Universitat Politècnica de Catalunya

Parallelism

Lab 5: Geometric (data) decomposition: solving the heat equationivide and Conquer parallelism with OpenMP: Sorting

Carlota Catot Bragós
Ferran Martínez Felipe
PAR1110

Quadrimestre Primavera 2017-2018

Contents

1 Introduction

Parallelism is the introductory subject to the parallelism strategies done in FIB. During all the course we have noticed this in the topic of each lab session: The first one was focused in learn about the parallelism toolkit used in PAR, the second one was focused on an introduction to OpenMP and the third and the fourth focused in learning about the most common task strategies used to generate parallelism.

This lab session is the final step in the introduction to OpenMP and parallelism. In this lab session we have gone to a real problem (an small one) in order to parallelize a piece of code using the most convenient task strategy.

In the following sections we are going to observe this procedure step by step, first by using Tareador to check dependencies in the code and later using OpenMP and Paraver to generating the real parallelism and improving the performance of the system.

2 Analysis with Tareador

In order to check the possible parallelism strategies that we can use in order to parallelize both Jacobi and Gauss-Seidel solvers, the first step to achieve is (as usual) use Tareador in order to simulate the dependencies obtained by using different task strategies.

In order to parallelize both solvers, the strategy used has been generate as finest grain tasks as possible. To do that, a parallel task has been created for each iteration in the inner loop of both Jacobi and Gauss-Seidel. In the case of Jacobi, to increase the parallelism of the whole code, the function copy_math has been parallelized.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {
    for (int i=1; i<= sizex-2; i++)
        for (int j=1; j<= sizey-2; j++) {
               tareador_start_task("Inner Copy Mat");
               v[ i*sizey+j ] = u[ i*sizey+j ];
          tareador_end_task("Inner Copy Mat");
        }
}

double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
             tareador_start_task("Inner Jacobi");
             utmp[i*sizey+j]= 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 = utmp[i*sizey+j] - u[i*sizey + j];

             tareador_disable_object(&sum);
             sum += diff * diff;
             tareador_enable_object(&sum);

             tareador_end_task("Inner Jacobi");

            }
      }
    }
    return sum;
}
```

Figure 1: Code for task decomposition for relax_jacobi and copy_math

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
    double unew, diff, sum=0.0;
    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("Inner Gauss");
            unew= 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 = unew - u[i*sizey+ j];
            tareador_disable_object(&sum);
            sum += diff * diff;
            tareador_enable_object(&sum);
            u[i*sizey+j]=unew;
            tareador_end_task("Inner Gauss");
        }
      }
    }

    return sum;
}
```

Figure 2: Code for task decomposition for relax_gauss

With this task decompositions, using Tareador is possible to check the dependencies in each solver. The dependency graph obtained by using both codes are the following (doing two iterations of each algorithm):

