Parallelism
1202

Lab 2: Geometric decomposition – solving the heat equation

Héctor Ramón Jiménez

Alvaro Espuña Buxo

May 16, 2014
Facultat d'Informàtica de Barcelona

1 Analysis of dependences for the heat equation

The analysis of the dependencies caused by the proposed task decomposition for the two solvers of the heat equation has been done using Tareador. Dependences appear when a task reads a memory position previously written by another task. Some dependences may impose task ordering constraints while other may impose data access constraints. For solving the heat equation two different solvers have been used, each with different dependence patterns.

For the Jacobi solver, there are two data dependencies between iterations, `sum` and `diff`. `diff` is not a real dependency, because it's value is not reused, but it's a global variable. `sum` presents the tipical acumulator dependence pattern. This dependences are only data-access dependences. Note that, to update the value in a cell we need to read its neighbours, but then we write the new value in a different matrix, so there's no task ordering dependence. This can be seen in figure 1. In the parallelization with OpenMP, we plan to guarantee these dependences using an `omp pragma` with `private` for `diff` and a reduction on `sum`.

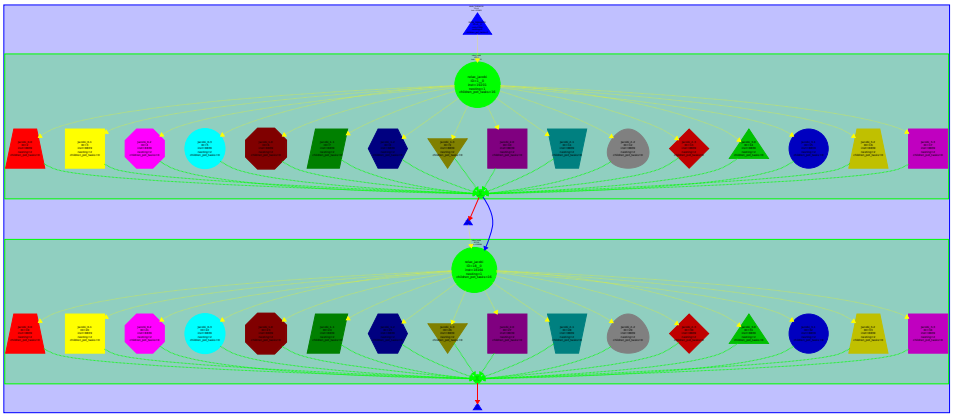

Figure 1: After isolating sum in `Tareador`, a Jacobi iteration is fully parallelizable.

For the Gauss-Seidel solver the dependences are a bit more complicated (as we can see in figure 2). First of all, we have a task ordering dependence. We are reusing the same matrix, and since we are reading all neighbours to update the current cell, we need to be careful about the order the calculations are performed. Also, there are data access dependences, because `unew`, `sum` and `diff` variables are global to the parallel zone. To solve the first part, we will use blocking. A block will be processed only when the upper block and the block of the left have been processed. This allows us to create individual parallelizable tasks. For the second part, we will use `reduce` on `sum` and `private` on `diff` and `unew`.

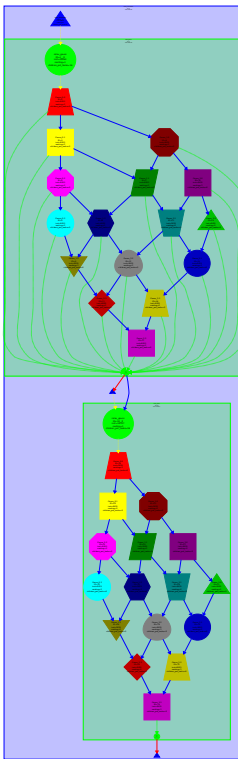

Figure 2: After isolating sum in tareador, a Gauss-Seidel iteration depends on the *left* and *upper* tasks.

The following table summarizes the predicted speed–up for the parallel execution for the two solvers, with 2, 4, 8 and 16 processors with respect to the simulation with just 1 processor:

Solver	2 threads	4 threads	8 threads	16 threads
Jacobi	1.955	3.419	5.467	6.868
Gauss-Seidel	1.774	2.498	2.498	2.498

Figure 3: Speed up expected. Simulated using `Paraver`.

We can see in figure 3 how we expect Jacobi to scale reasonably well, but for Gauss-Seidel, because of the data dependencies, using with the same granularity, we won't be able to get more speed-up adding more processors (we are bounded by the critical path).

