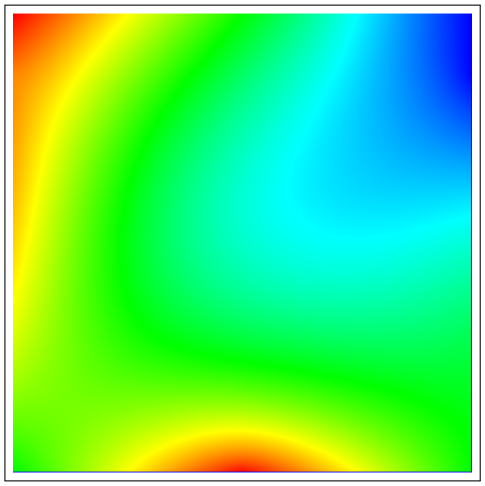
Solving the Heat Equation using several Parallel Programming Models

Lorién Nasarre
`lorien.nasarre@estudiantat.upc.edu`
cpds1117

Jakob Eberhardt
`jakob.eberhardt@estudiantat.upc.edu`
cpds1113

February 9, 2024



Contents

Listings

List of Figures

Setup

If nothing else is specified, the benchmarks included in this report were obtained with a problem size of 512x512 pixels. Apart from the do-across implementation discussed in section 1.5, the block number was fixed to 8. The included execution times were achieved on Boada. For some implementations, we also added benchmarks obtained in the local development environment.

Deliverable

1 Shared–memory parallelization with `OpenMP`

In this section, we present the results of our parallel implementations of the Jacobi and Gauß–Seidel solvers using `OMP`. This includes the sole parallel versions of the solvers as well as additional code parts that have been parallelized.

1.1 Jacobi

The Jacobi method employs an auxiliary matrix `utmp` which allows us to calculate the next iteration while retaining the current matrix in memory. Therefore, the threads can immediately access their data dependencies which allows us to fully exploit the potential for parallel execution. However, running the inner loops in parallel would be too fine-grain. As a result, we unify the iterations of the two outer loops using the `collapse` clause of `OMP`. Since we want to keep a working implementation of our convergence detection, we privatize the calculated difference for each thread and reduce it to a single value which represents the accumulated difference between the two iterations across the matrix. We also explicitly keep our inner loop variables `i` and `j` private. In a previous implementation, false cache invalidation of the read-only matrix `u` caused a significant impact on the program's performance when using more than two threads. To address this, we use the `firstprivate` pragma to pass a copy of the matrix to each thread. When being run in our local development environment, the implementation results in significant speedups which increase with more threads as shown in figure 1. However, the problem of falsely invalidated caches seems to persist on Boada, as can be seen in figure 2.

```
int i, j;
#pragma omp parallel for collapse(2) private(diff, i, j) reduction
    (+:sum) firstprivate(u)
```

Listing 1: This listing shows the single introduced `OMP` pragma to parallelize the Jacobi solver.

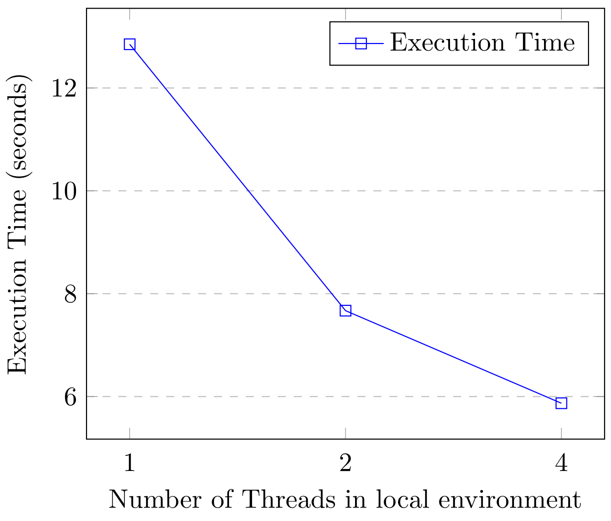


Figure 1: The execution time of the parallel Jacobi implementation with different amounts of threads running in the local development environment.

