PAR Laboratory Assignment

Lab5: Geometric (data) decomposition using implicit tasks: heat diffusion equation

Sergio Araguás Villas - par4107
David Pérez Barroso - par4121
30-12-2022
22-23 Q1

Index

Sequential heat diffusion program and analysis with Tareador

In this final laboratory assignment we will work on the parallelisation of a sequential code that simulates the diffusion of heat in a solid body using two different solvers for the heat equation (*Jacobi* and *GaussSeidel*).

We started the assignment by compiling the heat.c file and then executing it with the two solvers displaying their results, the next section shows the results obtained.

Jacobi execution

```
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: 4.358
Flops and Flops per second: (11.182 GFlop => 2565.93 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Fig.1. Display of heat.c with Jacobi

Gauss-Seidel execution

```
Iterations       : 25000
Resolution       : 254
Residual         : 0.000050
Solver           : 1 (Gauss-Seidel)
Num. Heat sources : 2
  1: (0.00, 0.00) 1.00 2.50
  2: (0.50, 1.00) 1.00 2.50
Time: 8.817
Flops and Flops per second: (8.806 GFlop => 998.84 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

Fig.2. Display of heat.c with Jacobi

If we compare both images (Fig.1. and Fig.2.) we can observe that there are slight differences between them. The cold zone is more cropped in the Gauss-Seidel execution than in the Jacobi and the lower hot zone seems to be more pronounced. We will use these images to check the future versions of the program.

Now we execute the Tareador script to visualize the dependencies graph of the two solvers executions, obtaining the below graphs. We note that the execution of tasks is almost sequential and a more parallel execution at this granularity level can't be achieved if we want to augment the parallelization of the code we should use a finer granularity .

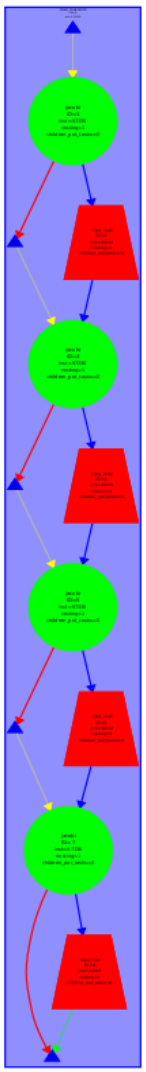

Fig.3. Dependency graph of Jacobi

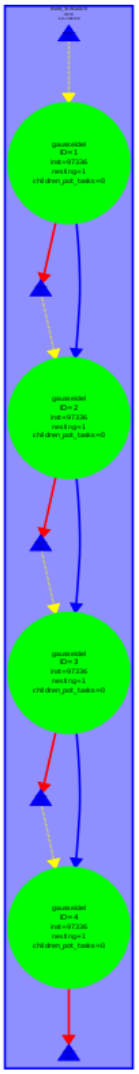

Fig.4. Dependency graph of Gauss-Seidel

So, we have to explore a finer granularity; the granularity that is wanted to achieve is a task per block. But before that, we have to find which variable is causing the sequentialization on the dependencies graph. We found it using the *DataView* option that *Tareador* offers over the nodes in graphs, observing that the variable *sum* had the fault of it.

To continue, we modify the code of *solver-tareador.c* file, adding tasks in the second loop of the *solve* function to achieve the new level of granularity and also uncomment the *tareador_disable_object* and *tareador_enable_object* directives to filter the dependences caused by the variable sum. The following image shows the changes in the code:

```
// 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) {
        tareador_start_task("inner solve");
        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;
          }
        }
        tareador_end_task("inner solve");
      }
    }
    tareador_enable_object(&sum);

    return sum;
}
```

Fig.5. Code changes for the new granularity level

Once the changes are done, we compile and execute Tareador with the two solvers (Jacobi and Gauss-Seidel) obtaining the next dependencies graphs (Fig.6. and Fig.7.):

