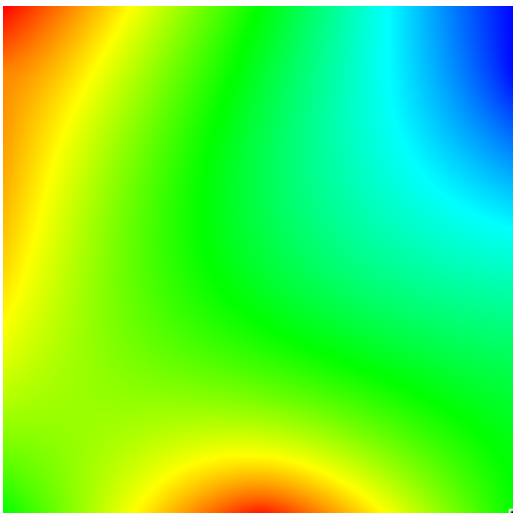
PAR Laboratory Assignment
Lab 5: Geometric (data) decomposition using implicit tasks: heat diffusion equation



David Cañadas López and Igor Duran Gallart, group 42,
19 October 2023, par4204 and par4206 (respectively)

December 29th, 2023

1.1 Sequential heat diffusion program and analysis with Tareador

We tackle this problem by analyzing both strategies using Tareador to generate the TDG of the resulting execution.

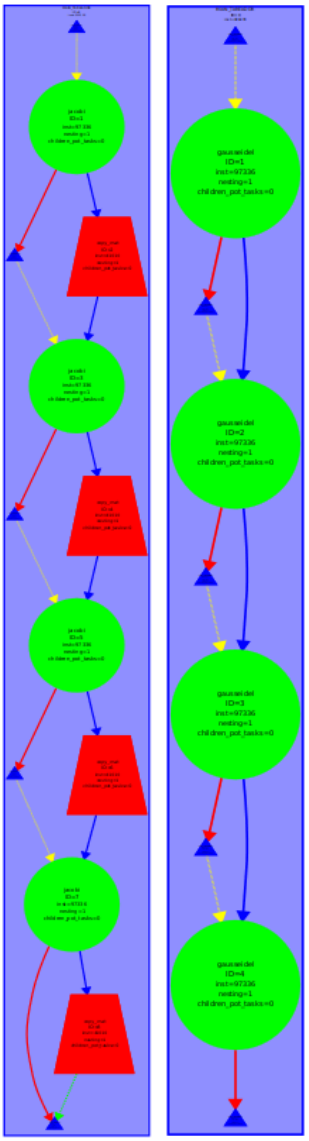


Figure 1. (Left) TDG of the initial Tareador Jacobi program. (Right) TDG of the initial Tareador Gauss-Seidel program.

We quickly see that because the TDGs generated by Tareador (Figure 1) are almost sequential, the naive solution must be optimized for any parallelism to be achieved. We then proceed by exploring the proposed granularity for the algorithms using Tareador.

