NTNU

Norwegian University of
Science and Technology

TDT4200 Parallel Computing

PS 5 - Pthreads and OpenMP

David Barahona Pereira

October 16, 2019

1 Parallelization with Pthreads

1.1 Task 1: Baseline measurement

1.1.1 a)

In order to set a baseline for future comparison, a reasonable set of parameters is used to run the serial and the parallel code. These parameter are shown in Table 1.1. The run-time of the application using these parameters is summarized in Table 1.2.

Table 1.1: Baseline parameters

Parameter	**Value**
Iterations	2048
Resolution	4096 x 4096
X	0.5
Y	0.5
Colors	1

Table 1.2: Baseline run-time

Algorithm	**Run-time**
Escape-Time	62.61 s
Mariani-Silver	7.22 s

1.1.2 b)

Given the Mariani-Silver's algorithm logic, the area of the Mandelbrot set that is being rendered can have an impact on the execution time. Take as an example Figures 1.1 and 1.2. These figures were rendered using the exact same parameters. However, the execution time to generate Figure 1.1 was 4,1 s while the time to generate Figure 1.2 was 3,43 s. This difference exists because Figure 1.2 has bigger areas in which the dwell value for the borders of a block is the same. Due to the properties of the Mandelbrot set, this implies that all the pixels inside the block also posses that same dwell value so there is no need to compute it. This difference makes possible that Figure 1.2 gets rendered faster than Figure 1.1 because it is performing less calculations.