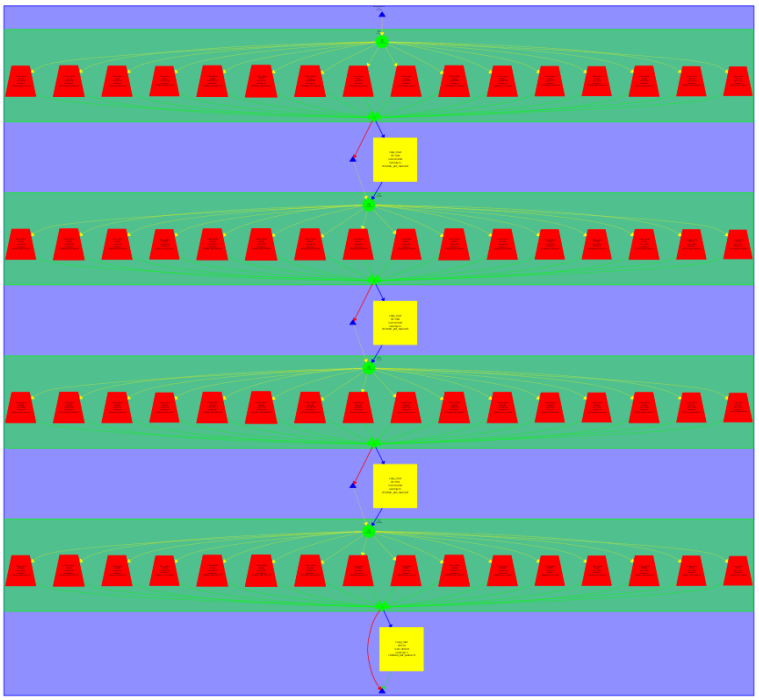

Fig.6. Jacobi solver task dependence graph

As we can see in the two dependencies graphs of the two solvers, we obtain a better level of parallelism than the other version.

To avoid the data race and to protect the sum variable, in the OpenMp implementation we can protect it with one of the following directives: *#pragma omp atomic*, *#pragma omp critical* or with *#pragma omp reduction*, choosing the last directive because is simpler to understand and better in performance as you don't pay the synchronization cost.

Finally, if we simulate the execution with 4 threads in Paraver we obtain the following traces results:

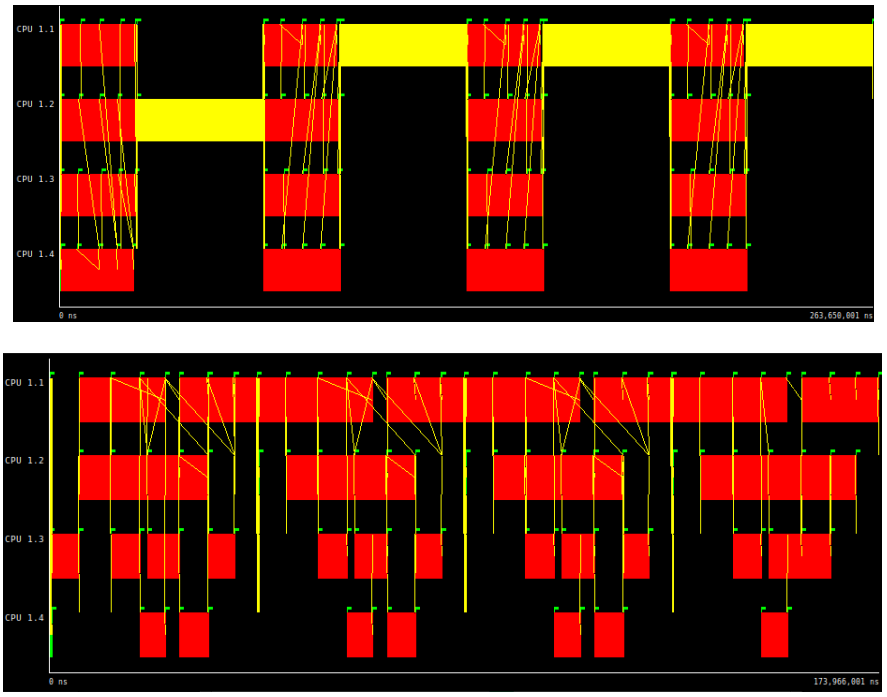

Fig.8. Jacobi (up) and Gauss-Seidel (bottom) trace with 4 cores in Paraver

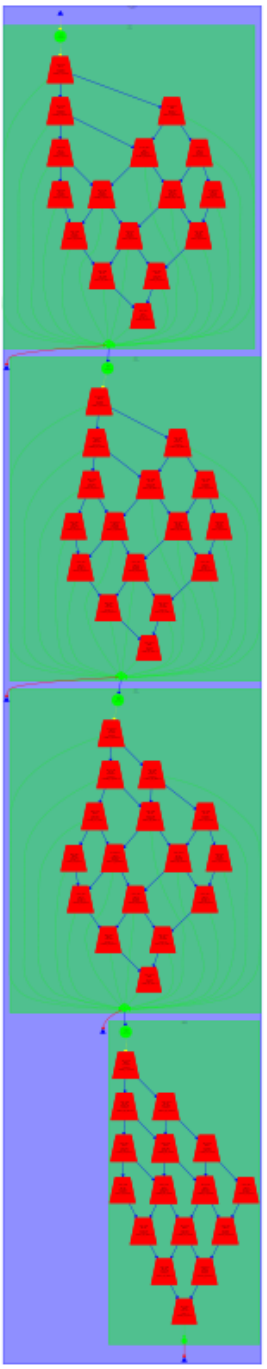

Fig.7. Gauss-Seidel solver task dependence graph

We can see that the parallelization of the *inner solve* task (red part) is the most notable in the traces and that the *copy_mat* task (yellow part) is causing the Jacobi version to be less efficient than the Gauss Seidel.