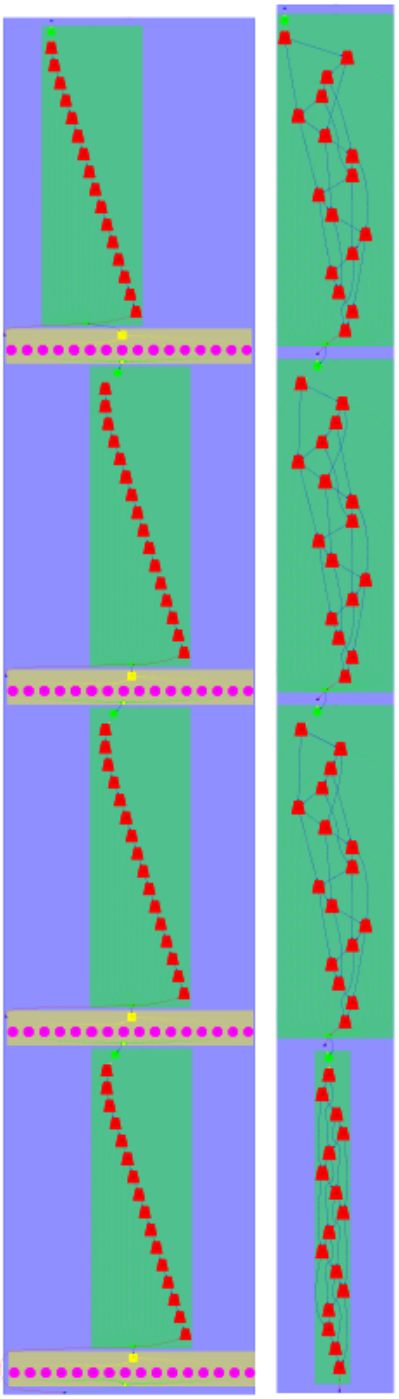


Figure 2. (*Left) TDG of the first optimization for the Tareador Jacobi program. (Right) TDG of the first optimization for the Tareador Gauss-Seidel program.*

This version of the algorithm achieves some parallelism, but a very serialized part is noticeable. When looking at the code again, we see that the variable `sum` is what produces dependencies between the tasks that are generated. We then continue our analysis by telling Tareador to ignore the variable `sum` (declaring it as independent) and generate the TDGs again.

