UNIVERSITAT POLITÈCNICA DE CATALUNYA

PARALELISMO

Deliverable5 - Descomposition

Geometric (data) decomposition using implicit tasks: heat diffusion equation

Autores:
Guo Haobin
Jin Haonan

Mayo Q2 2023-2024

INTRODUCTION

In this laboratory assignment, our focus will be on parallelizing a sequential code that simulates heat diffusion in a solid body. The code employs two different solvers for the heat equation: Jacobi and Gauss-Seidel. Our objective is to parallelize this code to improve its performance and enable efficient computation of heat diffusion.

1.Sequential heat diffusion program and analysis with Tareador

First of all, our intention is to use Tareador for analyzing the task dependence graphs generated by utilizing two distinct solvers with a coarse-grained approach. Here are the graphs we have obtained as a result:

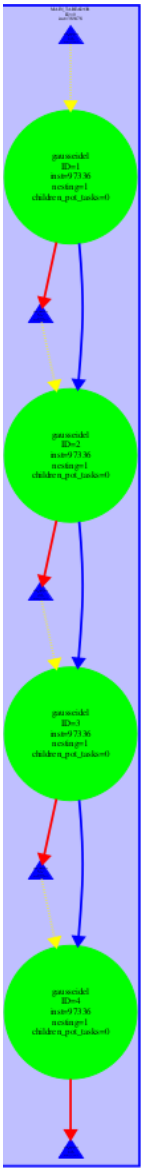


Figura 1: GaussTDG

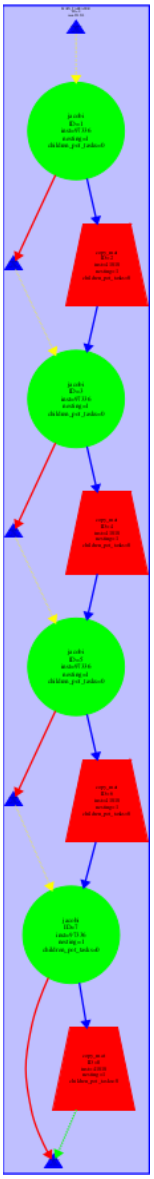


Figura 2: Jacobi TDG

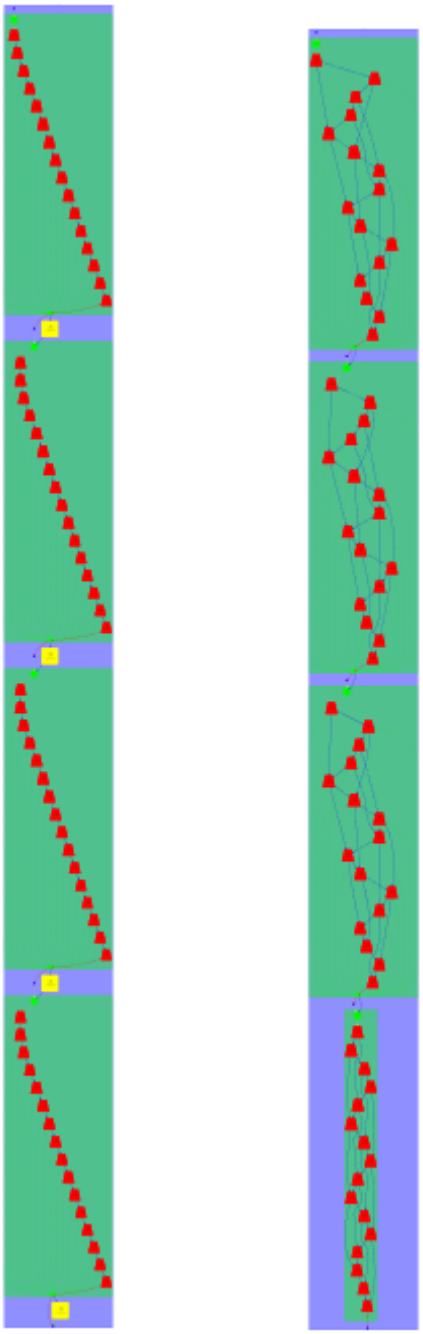
A notable observation is that the task dependence graph of the Jacobi version encompasses two distinct task types, with green tasks representing the "solve" operation and red tasks representing the "copy_mat" operation. Conversely, the Gauss version exhibits only a single task type as it does not involve any copying. Additionally, it is worth mentioning that the obtained results are less than satisfactory. Moreover, a further observation indicates that the tasks are sequential in nature. In order to address this issue, we propose transitioning from a coarse-grained approach to a finer-grained one.

In order to obtain a finer granularity, we have changed the code to create a task per block:

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

    int nblocksi=4;
    int nblocksj=4;

    //tareador_disable_object(&sum);
    for (int blocki=0; blocki<nblocksi; ++blocki) {
      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("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");
      }
    }
    //tareador_enable_object(&sum);