We think that this can be solved parallelizing the *copy_mat* task, creating an *inner copy_mat* tasks, obtaining the following results:

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

    int nblocksi=4;
    int nblocksj=4;

    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) {
        tareador_start_task("inner copy_mat");
        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];
          }
        }
        tareador_end_task("inner copy_mat");
      }
    }
}
```

Fig.8. copy_mat parallelized code

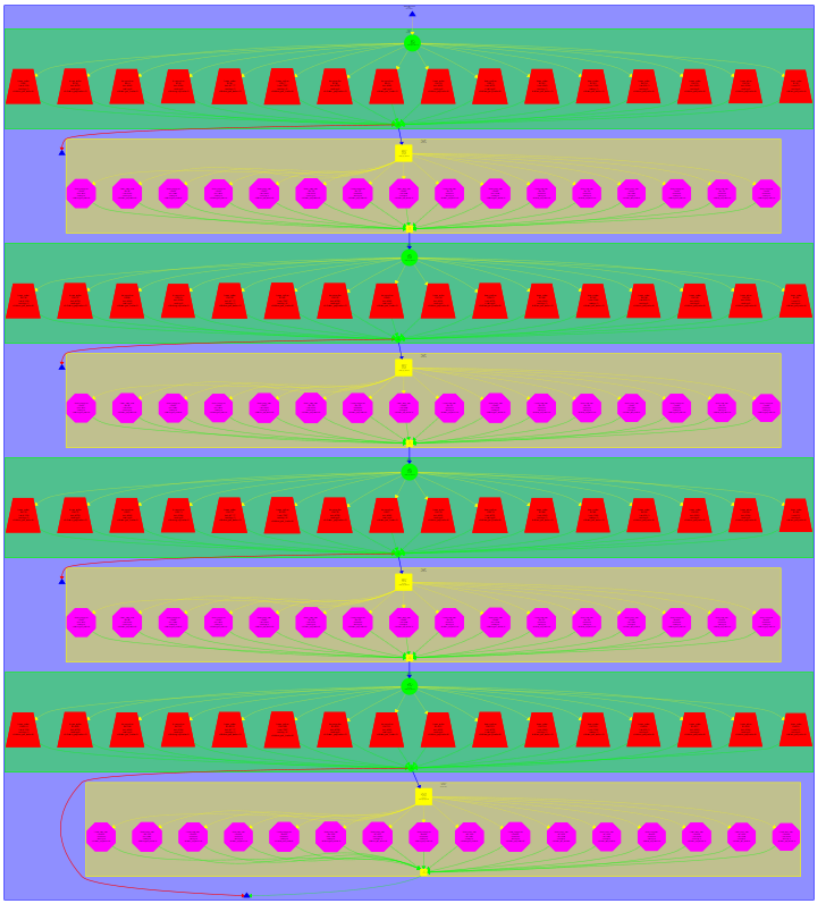

Fig.9. Jacobi solver task dependence graph with copy_mat paralelization

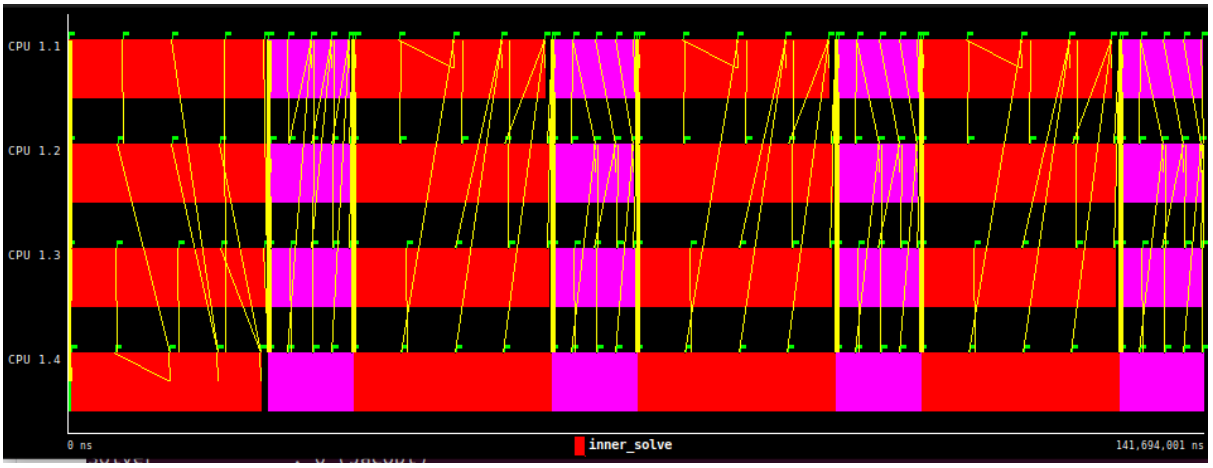

Fig.10. Jacobi trace with 4 threads with copy_mat parallelization

As it can be observed in Fig.9 the *copy_mat* task is parallelized obtaining the purple tasks instead of one yellow task as the previous version. If we observe the Fig.10 that is the trace obtained with the simulation with 4 threads, we can see that now the *copy_mat* is parallelized among the four threads obtaining a better performance and a better execution time than in the previous version Fig.8.

On the other hand the Gauss-Seidel solver isn't optimized as it doesn´t use the *copy_mat* function, so the traces and graph obtained from it are the same as before.

Parallelisation of the heat equation solvers

Jacobi solver

In this section we will first parallelise the sequential code for the heat equation code using the Jacobi solver with the implicit tasks generated in *#pragma omp parallel* , following a geometric block data decomposition by rows.

As it can be seen in the image below Fig.11., we added the directive *private(tmp, diff)* and the directive *reduction(+: sum)* to guarantee the dependencies discovered before.

