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

Rubén Dabrio (boada: 4309)
Sergi Campuzano (boada: 4305)

June 5, 2023

1

Laboratory 5 notebook

1.1 Sequential heat diffusion program and analysis with Tareador

You must include:

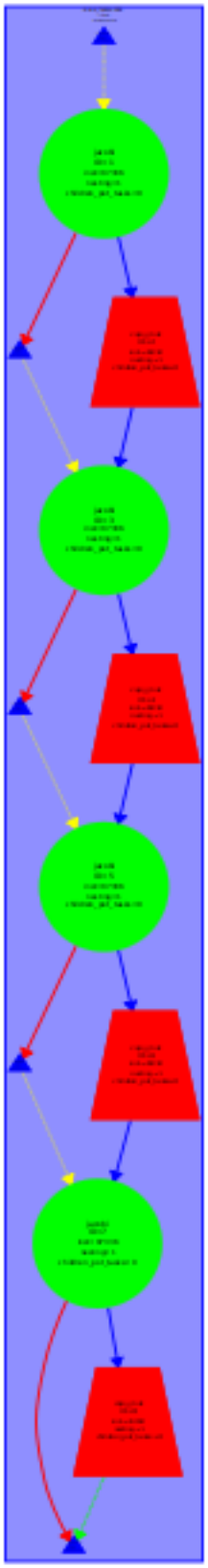
- The task dependency graph shown by *Tareador* for both solvers *Jacobi* and *Gauss-Seidel* for the default codes (naive version).
- The task dependency graph shown by *Tareador* for both solvers for the proposed task granularity, adding the dependences filter and new tasks per block in other parts of the code to increase parallelism.
- The excerpt of the last version of the *Tareador* code that you have modified in order to specify **one task per block** and exploit other parts of the code in addition to `solver` function.

Comments/Observations

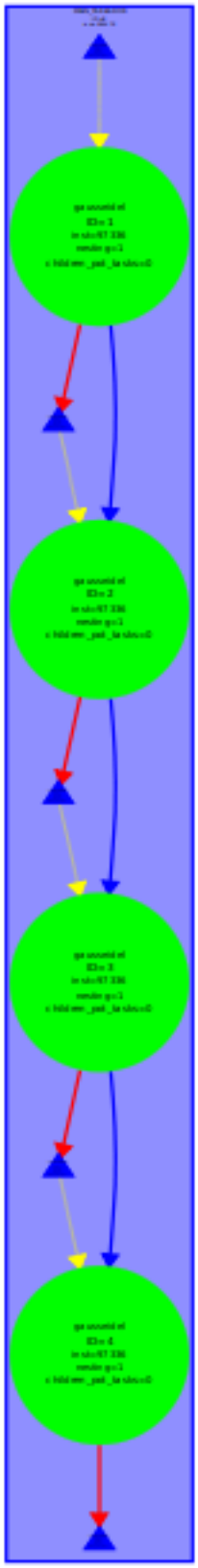
For the naive version: Is there any parallelism that can be exploited at the naive version granularity? For the proposed task granularity: Which variable was causing the serialisation of all the tasks? How will you protect the access to this variable in your OpenMP implementation? Are you obtaining more parallelism in the proposed task granularity than in the default version? Is the parallelism achieved the same for *Jacobi* and *Gauss-Seidel* solvers? What was the part of the code that you have parallelised?.

On the naive version we can see that all is practically secuential.
With Jacobi the only ridiculous paralellism we have is between the red and blue tasks. The green tasks have to wait for the two previous tasks before start and the red and blue have to wait the green task, so it is all practically secuential. With Gauss all is secuential.
For the proposed task granularity, the variable which is causing the serialization is "sum" and we will have to protect it with a reduction (which will acumulate the sum of all the tasks at the end of the execution of this internal loop execution tasks.
With the Jacobi version we have more parallelism as we can do the red tasks at the same time. We have to wait for all the firsts red tasks for executing the yellow one but then we can start again executing all the nexts red tasks.
With the Gauss version we have also increased the parallelism, it is not as we have obtain in the Jacobi version but the parallelism is better than what we had in the first Gauss version as we can start more red tasks than in the first version.
We have paralellized the solve function of the solver-tareador.c. We have made that each internal loop(the loop j) is a task.