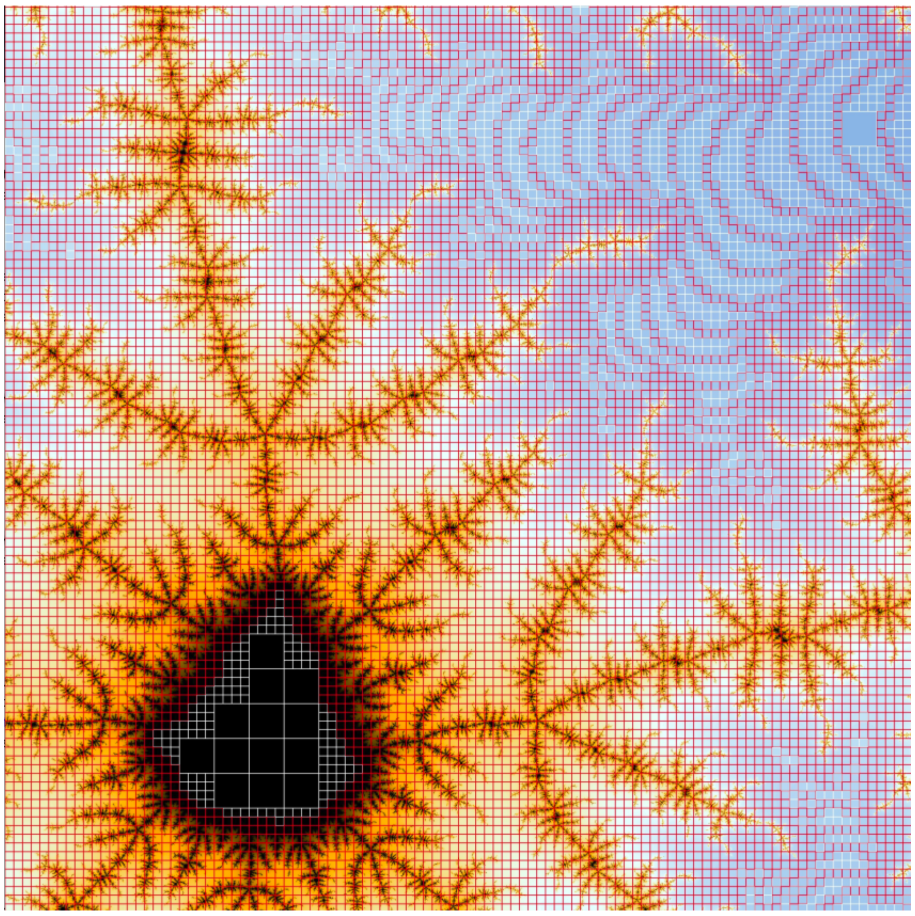

Figure 1.1: Rendered image with a few big blocks with the same dwell value

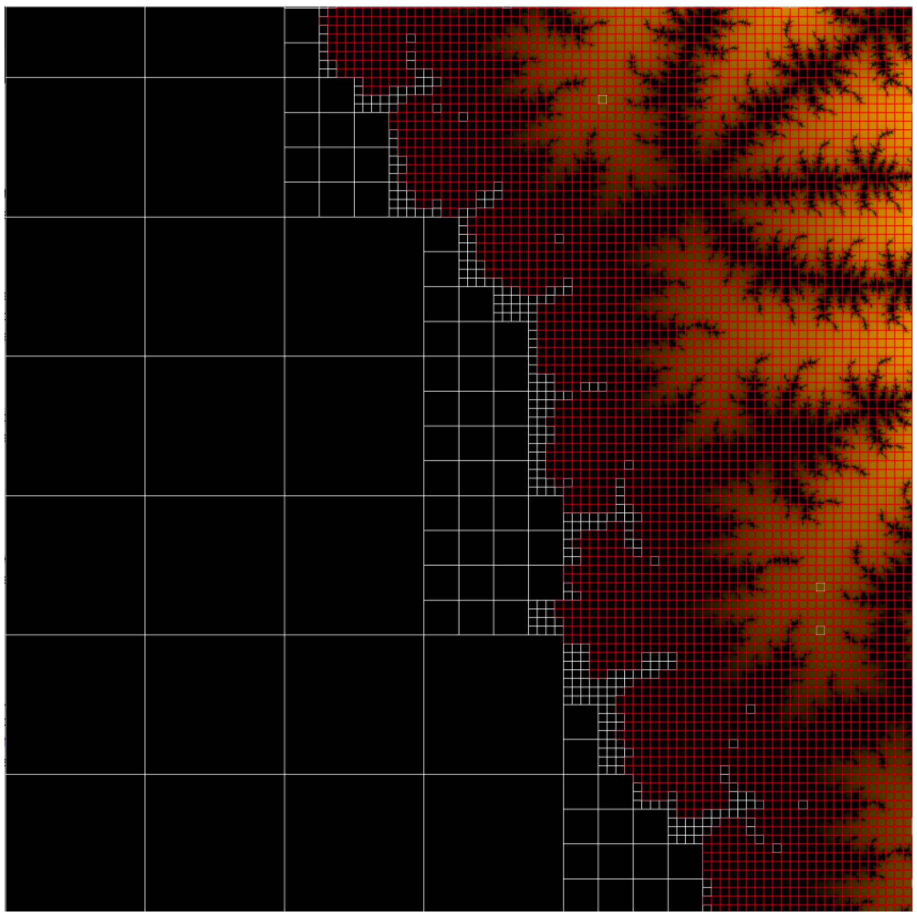
Figure 1.2: Rendered image with lots of big blocks with the same dwell value

1.2 Task 2: PThread Consumer/Producer

1.2.1 b)

After using pthreads to implement the Escape-Time and Mariani-Silver algorithms, the speedup due to the parallel execution can be measured. These results were obtained using 8 threads and they are summarized in Table 1.3. The speedup doesn't scale linearly with the number of threads and it saturates as it can be appreciated in Table 1.4. In this case, the speedup starts to saturate at around 8 threads. This is depends on the nature of the algorithm and also the system in which they run. The computer in which these measurements were taken has 4 physical cores able to run 8 threads simultaneously. Therefore, it makes sense that no much improvement is seen when having more than 8 threads executing the code.

Table 1.3: Speedup using pthreads

Algorithm	**Serial Run-time**	**Parallel Run-time**	**Speedup**
Escape-Time	62.61 s	13.51 s	4.63
Mariani-Silver	7.22 s	2.14 s	3.37

Table 1.4: Speedup using pthreads

Number of Threads	**Run-time**	**Speedup**
1	7.26 s	0.99
2	4.23 s	1.70
4	3.21 s	2.24
8	2.14 s	3.37
16	2.18 s	3.31
32	2.12 s	3.4
64	2.12 s	3.4
128	2.13 s	3.38

2 Matrix-Matrix Multiplication

2.1 Task 3: Baseline measurement

The code was instrumented in order to measure the execution time of the serial function. The way in which the code was instrumented can be appreciated in Listing 2.1. The instructions that instrument the code were placed there because the only part of the code that wants to be measured is the matrix multiplication itself. The execution time of the serial_mxm() function was 0.457 s.

```
void serial_mxm(const double *A,
                const double *B,
                double *C,
                int m, int n, int k)
{
  start = omp_get_wtime ( );
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      C[i*n + j] = 0;
      for (int l = 0; l < k; l++) {
        C[i*n + j] += A[i*k + l] * B[l*n + j];
      }
    }
  }
  end = omp_get_wtime ( );
}
```

Listing 2.1: Code instrumentation to measure the time of the serial function

2.2 Task 4: OpenMP parallelization

When using OpenMP to parallelize the code, an execution time of 0.0954 s was achieved. This means a speedup of approximately 4.8. In this case, the OpenMP implementation used 8 threads to run the application. As it can be noticed, using 8 threads doesn't imply a speedup of 8 since there is overhead during the OpemMP thread setup and the synchronization between threads.

2.3 Task 5: BLAS

When using the cblas library, the execution time was 0.09 s. The speedup compared to the serial version is 5.07. This speeup is comparable to the one obtained by using the OpenMP approach. The cblas library is able to provide such a good performance given that is carefully optimized by using CPU-specific assembly code including CPU features like SIMD-style instructions, instruction level parallelism, and cache-awareness.