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

Neus Mayol and Pol Verdura
group 11 - (par1117 and par1121)

Spring 2022-23

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Index

1

Sequential heat diffusion program and analysis with *Tareador*

As we see over Figure 1.1, the code can't be parallelised on both algorithms because we have defined really big tasks and all dependencies are serialized.

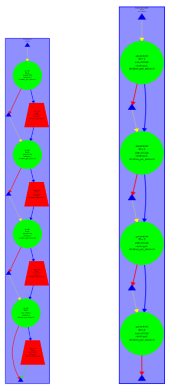


Figure 1.1: Jacobi and Gauss-Seidel TGD with a coarse-grained task definition

To see if we can exploit any sort of parallelism we try to refine the granularity, defining tasks as block size with the following code fragment:

```c
...
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_block");
        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++) {
                ...     //Loop body
            }
        }
        tareador_end_task("solve_block");
    }
}
...
```

The variable that is causing the serialisation of the tasks is the variable `sum` because is shared between all tasks and each thread updates `sum`. This can be solved using clause `reduction(+:sum)` when the tasks are created in OMP. Additionally, with *Tareador* we solve this problem with the clause `tareador_disable_object(&sum)` at the beginning of the outermost most loop and using the clause `tareador_enable_object(&sum)` at the end of the outermost loop.

If we display the TDG with the previous modifications we obtain the first and third figures from Figure 1.2. As we see, in the TDG there are parts of the code that can still be parallelised. When we update the matrix at function `copy_mat` this code fragment is serialized. If we copy the matrix using the last strategy (parallelisation with blocks) we will obtain the second and fourth TDG from Figure 1.2.

Finally, if we analyze the TDG from the previous strategy we can see that using *Jacobi* we gain a better parallelism than using *Gauss-Seidel*. The reason behind the difference could be that with *Jacobi*, we update the result on a new matrix. On the other hand, with *Gauss-Seidel* we use the same matrix to update the results.

