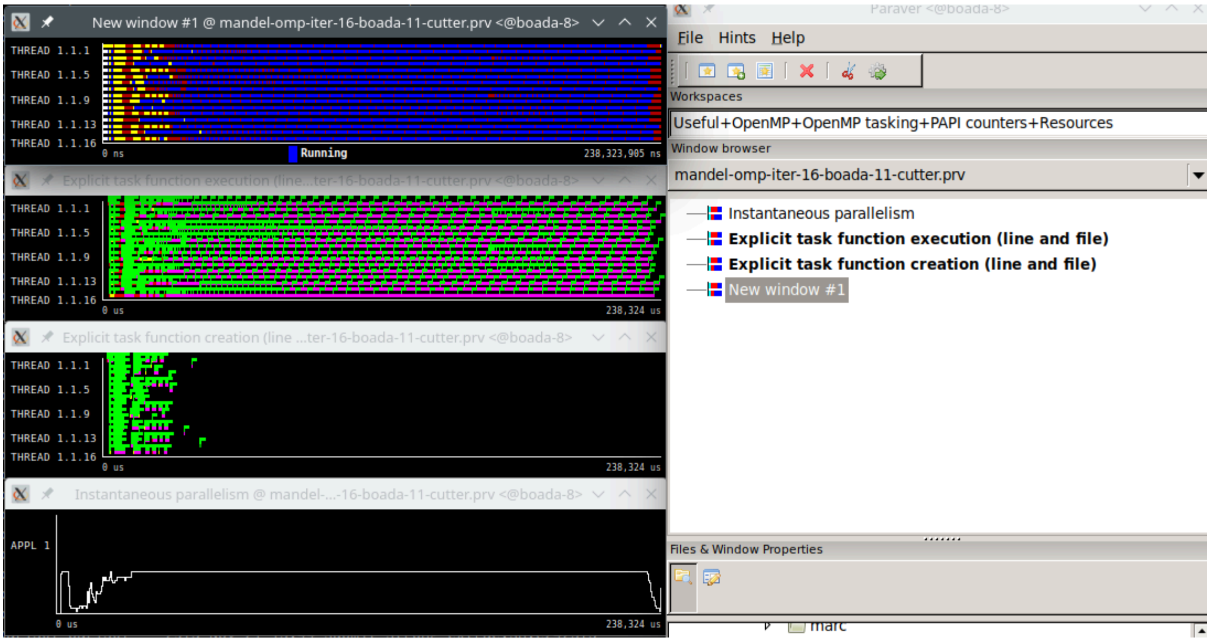
| Statistics about explicit tasks in parallel fraction | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Number of explicit tasks executed (total) | 8448.0 | 8448.0 | 8448.0 | 8448.0 | 8448.0 | 8448.0 | 8448.0 | 8448.0 | 8448.0 |
| LB (number of explicit tasks executed) | 1.0 | 0.95 | 0.64 | 0.43 | 0.45 | 0.37 | 0.31 | 0.34 | 0.37 |
| LB (time executing explicit tasks) | 1.0 | 1.0 | 1.0 | 1.0 | 0.99 | 0.99 | 0.99 | 0.98 | 0.97 |
| Time per explicit task (average us) | 367.01 | 369.52 | 378.16 | 396.37 | 400.24 | 414.76 | 417.52 | 419.97 | 425.12 |
| Overhead per explicit task (synch %) | 0.0 | 0.59 | 1.09 | 1.29 | 2.17 | 2.84 | 3.31 | 4.01 | 4.84 |
| Overhead per explicit task (sched %) | 0.08 | 0.34 | 0.65 | 0.98 | 1.51 | 1.8 | 2.18 | 2.89 | 3.64 |
| Number of taskwait/taskgroup (total) | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 | 64.0 |