    return sum;
}
```

Figure 3: Part of code of *solver-tareador-notp.c*

Following the refinement of granularity, we utilized Tareador once again to examine the task dependence graphs of both methods.



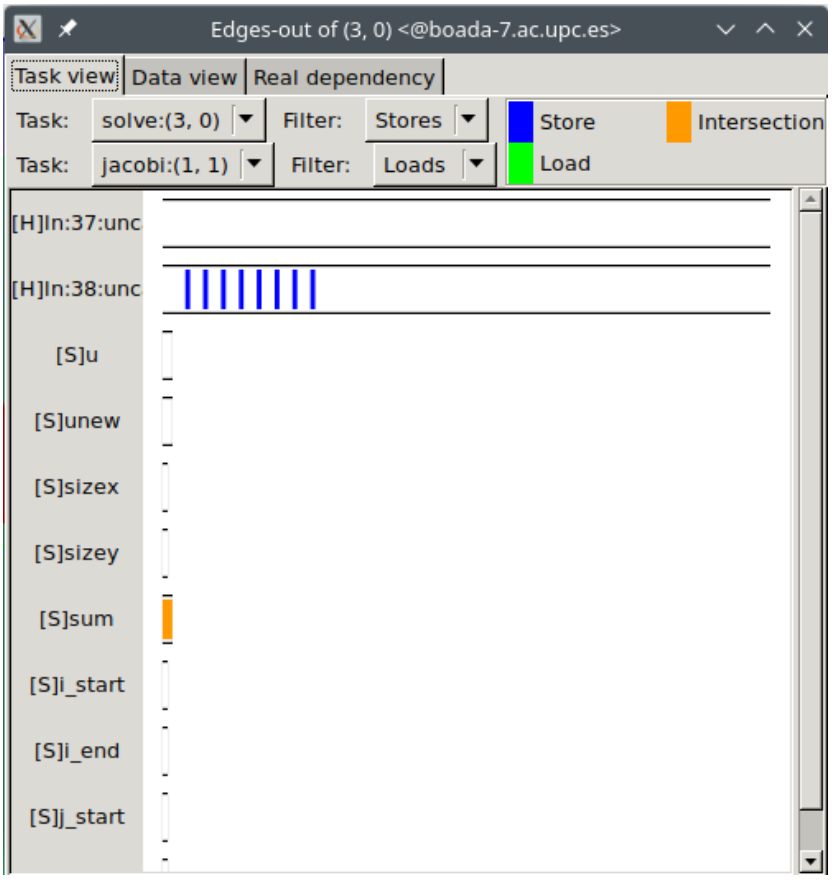


Figure 4: DataView option in Tareador
Figure 5: Task Dependency Graph with finer granularity (Jacobi, left)
Figure 6: Task Dependency Graph with finer granularity (Gauss-Seidel, right)

As depicted in Figures 5 and 6, although we have achieved finer-grained tasks, they still exhibit a sequential nature due to task dependencies. Notably, Figure 9 highlights that the variable "sum" is the root cause of this dependency. To explore the potential for parallelization using Tareador while excluding the "sum" dependency, we emulate the effect of protecting the dependences by uncommenting the relevant section of code: tareador_disable_object(&sum) and tareador_disable_object(&sum).

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

    int nblocksi=4;
    int nblocksj=4;

   tareador_disable_object(&sum);
    for (int blocki=0; blocki<nblocksi; ++blocki) {
      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("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");
      }
    }
    tareador_enable_object(&sum);

    return sum;
}
```

Figure 7: Part of code of *solver-tareador-notp.c*

After changing the code and compile it, we executed again for both cases and then obtaining the following results:

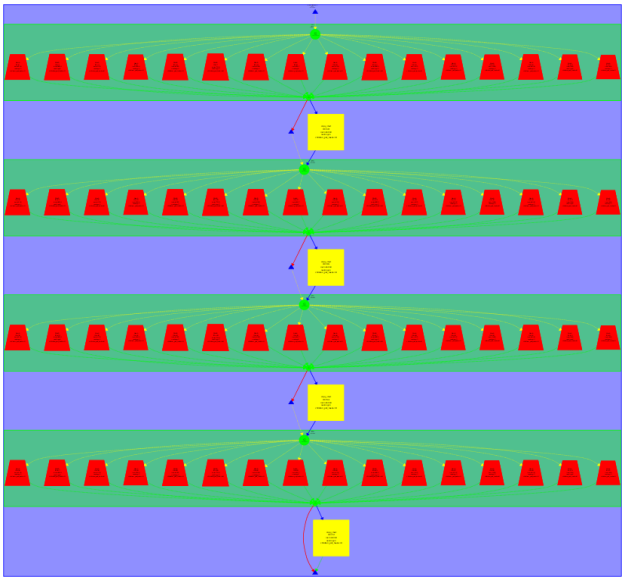


Figure 8: Task Dependency Graph without taking into account
the variable sum (Jacobi)

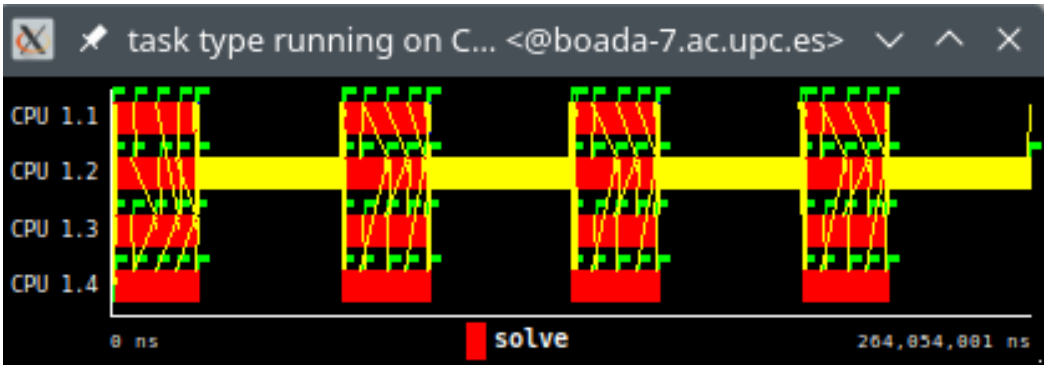
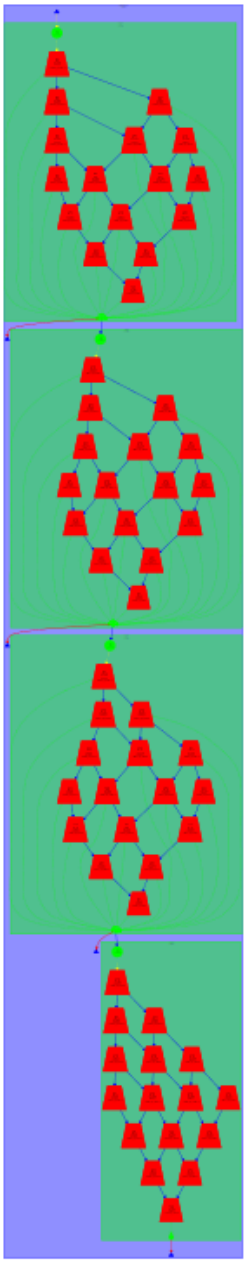