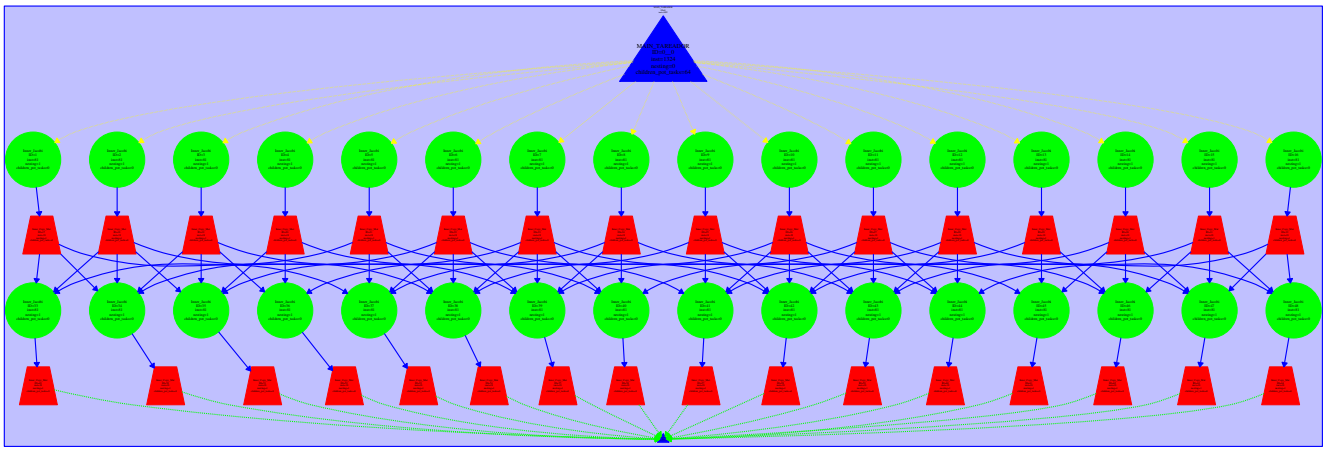

Figure 3: Jacobi and copy_mat decomposition obtained with Tareador

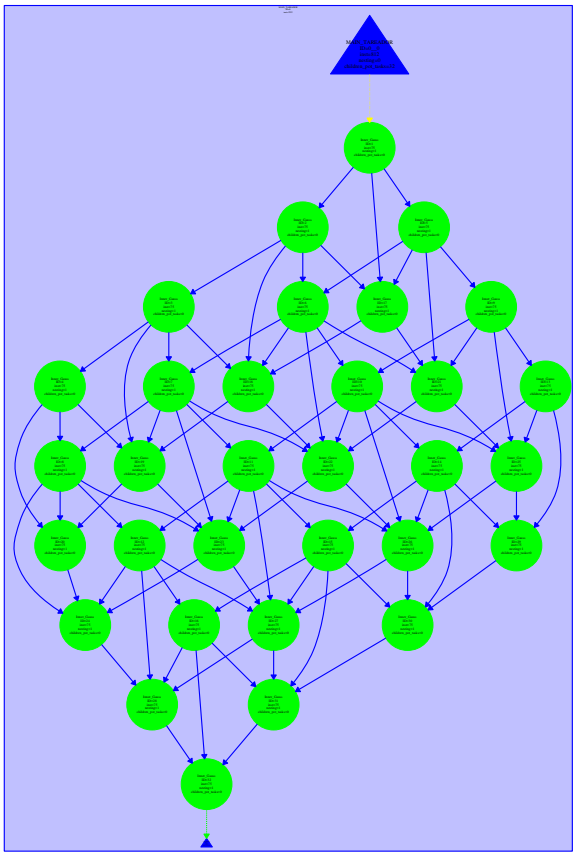

Figure 4: Gauss-Seidel decomposition obtained with Tareador

The first thing to comment is that both codes are sharing the variable sum between threads to store partial results, which has forced us to protect this variable in order to prevent data races. This variable can be protected on the OpenMP version of the code in different ways: using a critical pragma combined with a shared variable sum, using the pragma reduction, using the pragma atomic instead of the pragma critical, etc. In our case the solution chosen has been using the pragma reduction, which stores the partial results in a vector and after the parallel region makes the sum of the partial results of each thread.

In addition, we have to point out that each code is running by doing computations inside a matrix, and each iteration has a dependency with the neighbourhood cells. In the following sections we should take care to use the most appropiate parallel strategy in order to respect that dependencies.

3 OpenMP parallelization and execution analysis: Jacobi

In this section we are going to parallelize the relax_jacobi function, in order to transform the sequential execution of heat.c into a parallel one. To do that, we have generated the following code:

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    int howmany = omp_get_num_threads();

    #pragma omp parallel for private (diff) reduction(+: sum)
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
             utmp[i*sizey+j]= 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 = utmp[i*sizey+j] - u[i*sizey + j];
             sum += diff * diff;
         }
      }
    }

