

# **CPDS Laboratory Assignment 2: Solving the Heat Equation using several Parallel Programming Modes**

**Pau Adell Raventós: cpds1211  
Jaya García Fernández: cpds1216**

Master in Innovation and Research in Informatics (MIRI)

**Concurrency, Parallelism and Distributed Systems**

**Facultat d'Informàtica de Barcelona (FIB)**

**Contact:** [pau.adell@estudiantat.upc.edu](mailto:pau.adell@estudiantat.upc.edu) and [jaya.garcia@estudiantat.upc.edu](mailto:jaya.garcia@estudiantat.upc.edu)

2024-2025 Q1

**10/12/2024**

## Assignment 2

---

### 1 Introduction

The Heat Equation is a mathematical model that describes the diffusion of heat in a solid body. Due to its geometrical properties, many of the solvers that simulate this process implement parallelization techniques. In this assignment we will explore two type of solvers, the *Jacobi* and *Gauss-Seidel* each having different numerical properties. See in Figure 3 how the output of the simulation of the Heat equations change for each method. The goal is to parallelize the code and obtain speed-up over the sequential code with three different parallelization techniques: OpenMP, MPI and CUDA.

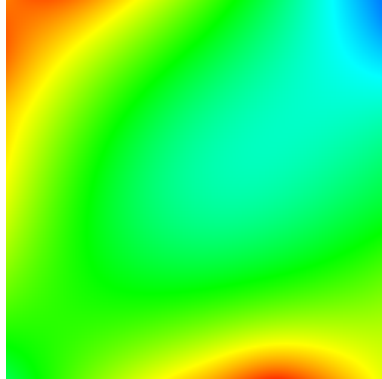


Figure 1: Jacobi Method

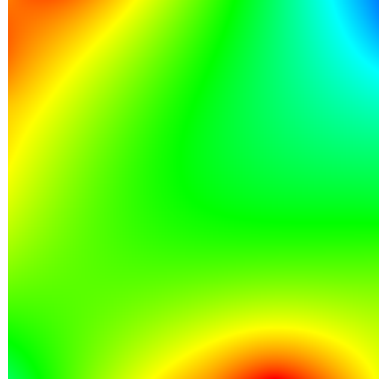


Figure 2: Gauss-Seidel Method

Figure 3: Result of the Heat equation simulation on a 2D plane with two heat sources.

Lets start with a brief description of how these two methods work in order to later understand the parallelization applied.

#### 1.1 Jacobi

We will explain it based on the sequential code and assuming the image is squared as we will divide the computation in equal blocks of size `NumberBlocks`. Here is the pseudo-code for the Jacobi algorithm:

## Assignment 2

---

### Algorithm 1 Jacobi Relaxation Method

---

**Input:**

- $u$  – Pointer to original 2D grid represented as 1D array
- $utmp$  – Pointer to the updated 2D grid represented as 1D array
- $size_x$  – Number of rows in the grid
- $size_y$  – Number of columns in the grid

**Output:**  $sum$  - total squared difference between values in  $u$  and new calculated values in  $utmp$

```
1:  $diff \leftarrow 0$ 
2:  $sum \leftarrow 0$ 
3:  $NB \leftarrow 8$ 
4:  $NBlocksX \leftarrow NB$ 
5:  $BlockSizeX \leftarrow size_x / NumberBlocksX$ 
6:  $NBlocksY \leftarrow NB$ 
7:  $BlockSizeY \leftarrow size_y / NumberBlocksY$ 
8: for all  $iblock \in [NumberBlocksX]$  do
9:   for all  $jblock \in [NumberBlocksY]$  do
10:    for all  $i \in BlockSizeX$  do
11:      for all  $j \in BlockSizeY$  do
12:         $Left \leftarrow u[i \cdot size_y + (j - 1)]$ 
13:         $Right \leftarrow u[i \cdot size_y + (j + 1)]$ 
14:         $Upper \leftarrow u[(i - 1) \cdot size_y + j]$ 
15:         $Lower \leftarrow u[(i + 1) \cdot size_y + j]$ 
16:         $utmp[i \cdot size_y + j] \leftarrow \frac{(Left + Right + Upper + Lower)}{4}$ 
17:         $diff \leftarrow utmp[i \cdot size_y + j] - u[i \cdot size_y + j]$ 
18:         $sum += sum + diff \cdot diff$ 
19:      end for
20:    end for
21:  end for
22: end for
23: return  $sum$ 
```