```
#pragma omp parallel private(tmp, diff) reduction(+:sum) // complete data sharing constructs here
{
  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;
```

Fig.11. Jacobi dependencies implementation

Once we compiled, we submit the program with 1 thread and with 8 threads and observe their execution time obtaining the next results:

1 Thread

par4121@boada-8:~/lab5$ cat heat-omp-jacobi-1-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: 2.300
Flops and Flops per second: (11.182 GFlop => 4861.12 MFlop/s)
Convergence to residual=0.000050: 15756 iterations

8 Threads

par4121@boada-8:~/lab5$ cat heat-omp-jacobi-8-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: 2.242
Flops and Flops per second: (11.182 GFlop => 4986.39 MFlop/s)
Convergence to residual=0.000050: 15756 iterations

As it can be observed from the text above, we obtain a slightly smaller execution time with 8 threads. Also we compare it the images obtained with the sequential version, proving that the program works correctly as it can be seen in the next images:

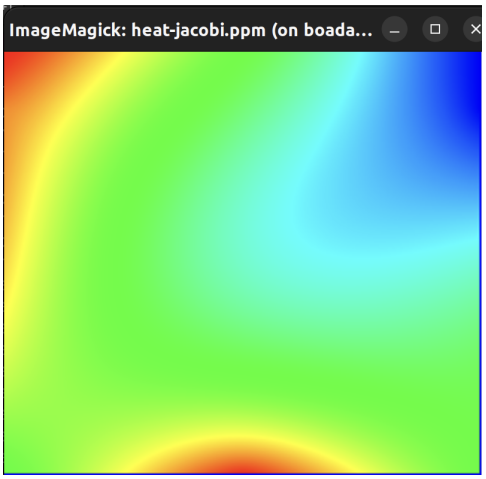

Fig.12. Heat display Jacobi sequential

Fig.13. Heat display Jacobi parallel

The next step is executing the modelfactors analysis with the *submit-strong-extrae.sh* script that execute the program with 1, 4, 8 and 16 threads, obtaining the following results:

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.85	2.14	2.06	2.10
Speedup	1.00	1.33	1.38	1.36
Efficiency	1.00	0.33	0.17	0.09

Table 1: Analysis done on Tue Dec 27 04:59:54 PM CET 2022, par4121

Overview of the Efficiency metrics in parallel fraction, ϕ=64.63%				
Number of processors	1	4	8	16
Global efficiency	99.63%	73.61%	62.05%	33.17%
Parallelization strategy efficiency	99.63%	83.43%	98.12%	97.91%
Load balancing	100.00%	86.25%	99.94%	99.86%
In execution efficiency	99.63%	96.73%	98.18%	98.05%
Scalability for computation tasks	100.00%	88.23%	63.23%	33.88%
IPC scalability	100.00%	89.60%	72.52%	45.74%
Instruction scalability	100.00%	99.97%	94.50%	82.79%
Frequency scalability	100.00%	98.50%	92.27%	89.46%

Table 2: Analysis done on Tue Dec 27 04:59:54 PM CET 2022, par4121

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)	1836.81	520.46	363.09	338.85
Load balancing for implicit tasks	1.0	0.86	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	6.88	176.62	6.87	7.39

Table 3: Analysis done on Tue Dec 27 04:59:54 PM CET 2022, par4121

Fig.14. Modefactor tables of Jacobi solver

As it can be observed from the tables, the speed up increases with the number of threads, the opposite happens with the global efficiency that highly decreases as more threads are added.
We observe in the second and third table that it is only parallelized the 64.63% of the code (we think that this should be bigger) and that the scalability for the computation tasks highly decreases.

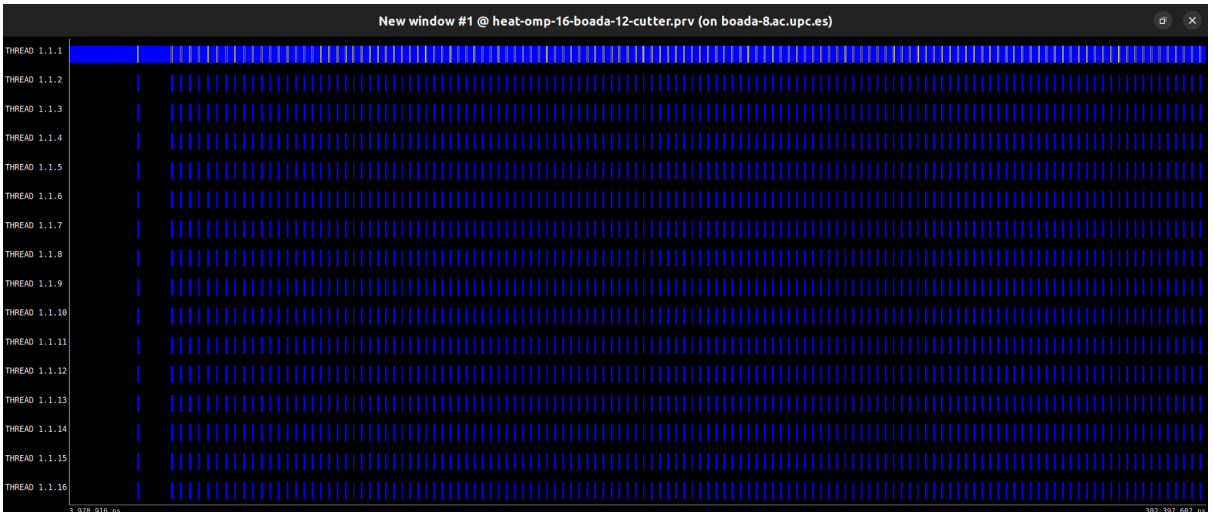

Fig.15. Time trace of jacobi execution with 16 threads

We executed the paraver analysis obtaining the time trace above (Fig.15.). We can see that all the threads are executing a task, but there are many black holes in the blue traces, this is because the threads are waiting for the first thread to finish the non-parallelized parts of the program.

This part is the function copy-mat, that in this version is not parallelized causing this lack of efficiency and the low value of the parallelization percentage. Parallelizing this function will reduce the black holes that we found on the trace among all the threads.

Optimization

The following image show the changes that we made in the code of *copy_mat* function to parallelize it:

```
// 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];
      }
   }

}
```

Fig.16. Code of the copy_mat function parallelized

To do that we first save in *nblocksi* the number of threads that allow us to send it to the *lowerb* and *upperb* functions. Finally with the *#pragma omp parallel directive*, we parallelized the code inside its brackets that will be executed by different threads.

Once implemented the new version, we executed the code with 16 threads, obtaining a better execution time than with the other version. as the following result shows:

```
par4121@boada-6:~/lab5$ cat heat-omp-jacobi-16-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: 0.342
Flops and Flops per second: (11.182 GFlop => 32660.22 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Now we executed the modelfactor analysis to compare and analyze the new version of the code.

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.87	0.70	0.41	0.23
Speedup	1.00	4.10	6.96	12.39
Efficiency	1.00	1.03	0.87	0.77

Table 1: Analysis done on Thu Dec 29 11:49:47 AM CET 2022, par4121

Overview of the Efficiency metrics in parallel fraction, ϕ=98.73%				
Number of processors	1	4	8	16
Global efficiency	99.68%	106.30%	93.76%	90.35%
Parallelization strategy efficiency	99.68%	97.10%	95.20%	92.43%
Load balancing	100.00%	98.66%	97.62%	97.25%
In execution efficiency	99.68%	98.41%	97.53%	95.04%
Scalability for computation tasks	100.00%	109.48%	98.48%	97.75%
IPC scalability	100.00%	110.15%	107.88%	113.00%
Instruction scalability	100.00%	99.95%	98.70%	97.03%
Frequency scalability	100.00%	99.44%	92.49%	89.15%

Table 2: Analysis done on Thu Dec 29 11:49:47 AM CET 2022, par4121

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)	1410.9	322.19	179.08	90.21
Load balancing for implicit tasks	1.0	0.99	0.98	0.97
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	4.5	8.18	10.79	9.74

Table 3: Analysis done on Thu Dec 29 11:49:47 AM CET 2022, par4121

Fig.17. Modelfactor analysis of the copy_mat version

As it can be seen in the first table, in this version the execution time is highly decreased as more threads are added obtaining better results than the other version. The same occurs with the speed-up that is better in this version than in the previous, here it increases mucho more as more threads are added. The efficiency, although it is slightly reduced, is better than in the version where copy_mat isn't parallelized.

If we observe the other tables, we can see that the parallelized part of the code has positively increased, explaining why this version has a better parallel performance than the previous version.

Finally if we take a look at the scalability values and its plots we can observe that the new version is much better scalable than the previous version, achieving close to the perfect line of scalability as it can be seen in the plot below.