Table 3: Analysis done on Thu Nov 21 01:44:46 PM CET 2024, par1307

## Paraver Analysis

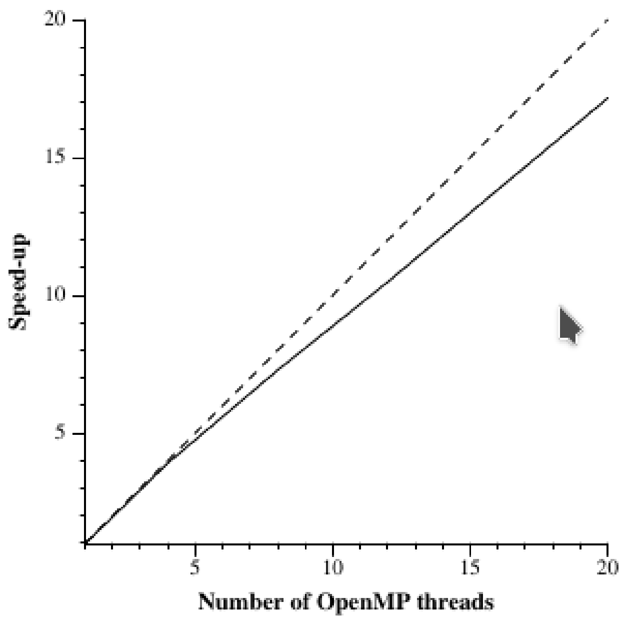
In the following picture we can see different paraver windows: execution trace, instantaneous parallelism and the explicit task function execution and creation.

Before we saw that parallelism was only exploited at the beginning, but now we see that parallelism is all over the execution.




## Strong Scalability

This graphics shows the speed-up against the number of OpenMP threads, we notice that this strategy's behaviour is almost the ideal, meaning that it indeed has a strong scalability.




par1307
Speed-up wrt sequential time (mandel funtion only)
Generated by par1307 on Thu Nov 21 01:34:40 PM CET 2024

| | **Number of threads** | | | | | |
|---|---|---|---|---|---|---|
| **Iterative: Finer Grain** | **1** | **4** | **8** | **12** | **16** | **20** |
| **Elapsed time (ns)** | 3.094647 | 0.780898 | 0.418766 | 0.291014 | 0.220544 | 0.177836 |

*Table 4: Elapsed time in (ns) when computing the iterative finer version.*

# Recursive task decomposition

## Leaf

### Code

```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 non_first_call= (NRows!=ROWS) | (NCols!=COLS);
        int equal;
        int x, y;

        equal = non_first_call;
        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; py < y + NRows; 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]);
        }

        //check if we can avoid computation of tile
        if (equal) {
          // Set all matrix positions with the same value
#pragma omp task
                {
                long color = (long)((M[y][x] - 1) * scale_color) + min_color;
                if (output2histogram) {
                  #pragma omp atomic
                  histogram[M[y][x] - 1] += (NRows * NCols);
                }
                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) {
            // Compute
#pragma omp task firstprivate(y,x)
                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);
            }

          }
        }
}
```

We added the leaf calls when reaching the last recursivity level (base case).

### Modelfactor Analysis

| Overview of whole program execution metrics | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Elapsed time (sec) | 1.62 | 1.24 | 1.32 | 1.33 | 1.37 | 1.37 | 1.37 | 1.37 | 1.37 |
| Speedup | 1.00 | 1.30 | 1.22 | 1.22 | 1.18 | 1.18 | 1.18 | 1.18 | 1.18 |
| Efficiency | 1.00 | 0.65 | 0.31 | 0.20 | 0.15 | 0.12 | 0.10 | 0.08 | 0.07 |