---

Note that as this is pseudo-code we have not added how should the " $i$  inside of  $iblock$ " be computed and it is important to understand that it will never have values that cannot access up/down/left/right values. Meaning that it will always start at position  $[1,1]$  of the first block as  $[0,0]$  does not have up and left values (and the same applies to all edges). Having explained this, it is clear that Jacobi's strategy aims to access the neighboring values and apply a sum of these in a relaxed way (by simply keeping 25%).

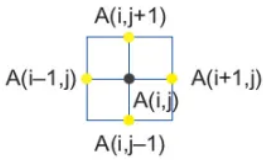

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Figure 4: Jacobi value computation

---

## Assignment 2

---

Most importantly, it keeps this value in a new updated grid, and for each position we compute the difference between the last and updated, and we keep that information as output. The total sum that we will have computed once we have finished the iteration will be helpful in being able to execute this method until the difference between values reaches a threshold that we consider to be sufficient. Clearly, the first few passes will make a big difference, but as we make more passes, the difference is reduced to a point where it is no longer relevant.

From this we can conclude that the method is embarrassingly parallel as we do not depend on previous values (we always use the origin matrix) and keep the new ones in a separated grid, which and can be heavily parallelized.

### 1.2 Gauss-Seidel

Now let us look at the sequential Gauss-Seidel algorithm. We will see that the computation of values is similar but with a different strategy, meaning we get different results compared to Jacobi and is simply another computational approach.

---

#### Algorithm 2 Gauss-Seidel Relaxation Method

---

**Input:**

- **u** – Pointer to original 2D grid represented as 1D array
- **size<sub>x</sub>** – Number of rows in the grid
- **size<sub>y</sub>** – Number of columns in the grid

**Output:** **sum** - total squared difference between the old values in **u** and the newly calculated values in the same grid

```
1: diff ← 0
2: sum ← 0
3: NB ← 8
4: NBBlocksX ← NB
5: BlocksSizeX ← sizex / NumberBlocksX
6: NBBlocksY ← NB
7: BlocksSizeY ← sizey / NumberBlocksY
8: for all iblock ∈ [NumberBlocksX] do
9:   for all jblock ∈ [NumberBlocksY] do
10:    for all i ∈ BlockSizeX do
11:      for all j ∈ BlockSizeY do
12:        Left ← u[i · sizey + (j - 1)]
13:        Right ← u[i · sizey + (j + 1)]
14:        Upper ← u[(i - 1) · sizey + j]
15:        Lower ← u[(i + 1) · sizey + j]
16:        unew ← (Left + Right + Upper + Lower) / 4
17:        diff ← unew - u[i · sizey + j]
18:        sum += sum + diff · diff
19:        u[i · sizey + j] ← unew
20:      end for
21:    end for
22:  end for
23: end for
24: return sum
```

---

---

## Assignment 2

---

The main difference that we can see is that we no longer use another 2D grid pointer to store the information. This means that if we update the position we have just calculated, the next variable that has that as a neighbor will use the updated value instead of the original one. This implies that the order in which the values are computed is relevant, and therefore the parallelization is not as trivial as the Jacobian method.

In fact, for the parallelization to be correct, one must wait for the value of the upper neighbor and the one to the left to not get wrong results. We have to follow a wavefront pattern parallelization as the one shown in Figure ??.