Figure 9: Timeline window with 4 threads of Jacobi version



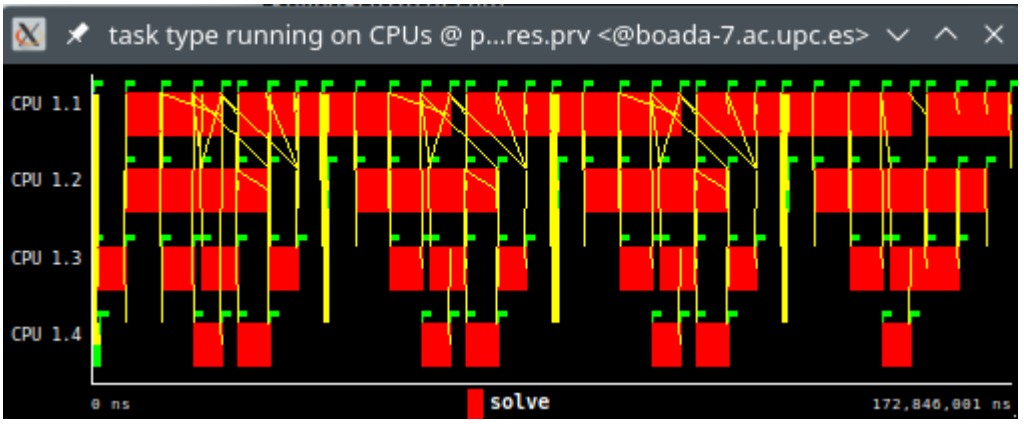


Figura 10: Gauss parallelism TDG(left)

Figure 11: Timeline window with 4 threads of Gauss-Seidel version(right)

Our analysis has revealed increased parallelism. Furthermore, upon closer examination of the previous figures, we have identified other noteworthy aspects. Firstly, in the Jacobi version, a specific portion of the code , the copy_mat function acts as a serialization point for the program. We plan to parallelize it at a later stage. Secondly, in the Gauss-Seidel version, we observe that tasks are executed in a wavefront fashion, leading to an imbalance in the workload distribution. Each thread undertakes a varying number of tasks. This wavefront behavior limits the scalability of the method. Moreover, to protect access to the "sum" variable in our OpenMP implementation, we can employ clauses such as atomic, critical, or reduction. Among these options, the most efficient approach is to use reduction (+:sum).

The figure below highlights the part of the code that we can parallelize in the Jacobi method

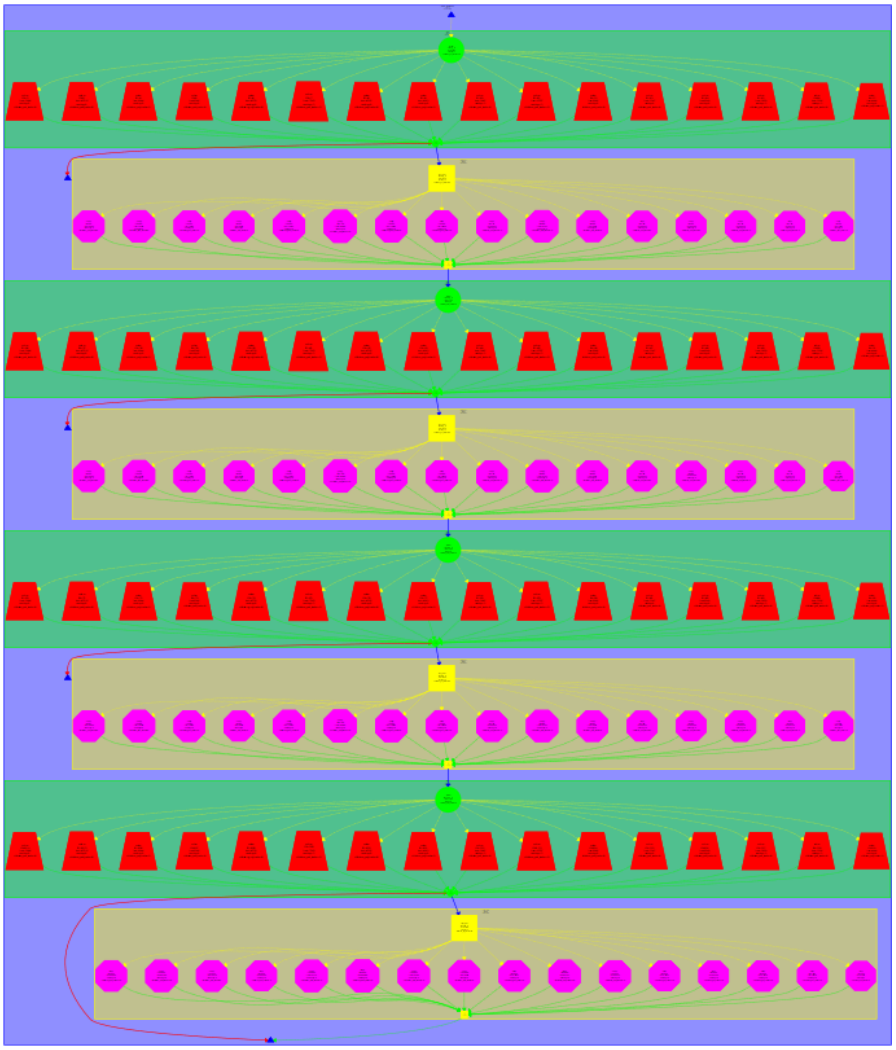


Figure 12: Task Dependency Graph having CopyMat parallelised (Jacobi)

