Assignment 2 - CPDS Parallelism Solving the Heat Equation using several Parallel Programming Models

Roberto Meroni
email: roberto.meroni@estudiantat.upc.edu

November 2023

1 Parallelization

The Jacobi and Gauss-Seidel methods are examples of an iterative approach used to solve partial differential equations. In the context of the heat equation, they update the temperature values at discrete points on a grid. The algorithms iteratively compute new temperature values based on the surrounding values from the previous iteration, following the so-called stencil pattern.

In the Jacobi method, as each grid point is updated based on the values from the previous iteration, there are no dependencies between the grid points, and all updates can be performed independently in parallel. The embarrassingly parallel nature of this method is offset by its slow convergence. In the Gauss-Seidel method, however, each grid point is updated in-place during each iteration. This generates loop-carried, so-called RAW dependencies, as each point can read neighboring values (the one above and the one to the left) only after they have been updated in the current iteration. The lower level of parallelism resulting from the wavefront processing of this method is compensated by its faster convergence, compared to the Jacobi method.

OpenMP

- Loop Parallelism

  For the purpose of simplifying the laboratory, both provided codes implemented the loop-blocking technique, where the grid was divided into smaller blocks. This is necessary for the Gauss-Seidel method, but is not a requirement for the the Jacobi method.

  In order to optimize the execution of the Jacobi method, the redundant division of the grid into blocks was removed. The `#pragma omp parallel` for directive was used to parallelize the remaining two loops, iterating over the grid points (i, j). As we are dealing with a rectangular iteration space, we used a `collapse(2)` clause to specify that both loops should be collapsed into a single loop construct, allowing for higher parallelism.

  For Gauss-Seidel method, we explored two different parallelization techniques: task parallelism (explicit task with dependencies) and do-across. For both techniques we used `#pragma omp parallel` directive in order to create a team of threads, however, they are managed differently.

  In the explicit task with dependencies technique, we used `#pragma omp single` to ensure that only one thread creates tasks. The tasks are executed concurrently by different threads, and dependencies between tasks are managed explicitly by using the `depend` clauses. These dependencies ensure that a given task does not proceed until the task that processes a block directly above and on its left is finished.

In the do-across technique, we used a `#pragma omp for collapse(2)` in order to parallelize two outer loops (block execution). As in the Gauss-Seidel method the order of execution is critical to fulfill dependencies, we added an ordered clause to define that 2 outer loops contain cross-iteration dependencies to make sure that blocks are executed in order. The `#pragma omp ordered` directive with `depend(sink)` clause ensures that the computation of a given block is executed only after the execution of the block directly above and to the left is complete.

- Thread Safety

  The `private(diff)` clause is used to ensure that each thread has its private variable diff. This prevents data races when multiple threads update the `diff` variable simultaneously.

  Additionally, in Gauss-Seidel method, it was necessary to mark as private an auxiliary variable `unew` used to compute a new value for the grid point. Such a variable does not exist in the Jacobi method. We have instead an auxiliary array used to store newly computed values, however, each thread updates a distinct segment of the array, thus it is not necessary to mark it as private.

  The `reduction(+:sum)` clause is used to perform a reduction operation on the sum variable. During each iteration, each thread computes its local sum of squared differences `(diff * diff)` and contributes to the global sum. The reduction operation aggregates these local sums, ensuring that the final sum variable contains the correct result.

MPI

- Distribution of Work

  The master process is responsible for distributing the algorithmic parameters and initial data to the worker processes. This includes sending (blocking communication with `MPI_Send`) the number of iterations, grid resolution, and information about the selected algorithm (Jacobi, or Gauss-Seidel). Additionally, distinct segments of the grid (consecutive rows, plus 2 additional rows for storing boundary values from adjacent processors) are sent to each worker process for local computation.

  Once the local computations are complete, the worker processes send their final results to the master process (blocking communication with `MPI_Send`). The master process collects the final values of the internal grid points for all segments using a non-blocking communication `MPI_Irecv`.

- Local Computations

  Jacobi

  Boundary exchange in the first iteration is skipped, as all processes already received it during the initial data distribution performed by the master process. Each process calculates the new values of each subset of rows, iterating over blocks. After that, they perform boundary data exchange: the first and the last row of internal points are sent to the previous and next processor respectively; the first internal row from the next processor is received and stored as the last row (ghost points), and the last internal row from the previous processor is received and stored as the first row (ghost points). As the first (master) process and the last process have only one adjacent process, they skip irrelevant communications. To minimize the overhead, non-blocking MPI communication with `MPI_Isend and MPI_Irecv` is used for all boundary exchanges, which enables overlapping computation (allows the copy from u to uhelp even if we are still waiting for the receiving) and communication, improving performance.