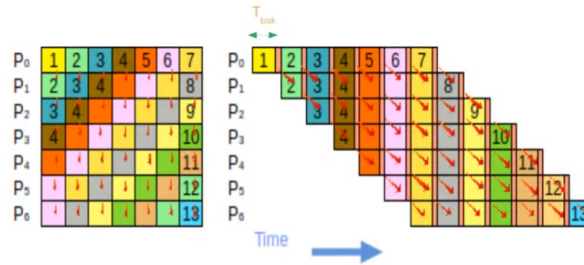


Figure 5: Wavefront pattern parallelization

## 2 Parallelization

Having seen the two methods used in this assignment, we now delve into the implementation of each parallelization technique.

### 2.1 OpenMP

#### Jacobi

As said before, as Jacobi method does not have any dependencies and uses a separated matrix to output the results, we can organize the threads in any way we want. It is not only a matter of preference how we organize the threads as doing it by rows, columns, and blocks will have different performance due to access data positions. It will not be the same accessing rows positions or columns as they will have non-continuous access in memory. In this case, we have chosen to follow the block structure as this is how the sequential code was given.

There is still one thing we have to take care of if we don't want to have problems with the parallelization. If we have different threads trying to modify our sum variable, we will need to synchronize in a way to not overwrite or miss access the variable.

This is a way of doing it with OpenMP:

```
1  #pragma omp parallel for collapse(2) reduction(+:sum) private(diff)
2  for (int ii=0; ii<nbx; ii++)
3      for (int jj=0; jj<nby; jj++)
4      ...
```

The OpenMP directives do as follows:

1. **omp parallel for:** tells to parallelize the loop that follows divided among the multiple threads to execute currently.

---

## Assignment 2

---

2. **collapse(2)**: Combines the two nested loops into a single iteration to distribute across threads (threads do a whole block).
3. **reduction(+:sum)**: specify **sum** as a reduction variable meaning each thread will maintain a private copy of **sum** during its execution and at the end of its parallel region all threads will combine its value with the operator **+**.
4. **private(diff)**: makes the variable **diff** private to each thread so each thread computing its difference won't affect the other threads (as it is a variable initialized before parallelization).

### Gauss-Seidel

As a wavefront parallelization method has to be chosen in this case due to the dependencies, we have tried using two OpenMP alternatives.

The first that we will be showing is using **explicit tasks with dependencies** (task + depend (in, out, inout)):

```
1  #pragma omp parallel
2  #pragma omp single
3  for (int ii=0; ii<nbx; ii++)
4      for (int jj=0; jj<nby; jj++) {
5          #pragma omp task depend(in: u[(ii-1)*sizey+jj], u[ii*sizey+(jj-1)]) \
6              depend(out: u[ii*sizey+jj]) \
7              private(diff,unew)
8              {
9                  double local_sum = 0.0;
10                 for (int i=1+ii*bx; i<=min((ii+1)*bx, sizex-2); i++) {
11                     for (int j=1+jj*by; j<=min((jj+1)*by, sizey-2); j++) {
12                         unew= 0.25 * ( u[ i*sizey + (j-1) ]+
13                             u[ i*sizey + (j+1) ]+
14                             u[ (i-1)*sizey + j ]+
15                             u[ (i+1)*sizey + j ] );
16                         diff = unew - u[i*sizey+ j];
17                         local_sum += diff * diff;
18                         u[i*sizey+j]=unew;
19                     }
20                 }
21                 #pragma omp atomic
22                 sum += local_sum;
23             }
24 }
```

The OpenMP directives do as follows:

- **omp parallel**: starts a parallel region. All threads are initialized and participate in the work inside of the region.
- **omp single**: ensures that the for loop defining tasks is executed by only one thread. However the tasks themselves are distributed among threads.
- **task directive**: a task is created for each block and we have added some dependencies (**depend**) and private (**private**) variables to it.
- **depend(in: u[(ii-1)\*sizey+jj], u[ii\*sizey+(jj-1)])**: A task can only start if the blocks above and to the left have finished processing.

## Assignment 2

---

- **depend(out: u[ii\*sizey+jj]:** marks that this task writes to the block preventing other tasks from interfering until it is completed.
- **private(diff, unew):** each task gets its private variable **diff** and **unew** ensuring thread safety.
- **omp atomic:** ensures that updates to sum from different tasks are thread-safe.