2 OpenMP parallelization and execution analysis

1. **Briefly comment the parallelization with OpenMP that has been done for the two solvers.**

   First of all, for the Jacobi solver, we optimized the sequential code to obtain almost 1.5x speed-up. Instead of performing a copy of the matrix every time, we just swap the pointers of the struct, so it's now done in constant time. Then, to allow parallelization we added `#pragma omp parallel for private(diff) reduction(+:sum)` before the first loop, thus solving the dependencies between loops. Following the specification of the assignment, we changed `nbx` to be the number of threads and `nby` to 1, hence executing the whole matrix row with the same thread. Because the default distribution is `static` 1, and there will be `nbx` threads, every thread will take a *row block*.

   For the Gauss-Seibel we did more work, because there are more dependencies. The first thing we did was adding `#pragma omp parallel for private(diff, unew) reduction(+:sum)` before the first loop. To avoid starting on a block whose dependencies have not been calculated we created an array (processed) that stores in the i-th position the greatest index of the *column block* that has been already treated by the thread i.

```
int *processed = malloc(num_threads * sizeof(int));

#pragma omp parallel for
for (int i = 0; i < num_threads; i++)
  processed[i] =  -1;

// ...
int thread_id = omp_get_thread_num();
for (int jj=0; jj<nby; jj++) {
  while (thread_id > 0 && processed[thread_id - 1] < jj) {
    #pragma omp flush(processed)
  }
  // ...
  processed[thread_id] = jj;
  #pragma omp flush(processed)
}
// ...
```

   We do an active wait if the previous needed block has not been processed yet. The flush is needed to comunicate the changes in `processed`, otherwise we would be trapped in an infinite loop.

2. **Plot the speedup achieved by the OpenMP parallel version, with respect to sequential execution time, for the two solvers. Comment the results that are obtained and justify the scalability that is obtained. Include Paraver timelines in order to support your explanation.**

   In figures 4 and 5 we can see how Jacobi scales better than Gauss-Seidel. This is because, Jacobi is fully parallelizable and Gauss-Seidel has to be computed *diagonally*. Gauss-Seidel's speedup converges rapidly to 2.5.

   Also is interesting to note that Jacobi performs better with 8 threads than with 16. This is because we only have 12 cores, so using 16 threads we are relying on hyperthreading.

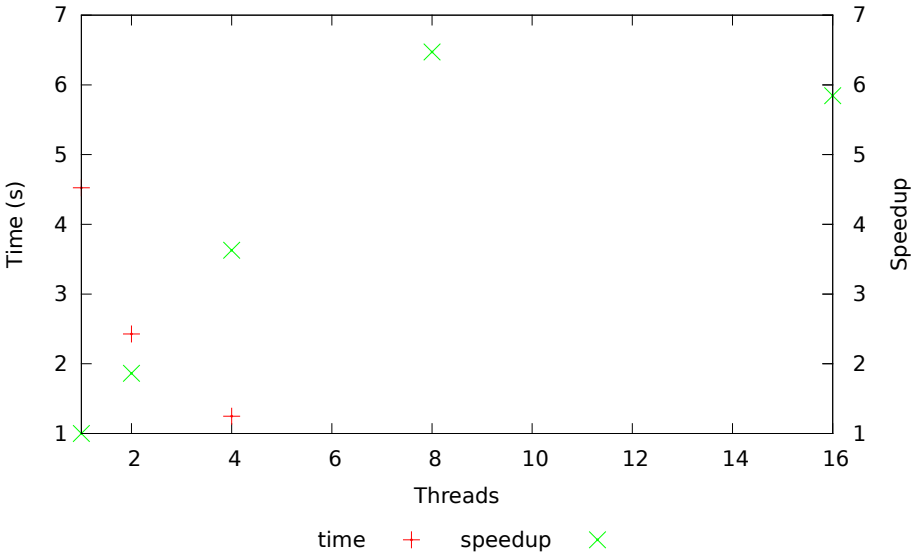

Figure 4: Jacobi parallelization results.

In figures 6 and 7 we can confirm using `Paraver` that Gauss-Seidel, due to task ordering dependences, some threads spend a lot of time idle. In Jacobi version, though, the parallelization exploits better the multicore usage.