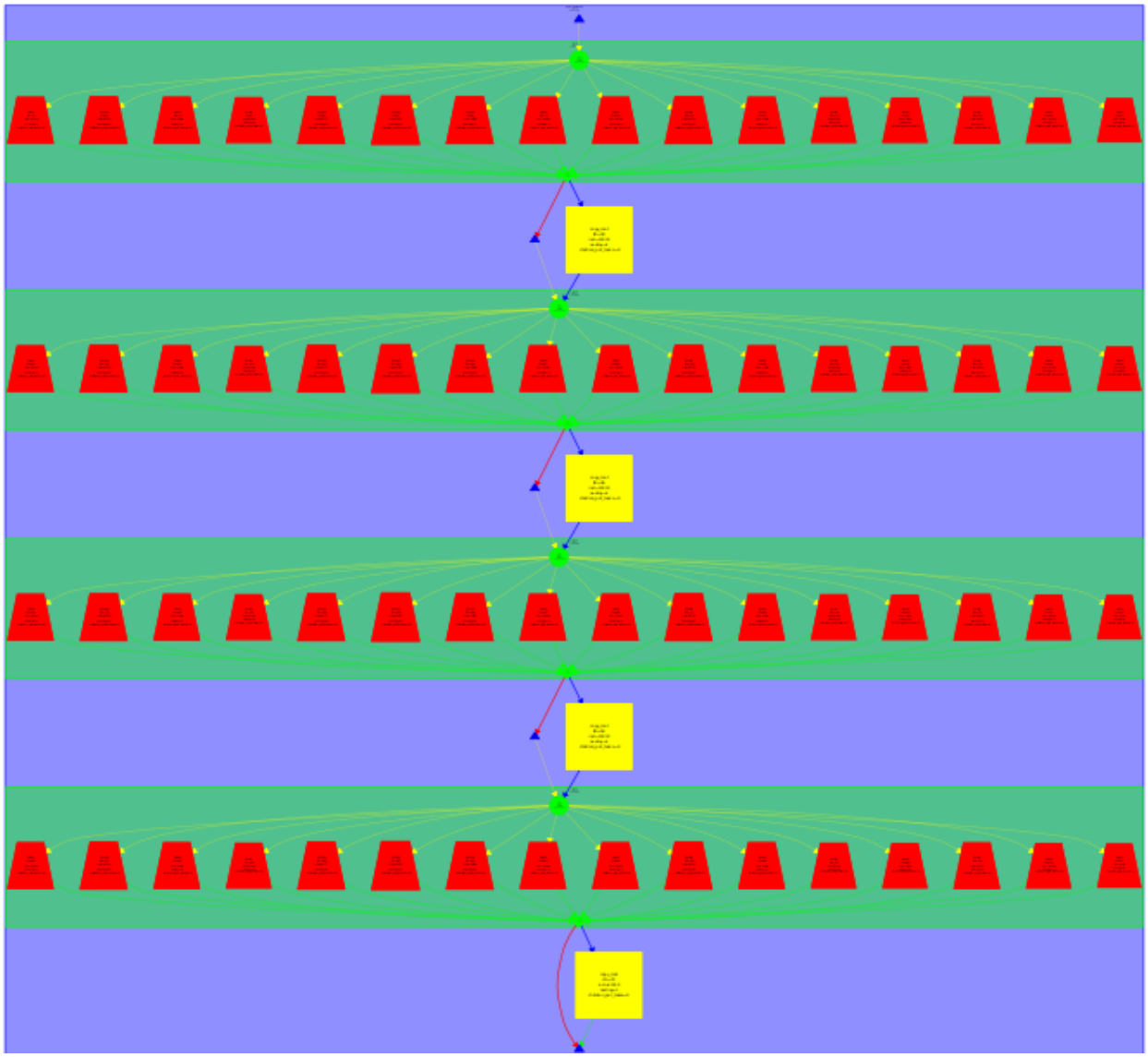
JacobiNaiveVersion:



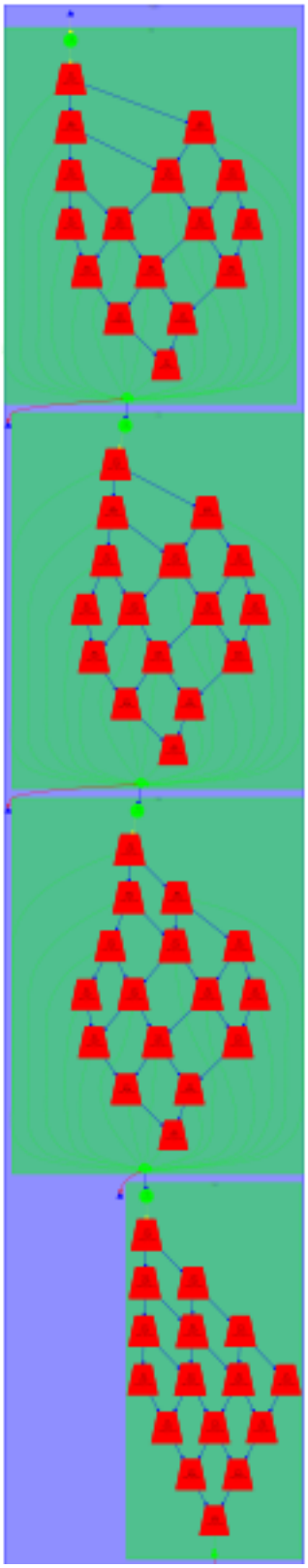
GaussNaiveVersion:



Jacobi version2:



Gauss version2:



```
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("block");
        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("block");
      }
    }
    //tareador_enable_object(&sum);

    return sum;
}
```

1.2 OpenMP parallelization and execution analysis: *Jacobi*

You must include:

- **An excerpt of the code to show the OpenMP annotations you have added to the code after the optimizations (Optimization section - Jacobi solver).**
- **The *Modelfactor* tables and the plot of scalability for the first implementation (with some speedup - Overall Analysis section - Jacobi solver) and your last optimized implementation.**
- **Captures of the window timelines for both first (with some speedup - Detailed Analysis section - Jacobi solver) and last implementation.**

Comments/Observations

What is the syncrhonization mechanism you have used to allow an efficient data sharing (speedup larger than 1 - Overall Analysis section)? What was the region of the code that was provoking the low value for the *parallel fraction* in your first parallelisation? (Detailed Analysis section) Compare the *parallel fraction* of your first and last versions (Overall Analysis of the Optimized Code section). Is the execution time reduced from your first to the last version?. Have you increased the scalability? (Overall Analysis of the Optimized Code section) Compare the timelines of both executions under the point of view of instantaneous parallelism (Detailed Analysis of the Optimized Code section).

We have used a mechanism of reduction(+:sum) to allow an efficient data sharing. We have also add "tmp" and "diff" as a private variables. In the first version we had a parallel fraction of 65.24 percent and in the last version we have improved the parallel fraction to 98.80 percent. What we have basically changed between the versions is the code of the function "copymat" in the "solver.omp" document. The execution time is reduced from our first version to the last version. The minimum elapsed time is reduced considerably when we have more than 1 thread. For example, with 8 threads we had 2.05 and now 0.41. We have also increased the scalability when we have more than 1 thread. For example for 8 threads we had 62 percent of scalability and now 96 percent. With the instantaneous parallelism we can observe that with the first version we are a lot of time executing and in the last version we only see a white bar which means that the execution finish earlier.