In summary, the parallelization strategy is the next; we first divide the grid into blocks (defined before creating the tasks so that it can be used as depend). We respect data dependencies, meaning that blocks will wait for neighboring blocks to finish before processing their own block. Each block computes its values sequentially and using private variables to avoid race conditions. Once the block is finished, we safely add the whole block difference to our total sum.

And the second way is using a **do cross** (ordered + depend (sink, source)):

```
1  #pragma omp parallel reduction(+:sum) private(unew, diff)
2  {
3      #pragma omp for ordered(2)
4      for (int ii=0; ii<nbx; ii++){
5          for (int jj=0; jj<nby; jj++) {
6              #pragma omp ordered depend(sink: ii-1, jj) depend(sink: ii,jj-1)
7              for (int i=1+ii*bx; i<=min((ii+1)*bx, sizex-2); i++) {
8                  for (int j=1+jj*by; j<=min((jj+1)*by, sizey-2); j++) {
9                      unew= 0.25 * ( u[ i*sizey + (j-1) ]+
10                      u[ i*sizey + (j+1) ]+
11                      u[ (i-1)*sizey + j ]+
12                      u[ (i+1)*sizey + j ]);
13                      diff = unew - u[i*sizey+ j];
14                      sum += diff * diff;
15                      u[i*sizey+j]=unew;
16                  }
17              }
18              #pragma omp ordered depend(source)
19          }
20      }
21  }
```

- **omp parallel:** starts a parallel region. All threads are initialized and participate in the work inside of the region.
- **reduction(+:sum):** specify **sum** as a reduction variable meaning each thread will maintain a private copy of sum during its execution and at the end of its parallel region all threads will combine its value with the operator +.
- **private(diff):** makes the variable **diff** private to each thread so each thread computing its difference won't affect the other threads (as it is a variable initialized before parallelization).
- **for ordered(2):** indicates that the loop nest has two levels and the execution order between iterations is constrained by the dependencies specified with **#pragma omp ordered**
- **ordered depend(sink: ii-1, jj) depend(sink: ii, jj-1):** manages the dependencies between blocks, specifying that it depends on the completion of block above (b[ii-1][jj]) and left (b[ii][jj-1]).
- **ordered depend(source):** indicate that this block's results are now available to other blocks that depend on it.

---

## Assignment 2

---

In summary, it follows the same idea as the explicit task with dependencies, but in this case we use `#pragma omp ordered depend` directives. We would consider using `ordered` as it is simpler as it is tied to the iteration space of the nested loops, and not having to use `#pragma omp parallel` and `#pragma omp single` even though it is more general and flexible.

### Results

Here we can find the results obtained for the different executions of our OpenMP code. Table 1 contains the execution time of the sequential code for the two methods. Tables 2, 3 and 4 contain the results of the parallelized code with different number of threads and resolution sizes. Note that all the speed-up values have been computed using the results in Table 1. The reason for this is that we consider important to see the overhead that OpenMP can generate even if we only use one thread.

Table 1: Execution time results in seconds of the sequential versions of the code (own creation).

Resolution	Jacobi	Gauss-Seidel
256	2,6	4,67
512	17,78	40
1024	87,49	168,03

Table 2: Execution results with 256x256 resolution of the OpenMP parallelization (own creation).

Threads	Jacobi		Gauss Seidel (tasks)		Gauss Seidel (do-across)	
	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>
1	2,72	0,96	5,4	0,86	4,8	0,97
2	2,4	1,08	5,04	0,93	4,56	1,02
4	2,78	0,94	5,73	0,82	3,53	1,32
8	2,47	1,05	3,15	1,48	1,51	3,09
16	3,08	0,84	3,98	1,17	1,62	2,88

Table 3: Execution results with 512x512 resolution of the OpenMP parallelization (own creation).