During this exercise we noticed that past 16 threads (17 or more), because of the accuracy of floating point arithmetic, the results were wrong. Both using Jacobi and Gauss-Seidel stopped earlier than expected: the program did less iterations than the original. One possible solution would be to adjust the threshold as a function of the number of threads.

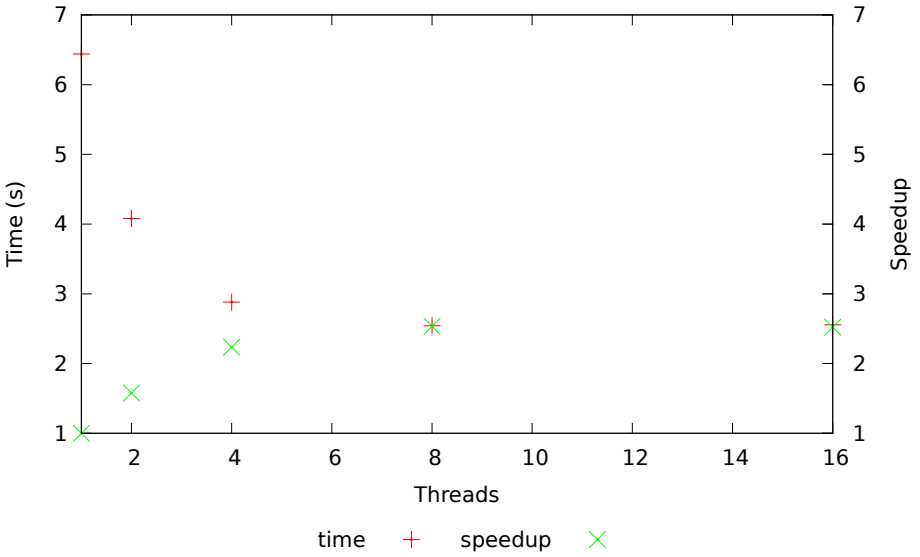

Figure 5: Gauss-Seidel parallelization results.

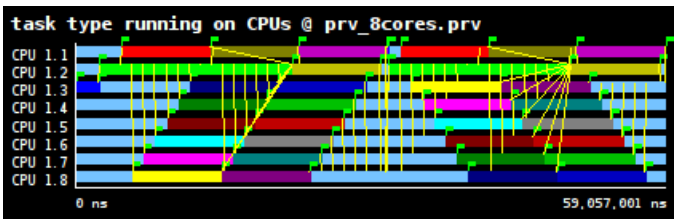

Figure 6: Jacobi `Paraver` simulation with 8 cores.

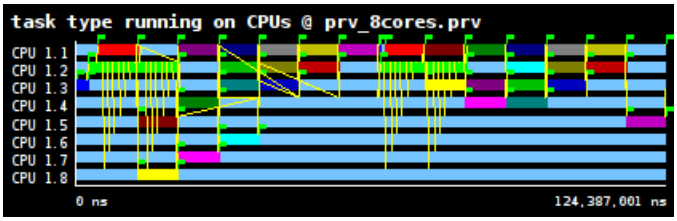

Figure 7: Gauss-Seidel `Paraver` simulation with 8 cores.

3. **In the parallelization of Gauss-Seidel, analyze the impact on the parallel execution time of the block size `by` (or equivalently the number of blocks `nby`) [1]. For example try with `nby=2*nbx`, `nby=4*nbx`, ... Is there any performance impact? Is there an optimal point? Justify your answer.**

nby	1 thread	2 threads	4 threads	8 threads	16 threads
2*nbx	6.452	4.051	2.311	1.421	1.454
4*nbx	6.433	3.597	1.980	1.275	1.585
8*nbx	6.117	3.211	1.689	1.375	2.369
16*nbx	5.708	2.763	1.666	1.678	3.784
32*nbx	4.905	2.642	1.805	2.629	-

Figure 8: Impact of the number of blocks `nby`. Time is in seconds.

The number of blocks `nby` has an obvious performance impact, affecting considerably the parallel execution times. This is because the `nby` value is directly related with the number of performed flushes. There are going to be at least `nby*nbx` flushes to mark each block as processed. To this flushes we have to add the flushes that each thread performs when it's waiting its dependencies to be processed. Thus, the optimal value is going to be one that finds a balance between these two types of flushes (one that keeps `nby` reasonably low but allowing to solve the dependencies quickly). In the obtained results this value seems to be `4*nbx` with 8 threads.

[1] The resulting image should be the same.