Table 1: Analysis done on Thu Nov 28 12:42:51 PM CET 2024, par1307

| Overview of the Efficiency metrics in parallel fraction, $\phi$=99.98% | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Global efficiency | 99.94% | 64.97% | 30.56% | 20.31% | 14.73% | 11.76% | 9.82% | 8.40% | 7.35% |
| Parallelization strategy efficiency | 99.94% | 65.20% | 32.39% | 21.79% | 16.27% | 13.10% | 10.93% | 9.38% | 8.25% |
| Load balancing | 100.00% | 65.43% | 32.51% | 21.87% | 16.34% | 13.15% | 10.97% | 9.42% | 8.28% |
| In execution efficiency | 99.94% | 99.66% | 99.63% | 99.63% | 99.61% | 99.59% | 99.60% | 99.60% | 99.61% |
| Scalability for computation tasks | 100.00% | 99.64% | 94.34% | 93.21% | 90.51% | 89.81% | 89.79% | 89.53% | 89.20% |
| IPC scalability | 100.00% | 99.66% | 99.61% | 99.53% | 99.53% | 99.46% | 99.55% | 99.52% | 99.50% |
| Instruction scalability | 100.00% | 100.06% | 100.07% | 100.06% | 100.06% | 100.06% | 100.06% | 100.06% | 100.06% |
| Frequency scalability | 100.00% | 99.91% | 94.65% | 93.59% | 90.88% | 90.24% | 90.15% | 89.91% | 89.59% |

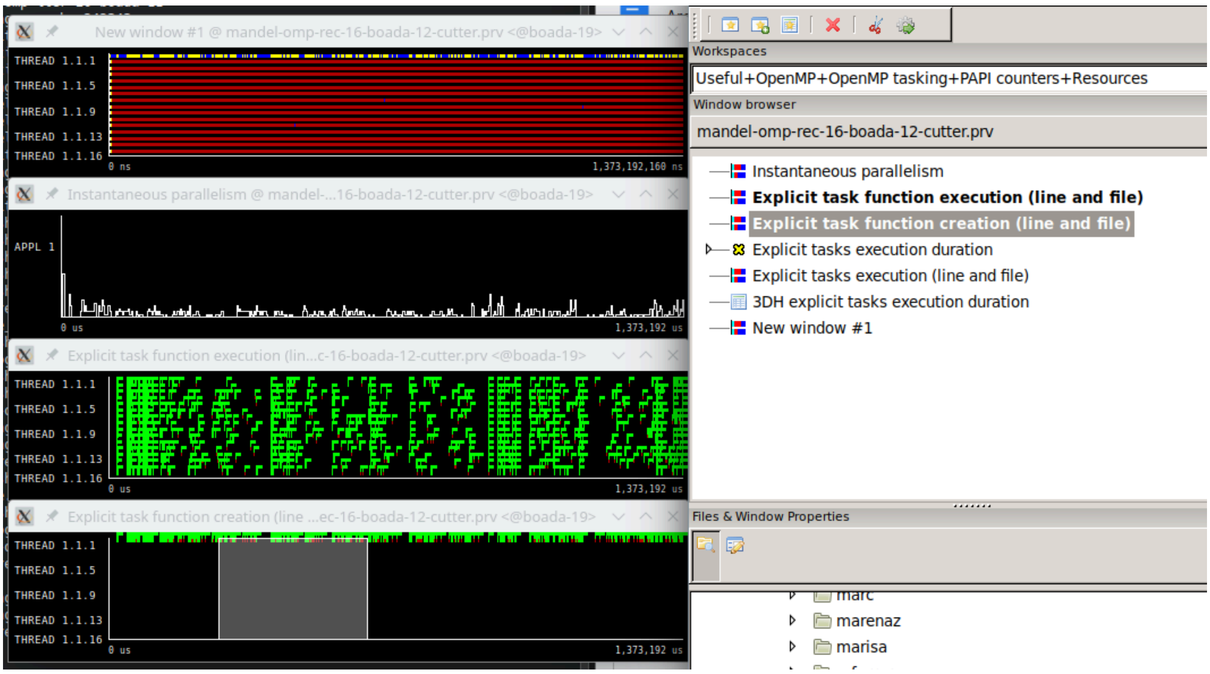
Table 2: Analysis done on Thu Nov 28 12:42:51 PM CET 2024, par1307