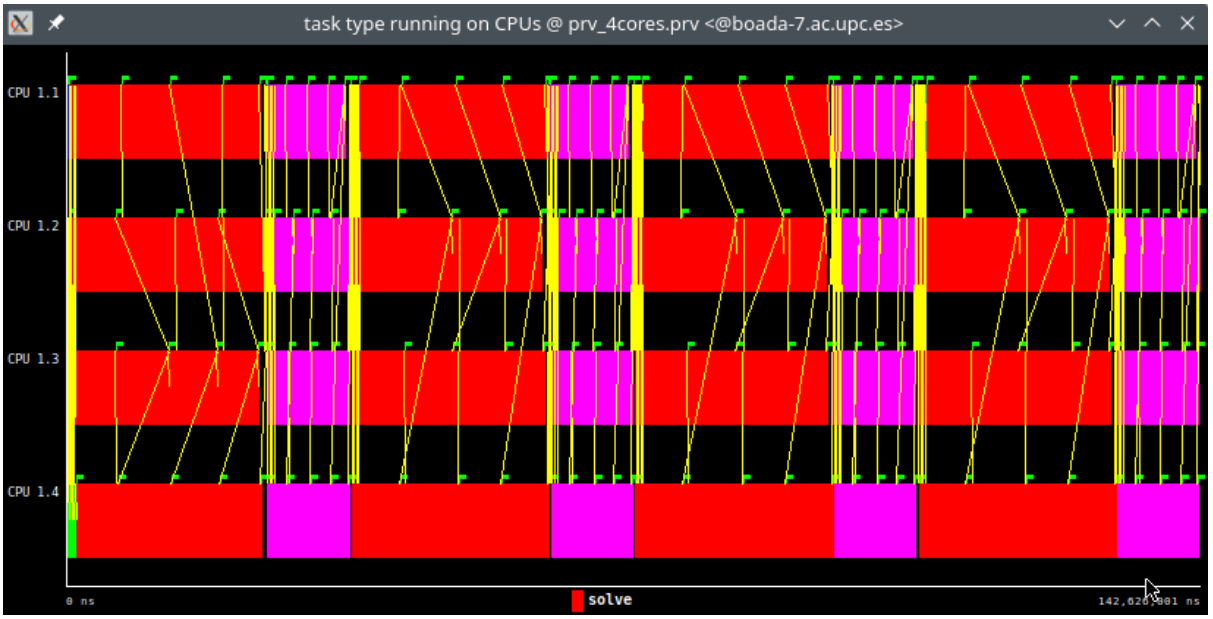


Figure 13: Timeline window with 4 threads having CopyMat parallelised (Jacobi)

As observed, the previously serialized section of the Jacobi version has now been effectively distributed among all threads, ensuring parallel execution. Furthermore, each thread performs its workload in a balanced and evenly distributed manner.

1.2. OpenMP parallelization and execution analysis: Jacobi

In this section, our goal is to parallelize the sequential code for the heat equation using the Jacobi solver. The first step involves parallelizing the "solve" function while considering the discovered dependency on the variable "sum." Additionally, we have chosen to make the variables "diff" and "tmp" private. To accomplish this, we have added the clause "#pragma omp parallel private (tmp, diff) reduction (+:sum)," as illustrated in the following image:

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

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;

    #pragma omp parallel private(tmp, diff) reduction(+:sum)
    {
      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);
        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;
          }
        }
      }
    }

    return sum;
}
```

Figure 14: Part of code of solver-omp-jacobi.c (solve)

After making the necessary modifications to the code, we proceeded to execute it with 1 and 8 threads. Additionally, we verified the correctness of the parallelized code by comparing the resulting image with the original image obtained in the first section. The following are the results obtained from the execution:

```
par2111@boada-7:~/lab5$ cat heat-omp-jacobi-1-boada-11.txt
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 2.127
Flops and Flops per second: (11.182 GFlop => 5257.76 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

```
par2110@boada-8:~/lab5$ cat heat-omp-jacobi-8-boada-12.txt
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 2.334
Flops and Flops per second: (11.182 GFlop => 4791.28 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 16: Result of the execution with Jacobi method having solver parallelised

As observed, the parallelized version of the code has demonstrated improved execution time compared to the sequential version depicted in Figure 1. However, there is not a significant difference in execution time between running the program with 1 thread and 8 threads.

1.2.1 Overall Analysis

After validating and observing the results of the previous code we execute it with the script submit-strong-extrae.sh to trace the execution for 1, 4, 8, 16 threads and analyse it with the modelfactor. These are the results:

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.82	2.14	1.98	2.14
Speedup	1.00	1.32	1.42	1.32
Efficiency	1.00	0.33	0.18	0.08

Table 1: Analysis done on Wed May 24 11:05:36 AM CEST 2023, par2110

Overview of the Efficiency metrics in parallel fraction, ϕ=64.72%				
Number of processors	1	4	8	16
Global efficiency	99.65%	70.85%	58.13%	33.05%
Parallelization strategy efficiency	99.65%	84.47%	98.19%	97.87%
Load balancing	100.00%	87.05%	99.94%	99.82%
In execution efficiency	99.65%	97.03%	98.26%	98.05%
Scalability for computation tasks	100.00%	83.87%	59.20%	33.77%
IPC scalability	100.00%	84.45%	69.02%	45.78%
Instruction scalability	100.00%	99.97%	92.79%	82.30%
Frequency scalability	100.00%	99.35%	92.43%	89.61%

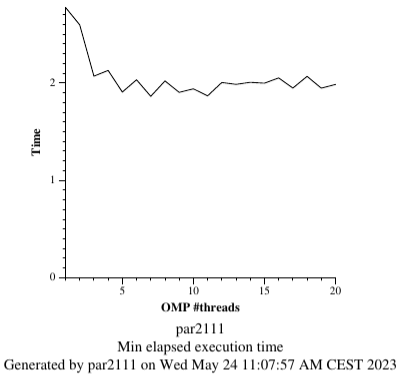
Table 2: Analysis done on Wed May 24 11:05:36 AM CEST 2023, par2110

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)	1816.93	541.56	383.65	336.31
Load balancing for implicit tasks	1.0	0.87	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	6.33	171.84	7.19	7.62

Table 3: Analysis done on Wed May 24 11:05:36 AM CEST 2023, par2110

We also execute it with the script submit-strong-omp.sh to obtain the scalability plot:

Based on the performance plot and the values of the speedup in the first table, it is evident that the scalability of the parallelized code is not satisfactory. Additionally, the parallel fraction, which is currently at 64.72%, can be further improved. This indicates that there is room for optimization in terms of parallel execution and efficiency. It would be beneficial to analyze and identify potential bottlenecks or limitations in the parallel implementation to enhance both scalability and parallel fraction.



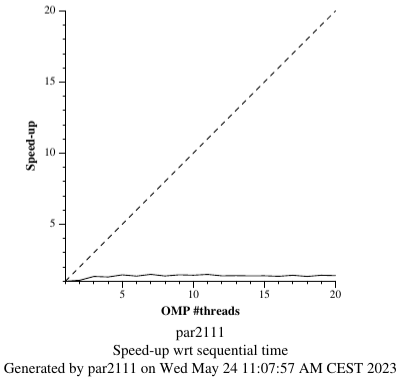


Figure 17: Scalability plot of the Jacobi version having solver parallelised.

1.2.2 Detailed Analysis

In addition, we use the Paraver in order to analyse in more detail the behaviour of the code with 16 threads:

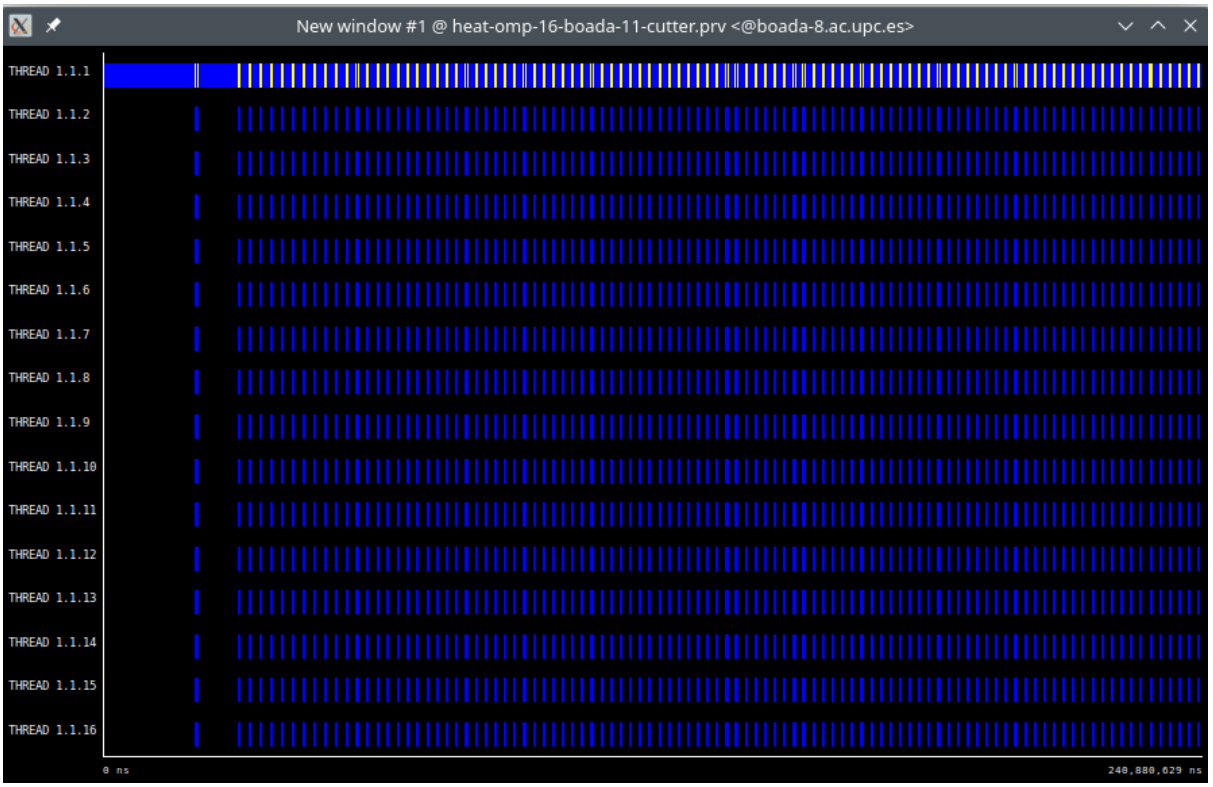


Figure 18: Timeline window of the Jacobi version having solver parallelised with 16 threads

Figure 19: Instantaneous Parallelism copy_mat not parallelised with 16 threads

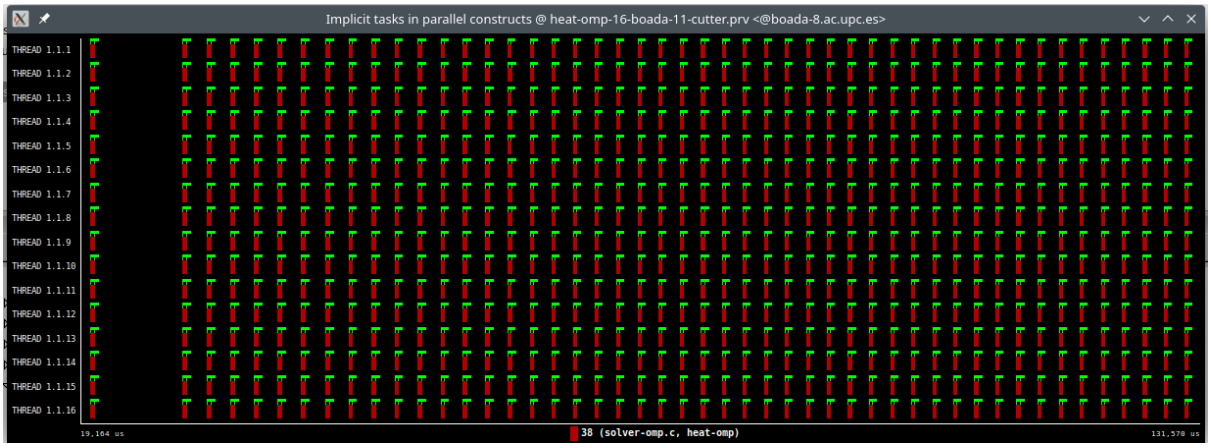


Figure 20: Implicit Tasks in Parallel Constructions copy_mat not parallelised with 16 threads

By examining the timeline of the trace and the other two Hints, it becomes apparent that there are certain sections in the program that still exhibit sequential behavior. These sequential portions can be attributed to a specific function within the code that has not yet been parallelized. After careful analysis, we have determined that by parallelizing the "copy_mat" function, we can increase the level of parallelism and subsequently improve the overall performance of the program.

1.2.3 Optimization

In order to do the parallelisation of copy_mat, we added the clause: #pragma omp parallel to create the parallel region and also we deleted the outer loop of for:

```
// Function to copy one matrix into another
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;
    #pragma omp parallel
    {
      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);
        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];
      }
    }
}
```

Figure 21: Part of code of solver-omp-Jacobi-Par.c (copy_mat)

```
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 0.349
Flops and Flops per second: (11.182 GFlop => 32024.10 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Figure 22: Result of the execution with Jacobi method having solver and copy_mat parallelised

1.2.4 Overall Analysis of the Optimised Code

We also use the script submit-strong-extrae.sh to get more information about the program executed in different numbers of threads.