Initial parallelism:

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.83	2.15	2.05	2.16
Speedup	1.00	1.32	1.38	1.31
Efficiency	1.00	0.33	0.17	0.08

Table 1: Analysis done on Thu May 25 10:47:34 AM CEST 2023, par4309

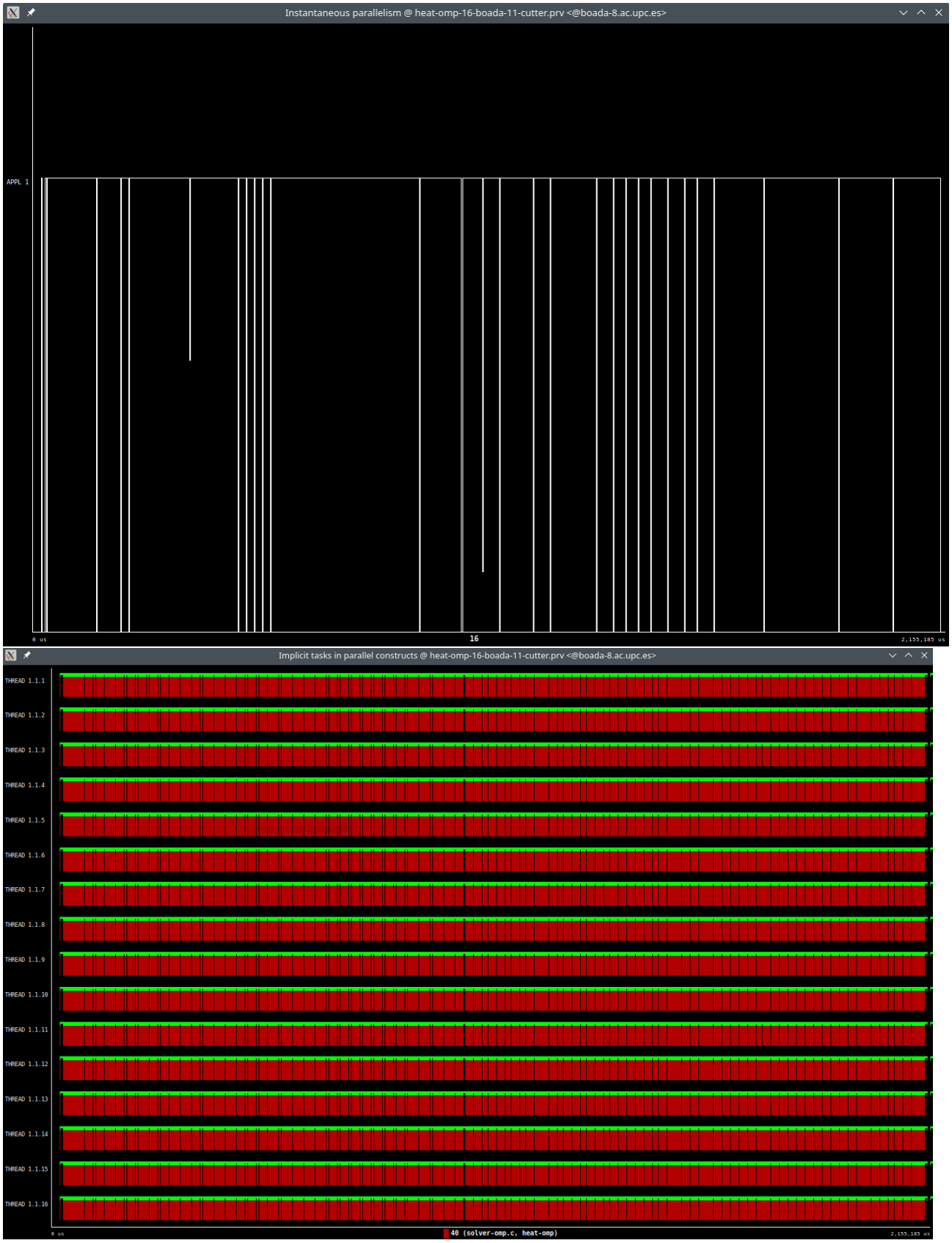
Overview of the Efficiency metrics in parallel fraction, ϕ=65.24%				
Number of processors	1	4	8	16
Global efficiency	99.65%	72.70%	61.25%	34.18%
Parallelization strategy efficiency	99.65%	82.42%	98.08%	97.81%
Load balancing	100.00%	85.41%	99.90%	99.85%
In execution efficiency	99.65%	96.51%	98.17%	97.97%
Scalability for computation tasks	100.00%	88.20%	62.45%	34.94%
IPC scalability	100.00%	88.56%	72.02%	46.54%
Instruction scalability	100.00%	99.97%	93.79%	83.66%
Frequency scalability	100.00%	99.63%	92.46%	89.74%