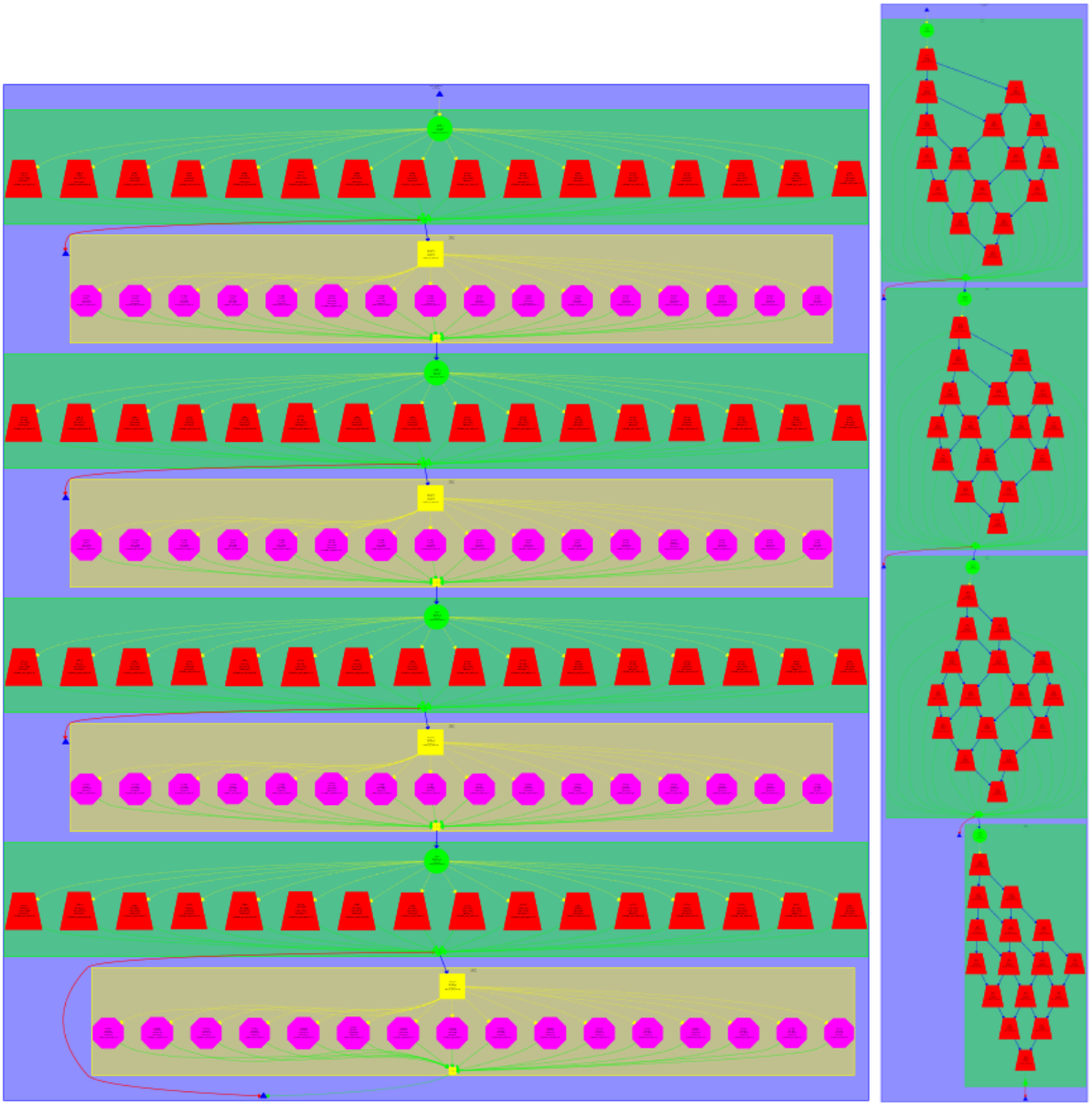


Figure 3. *(Left) TDG of the optimized Tareador Jacobi program. (Right) TDG of the optimized Tareador Gauss-Seidel program.*

This optimized version exploits much better the potential parallelism of each algorithm. However, there is a clear difference between the two TDGs generated (Figure 3), with the Jacobi one showing a better parallelization of the tasks generated: this is because of the way each algorithm computes its solution: the Jacobi program uses two different matrices for reading the input and writing the solution, while the Gauss-Seidel program reads and writes from the same input matrix, thus creating more dependencies between tasks and, in turn, reducing the amount of possible parallel tasks.

The code we modified is the following (see Figures 4 and 5):

```
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        tareador_start_task("in-copy");
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++)
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++)
            v[i*sizey+j] = u[i*sizey+j];
        tareador_end_task("in-copy");
      }
    }
```

Figure 4. Code modified (copy_mat) to implement the "one task per block" granularity in Tareador.

```
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        tareador_start_task("solve");
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
            tmp = 0.25 * ( u[ i*sizey      + (j-1) ] +  // left
                           u[ i*sizey      + (j+1) ] +  // right
                           u[ (i-1)*sizey + j     ] +  // top
                           u[ (i+1)*sizey + j     ] ); // bottom
            diff = tmp - u[i*sizey+ j];
            sum += diff * diff;
            unew[i*sizey+j] = tmp;
          }
        }
        tareador_end_task("solve");
      }
    }
```

Figure 5. Code modified (solve) to implement the "one task per block" granularity in Tareador

1.2 OpenMP parallelization and execution analysis: *Jacobi*

The first program we will parallelize using OpenMP will be the implementation of the Jacobi algorithm. Using the proposed geometric data decomposition for both u and utmp matrices produces the following results:

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	3.08	2.18	2.02	2.16
Speedup	1.00	1.41	1.53	1.42
Efficiency	1.00	0.35	0.19	0.09

Table 1: Analysis done on Thu Dec 14 04:02:33 PM CET 2023, par4204

Overview of the Efficiency metrics in parallel fraction, ϕ=67.13%				
Number of processors	1	4	8	16
Global efficiency	99.75%	81.32%	66.31%	36.65%
Parallelization strategy efficiency	99.75%	98.98%	98.08%	97.48%
Load balancing	100.00%	99.96%	99.88%	99.89%
In execution efficiency	99.75%	99.02%	98.20%	97.59%
Scalability for computation tasks	100.00%	82.16%	67.60%	37.60%
IPC scalability	100.00%	85.45%	77.50%	50.45%
Instruction scalability	100.00%	97.54%	94.88%	83.66%
Frequency scalability	100.00%	98.58%	91.94%	89.07%

Table 2: Analysis done on Thu Dec 14 04:02:33 PM CET 2023, par4204

Statistics about explicit tasks in parallel fraction				
Number of processors	1	4	8	16
Number of implicit tasks per thread (average us)	1000.0	1000.0	1000.0	1000.0
Useful duration for implicit tasks (average us)	2060.42	626.93	380.97	342.53
Load balancing for implicit tasks	1.0	1.0	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	5.11	0	0	0

Table 3: Analysis done on Thu Dec 14 04:02:33 PM CET 2023, par4204

Figure 6. Modelfactors tables of the first parallel implementation of the Jacobi solver

The modelfactors tables (Figure 6) show a very poor efficiency scaling, and the speedup is less than appropriate. Analyzing the second table we see that the parallel fraction is to blame for this, which is caused by not parallelizing the function that copies one matrix to the other (`copy_mat`, in this case). We then repeat the execution, optimizing this function to increase the parallel fraction of the program.