    return sum;
}
```

Figure 5: Code for task decomposition for relax_jacobi using #pragma parallel for

As we can see, we have used the #pragma omp parallel for in order to divide the code execution into different threads. The data decomposition strategy chosen is a block data decomposition, in which we divide all the iterations of the inner loop in blocks of consecutive iterations of the same size (or nearly the same size) by combining an adaptative iterator init and end in the init loop and a task decomposition made in the outer loop.

We can see an example of block data decomposition for a number of threads equal to four in the following figure:

Figure 6: Block data decomposition example

However, the code performance of this parallel version is poor. To improve the parallelism of the

whole code, we have parallelized the copy_mat function. We can see an example of how to parallelize the copy_mat function in the following code:

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for
    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```

Figure 7: Code to parallelize the copy_mat function

The only thing that we have done here is parallelize the outer loop of the copy_mat function. Doing that, all the inner iterations will be assigned to each of the iterations of the outer loop. As we are going to see in the next lines, doing this change we are going to increase the performance of the program.

After parallelizing this two functions, we can use Paraver to check that the performance of the program is correct. To do that, we can observe in the following trace the time that each processor is executing code (in blue). We can check that the parallel region of the current program domains the execution of the program.

Figure 8: Jacobi and copy_mat decomposition obtained with Tareador

Finally, we can compare the Execution Time and the Speed-Up of both programs (the one with the sequential copy_mat and the other with the parallel copy_mat). In the following plots we can check the differences obtained with each code:

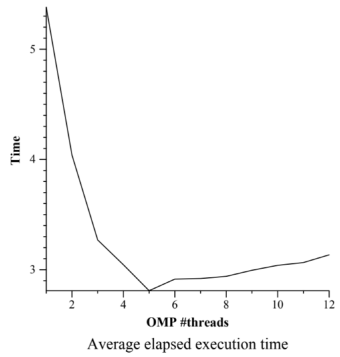

Figure 9: Execution Time parallel Jacobi

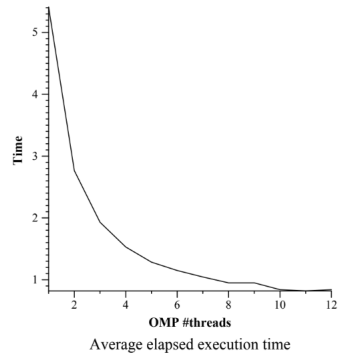

Figure 10: Execution Time parallel Jacobi and Copy Mat

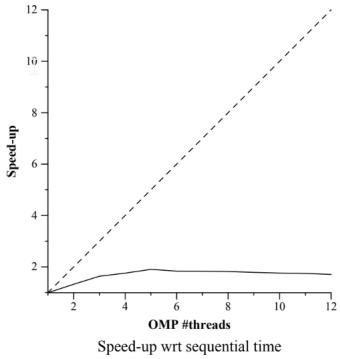

Figure 11: Speed-Up parallel Jacobi

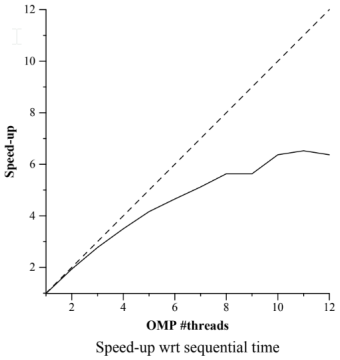

Figure 12: Speed-Up parallel Jacobi and Copy Mat

As we can see the Execution Time and the Speed-Up obtained with the code that uses the parallelized copy_mat has an orders of magnitude better performance than the sequential one (the Speed-Up of the sequential code is nearly 0).

Finally, we have attached the Heatmaps obtained with the sequential Jacobi and the parallel Jacobi, in order to check if the parallel version that we have done is correct. There is no difference between images.

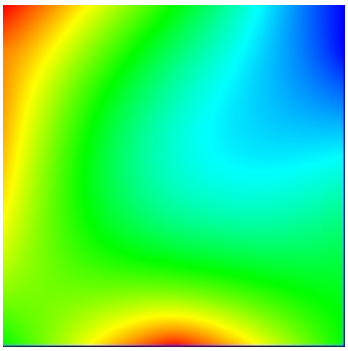

Figure 13: Sequential Heatmap generated

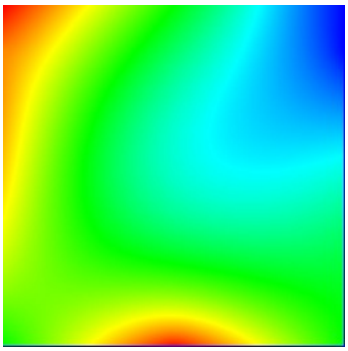

Figure 14: Parallel Heatmap generated

4 OpenMP parallelization and execution analysis: Gauss-Seidel

In this section we are going to parallelize the relax_gauss, in order to transform the sequential execution of heat-omp.c into a parallel one, for doing we change the function relax_gauss on the solver-omp.c code. We generate the following function:

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    int howmany = 4;

    #pragma omp parallel for ordered(2) private(unew, diff) reduction(+:sum)
    for (int blockid_row = 0; blockid_row < howmany; ++blockid_row) {
        for (int blockid_col = 0; blockid_col < howmany; ++blockid_col) {
            int i_start = lowerb(blockid_row, howmany, sizex);
            int i_end = upperb(blockid_row, howmany, sizex);
            int j_start = lowerb(blockid_col, howmany, sizey);
            int j_end = upperb(blockid_col, howmany, sizey);

            #pragma omp ordered depend (sink: blockid_row-1, blockid_col)
            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++) {
                    unew= 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 = unew - u[i*sizey+ j];
                    sum += diff * diff;
                    u[i*sizey+j]=unew;
                }
            }
            #pragma omp ordered depend(source)
        }
    }
    return sum;
}
```

Figure 15: Code for task decomposition for relax_gauss using OpenMP ordered clause

By seeing the code we can see that parallelize the Gauss-Seidel function is more complicated than Jacobi one. That's because in this one we can find dependencies between iterations, and to solve this we use a block distribution cause we think that is a better idea, as opposed to the row decomposition which was initially implemented. By using block decomposition, we can avoid the dependency that we can see on the i - 1 and the j - 1 elements.