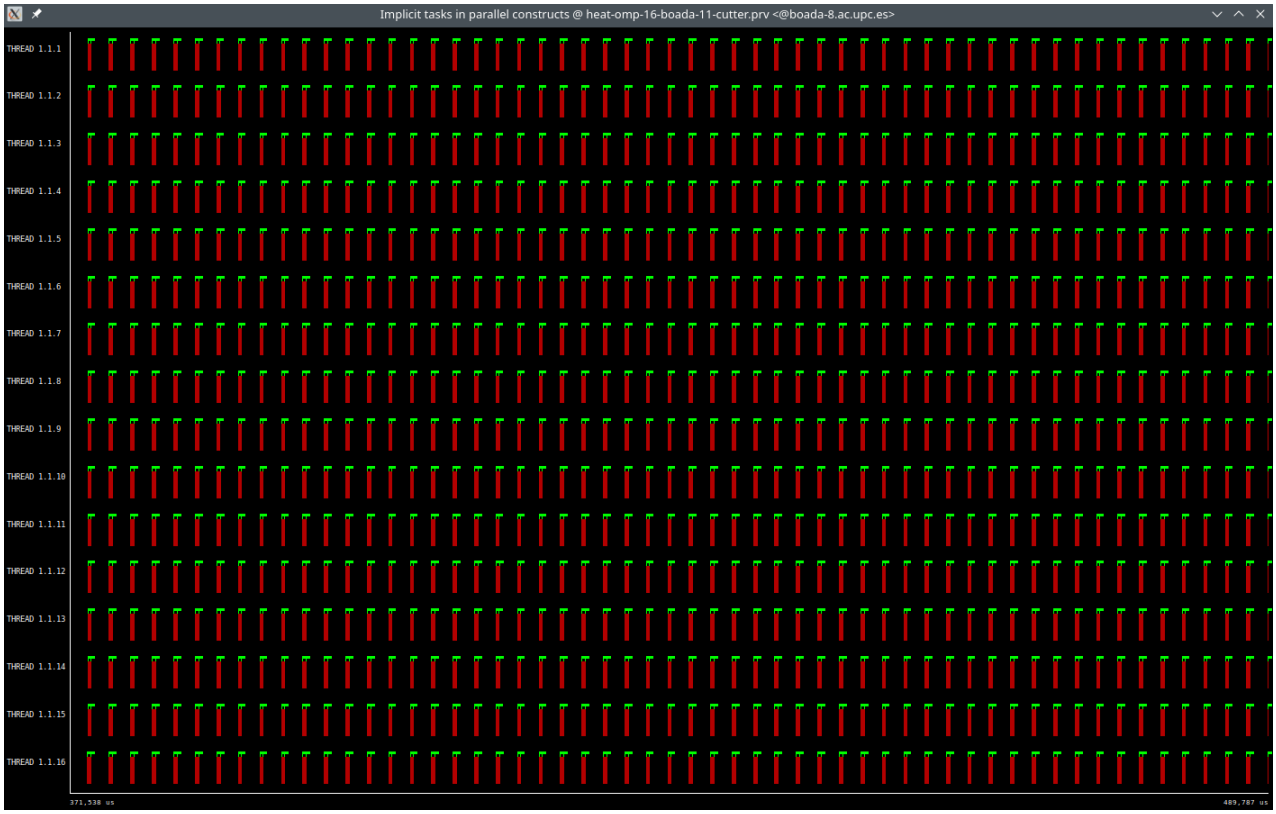
Table 2: Analysis done on Thu May 25 10:47:34 AM CEST 2023, par4309

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)	1840.58	521.68	368.39	329.22
Load balancing for implicit tasks	1.0	0.85	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	6.48	191.66	7.26	7.34

Table 3: Analysis done on Thu May 25 10:47:34 AM CEST 2023, par4309

```
// 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;
          }
        }
      }
    }
```


Instantaneous parallelism @ heat-omp-16-boada-11-cutter.prv <@boada-8.ac.upc.es>
APPL 1
0 us
16
2,155,185 us
Implicit tasks in parallel constructs @ heat-omp-16-boada-11-cutter.prv <@boada-8.ac.upc.es>
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
THREAD 1.1.9
THREAD 1.1.10
THREAD 1.1.11
THREAD 1.1.12
THREAD 1.1.13
THREAD 1.1.14
THREAD 1.1.15
THREAD 1.1.16
0 us
40 (solver-omp.c, heat-omp)
2,155,185 us



Optimized parallelism:

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.82	0.70	0.41	0.23
Speedup	1.00	4.02	6.87	12.11
Efficiency	1.00	1.00	0.86	0.76