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	3.12	0.80	0.45	0.25
Speedup	1.00	3.88	6.94	12.61
Efficiency	1.00	0.97	0.87	0.79

Table 1: Analysis done on Thu Dec 14 04:44:05 PM CET 2023, par4204

Overview of the Efficiency metrics in parallel fraction, ϕ=98.82%				
Number of processors	1	4	8	16
Global efficiency	99.73%	99.62%	92.43%	90.00%
Parallelization strategy efficiency	99.73%	97.88%	95.29%	92.57%
Load balancing	100.00%	99.15%	97.82%	98.61%
In execution efficiency	99.73%	98.72%	97.42%	93.87%
Scalability for computation tasks	100.00%	101.78%	97.00%	97.23%
IPC scalability	100.00%	104.57%	105.54%	111.26%
Instruction scalability	100.00%	99.04%	99.00%	98.42%
Frequency scalability	100.00%	98.27%	92.84%	88.79%

Table 2: Analysis done on Thu Dec 14 04:44:05 PM CET 2023, par4204

Statistics about explicit tasks in parallel fraction				
Number of processors	1	4	8	16
Number of implicit tasks per thread (average us)	2000.0	2000.0	2000.0	2000.0
Useful duration for implicit tasks (average us)	1536.55	377.42	198.01	98.77
Load balancing for implicit tasks	1.0	0.99	0.98	0.99
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	4.18	0	0	0

Table 3: Analysis done on Thu Dec 14 04:44:05 PM CET 2023, par4204

Figure 7. Modelfactors tables of the optimized parallel implementation of the Jacobi solver

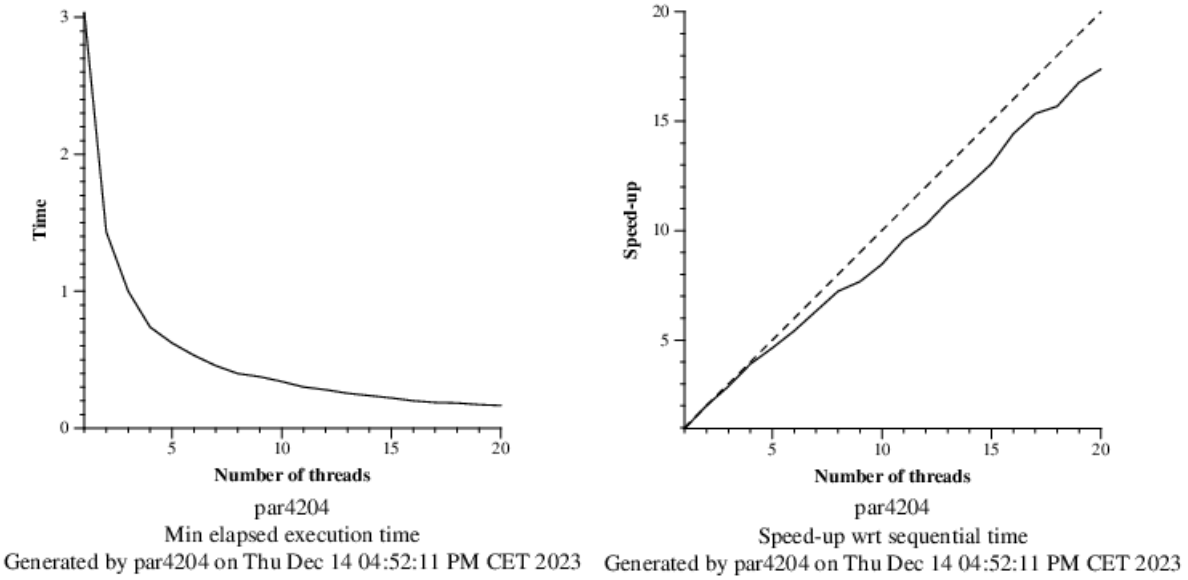


Figure 8. Scalability plots for the optimized implementation of the Jacobi solver

The optimization we made reported a greater parallel fraction, producing much better efficiency and speedup when scaling (Figure 7). As expected, the execution time is greatly reduced (Figure 8).