| 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) | 3013.0 | 3013.0 | 3013.0 | 3013.0 | 3013.0 | 3013.0 | 3013.0 | 3013.0 | 3013.0 |
| LB (number of explicit tasks executed) | 1.0 | 0.56 | 0.65 | 0.89 | 0.8 | 0.87 | 0.78 | 0.79 | 0.81 |
| LB (time executing explicit tasks) | 1.0 | 0.5 | 0.67 | 0.9 | 0.81 | 0.7 | 0.64 | 0.6 | 0.76 |
| Time per explicit task (average us) | 125.93 | 127.17 | 130.71 | 135.99 | 137.67 | 140.69 | 140.55 | 140.43 | 140.56 |
| Overhead per explicit task (synch %) | 0.0 | 224.8 | 905.69 | 1516.49 | 2210.39 | 2810.39 | 3456.56 | 4113.86 | 4752.27 |
| Overhead per explicit task (sched %) | 0.23 | 0.82 | 1.05 | 1.02 | 0.97 | 0.94 | 0.98 | 0.95 | 0.91 |
| 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 Thu Nov 28 12:42:51 PM CET 2024, par1307

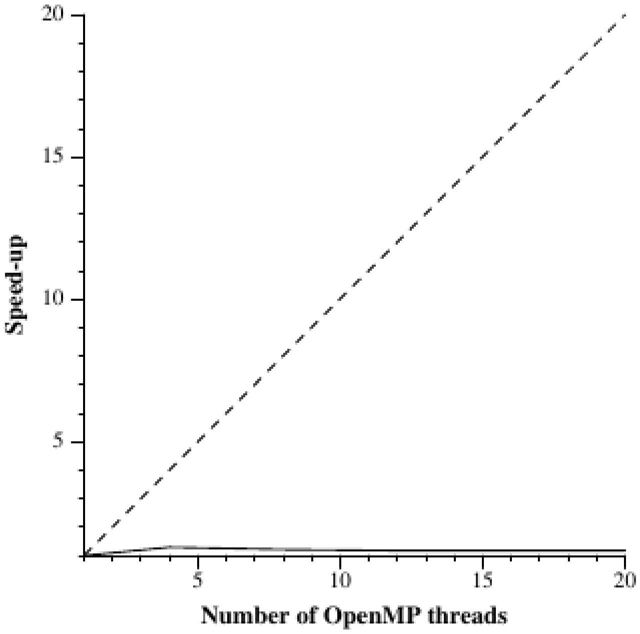
Thanks to the Modelfactor tables, we can observe a point (8 processors) where the speedup reaches its peak. Moreover, efficiency drops sharply beyond 4 processors, which could be attributed to poor load balancing management. This is confirmed in the third table, where we can see there is indeed significant load imbalance.

## Paraver Analysis




In this execution of wxparaver over this particular parallel implementation, we can see all the tasks created at leaf level. The explicit task execution graph shows a huge Load Unbalance, as there are a lot of spaces where threads are stopped waiting for other threads to finish. This is not efficient.

## Strong Scalability




par1307
Speed-up wrt sequential time (mandel funtion only)
Generated by par1307 on Thu Nov 28 12:49:07 PM CET 2024

1 1.611466
4 1.242143
8 1.323873
12 1.368214
16 1.369712
20 1.370270

Using the strong scalability analysis, however, we can observe severe efficiency problems, which might be caused by a poor parallelisation strategy, big Load Unbalance or Memory Contention problems.

# Tree

## Code

```c
    } 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);
      }

    }
  }
}
```

We added the recursive call tasks, which create the Tree shaped core of this parallelisation strategy.

## Modelfactor Analysis

| Overview of whole program execution metrics | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Elapsed time (sec) | 1.62 | 0.82 | 0.44 | 0.32 | 0.25 | 0.22 | 0.19 | 0.17 | 0.16 |
| Speedup | 1.00 | 1.97 | 3.67 | 5.10 | 6.39 | 7.42 | 8.57 | 9.48 | 10.40 |
| Efficiency | 1.00 | 0.99 | 0.92 | 0.85 | 0.80 | 0.74 | 0.71 | 0.68 | 0.65 |