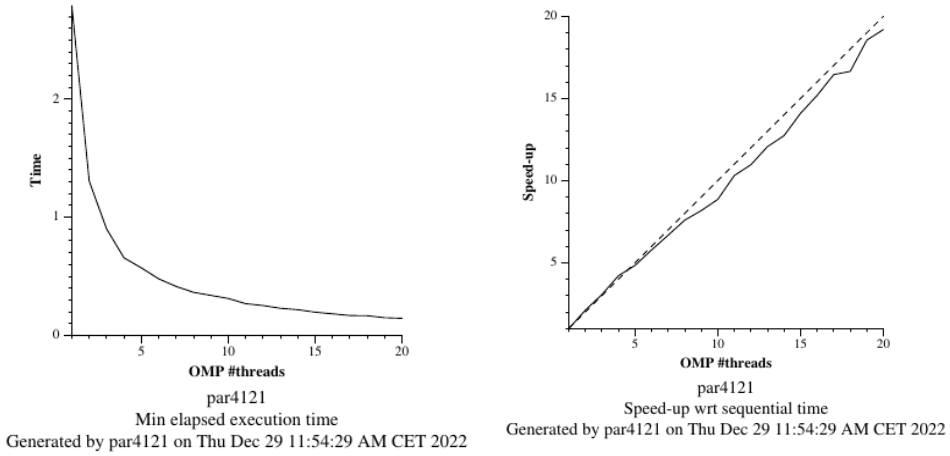

Fig.18. Plots of the copy_mat version

Finally we executed the paraver analysis obtaining the time trace of the Fig.19.

Fig.19. Time trace of the copy_mat version

If we compare the trace of the two versions, we can observe that the trace of the new version has less black holes among the threads, meaning that the waiting time to thread one of the threads is decreased due the parallelization of the copy_mat function, obtaining a better performance and a better parallelization than in the previous version, also although the solve execution time is the same we obtaining a better execution time due the copy of the matrix is now parallelized obtaining a better total execution time.

Gauss-Seidel

Now we will try to parallelize this code considering the dependences of the Gauss seidel approach. The parallelization strategy will consist of block by row data decomposition with the use of implicit tasks in parallel regions. We have also taken into account the annex of the lab to resolve memory consistency problems, leaving us with this code.

```
// 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=nblocksi;
    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) private(tmp) 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;
}
```

Fig.20. Gauss-seidel dependencies implementation

We have started the parallelization like in the Jacobi's code and used atomic directives to preserve memory consistency. We have added a vector mat[] to keep track of the calculations, then we have added the expression *u == unew* to assure the Gauss-Seidel solver has been chosen and *blocki* != 0 to assure we aren't in block 0 (because this would create a vector index out of bound error later). If the conditions are fulfilled the code below will make the program wait until all dependencies are calculated in the *while(cont <= blockj)*, if all dependencies are calculated we can freely calculate the current position. Then we update the mat position to mark that it has been calculated.

Once the code is completed we compile and execute the program with 1 and 8 threads.

1 Thread
Iterations : 25000
Resolution : 254
Residual : 0.000050
Solver : 1 (Gauss-Seidel)
Num. Heat sources : 2
1: (0.00, 0.00) 1.00 2.50
2: (0.50, 1.00) 1.00 2.50
Time: 5.176
Flops and Flops per second: (8.806 GFlop => 1701.32 MFlop/s)
Convergence to residual=0.000050: 12409 iterations

8 Threads
Iterations : 25000
Resolution : 254
Residual : 0.000050
Solver : 1 (Gauss-Seidel)
Num. Heat sources : 2
1: (0.00, 0.00) 1.00 2.50
2: (0.50, 1.00) 1.00 2.50
Time: 1.487
Flops and Flops per second: (8.806 GFlop => 5921.47 MFlop/s)
Convergence to residual=0.000050: 12409 iterations

As expected the time with 8 threads is smaller than the 1 thread execution, and if we compare the resulting images (Fig.21 and 22) of the execution we can assure they do the same work as the images look the same.

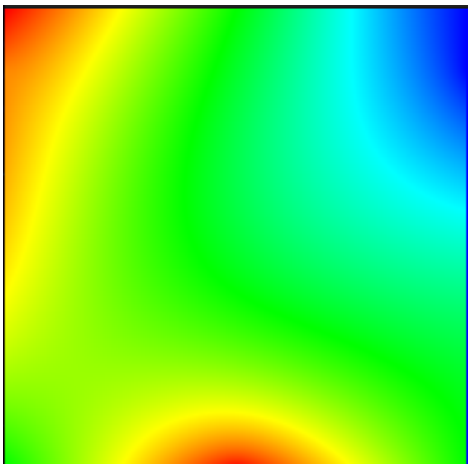

Fig.21. Heat display Gauss-seidel parallelized with 1 thread

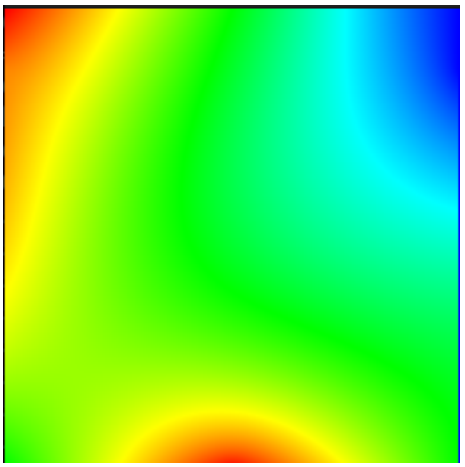

Fig.22. Heat display Gauss-seidel parallelized with 8 threads

In the same way we have done with Jacobi solver, now we generate the modelfactor tables:

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	6.92	3.10	1.77	0.98
Speedup	1.00	2.23	3.91	7.05
Efficiency	1.00	0.56	0.49	0.44

Table 1: Analysis done on Tue Dec 27 05:04:38 PM CET 2022, par4107

Overview of the Efficiency metrics in parallel fraction, ϕ=99.09%				
Number of processors	1	4	8	16
Global efficiency	99.96%	55.90%	49.39%	45.22%
Parallelization strategy efficiency	99.96%	78.49%	99.73%	99.46%
Load balancing	100.00%	78.59%	99.98%	99.93%
In execution efficiency	99.96%	99.87%	99.75%	99.52%
Scalability for computation tasks	100.00%	71.22%	49.53%	45.47%
IPC scalability	100.00%	78.36%	76.63%	73.76%
Instruction scalability	100.00%	92.08%	70.22%	68.66%
Frequency scalability	100.00%	98.71%	92.04%	89.79%

Table 2: Analysis done on Tue Dec 27 05:04:38 PM CET 2022, par4107

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)	6858.08	2407.53	1730.91	942.63
Load balancing for implicit tasks	1.0	0.79	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	2.76	1320.14	4.75	5.4

Table 3: Analysis done on Tue Dec 27 05:04:38 PM CET 2022, par4107

Fig.23. Modelfactor tables

In the tables above, we can see that the major problem of this parallelization is its scalability, given that as we add more threads the Global efficiency decreases.

The following plots (Fig.24.) show more graphically this conclusion, where we see that the speedup is far from optimal as it is separated from the ideal speedup.

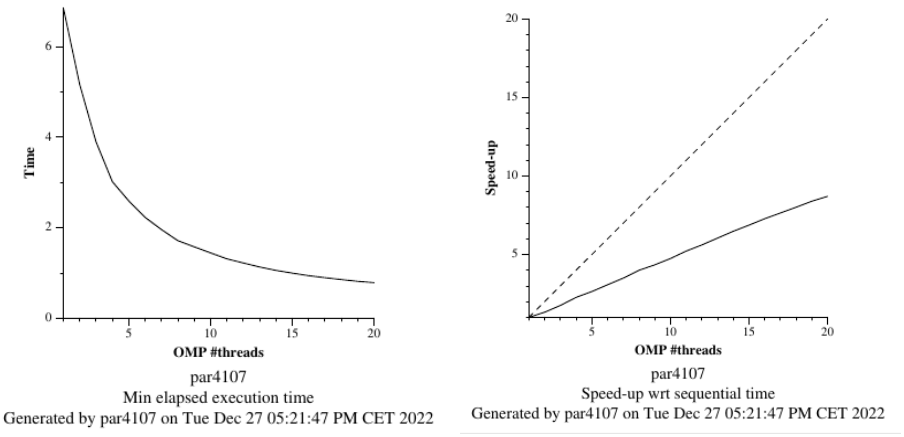

Fig.24. Scalability plots using Gauss-Seidel solver

If we take a look at the timeline generated using 16 threads (Fig.25.) we can see that thread 1 is the thread dedicated to the creation of tasks (yellow) and we can also note that threads aren't waiting (black spaces) like in Jacobi solver before the optimization (Fig.15) because it doesn´t depend on the *copy_mat* function.

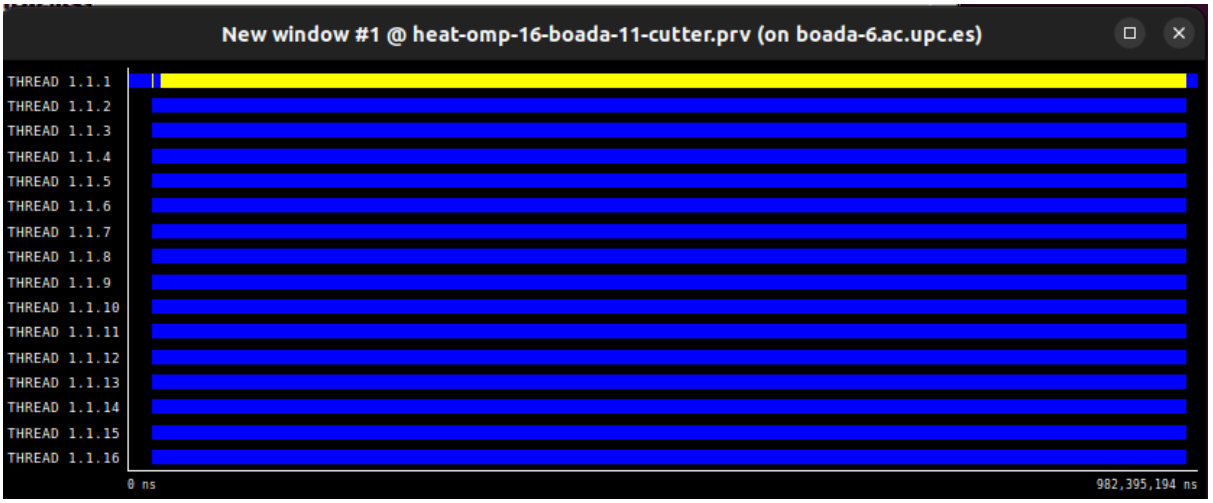

Fig.25. 16 threads trace timeline Gauss-Seidel solver

Optimization

We can explore a range of values of the number of blocks in order to improve the parallelism of the code by assigning *userparam* to the number of blocksj like this:

```
// 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};
```