Threads	Jacobi		Gauss Seidel (tasks)		Gauss Seidel (do-across)	
	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>
1	18,71	0,95	41,02	0,98	40,27	0,99
2	12,46	1,43	29,58	1,35	37,98	1,05
4	19,43	0,92	18,82	2,13	27,96	1,43
8	15,67	1,13	14,29	2,80	10,98	3,64
16	14,29	1,24	14,57	2,75	11,25	3,56

Table 4: Execution results with 1024x1024 resolution of the OpenMP parallelization (own creation).

Threads	Jacobi		Gauss Seidel (tasks)		Gauss Seidel (do-across)	
	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>
1	90,87	0,97	174,35	0,96	168,64	1,00
2	58,24	1,51	96,66	1,74	159,77	1,05
4	57,37	1,53	64,29	2,61	118,11	1,42
8	54,75	1,61	50,74	3,31	43,61	3,85
16	52,36	1,68	51,6	3,26	44,53	3,77



---

## Assignment 2

---

In these results we see how we get a speed-up with most executions, although the change is more significant the larger the datasets. This is the case of the speed-up achieved with 1024 resolution compared to the one with 256.

### 2.2 MPI

The idea here is similar to the one in OpenMP: we first split the matrix equally between the workers, send them their corresponding part and let them make their computation. Once that is done, each worker returns their computed part to the master. Our idea is to use blocking communications to share the necessary data between the threads. Our hypothesis is that using blocking communication won't be as inefficient as there won't be many data transfers, in comparison to using non-blocking transmissions that will make the code slower with the synchronization requirements.

The code works as follows, first the master sends all the necessary parameters, waits for the workers to finish their computation and collects the corresponding results and aggregates them into its matrix. This can be seen in the following snippets of code. In the first one, the master sends all the parameters to the workers and in the second one the workers send their result back to the master.

```
1 // MASTER CODE
2   np = param.resolution + 2;
3   int rows = (param.resolution/numprocs) + 2;
4   int size_rows = rows*np;
5
6   // send to workers the necessary data to perform computation
7   for (int i=1; i<numprocs; i++) {
8       MPI_Send(&param.maxiter, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
9       MPI_Send(&param.resolution, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
10      MPI_Send(&param.algorithm, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
11      unsigned int offset = np*i*(rows-2);
12      MPI_Send(&param.u[offset], size_rows, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
13      MPI_Send(&param.uhelp[offset], size_rows, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
14  }
15  ...
16  // Collect the results from the workers
17  for (int i = 1; i < numprocs; i++) {
18      unsigned int offset = np*i*(rows-2) + np;
19      MPI_Recv(&param.u[offset], np*(rows-2), MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
20             MPI_STATUS_IGNORE);
21  }
22  ...
23 // WORKER CODE
24   MPI_Send(&u[np], (rows-2)*np, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

Notice that the resolution and the number of processors is assumed to be a power of 2. This is not incorrect but it must be taken into account. Also see that this code is shared by both the Jacobi and the Gauss-Seidel method. The difference between the methods resides in the communication at each iteration of the computations.

As a last note, each thread computes its own value for the residual, hence we need to take a consensus on when to finish the iterations. To do this, we make a reduction on the residual value and make this value available to all the threads. The reduction is done by summing up all the residual values. This can be seen in the following pseudo-code. This is done at each iteration of the algorithm. The `MPI_Allreduce` method is the same as making a simple `MPI_Reduce` and then an `MPI_Broadcast` to all the workers. The `MPI_IN_PLACE` directive is used to indicate the value of the residual variable in the corresponding thread.

## Assignment 2

---

```
1 // MASTER AND WORKER CODE
2 MPI_Allreduce(MPI_IN_PLACE, &residual, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