So we can see that in this sections code changes are more significantly than in the previous one. In this code we continue use howmany to represent the number of blocks that will be distributed, but the main difference is that now, there are 2 loops, one to iterative over the block's x-coordinate, and one to iterate over the y-coordinate. There are also new limits which determine how many columns contain each block.

These code is implemented to enable the use of the OpenMP ordered directives, that will help us to work with block descomposition.

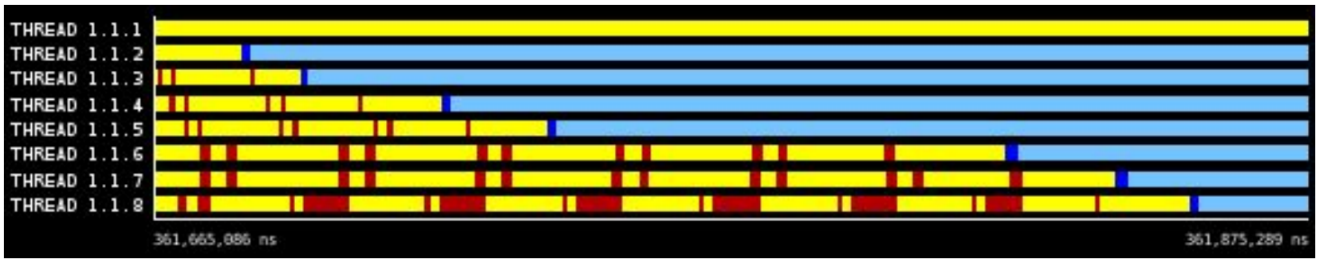

Figure 16: Gauss-seidel paraver trace

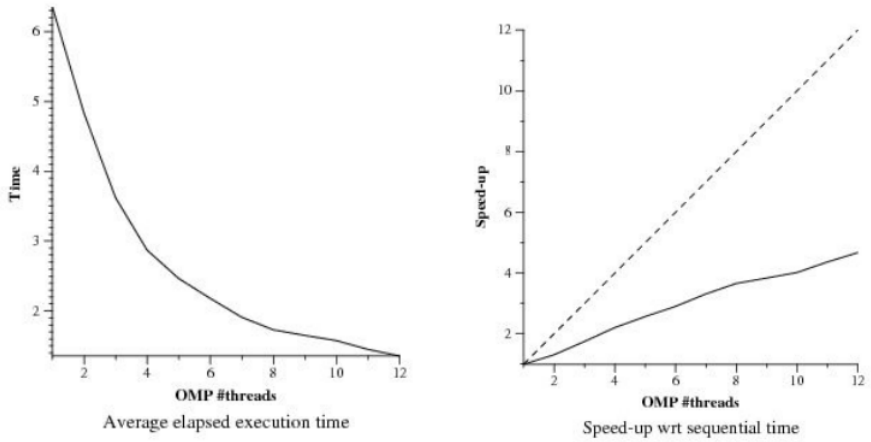

Figure 17: Strong scalability graphics

By taking a look on the paraver trace on figure 16 and the Scalability graphics on figure 17 we can conclude that at list is much better than Jacobi one.

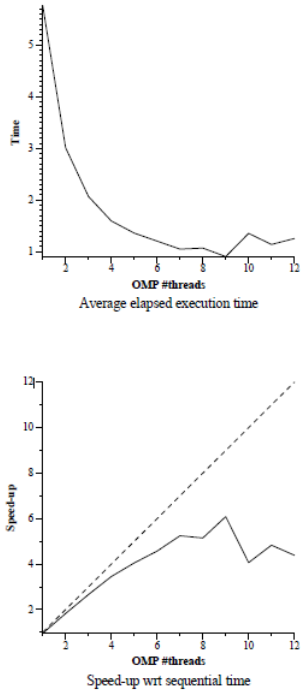

Figure 18: Strong scalability graphics for omp_get_num_threads()

In the previous image we can see that if we change a bit the code, we can get in some threads a better parallelization but it still not optimizated at all. The code was changed in this way:

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    #pragma omp parallel
    {
        int howmany = omp_get_num_threads();

        #pragma omp for ordered(2) private(unew, diff) reduction(+:sum)
        ...
    }
}
```

Figure 19: Better parallelization of Gauss-Seidel

In figure 18 we also can conclude that the optimum value of threads is 9, because is the point were we see that the code is better parallelized.

5 Conclusion

In this laboratory session we learn about solving a real problem, in this way we learn about Jacobi and Gauss-Seidel functions, we can say that it's a better experience to work with more real problems, also if the real problem is not a big problem.

We can conclude that in this subject we learn the basic thing about parallelization, with good material, such as boada and paraver to learn the basic concepts of OpenMP and parallelism. It has been a good experience and we learn a lot in this lab sessions.