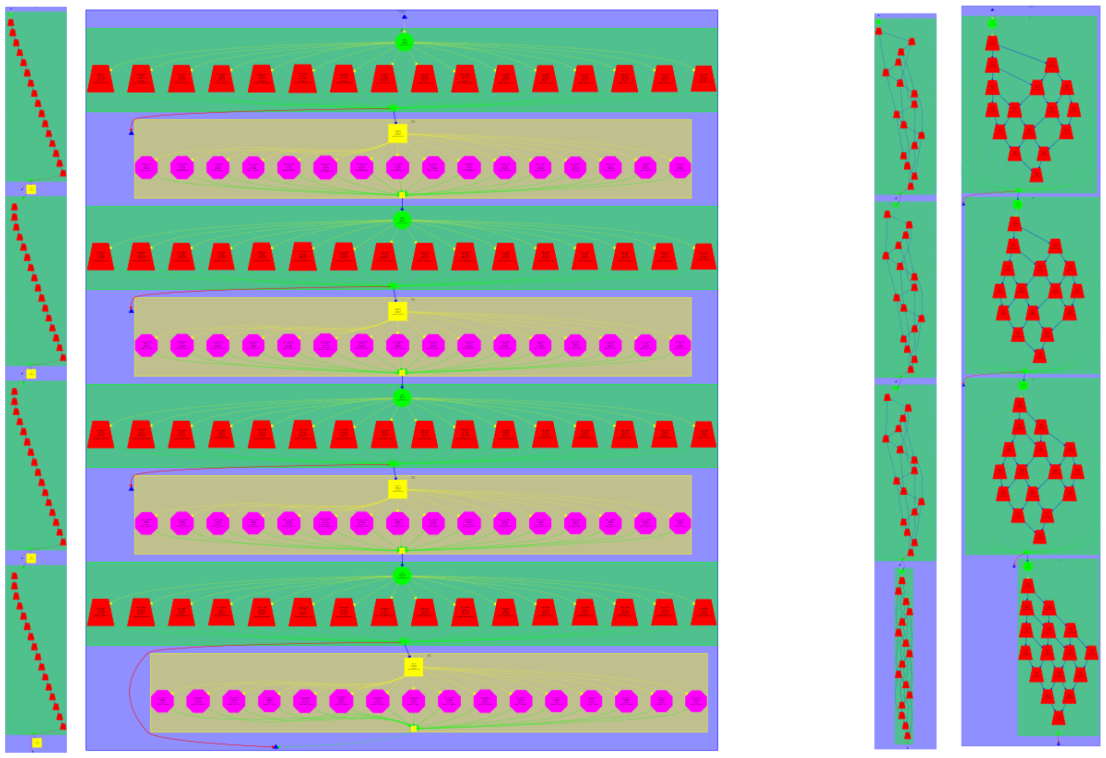


Figure 1.2: TDG of Jacobi (left) and Gauss-Seidel (right) with block strategy

2

Parallelisation of the heat equation solvers

2.1 Jacobi solver

If we parallelise the function `solve`, we see on Table 2.2 that if we increase the number of processors it does not scale correctly. On the other hand, we can see that the parallel fraction of the code is really bad, that's because we did not parallelise the function `copy_mat`, creating then a bad scalability.

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.79	2.12	1.97	2.15
Speedup	1.00	1.32	1.41	1.30
Efficiency	1.00	0.33	0.18	0.08

Table 2.1: Analysis done on Sun Jun 4 04:46:19 PM CEST 2023, par1121

Overview of the Efficiency metrics in parallel fraction, ϕ=65.41%				
Number of processors	1	4	8	16
Global efficiency	99.65%	77.02%	57.73%	33.07%
Parallelization strategy efficiency	99.65%	86.95%	98.37%	97.95%
Load balancing	100.00%	87.86%	99.90%	99.86%
In execution efficiency	99.65%	98.96%	98.47%	98.10%
Scalability for computation tasks	100.00%	88.59%	58.69%	33.76%
IPC scalability	100.00%	88.82%	69.11%	46.33%
Instruction scalability	100.00%	99.97%	92.32%	81.41%
Frequency scalability	100.00%	99.77%	91.99%	89.50%

Table 2.2: Analysis done on Sun Jun 4 04:46:19 PM CEST 2023, par1121

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)	1819.68	513.52	387.56	336.91
Load balancing for implicit tasks	1.0	0.88	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	6.46	138.39	6.59	7.23

Table 2.3: Analysis done on Sun Jun 4 04:46:19 PM CEST 2023, par1121

If we optimize the code by parallelising the function `copy_mat` (the following fragment), we can see that the parallel fraction of this code is much better than before as shown over Table 2.5. Moreover, if we compare Table 2.1 and Table 2.4, we will see that with this upgrade the speedup with 16 processor is 9,65 faster than before, which means we will have a faster execution time parallelising the function `copy_mat` and the program scales correctly when we add more processors.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {
    int nblocksi=omp_get_max_threads();
    int nblocksj=1;