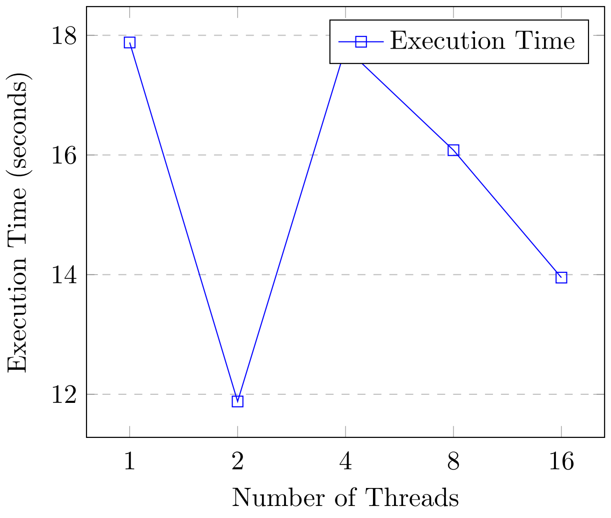


Figure 2: The execution time of the parallel Jacobi implementation with different amounts of threads. The increased execution time when utilizing more than two threads can probably be accounted to the loss of data locality as described in the previous section.

1.2 Parallel Jacobi and other code parts

Besides the actual solver implementations, the project bears many other opportunities to exploit parallelism. In this section, we will focus on `OMP` pragmas applied to the `misc.c` file to further enhance performance. The initialization for the iterative solver includes four loops dedicated to setting up the boundary columns with initial values. These loops can be run in parallel using pragmas similar to the one shown in the next listing.

```c
/* top row */
#pragma omp parallel for private(j, dist)
for( j=0; j<np; j++ )
```

Listing 2: Many loops that are part of the initialization can be parallelized using simple OMP pragmas.

Copying the `u` matrix into `uhelp` can also run completely parallel by applying the `OMP` pragma as shown in the listing below.

```c
// Copy u into uhelp
#pragma omp parallel for
for(int j=0; j<np; j++) {
    double *pu = param->u + j*np;
    double *putmp = param->uhelp + j*np;
    for(int i=0; i<np; i++) {
        *putmp++ = *pu++;
    }
}
```

Listing 3: Copying u into uhelp in parallel.

Lastly, we can accelerate the image generation of the program by parallelizing the search for the minimum and maximum value within the grid. However, we have to be careful when updating the values of the shared minimum and maximum after going parallel to avoid wrong results caused by race conditions among threads. Therefore, we define a critical zone within our parallel region as shown in the following code.

```c
#pragma omp critical
{
    if(local_max > max) max = local_max;
    if(local_min < min) min = local_min;
    }
```

Listing 4: Copying u into uhelp in parallel.

The following plots show the gained performance when combining the parallel solver and the previously described parallelizations. The differences are marginal, we have to consider the very small problem size of a resolution of 512x512 pixels.

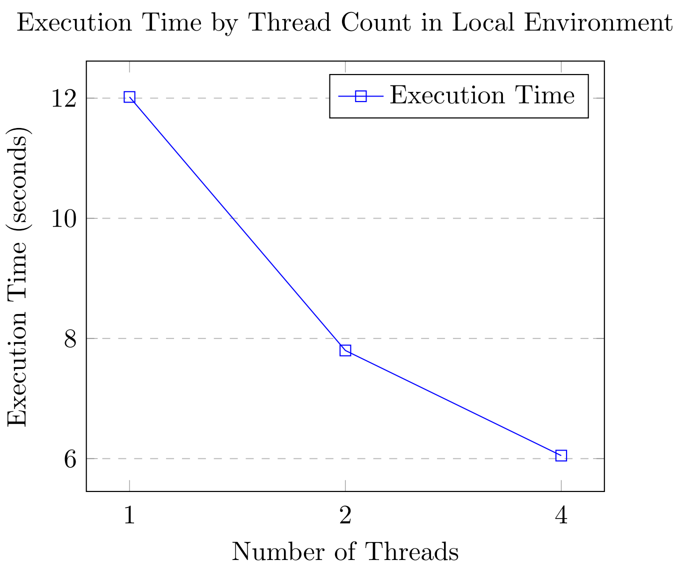


Figure 3: The execution time of the parallel Jacobi implementation in combination with other code regions running in parallel in a local environment