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.85	0.74	0.42	0.24
Speedup	1.00	3.86	6.78	11.93
Efficiency	1.00	0.96	0.85	0.75

Table 1: Analysis done on Wed May 24 11:22:18 AM CEST 2023, par2111

Overview of the Efficiency metrics in parallel fraction, ϕ=98.86%				
Number of processors	1	4	8	16
Global efficiency	99.62%	99.40%	91.08%	86.64%
Parallelization strategy efficiency	99.62%	96.67%	95.26%	93.40%
Load balancing	100.00%	98.16%	97.55%	98.06%
In execution efficiency	99.62%	98.48%	97.66%	95.24%
Scalability for computation tasks	100.00%	102.83%	95.62%	92.77%
IPC scalability	100.00%	103.48%	104.40%	107.18%
Instruction scalability	100.00%	99.95%	99.01%	96.98%
Frequency scalability	100.00%	99.43%	92.51%	89.25%

Table 2: Analysis done on Wed May 24 11:22:18 AM CEST 2023, par2111

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)	1403.55	341.23	183.48	94.56
Load balancing for implicit tasks	1.0	0.98	0.98	0.98
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	5.3	21.55	10.91	8.51

Table 3: Analysis done on Wed May 24 11:22:18 AM CEST 2023, par2111

Compared to the previous version, there are notable improvements in various aspects. The parallel fraction has significantly increased, reaching 98.86%. This indicates that a larger portion of the code is now effectively parallelized. Other enhancements include improvements in speedup, global efficiency, and scalability for computational tasks.

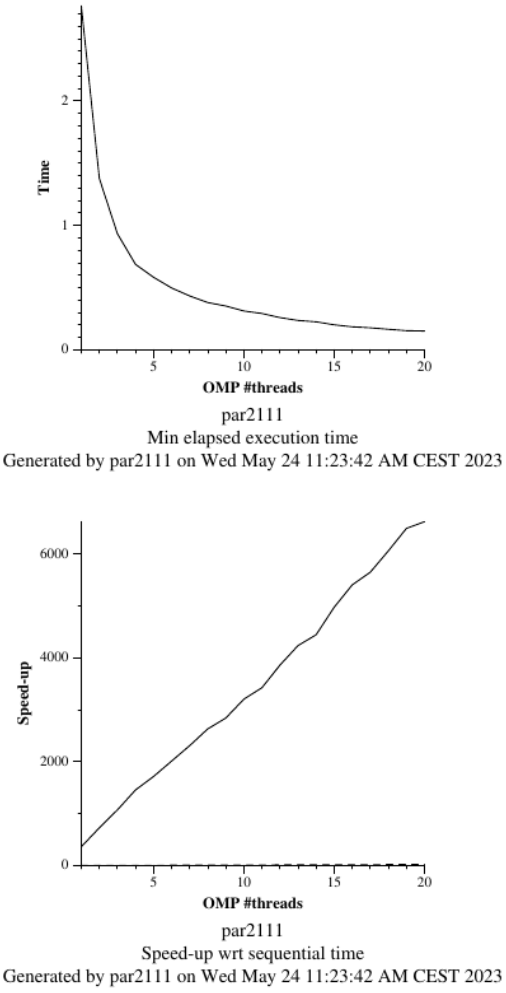


Figure 23: Scalability plot of the Jacobi version having solver and copy_mat parallelised.

Indeed, the scalability plot (Figure 20) highlights a significant improvement in scalability compared to previous versions. In the first graph, the execution time demonstrates a sharp decrease as the number of threads increases, indicating a successful utilization of parallel resources. This reduction in execution time reflects the efficient distribution of computational tasks among multiple threads.

The second graph, which represents speedup, showcases an almost linear relationship, closely approaching the ideal speedup. This implies that the parallelized code is effectively leveraging the available resources and efficiently dividing the workload, resulting in improved performance.

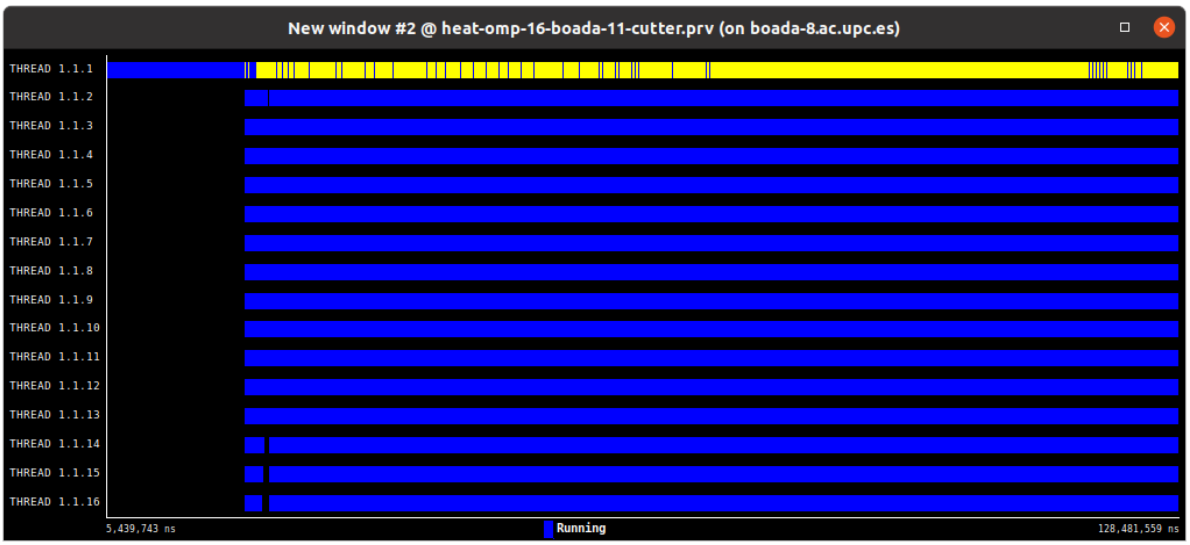


Figure 24: Timeline window of the Jacobi version having solver and copy_mat parallelised with 16 threads