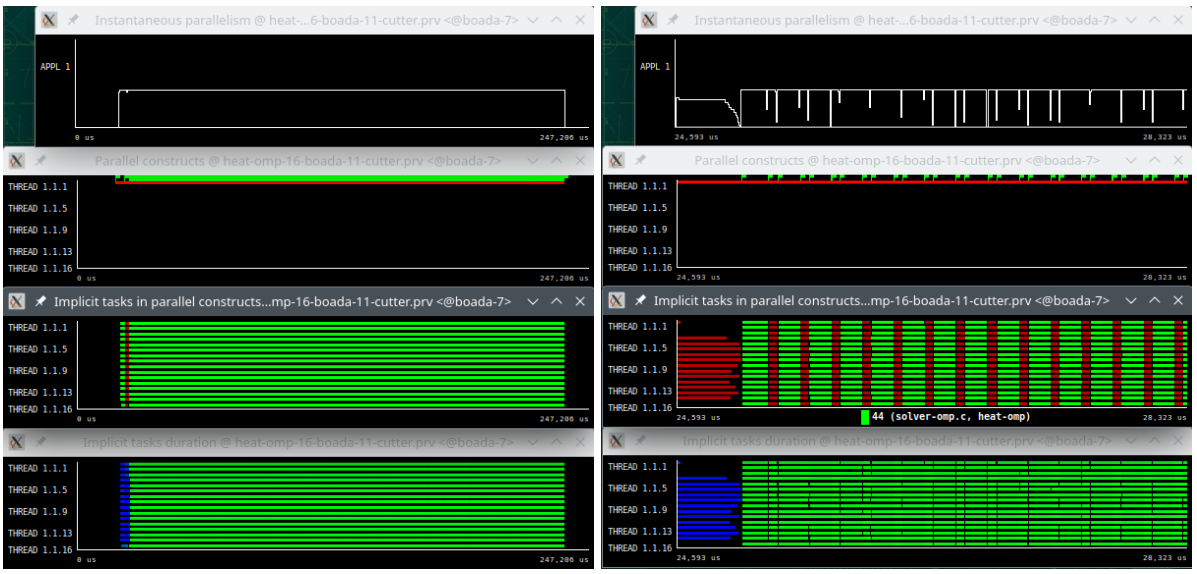


Figure 9. Paraver trace of the execution with 16 threads, full (left) and zoomed in (right).

In the analysis we made of the execution of the program with 16 threads (Figure 9) we can see that the full size timeline is quite misleading because it shows the whole execution as a single parallel region. If we zoom in, the starting and ending points of each parallel region created for each call to `solve` are now visible (second and third timelines) as well as how it reflects on the instantaneous parallelism of the program (first timeline).

1.3 OpenMP parallelization and execution analysis: *Gauss-Seidel*

Next we will parallelize using OpenMP the implementation of the Gauss-Seidel algorithm. This algorithm was trickier to work with: we needed to implement our own data structure to manage the dependencies introduced by using the same matrix for writing the solution. The code we modified for the first version of the implementation is the following (see Figure 10):

```
// 2D-blocked solver: one iteration step
double solve_gauss (double *u, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int P=omp_get_max_threads();
    int nblocksi=P;
    int nblocksj=20;
    int next[P];

    for (int i=0; i<P; i++) next[i]=0;

    #pragma omp parallel reduction(+: sum) private(tmp, diff) num_threads(P)
    {
        int myid = omp_get_thread_num();
        int tmp_next;

        int blocki = omp_get_thread_num();
        int i_start = lowerb(blocki, nblocksi, sizex);
        int i_end = upperb(blocki, nblocksi, sizex);
        for (int blockj=0; blockj<nblocksj; ++blockj) {
            int j_start = lowerb(blockj, nblocksj, sizey);
            int j_end = upperb(blockj, nblocksj, sizey);

            if (myid > 0) {
                do {
                    #pragma omp atomic read
                    tmp_next = next[myid-1];
                } while (tmp_next<=next[myid]);
            }

            for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
                for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
                    tmp = 0.25 * (  u[ i*sizey     + (j-1) ] +  // left
                                    u[ i*sizey     + (j+1) ] +  // right
                                    u[ (i-1)*sizey + j     ] +  // top
                                    u[ (i+1)*sizey + j     ] ); // bottom
                    diff = tmp - u[i*sizey+ j];
                    sum += diff * diff;
                    u[i*sizey+j] = tmp;
                }
            }
            #pragma omp atomic update
            next[myid]++;
        }
    }
    return sum;
}
```

Figure 10. Code modified (solve_gauss) implementing the ordering constraints for the Gauss-seidel solver with nblocksj=20.