    #pragma omp parallel firstprivate(nblocksi, nblocksj)
    {
        ... //copy_mat body
    }
}
```

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	2.83	0.71	0.42	0.23
Speedup	1.00	4.00	6.67	12.55
Efficiency	1.00	1.00	0.83	0.78

Table 2.4: Analysis done on Sun Jun 4 06:38:56 PM CEST 2023, par1121

Overview of the Efficiency metrics in parallel fraction, ϕ=98.80%				
Number of processors	1	4	8	16
Global efficiency	99.69%	103.82%	89.67%	92.45%
Parallelization strategy efficiency	99.69%	96.49%	95.51%	92.81%
Load balancing	100.00%	98.08%	97.73%	98.03%
In execution efficiency	99.69%	98.38%	97.73%	94.67%
Scalability for computation tasks	100.00%	107.60%	93.88%	99.62%
IPC scalability	100.00%	108.56%	104.27%	114.28%
Instruction scalability	100.00%	99.95%	97.46%	97.72%
Frequency scalability	100.00%	99.16%	92.39%	89.20%

Table 2.5: Analysis done on Sun Jun 4 06:38:56 PM CEST 2023, par1121

Finally, in Figure 2.1 we compare the timelines from both versions, with 16 processors we see that having a serial strategy of `copy_mat` is a bottle neck for our program and slows down the execution time as it is shown over Figure 2.2.

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)	1394.51	324.02	185.67	87.49
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)	4.36	18.78	10.74	7.98

Table 2.6: Analysis done on Sun Jun 4 06:38:56 PM CEST 2023, par1121

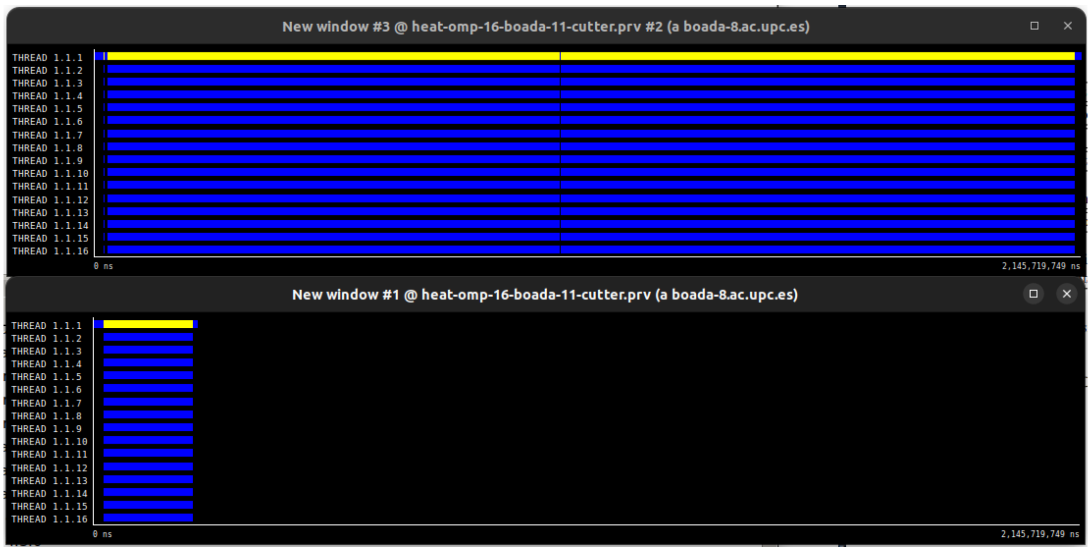

New window #3 @ heat-omp-16-boada-11-cutter.prv #2 (a boada-8.ac.upc.es)
THREAD 1.1.1 … THREAD 1.1.16
0 ns
2,145,719,749 ns
New window #1 @ heat-omp-16-boada-11-cutter.prv (a boada-8.ac.upc.es)
THREAD 1.1.1 … THREAD 1.1.16
0 ns
2,145,719,749 ns

Figure 2.1: Timelines of Jacobi solver with parallelising `copy_mat` (down) and without parallelising `copy_mat` (up)

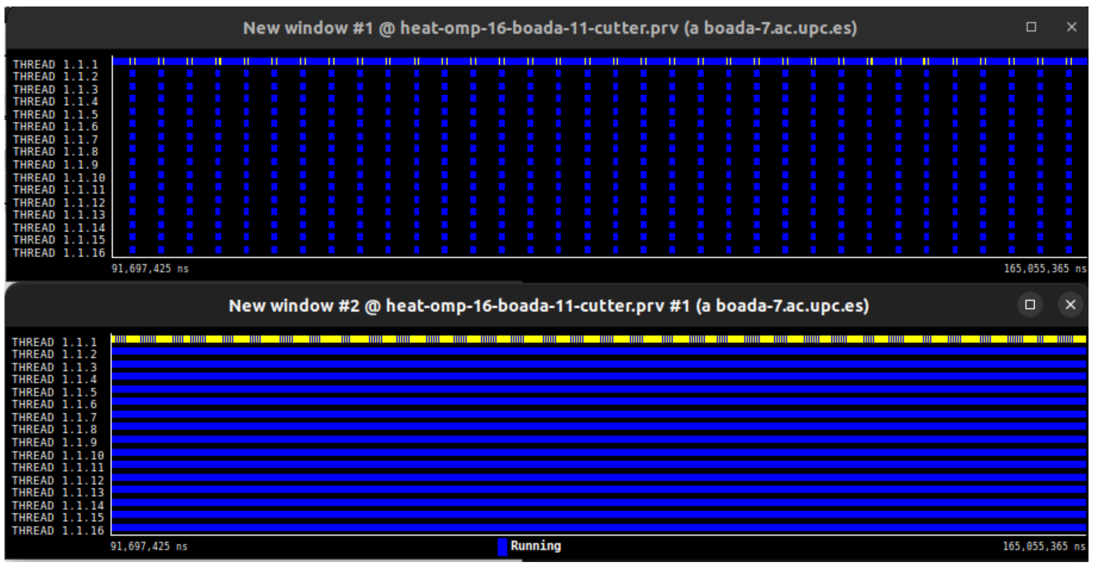

New window #1 @ heat-omp-16-boada-11-cutter.prv (a boada-7.ac.upc.es)
THREAD 1.1.1 … THREAD 1.1.16
91,697,425 ns
165,055,365 ns
New window #2 @ heat-omp-16-boada-11-cutter.prv #1 (a boada-7.ac.upc.es)
THREAD 1.1.1 … THREAD 1.1.16
91,697,425 ns
Running
165,055,365 ns

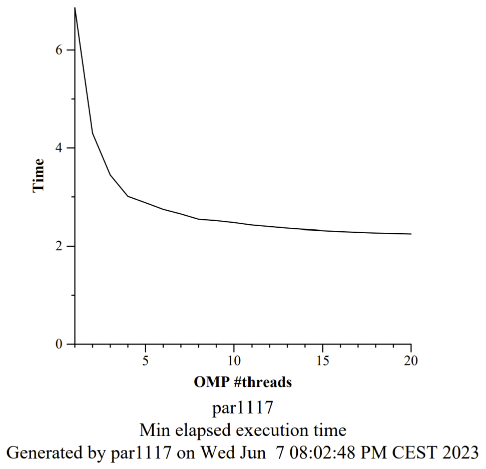
Figure 2.2: Timelines from Figure 2.1 zoomed in

2.2 Gauss-Seidel solver

To use Gauss-Seidel solver we modified the following lines from de function `solver`:

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

    if(u != unew) {
        ... // Jacobi body
    }
    else {  //(u == unew) Gauss-Seidel
        int nblocksi = omp_get_max_threads();
        int nblocksj = 4;
        int lights[nblocksi];

        for(int i = 0; i < nblocksi; ++i) {
            if(i == 0) lights[i] = nblocksj;
            else lights[i] = 0;
        }

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

            //SYNC
            int aux = 0;
            while(aux  <= blockj) {
                #pragma omp atomic read
                aux = lights[blocki];
            }

            //KERNEL
            ... //computation body

            //UPDATE
            if(blocki < nblocksi-1) {
                #pragma omp atomic update
                lights[blocki+1]++;
            }
        }
    }
    return sum;
}
```