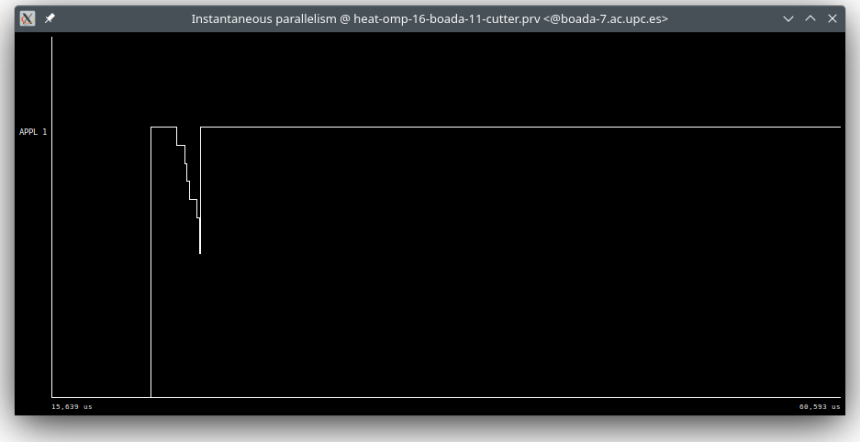


Figure 25: Instantaneous Parallelism copy_mat parallelised with 16 threads

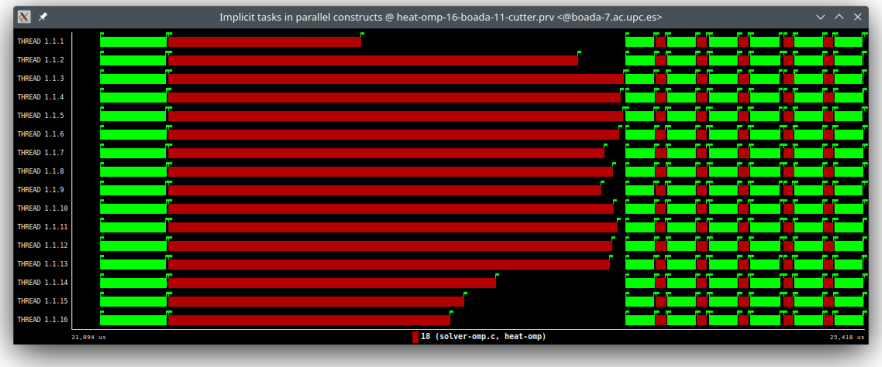


Figure 26: Implicit Tasks in
Parallel Constructions
copy_mat parallelised with 16 threads

Given that we exclusively parallelized the "copy_mat" function, the execution time for invocations of the "solve" function is expected to remain unchanged. Consequently, the performance enhancement in the program can be attributed to the parallelization of the "copy_mat" function, as previously discussed. It is evident from the timeline that the significant reduction in black spaces, which represent thread idle time, has led to a decrease in waiting time for execution.

In summary, the parallelization of the "copy_mat" function has resulted in an improved program performance. This enhancement is characterized by improved scalability, a notable reduction in execution time, and an adequate speedup. These positive outcomes can be observed by examining various metrics, demonstrating the overall improved performance of the program.

1.3 OpenMP parallelization and execution analysis: Gauss- Seidel

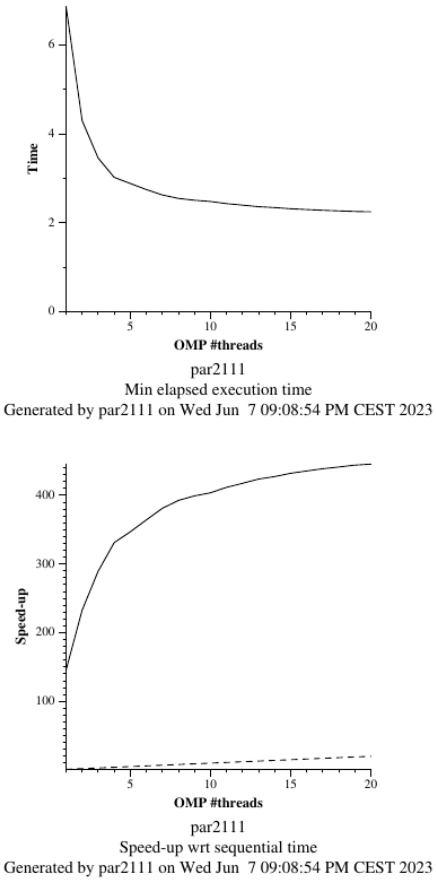
To initiate the parallelization process, we begin with the foundational principles of Jacobi's solver. Initially, we establish a vector to facilitate the tracking of calculated values. Subsequently, we evaluate whether to employ Jacobi or Gauss-Seidel by examining the expression "u == unew." In the case that "u == unew" evaluates to true, signifying the selection of the Gauss-Seidel solver, an additional verification is performed to ensure we are not operating within block 0. This precautionary measure aims to prevent any potential index out of bounds errors that may occur in the subsequent expression.

```
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=omp_get_max_threads();
    int nblocksj = 4;

    int mat[24] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    #pragma omp parallel private(diff) reduction(+:sum)
    {
      int cont;
      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((u==unew) && blocki !=0){
                do{
                        #pragma omp atomic read
                        cont = mat[blocki-1];
                }while(cont <= blockj);
        }
        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;
          }
        }
        if(u==unew){
                #pragma omp atomic write
                mat[blocki] = mat[blocki] + 1;
        }

      }
    }

    return sum;
}
```

Figure 27: Part of code of solver-omp-gauss.c (solve)

At this stage, we access the "mat" vector, which stores information concerning the calculated blocks. To synchronize the threads appropriately, we employ a while loop that blocks execution until all necessary dependencies are computed. Once all the dependencies have been determined, we proceed with the algorithm to calculate the current position accordingly. To inform the blocked threads of the completion of a dependency calculation, we increment the corresponding position within the "mat" vector.

With respect to the "#pragma omp" directives, as elaborated in the accompanying documentation, we make use of the "#pragma omp atomic write" and "#pragma omp atomic read" directives to maintain memory consistency throughout the parallel execution.
The first implementation of Gauss we will use 4 blocks in the j dimension and submit the execution of the binary using the submit-omp.sh script to validate the parallelisation



Upon analyzing the scalability plots, it becomes apparent that the results are not as favorable as those observed in the optimized Jacobi version. In the second graph, although the speed-up does not exhibit a relatively linear trend, it falls short of the ideal speed-up. However, it is noticeably better than the Jacobi version without the parallelization of the "copy_mat" function. This discrepancy in performance could indeed be influenced by the characteristics and impact of the "copy_mat" function.

Figure 28: Scalability plot of the Gauss-Seidel version.

Now we will modify our code accordingly to make use of the value and change the number of blocks in the j dimension without recompiling the code for each new value to test. Using the value in userparam directly as the number of blocks (i.e. nblocksj=userparam). Compile and execute with different values and check that it has the expected results and effect.