Table 1: Analysis done on Thu Nov 28 12:58:20 PM CET 2024, par1307

| Overview of the Efficiency metrics in parallel fraction, $\phi$=99.98% | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Global efficiency | 99.88% | 98.54% | 91.61% | 85.03% | 79.88% | 74.18% | 71.39% | 67.78% | 65.05% |
| Parallelization strategy efficiency | 99.88% | 98.53% | 94.63% | 90.97% | 87.00% | 82.36% | 79.40% | 75.73% | 72.95% |
| Load balancing | 100.00% | 99.30% | 96.46% | 96.64% | 95.00% | 93.02% | 90.15% | 89.02% | 84.76% |
| In execution efficiency | 99.88% | 99.22% | 98.10% | 94.13% | 91.58% | 88.54% | 88.07% | 85.07% | 86.06% |
| Scalability for computation tasks | 100.00% | 100.00% | 96.81% | 93.47% | 91.81% | 90.07% | 89.92% | 89.51% | 89.18% |
| IPC scalability | 100.00% | 99.86% | 99.83% | 99.76% | 99.68% | 99.60% | 99.60% | 99.50% | 99.51% |
| Instruction scalability | 100.00% | 100.16% | 100.16% | 100.16% | 100.16% | 100.16% | 100.15% | 100.15% | 100.15% |
| Frequency scalability | 100.00% | 99.99% | 96.82% | 93.55% | 91.96% | 90.29% | 90.14% | 89.82% | 89.48% |

Table 2: Analysis done on Thu Nov 28 12:58:20 PM CET 2024, par1307

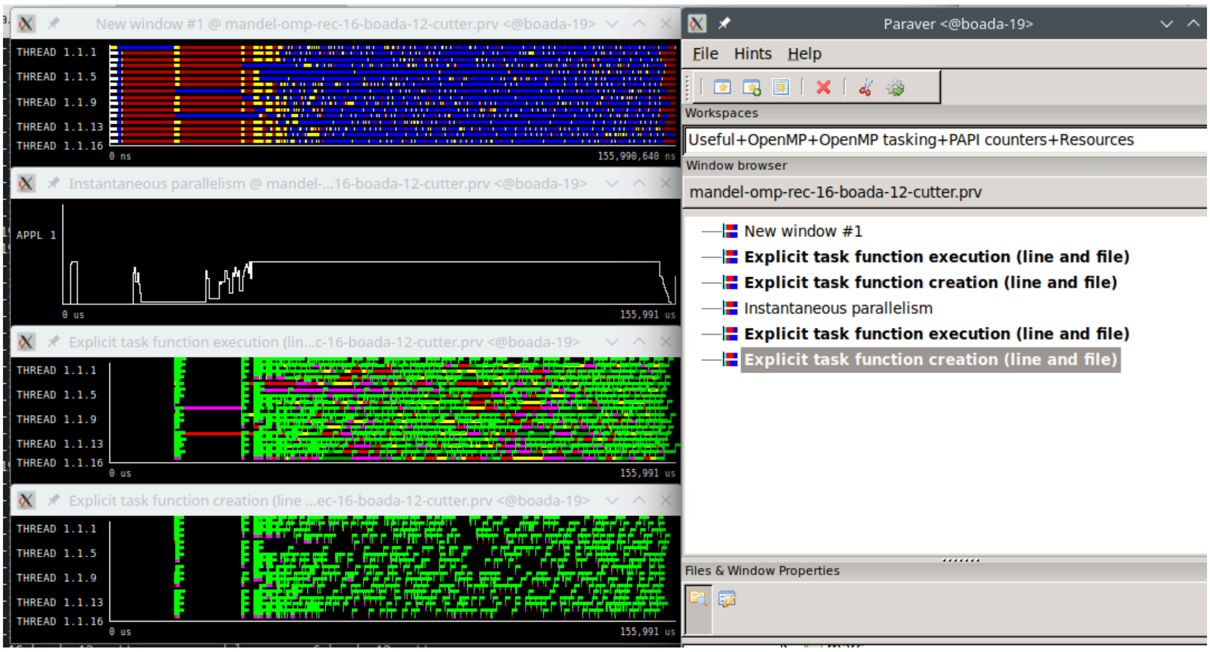
| Statistics about explicit tasks in parallel fraction | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Processors | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| Number of explicit tasks executed (total) | 7029.0 | 7029.0 | 7029.0 | 7029.0 | 7029.0 | 7029.0 | 7029.0 | 7029.0 | 7029.0 |
| LB (number of explicit tasks executed) | 1.0 | 0.78 | 0.6 | 0.61 | 0.68 | 0.45 | 0.55 | 0.49 | 0.64 |
| LB (time executing explicit tasks) | 1.0 | 1.0 | 0.98 | 0.96 | 0.94 | 0.92 | 0.89 | 0.89 | 0.85 |
| Time per explicit task (average us) | 228.07 | 228.64 | 236.32 | 244.71 | 249.29 | 254.62 | 255.34 | 256.98 | 257.74 |
| Overhead per explicit task (synch %) | 0.0 | 1.3 | 5.37 | 9.52 | 14.26 | 20.24 | 24.21 | 29.56 | 33.86 |
| Overhead per explicit task (sched %) | 0.12 | 0.16 | 0.23 | 0.27 | 0.44 | 0.74 | 1.02 | 1.46 | 1.76 |
| 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 Thu Nov 28 12:58:20 PM CET 2024, par1307