Gauss-Seidel

To parallelize the Gauss-Seidel algorithm, we divide the matrix into blocks. To get the same result as sequential, each block has to be computed after the neighbor block on the top and the neighbor block on the left. We divide the matrix into a subset of rows, and we assign each subset of rows to one process (in this case, one block line to every processor). For the computation we also need the cells above the block and below the block, and, since the memory is not shared between the process, we need to send and receive these rows between adjacent processors. Even in the sequential code, each block is executed strictly after the neighbor block on the left, so this dependency is naturally satisfied. With the block division that we made, each block already has the columns on the left and right, because they are computed by the same processor, so we need just to exchange the bottom and the top rows. At the end of the computation of each block, we send (`MPI_Isend`) the bottom row of the block to the block below. We do this through non-blocking communication so that the processor can start immediately to compute the block on the right (when this has been received from the block above). For receiving from the block above, we use blocking communication, so that we satisfy the dependency. We also need to send the first row to the halo of the processor above, but this is not needed until the next iteration, so we do this by sending the entire first row to the process above and storing it in his bottom halo at the end of every iteration.

For both methods, after each iteration, each process calculates the residual value locally and then performs the reduction in order to combine values across all processes to obtain the global residual using `MPI_AllReduce`. This allows us to check for convergence and terminate the iterations when the residual falls below a certain threshold.

CUDA

The computation of the Jacobi algorithm is a series of sums of elements of an array (or matrix) with no dependencies, so is perfect to be executed by a GPU, which can perform a huge number of simple operations in parallel.
In order to do the computation on the GPU, we first allocate memory (with `cudaMalloc`) on the GPU for the 2 matrix that we use for the Jacobi algorithm, and we copy them from the CPU (with `cudaMemcpy`).
For the computation, we modify the kernel function "gpu_Residual". We still use the Stencil as for the computation on the CPU, but instead of using a for loop, every thread computes the value of a single cell of the matrix and does the difference from the value of the previous iteration. Now we obtained an array containing all the residual values of the inner matrix (the full matrix, but without the halo), which we are going to sum together to obtain a global convergence criterion. The fastest way to do it is once again with the GPU. We pass the array to the function "gpu_Reduction": here, to take advantage of the shared memory within a block, we sum up all the values computed by threads of the same block, and we later copy it into an array where each cell corresponds to one block. If the total size of the values to sum is greater than the maximum size of a block (1024), in order to have a single global residual value we'll need to sum up the reducted values from each block, and the most natural way to do it is by using the function "gpu_Reduction" once again. This time the reduction can be computed by a single block, so we obtain a single residual value that we copy back to the CPU memory (with `cudaMemcpy`). Once the convergence criterion of the whole cycle is satisfied, we just copy the final matrix from GPU to CPU and, since we don't need it anymore, we free the memory allocated on the GPU.

2 Parallel Execution

OpenMP

Solver	Processors	Execution Time	Speedup
Jacobi	1	2.838	0.91
	2	1.799	1.44
	4	2.276	1.14
	8	2.280	1.14
Gauss-Seidel task dependency	1	5.004	0.90
	2	3.387	1.34
	4	3.079	1.47
	8	2.964	1.53
Gauss-Seidel do-across	1	4.576	0.99
	2	4.105	1.10
	4	3.318	1.36
	8	1.440	3.14

We obtained inconsistent scaling results for the Jacobi method, as the speedup increases when the code is run with 2 processors, but decreases once the code is run with 4 or 8 processors. This could be explained by the fact that we parallelized computation of every grid point, as opposed to block, and the small size of the problem.

For both parallelization techniques of the Gauss-Seidel method, however, we could observe strong scaling, as the speedup increases with the number of processors used.

In all cases, the execution of parallel code using only 1 processor results in a longer runtime, which can be explained by redundant overhead related to the creation and management of multiple threads scheduled on the same processor.

MPI

Solver	Processors	Execution Time	Speedup
Jacobi	1	2.391	1.08
	2	2.349	1.10
	4	509.055	0.01
Gauss-Seidel	1	5.312	0.85
	2	4.147	1.09
	4	422.671	0.01

For both methods, we obtained inconsistent results, as the speedup decreases substantially with the addition of new processors. We observed strong scaling for the execution with 1 and 2 processors for Jacobi method, and 2 processors for Gauss-Seidel method. The execution with 8 processors resulted in timeout after several minutes for both methods.

CUDA

The code correctly computes the matrix and produces an identical image compared to the one generated from sequential code (once we change the precision of the values from float to double), in the same number of iterations of the CPU, but approximately 20x faster:

```
Iterations :  25000
Resolution :  256
Num.  Heat sources :  2
1:  (0.00, 0.00) 1.00 2.50
2:  (0.50, 1.00) 1.00 2.50
Execution on CPU (sequential)
-----------------------------
Time on CPU in ms.= 9032.824219 (11.463 GFlop =>1269.03
MFlop/s)
Convergence to residual=0.000050:  15901 iterations
Execution on GPU
----------------
Time on GPU in ms.  = 420.149963 (11.463 GFlop => 27283.04
MFlop/s)
Convergence to residual=0.000050:  15901 iterations
```