However, if false sharing on the `next` vector is a matter of concern, we could add padding between each element. For instance, assuming a cache line size of 64 bytes and an int size of 4 bytes, we would need 60 bytes of padding (an array of 15 unused int values) for each element.

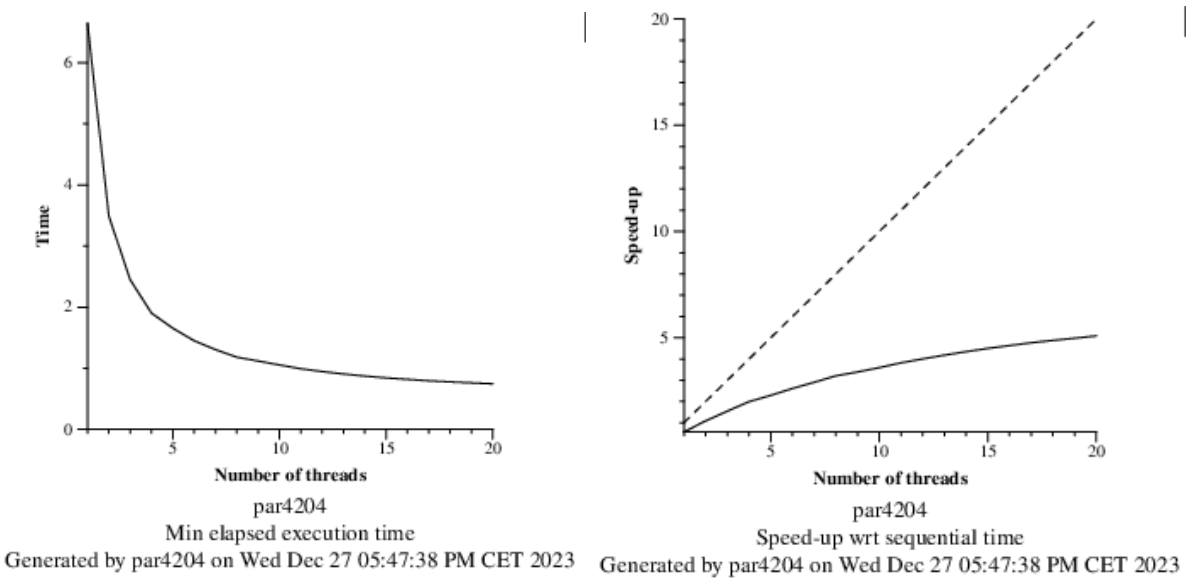


Figure 11. Scalability plots for the first implementation of the Gauss-seidel solver with nblocksj=20.

As shown in the scalability plots (Figure 11) generated by the execution of the program with 20 blocks in the j dimension, the speedup is not linear and far from perfect. To achieve better performance, we implemented the strategy of using the parameter that the user supplies the program to find a better value for `nblocksj` by changing the following code (see Figure 12):

```
int P=omp_get_max_threads();
int nblocksi=P;
int nblocksj=P*userparam;
int next[P];
```

Figure 12. Code modified (solve_gauss) to set blocksj according to the user's parameter for the Gauss-seidel solver.

The goal of this optimization is to have better control over the ratio between computation and synchronization. This is achieved by changing the amount of columns the blocks are divided into: the more blocks per line, the finer the granularity of the tasks (which results in more tasks created, less work per task, and more synchronization between them to have a coherent execution), and the contrary: the fewer blocks per line, the coarser the granularity of the tasks (which results in less tasks created, more computation per task, and less synchronization between them).

In Figure 12 we show the code we used to implement this. The number of blocks per column is proportional to the number of threads used, and the user-supplied parameter acts as a multiplier of that amount.