### Jacobi

As explained in previous sections, the given implementation of the Jacobi method uses an auxiliary input matrix to compute the results of each cell. Therefore, we can parallelize almost all the main work and let each worker compute their part of the matrix individually. For this, the workers must communicate between them, to obtain the previous and next rows of their block of rows. The communication between the workers can be seen in the following snippet. Notice that the master in this part would only send and receive rows from the first worker.

```
1 // WORKER CODE
2 MPI_Send(&u[np], np, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD);
3 MPI_Recv(&u[0], np, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
4
5 if (myid < numprocs-1) {
6     MPI_Send(&u[np*(rows-2)], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD);
7     MPI_Recv(&u[np*(rows-1)], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD,
8             MPI_STATUS_IGNORE);
9 }
10 residual = relax_jacobi(u, uhelp, rows, np);
```

### Gauss-Seidel

Regarding the Gauss-Seidel method, we cannot do the same as with the Jacobi as now we don't have an auxiliary matrix and the order of computation matters, therefore the first values computed must be sent to the next threads so that they can use them correctly for their computation. Now at each iteration each worker only sends its last row to the previous worker and receives the last row from the next worker, except the master that only receives from the first worker. This is done after the relaxation of the matrix of the corresponding thread is done. See the following snippet for a more clear picture.

```
1 // MASTER CODE
2 residual = relax_gauss(param.u, rows, np);
3 if (numprocs > 1)
4     MPI_Recv(&param.u[np*(rows-1)], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD,
5             MPI_STATUS_IGNORE);
6
7 // WORKER CODE
8 residual = relax_gauss(u, rows, np);
9 MPI_Send(&u[np], np, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD);
10 if (myid < numprocs-1)
11     MPI_Recv(&u[np*(rows-1)], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD,
12             MPI_STATUS_IGNORE);
```

Now we also need to add some communication inside the `relax_gauss`. Before getting into the actual computation, we need to have all the necessary data available first. For that reason we first make an `MPI_Recv` from the previous worker with the necessary blocks for the computation. Notice that the master doesn't make these call. Once the worker has the appropriate data, we can make the computation and at the end, we send the computed values to the next worker. This is all shown in the following code.

## Assignment 2

```
1 // WORKER AND MASTER CODE
2 if (numprocs > 1 && myid > 0) {
3     for (int jj = 0; jj < nby; jj++)
4         MPI_Recv(&u[jj*by + 1], by, MPI_DOUBLE, myid - 1, 0, MPI_COMM_WORLD,
5                 MPI_STATUS_IGNORE);
6 }
7
8 for (int jj = 0; jj < nby; jj++) {
9
10    for (int i = 1; i < sizex - 1; i++) {
11        for (int j = 1 + jj*by; j <= min((jj+1)*by, sizey-2); j++) {
12            unew = 0.25* ( u[ i*sizey + (j-1) ]+ // left
13                          u[ i*sizey + (j+1) ]+ // right
14                          u[ (i-1)*sizey + j ]+ // top
15                          u[ (i+1)*sizey + j ]); // bottom
16            diff = unew - u[i*sizey + j];
17            sum += diff*diff;
18            u[i*sizey + j] = unew;
19        }
20    }
21
22    if (myid < numprocs - 1) {
23        int offset = (sizex - 2)*sizey + jj*by + 1;
24        MPI_Isend(&u[offset], by, MPI_DOUBLE, myid + 1, 0, MPI_COMM_WORLD, &request);
25    }
26 }
```

### Results

In this section we present the results obtained when running the simulations with the implemented code. Tables 5, 6 and 7 show the results on resolutions 256, 512 and 1024 respectively. It can be appreciated that the results are mostly bad, as they show that the code is not really scalable. For the Jacobi method we see some minor improvement on the speed-up with high resolutions using up to two threads. However, the Gauss-Seidel method doesn't seem to improve at all the code in any case. Notice how using more than two threads leads to awful results and really bad scalability. This probably happens due to the big overheads in the communication between workers. We could wonder now if a non-blocking communication would be better for this, but we would still need to add many synchronization mechanisms that would also slow the code. In conclusion, it seems that the bottleneck here is the communication process, which is not avoidable with this technique.

Table 5: Execution results with 256x256 resolution of the MPI parallelization (own creation).