Using the previous code we obtain the plots from Figure 2.3. We see that the code does not scale correctly when we increase the number of threads. That's because if we check the previous results from *Tareador*, the last TDG from Figure 1.2 shows that tasks are executed in diagonal, doing so, even if we increase the number of threads, we will only have one instant of time with 4 concurrent tasks.



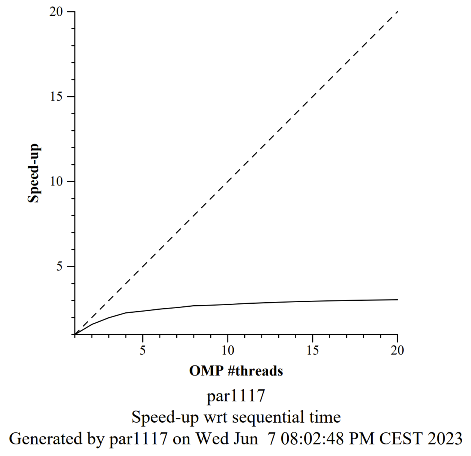


Figure 2.3: Gauss-Seidel statisics with 4 blocks for each thread

Then, if we increase the number of blocks for each task, we will be spending more time executing 4 simultaneous tasks as shown over Figure 2.4, but in exchange, we will have more overheads because we increase the number of tasks.