In these tables we can see a much better speedup compared to the previous strategy, as it doesn't get stuck at all when using an increasingly big number of processors. Furthermore, the efficiency of the parallelization strategy remains over 70% even at a high number of processors.

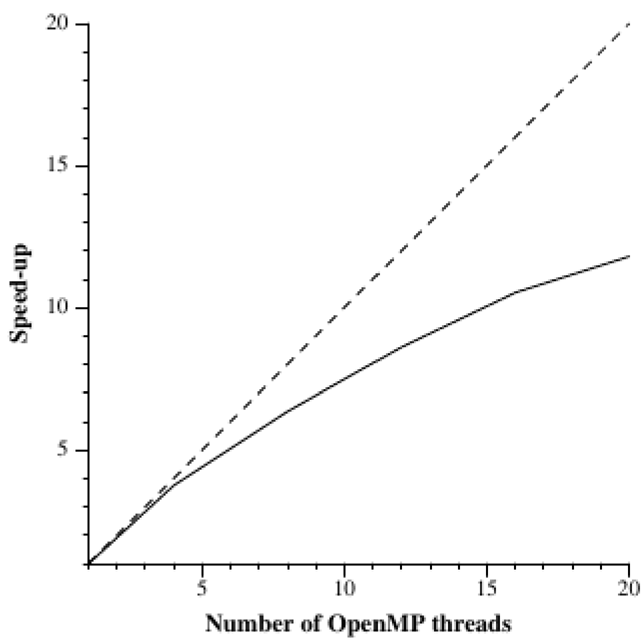
Furthermore and regarding the efficieny of this parallelisation strategy, we can see that it drops to ~70% when using 20 threads, but that's not too bad.

## Paraver Analysis




In this execution of wxparaver over this particular parallel implementation, we can see all the tasks created at each tree level. All the graphs look reasonably balanced, therefore there is not an apparent big Load Unbalance or task creation overhead problem causing a bottleneck that can be seen by the naked eye on these graphs.

## Strong Scalability




par1307
Speed-up wrt sequential time (mandel funtion only)
Generated by par1307 on Thu Nov 28 01:02:50 PM CET 2024

1 1.612781
4 0.427873
8 0.252638
12 0.186513
16 0.152506
20 0.136090

In this graph we can see how the speedup has more of a logarithmic shape, which is expected, but the efficiency and overall health of the speedup evolution is much better than that of the previous strategy.

# Summary of the elapsed execution time

| | Number of threads | | | | | |
|---|---|---|---|---|---|---|
| **Version** | **1** | **4** | **8** | **12** | **16** | **20** |
| **Iterative: Tile** | 3.089155 | 0.910804 | 0.698836 | 0.709182 | 0.709561 | 0.709295 |
| **Iterative: Finer Grain** | 3.094647 | 0.780898 | 0.418766 | 0.291014 | 0.220544 | 0.177836 |
| **Recursive: Leaf** | 1.611466 | 1.242143 | 1.323873 | 1.368214 | 1.369712 | 1.370270 |
| **Recursive: Tree** | 1.612781 | 0.427873 | 0.252638 | 0.186513 | 0.152506 | 0.136090 |

*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*