Threads	Jacobi		Gauss-Seidel	
	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>
1	2,48	1,05	4,69	0,996
2	2,53	1,03	5,50	0,85
4	152,21	0,001	157,74	0,03

## Assignment 2

Table 6: Execution results with 512x512 resolution of the MPI parallelization (own creation).

Threads	Jacobi		Gauss-Seidel	
	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>
1	17,6	1,01	39,98	1,00
2	18,17	0,98	43,92	0,91
4	407,23	0,04	298,67	0,13

Table 7: Execution results with 1024x1024 resolution of the MPI parallelization (own creation).

Threads	Jacobi		Gauss-Seidel	
	<i>Time (s)</i>	<i>Speed-up</i>	<i>Time (s)</i>	<i>Speed-up</i>
1	88,73	0,99	171,40	0,98
2	77,45	1,14	180,01	0,93
4	267,74	0,33	420,40	0,40

### 2.3 CUDA

For the last part, where we used CUDA, we only had to implement the Jacobi method. We made two versions, one to start understanding CUDA where we only used one kernel call to compute the new heat values and then let the CPU in charge of computing the residual, and another where we make use of the reduction method.

#### 2.3.1 Simple Method

In this section we won't be discussing how we had to define all the variables in both the CPU and GPU with their corresponding size, we will only focus on the relevant details for the parallelization. The following pseudo-codes show how the kernel is used to solve the problem and how it is implemented, respectively.

```
1 // simple version
2 gpu_Heat<<<Grid,Block>>>(dev_u, dev_uhelp, np);
3 cudaDeviceSynchronize(); // Wait for compute device to finish.
4 cudaMemcpy(param.u, dev_u, imageSize, cudaMemcpyDeviceToHost);
5 cudaMemcpy(param.uhelp, dev_uhelp, imageSize, cudaMemcpyDeviceToHost);
6 residual = cpu_residual (param.u, param.uhelp, np, np);
```

```
1 __global__ void gpu_Heat (float *h, float *g, int N) {
2     int i = blockIdx.x * blockDim.x + threadIdx.x;
3     int j = blockIdx.y * blockDim.y + threadIdx.y;
4     if( i > 0 && i < (N-1) && j > 0 && j < (N-1) ) { // no edges
5         g[N * i + j] = 0.25f * (
6             h[N * (i + 1) + j] +
7             h[N * (i - 1) + j] +
8             h[N * i + j - 1] +
9             h[N * i + j + 1]);
10    }
11 }
```

The `blockIdx` and `threadIdx` variables, combined with `blockDim`, calculate the global indices  $i$  and  $j$  for each thread in the 2D grid of threads of the kernel call. This means we will use as many threads as threads per Block

## Assignment 2

---

(in this case being 8x8), times the number of blocks in the `Grid` which varies in size depending on the resolution. For a total of `Block · Block · Grid · Grid` threads which will be computing the new values and storing them in a new grid. This is a simple solution that gives more or less good results as we'll see later but this can be greatly improved.

### Reduction Method

The second method tries to improve the first one by parallelizing the calculation of the difference between matrices instead of letting the CPU compute it sequentially.

The idea is to compute the heat values as before, and now also compute the difference values ourselves in the same call saving them in their corresponding position. The next pseudo-code shows the new kernel to do so.

```
1 __global__ void gpu_Diff(float *u, float *utmp, float* diffs, int N) {
2     int i = (blockIdx.y * blockDim.y) + threadIdx.y;
3     int j = (blockIdx.x * blockDim.x) + threadIdx.x;
4
5     if (i > 0 && i < N-1 && j > 0 && j < N-1){
6         utmp[i*N+j] = 0.25 * (u[i*N + (j-1)] + // left
7                               u[i*N + (j+1)] + // right
8                               u[(i-1)*N + j] + // top
9                               u[(i+1)*N + j]); // bottom
10        diffs[i*N+j] = utmp[i*N+j] - u[i*N+j];
11        diffs[i*N+j] *= diffs[i*N+j];
12    }
13 }
```