Table 1: Analysis done on Thu May 25 11:22:27 AM CEST 2023, par4309

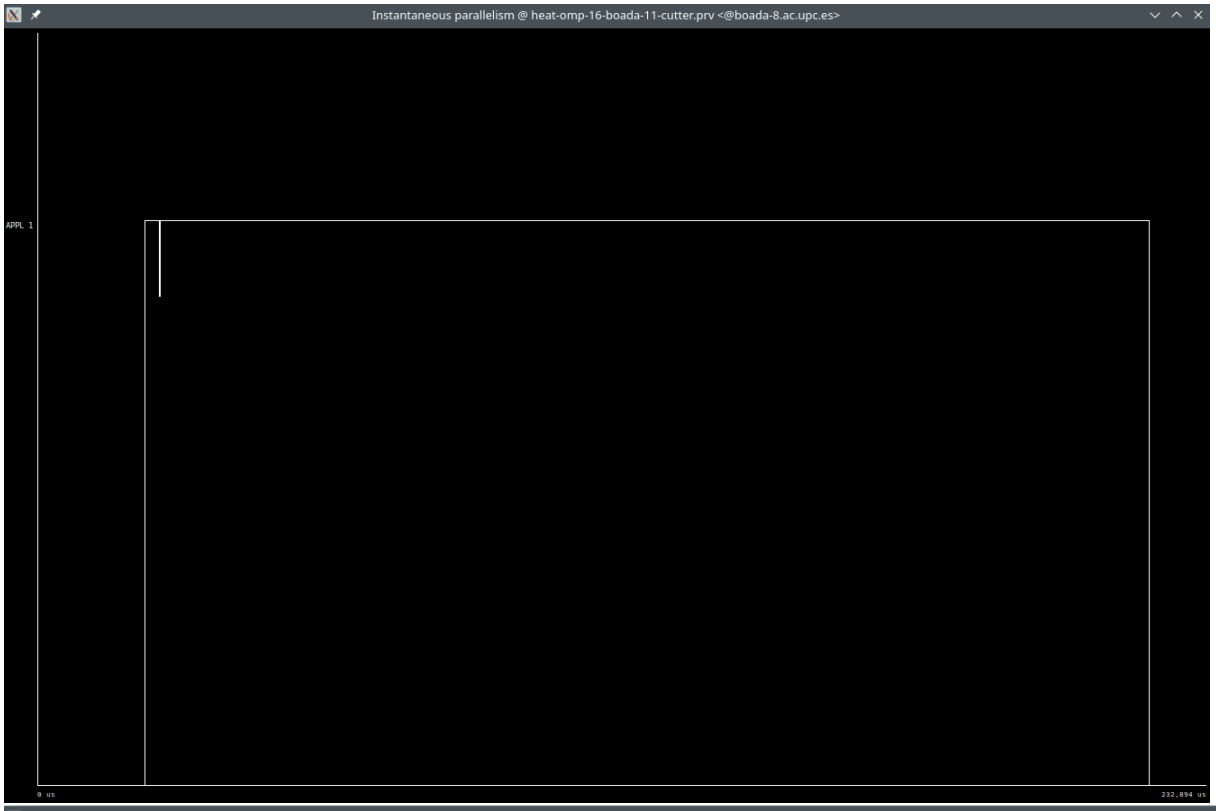
Overview of the Efficiency metrics in parallel fraction, ϕ=98.80%				
Number of processors	1	4	8	16
Global efficiency	99.68%	103.70%	92.34%	88.22%
Parallelization strategy efficiency	99.68%	97.17%	95.25%	92.61%
Load balancing	100.00%	99.06%	97.53%	97.36%
In execution efficiency	99.68%	98.09%	97.66%	95.12%
Scalability for computation tasks	100.00%	106.72%	96.95%	95.26%
IPC scalability	100.00%	108.59%	105.48%	110.41%
Instruction scalability	100.00%	99.95%	98.88%	96.87%
Frequency scalability	100.00%	98.33%	92.95%	89.06%

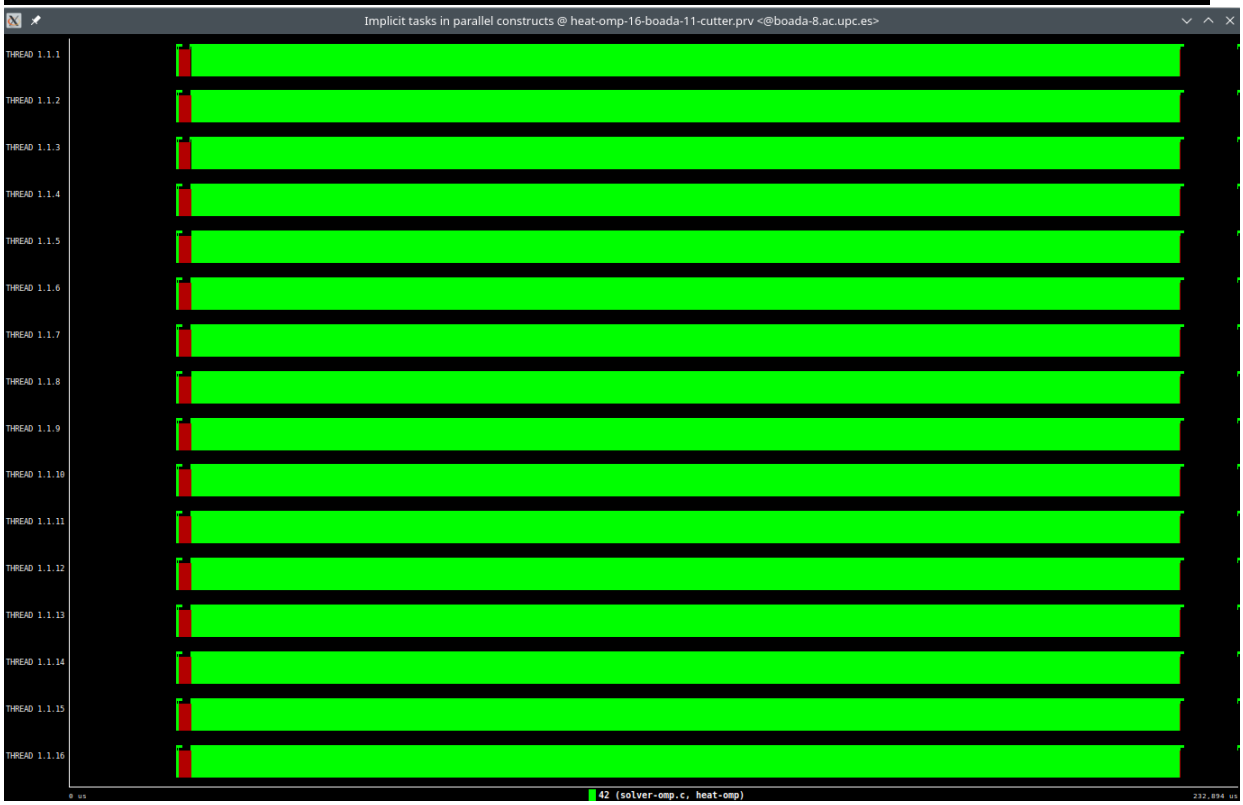
Table 2: Analysis done on Thu May 25 11:22:27 AM CEST 2023, par4309