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

    int nblocksi=omp_get_max_threads();
    int nblocksj = userparam;

    int mat[24] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    #pragma omp parallel private(diff) reduction(+:sum)
    {
      int cont;
      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((u==unew) && blocki !=0){
```

Figure 30: Modified code of solver-omp-gauss.c (solver-omp-gauss-userparam.c)

After that, to explore the range of values of blocks in j, we have submitted the script submit-userparam-omp.sh with 16 threads:

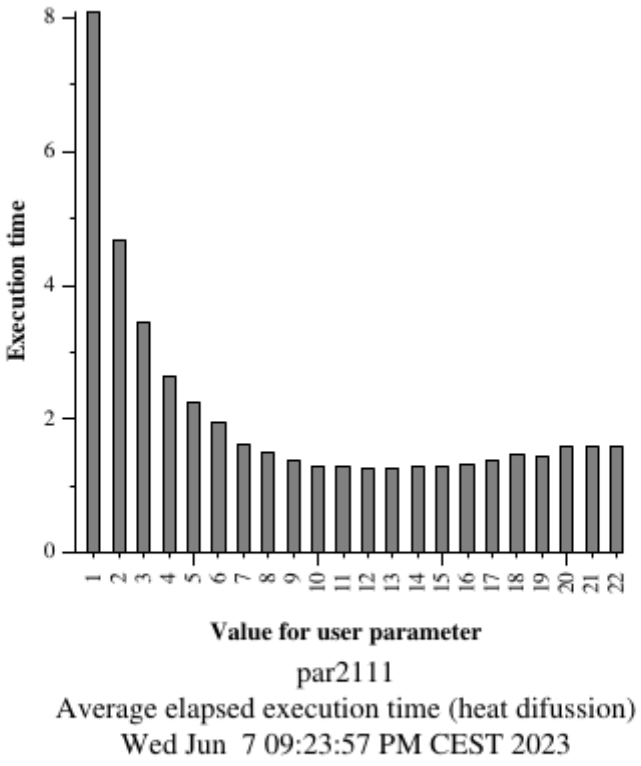
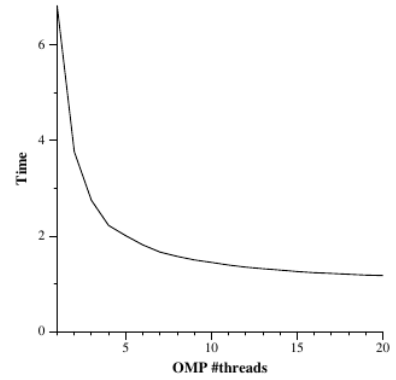
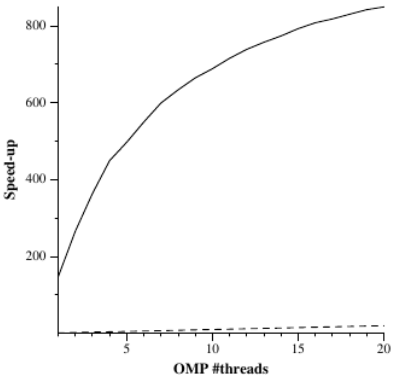


Figure 31: Graph of user parameter values in Gauss-Seidel with 16 threads

Analysing the previous plot, we have expected that the best performance is with userparam = 10. So let's see how is the scalability plot with value 10 in nblockj.





As anticipated, we have generated the user parameter plots with varying numbers of threads, as shown in Figure 31. From these plots, we can observe that increasing the user parameter (number of blocks) while using fewer threads leads to improved results. This finding supports our previous explanation regarding the benefits of blocking and synchronization.

Surprisingly, even though we expected that more blocks/tasks would result in increased synchronization overhead and potentially worse performance, the implemented data decomposition strategy successfully mitigates this overhead. By leveraging the principle of locality, the program manages to capitalize on the gains from increased blocks, outweighing the synchronization overhead.

Figure 32:Scalability plot with userparam 10

However, it is important to note that the synchronization overhead becomes noticeable when 16 threads are utilized. In such cases,comparing the results obtained with nblockj = 4 and nblockj = 10, we see a little performance even though it is not that much but we need to remember it is crucial to select the appropriate granularity of the user parameter. This choice helps strike a balance between maximizing parallelism and minimizing the impact of synchronization overhead.

In conclusion, the results confirm the effectiveness of the blocking and synchronization techniques implemented in the program. By appropriately selecting the number of blocks and threads, we achieve improved performance and leverage the principle of locality to reduce synchronization overhead.

1.3 CONCLUSIONS

In this laboratory assignment, our focus was on exploring the data decomposition strategy using two different methods: Jacobi and Gauss-Seidel, for the heat diffusion program.

We began by analyzing the program of both methods using Tareador to understand their dependencies. For the parallelization of the Jacobi version, we initially parallelized only the solve function and examined the results. However, we discovered that the outcomes were not as expected, leading us to realize that there was another part that could be parallelized. As a result, we parallelized the copy_mat function and re-evaluated the results. This time, we observed a significant improvement, with a noticeable difference from the previous version. Additionally, we implemented an explicit version of the Jacobi method and found that although the execution time was similar, the explicit version had higher overheads and a less efficient parallelization strategy compared to the implicit version.

Moving on to the parallelization of the Gauss-Seidel version, we protected the dependencies by writing to the same matrix and carefully analyzed the dependent variables identified by Tareador. However, we found that the scalability of the Gauss-Seidel method was not as good as the Jacobi version. We further studied the optimal value for blocking and analyzed how the data decomposition strategy, number of threads, and number of blocks influenced the performance of Gauss-Seidel's method. We determined the best value for the user parameter (userparam) and generated scalability plots to showcase the improvements achieved. While the new scalability of the Gauss-Seidel version approached that of the improved Jacobi version, the elapsed time in the Jacobi method was still faster than Gauss-Seidel. Unfortunately, we were unable to solve the Gauss-Seidel version using explicit tasks, so a comparison to the implicit version was not possible.

In summary, both the Jacobi and Gauss-Seidel versions have their own advantages and disadvantages, and neither one is definitively superior to the other. The effectiveness of each method depends on factors such as the specific problem, the available hardware resources, and the desired performance criteria.

1.4 FINAL SURVEY

• Modelfactors: 10. The Modelfactors tool is highly valuable and provides clear and comprehensive information through its tables. It allows for a detailed analysis of the program, and in my opinion, it is sometimes easier to understand than Paraver due to its clear presentation of the number and percentage of different elements. I would highly recommend its use for future editions.
• Tareador: 9. Tareador greatly assists in visualizing task dependencies and granularity, and it is user-friendly and straightforward to utilize.
• Extrae + Paraver: 6. The timeline window and other features of Extrae and Paraver have proven useful for understanding and analyzing program performance. However, there are some drawbacks such as long loading times, especially on non-university computers. Additionally, occasional virtual machine crashes while attempting to open it can be frustrating.