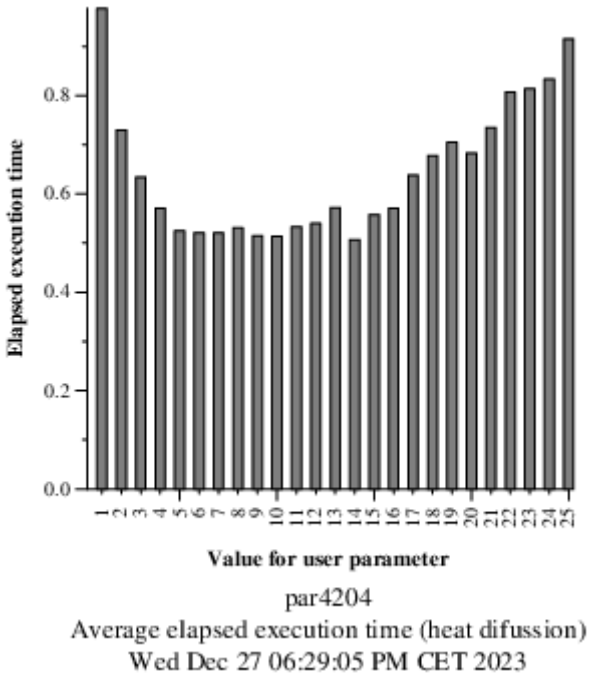


Figure 13. Relation of the execution time and userparam value for the execution of the program with 16 threads.

In figure 13 we appreciate that the best value for the user-supplied parameter is 14, since it produces the lowest execution time when executing the program with 16 threads.

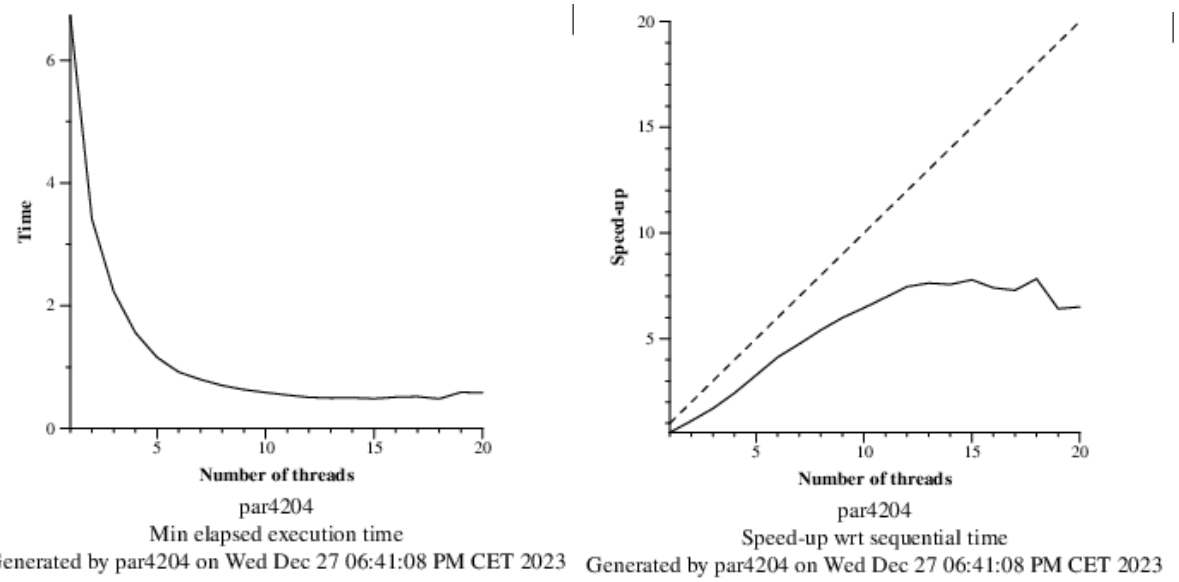


Figure 14. Scalability plots for the second implementation of the Gauss-seidel solver with the best value for userparam (14).

When checking the scalability of the version with `userparam` equal to 14 (Figure 14) and comparing it to the version of the program where `nblocksj` is always equal to 20 (Figure 11) we can conclude that a `nblocksj` value of 14 reports better performance than a value of 20 for all the amounts of threads tested. However, for even larger amounts of threads, the trend of the first version of the program seems to be stable and going upwards, while the second

version drops near the end, possibly having worse performance on the long run. This hypothesis also depends on the number of cores on the machines that execute the program: speedup could be severely worsened if too many threads are used and there are not sufficient cores available to process all in parallel.

In conclusion, the Gauss-Seidel algorithm is better for situations where the memory usage constraint is tighter but performance is not as critical, whereas the Jacobi algorithm is better for situations where maximum performance is required and there is plenty of memory to use.