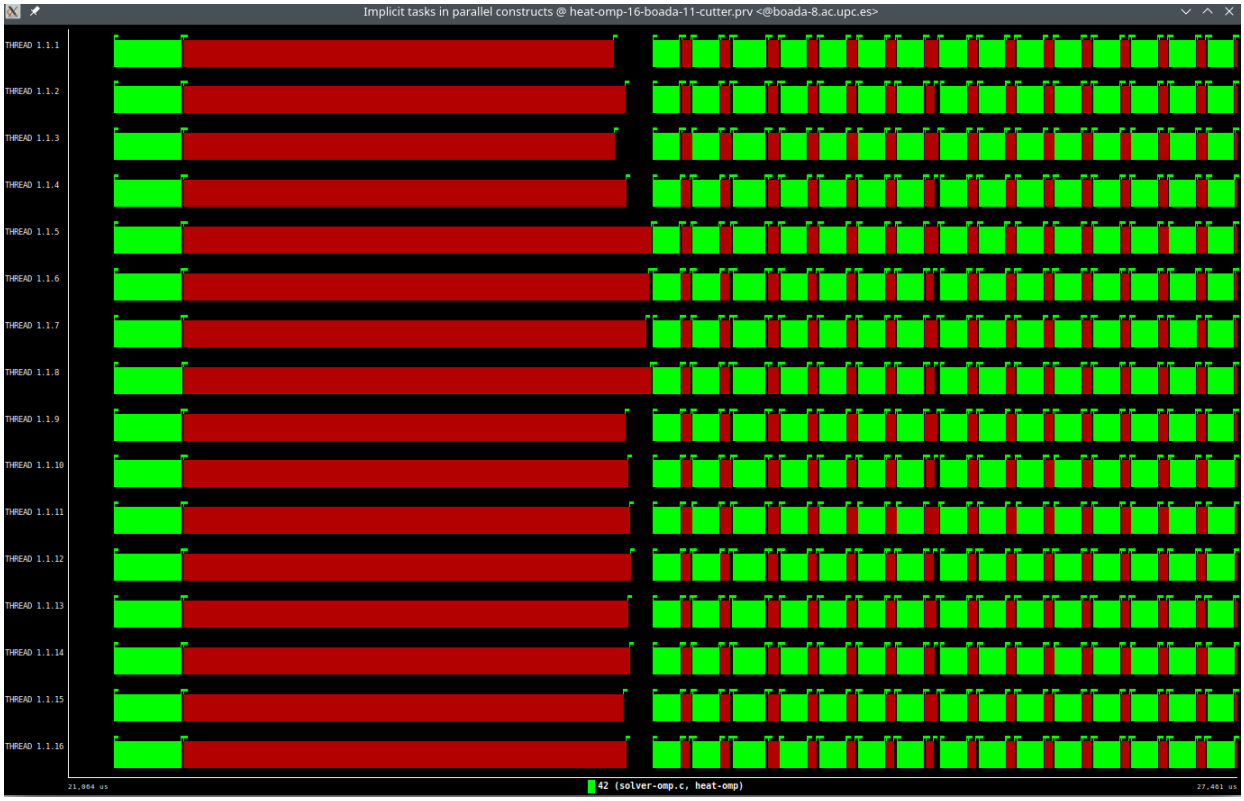
Statistics about explicit tasks in parallel fraction				
Number of processors	1	4	8	16
Number of implicit tasks per thread (average us)	2000.0	2000.0	2000.0	2000.0
Useful duration for implicit tasks (average us)	1388.45	325.26	179.02	91.1
Load balancing for implicit tasks	1.0	0.99	0.98	0.97
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	4.49	8.11	11.04	7.89

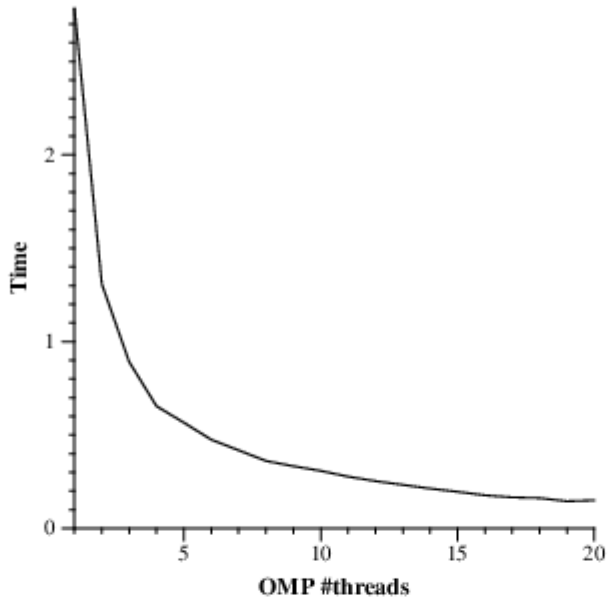
Table 3: Analysis done on Thu May 25 11:22:27 AM CEST 2023, par4309

```
// 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];
        }
    }


}
```