Notice how we have only added a new variable that accesses its own position using iterators based on the thread being computed. But now we also want to be able to compute in a similar way using blocks this difference value and to be able to do the total sum of it. To do this we will use a kernel that can, in parallel, make a sum of the components of its block and store them in a variable.

```
1 __global__ void gpu_Heat_reduction(float *idata, float *odata, int N) {
2     extern __shared__ float sdata[];
3     unsigned int s;
4
5     unsigned int tid = threadIdx.x;
6     unsigned int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
7     unsigned int gridSize = blockDim.x * 2 * gridDim.x;
8     sdata[tid] = 0;
9     while (i < N) {
10         sdata[tid] += idata[i] + idata[i + blockDim.x];
11         i += gridSize;
12     }
13     __syncthreads();
14
15     for (s = blockDim.x / 2; s > 32; s >>= 1) {
16         if (tid < s)
17             sdata[tid] += sdata[tid + s];
18         __syncthreads();
19     }
20     if (tid < 32) {
21         volatile float *smem = sdata;
22         smem[tid] += smem[tid + 32];
23         smem[tid] += smem[tid + 16];
24     }
```

## Assignment 2

---

```
24     smem[tid] += smem[tid + 8];
25     smem[tid] += smem[tid + 4];
26     smem[tid] += smem[tid + 2];
27     smem[tid] += smem[tid + 1];
28 }
29
30 if (tid == 0)
31     odata[blockIdx.x] = sdata[0];
32 }
```

The main idea for this kernel is to distribute the data from `idata` between threads, summing the elements and applying a reduction within a block. The sum is done in a strided manner inside the shared memory, to reduce global memory access latency, processing two elements per thread for efficiency: `sdata[tid] += idata[i] + idata[i + blockDim.x];`.

Once the sum has been done and we have waited for all threads to finish, a reduction is performed in a pairwise manner ( $s \gg 1$ ). This strategy is fine but it can be improved if the reduction is unrolled manually for maximum efficiency, not needing explicit `_synchrthreads()` so we have done it for the reduced amount of 32 threads to do the reduction.

Having the kernel, now we only need to call it to compute the results. However, if we call it only once, we will have the sum of the blocks in a variable but we had more than one block, hence we would be reducing the problem to `numBlock` size. To avoid this, another call has to be done of that size in order to reduce the results of the blocks to only one final value.

```
1 // with reductions
2 gpu_Diff<<<Grid, Block>>>(dev_u, dev_uhelp, dev_diffs, np);
3 gpu_Heat_reduction<<<numBlocks, numThreads, numThreads*sizeof(float)>>>(dev_diffs, dev_red1, np * np);
4 gpu_Heat_reduction<<<1, numBlocks, numBlocks*sizeof(float)>>>(dev_red1, dev_red2, numBlocks);
5 cudaDeviceSynchronize(); // Wait for compute device to finish.
6 cudaMemcpy(res, dev_red2, sizeof(float)*numBlocks, cudaMemcpyDeviceToHost);
7 residual = res[0];
```

---

## Assignment 2

---

### 2.3.2 Results

Here we present the results of both versions of the CUDA implementations. Table 8 holds the values for the first method while Table 9 shows the results for the second method. See how the improvement is huge between the two implementations as we go from having speed-ups of approximately 2 times the sequential to speed-ups of 24, 114 and 300 times. This difference is more noticeable the bigger the resolution is. We conclude then, that the second method is far better than the first, and actually better than any parallelization we have come up with previously.

Table 8: Execution results with different resolution values using the CPU and GPU units with the simple method (own creation).

	Resolution 256	Resolution 512	Resolution 1024
<b>CPU</b>	8,74	50,03	210,46
<b>GPU</b>	4,54	25,05	109,80
<b>Speed-up</b>	1,93	1,99	1,91

Table 9: Execution results with different resolution values using the CPU and GPU units with the reduction method (own creation).

	Resolution 256	Resolution 512	Resolution 1024
<b>CPU</b>	8,74	49,0	209,60
<b>GPU</b>	0,36	0,43	0,70
<b>Speed-up</b>	24,1	113,95	299,40