Fig.26. Code with nblocksj upgrade

If we execute **./submit-userparam-omp.sh** we can see in Fig.27. that the execution time decreases as the value used as parameter is higher. This is because the parameter value increases the number of blocksj, creating more loops that can be parallelized. We can conclude that, with the information of this plot, the optimal value for 16 threads is 12.

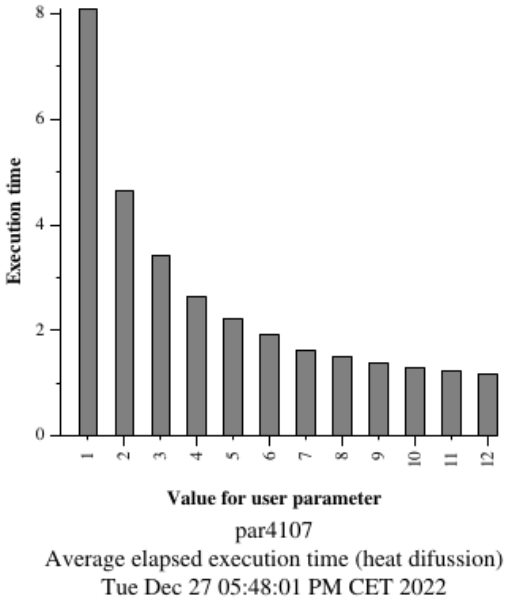

Fig.27. Execution time for parameter value

Said that, if we take a look at the scalability plots of the code with 12 as the value for *nblocksj* (*Fig.28.*), we can see that the time compared to the plot of the previous version (Fig.24.) is less with fewer threads but higher with more threads. We also see that the speedup is higher with less threads compared to the previous versions but less with a high number of threads.

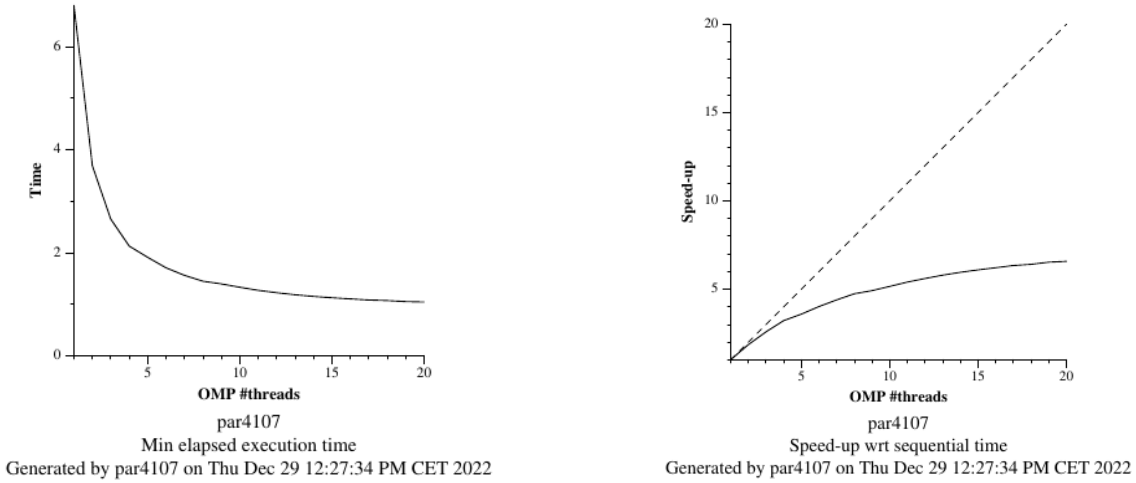

Fig.28. Scalability plots for Gauss-Seidel solver with 12 blocksj

Conclusions

We have simulated the heat diffusion in a solid body using 2 equations solvers: Jacobi and Gauss-Seidel.

At first sight the Gauss-Seidel solver had more potential for parallelization than the Jacobi solver, because it did not depend on the copy_mat function as it can be seen in Fig.8.

But, giving that, we also parallelized the copy_mat function to achieve a better performance and parallelization in the Jacobi solver. With the results obtained, we come to the conclusion that the Jacobi solver has higher potential for parallelization than the Gauss-Seidel solver, as the scalability of the Jacobi solver is nearly ideal (Fig.18.) and the Gauss-Seidel is far from optimal as it can be seen in Fig.28.