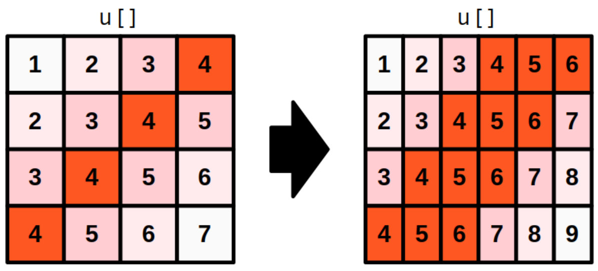


Figure 2.4: Task execution order using Gauss-Seidel solver

With the following code modifications we can choose how many blocks we assign to each thread:

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

    if(u != unew) {
        ... // Jacobi body
    }
    else {  //(u == unew) Gauss-Seidel
        int nblocksi = omp_get_max_threads();
        int nblocksj = userparam*nblocksi;

        ... //Gauss-Seidel code
    }
    return sum;
}
```

After changing the `userparam` value, we executed the program using `submit-userparam-omp.sh`, and we obtained the following plots, displayed in figure 2.5. As we can see, we achieve the minimum execution time when `userparameter = 9`.

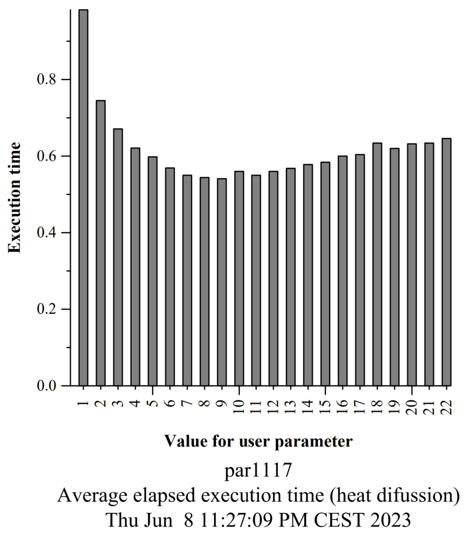


Figure 2.5: Gauss-Seidel solver time statistics using different values of variable `userparam`.

That's why we selected 9 as the value to execute the program using `submit-strong-omp.sh`. The execution gave us the following plots in figure 2.6.

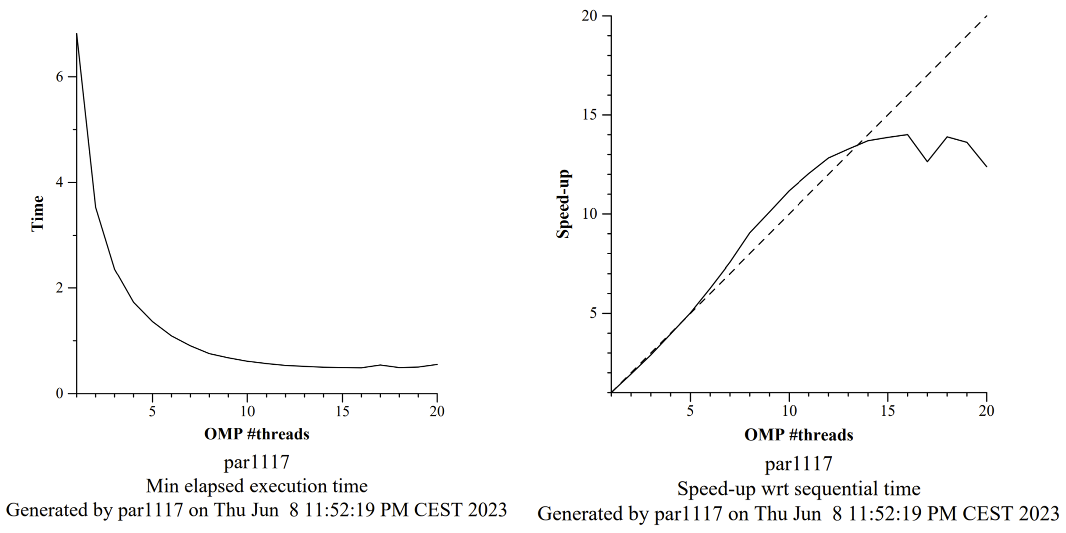


Figure 2.6: Gauss-Seidel statisics with `nblocksj = 9*nblocksi`.

Where, in comparision with figure 2.3, we can appreciate an incredibly better performance. In figure 2.3 time values decreasment stops before and consequently, we obtain bigger values of Min elapsed time and a worst speedup. On the other side, in 2.6 we can appreciate a better performance because we have created the proper amount of blocks in order to take advantage of parallelism without having a big synchronisation overhead. We could say that `userparam = 9` is the equilibrium point between these two factors.