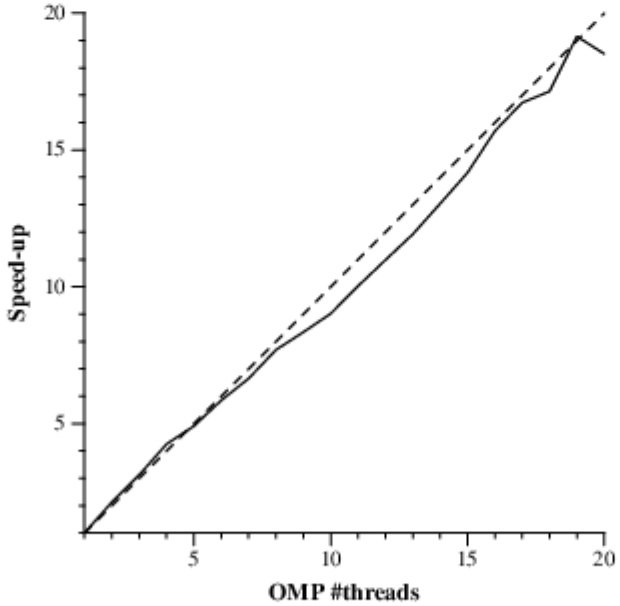

Instantaneous parallelism @ heat-omp-16-boada-11-cutter.prv <@boada-8.ac.upc.es>
APPL 1
0 us
232,894 us


Implicit tasks in parallel constructs @ heat-omp-16-boada-11-cutter.prv <@boada-8.ac.upc.es>
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
THREAD 1.1.9
THREAD 1.1.10
THREAD 1.1.11
THREAD 1.1.12
THREAD 1.1.13
THREAD 1.1.14
THREAD 1.1.15
THREAD 1.1.16
0 us
42 (solver-omp.c, heat-omp)
232,894 us


Implicit tasks in parallel constructs @ heat-omp-16-boada-11-cutter.prv <@boada-8.ac.upc.es>
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
THREAD 1.1.9
THREAD 1.1.10
THREAD 1.1.11
THREAD 1.1.12
THREAD 1.1.13
THREAD 1.1.14
THREAD 1.1.15
THREAD 1.1.16
21,864 us
42 (solver-omp.c, heat-omp)
27,461 us



par4309
Min elapsed execution time
Generated by par4309 on Thu May 25 11:34:03 AM CEST 2023



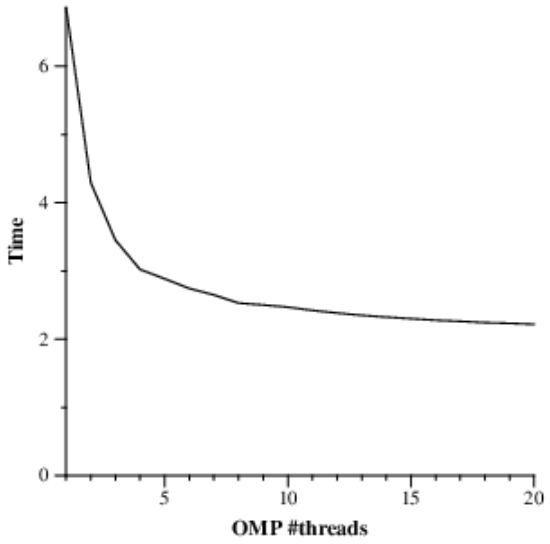
par4309
Speed-up wrt sequential time
Generated by par4309 on Thu May 25 11:34:03 AM CEST 2023

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

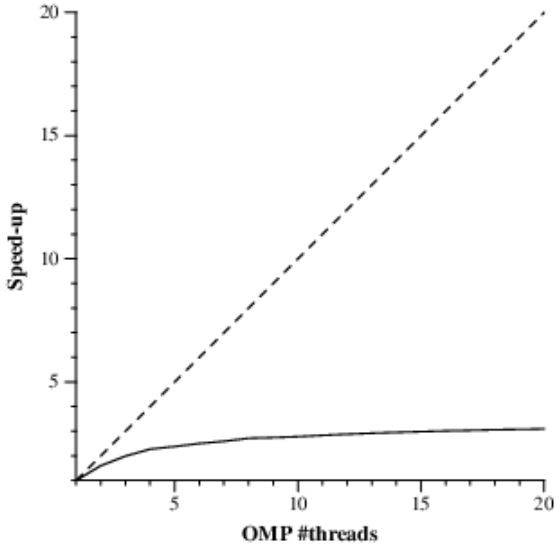
You must include:

- **The plot of scalability when using 4 blocks in the `j` dimension (Overall Analysis section).**
- **An excerpt of the OpenMP code to show the modifications done: OpenMP annotations and synchronization mechanisms (once you make `nblocksj=userparam`) (Detailed Analysis and Optimization section).**
- **The plots obtained with `submit-userparam-omp.sh` (number of threads 16 and `nblocksj=userparam`) and strong scalability for the best number value of `nblocksj=userparam`.**

```
// 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=4;
    int finished[nblocksi][nblocksj];
    for (int i = 0; i < nblocksi; ++i) {
        for (int j = 0; j < nblocksj; ++j) finished[i][j] = 0;
        }

    #pragma omp parallel reduction(+: sum) private(tmp, diff) shared(finished)
    {
      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 prevFinished = 0;
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);

        if(u==unew && blocki!=0){ //Only enter if we are in Gauss and we are not in the first row
            do{
            //Check if the previous block has finished
            #pragma omp atomic read
            prevFinished= finished[blocki-1][blockj]; // COMPLETE
            }
            while(prevFinished != 1);//Wait if prev has not finished
        }

        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
            finished[blocki][blockj] = 1;
        }
      }
    }
    return sum;
}
```

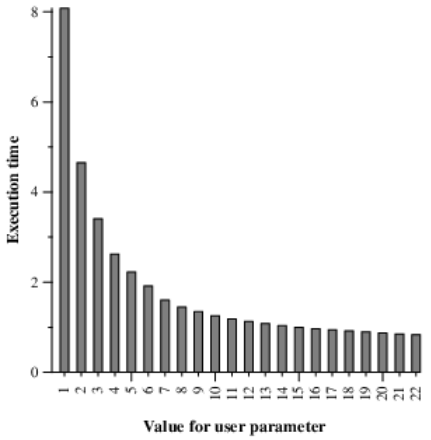


par4309
Min elapsed execution time
Generated by par4309 on Thu Jun 1 11:26:28 AM CEST 2023

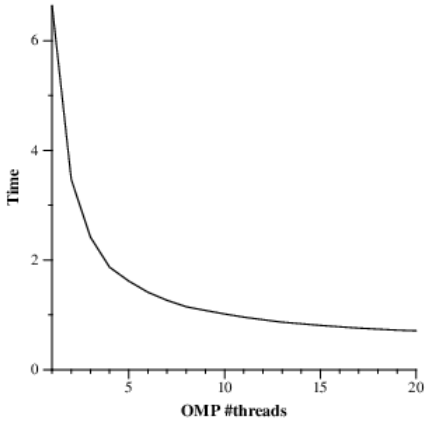


par4309
Speed-up wrt sequential time
Generated by par4309 on Thu Jun 1 11:26:28 AM CEST 2023

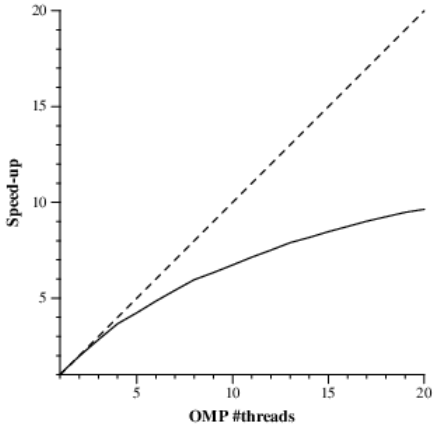
```c
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {

    double tmp, diff, sum=0.0;

    int nblocksi=omp_get_max_threads();
    int nblocksj=userparam;
    int finished[nblocksi][nblocksj];
    for (int i = 0; i < nblocksi; ++i) {
        for (int j = 0; j < nblocksj; ++j) finished[i][j] = 0;
        }

    #pragma omp parallel reduction(+: sum) private(tmp, diff) shared(finished)
    {
      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 prevFinished = 0;
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);

        if(u==unew && blocki!=0){ //Only enter if we are in Gauss and we are not in the first row
            do{
            //Check if the previous block has finished
            #pragma omp atomic read
            prevFinished= finished[blocki-1][blockj]; // COMPLETE
            }
            while(prevFinished != 1);//Wait if prev has not finished
        }

        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
            finished[blocki][blockj] = 1;
        }
      }
    }
    return sum;
}
```



par4309
Average elapsed execution time (heat difussion)
Thu Jun 1 11:34:29 AM CEST 2023



par4309
Min elapsed execution time
Generated by par4309 on Thu Jun 1 11:38:52 AM CEST 2023



par4309
Speed-up wrt sequential time
Generated by par4309 on Thu Jun 1 11:38:52 AM CEST 2023

Comments/Observations

Do you observe a linear speedup for 4 blocks in the j dimension? (Overall Analysis section). Reason *why* changing the number of blocks in the j dimension changes the ratio between computation and synchronisation (Detailed Analysis and Optimization, Number of blocks tune section) and explain the plots that you have included, comparing the strong scalability for both cases `nblocksj=4` and `nblocksj=best`.

No, it is not linear. The speedup takes a curve that when we have more than 5 threads it is practically horitzontal, so there the speed-up remains practically constant. When we change the number of blocks of j, increasing the number, we can observe that we get a better speed-up. Until we have 5 threads it is linear. Then, it starts to curve but the speed-up increase with a big number of threads. On the first case, with nblocksj=4, we observe that the time is redused so much when we increase the number of threads until 5

and then it is redused but not too much and the final time with 20 threads is 2,5. In the second graphic we can see that we have less scalability as we can observe that the function of speed-up is practically horitzontal. On the other case, with nblocksj=best, we have selected 22 as the best value because it has the least execution time from values 1 to 22. In the first graph we observe that the time is redused so much when we increase the number of threads until 8 and then it is redused but not too much and the final time with 20 threads is 1. In the second graphic we can observe that the speed-up is better than with nblocks=4 as the speed-up remains increasingly more when the threads increase too.

Final Survey

We would like to get some feedback from you so that we can continue improving the practical part of the course. In particular, we are interested in your opinion about the usage of `modelfactors`. Can you please tell us briefly your opinion about it? Was `modelfactors` useful for you in order to understand the performance of your parallel application? Would you advice its use in the laboratory assignments of this course in future editions?

We have had some problems with lab4 as we can not done all the questions at home because we could not open paraver in our home. About the modelfactor tables, it is a little annoying to wait for some modelfactor tables that sometimes take a lot of time to do the exection. In general we are satisfied about the work done and we think we have learnt so interesting things doing the laboratory sessions.

From 0 to 10, how would you rate:

- `modelfactors`: **7**
- `Tareador`: **10**
- `Extrae + Paraver`: **8**

On the other hand, we have made an effort to reduce the volume of things to deliver in the lab documents, trying to guide more the information we require. For you information, previous semesters student has to deliver documentation from lab1 to lab5, and documents had to be delivered as a technical report (introduction of the problem, analysis, implementation, scalability analysis and conclusions).

From 0 to 10, how would you rate (0 - too much, 10 - very well) the volume of documentation to be delivered.

Volume of documentation to be delivered: 9
We think that the work to deliver is adequate. We also think that do the sessions in pairs is usefull and the lab slides provided have helped us a lot

Feel free to include any other comment that you want to add about the practical sessions.

We think that it is important to use third languages and it has been a good manner to practise english and increase our level