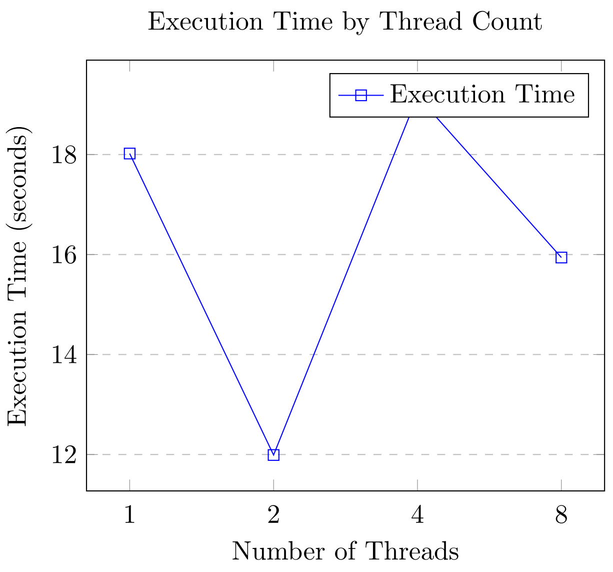


Figure 4: The execution time of the parallel Jacobi implementation in combination with other code regions running in parallel

1.3 Gauß–Seidel

Since the Gauß-Seidel solver only keeps one matrix in memory at a time, we have to consider the data dependencies in between iterations more carefully. In particular, we can only start computing a block once its left neighbor as well as the block above it is finished. The two following implementations respect these dependencies using an explicit task definition approach as well as employing the do-across pattern.

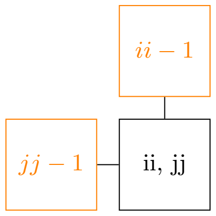


Figure 5: The values for the next iteration of a block depend on its top and left neighbor in the stencil

1.4 Gauß–Seidel with explicit tasks

```
#pragma omp parallel
#pragma omp single
for (int ii=0; ii<nbx; ii++){
    for (int jj=0; jj<nby; jj++){
   #pragma omp task ...
```

Listing 5: Start of the parallel region in the Gauß-Seidel implementation

This listing above shows a part of the parallel Gauß-Seidel implementation. The start of the parallel region is defined using the `parallel` pragma. One thread will create the tasks which will be picked up later. Each task corresponds to one block which has explicit dependencies which are explained in the following.

```
#pragma omp task private(diff, unew)
```

Listing 6: Private variables of the Gauß-Seidel implementation

Just like in the Jacobi implementation, we also want to keep private values for `diff` to detect convergence correctly. Additionally, we privatize `unew` to prevent the threads from constantly invalidating the cache by accessing one shared matrix. By giving each thread a private `unew`, we can prevent cache misses and exploit data locality.

```
depend (
    in:
    u[ii * bx* sizey+(jj - 1) * by],
    u[(ii - 1) * bx* sizey+jj * by]
)
```

Listing 7: Explicit input block dependencies within a iteration

Dependencies specified as `in` will block the respective task until its dependencies have been computed. The input dependencies among the blocks are defined in the pragma shown above. The first corresponds to the left neighbor block, and the second one (`ii - 1`) represents the top neighbor.

```
depend (out: u[ii*by*sizex+bx*jj])
```

Listing 8: Explicit declaration of the task output

In this case, we eventually unblock later tasks by defining the task output using `out` in a second `depend` section. Once a thread finishes the inner loops, the computed block will be available for other blocks which might depend on it.

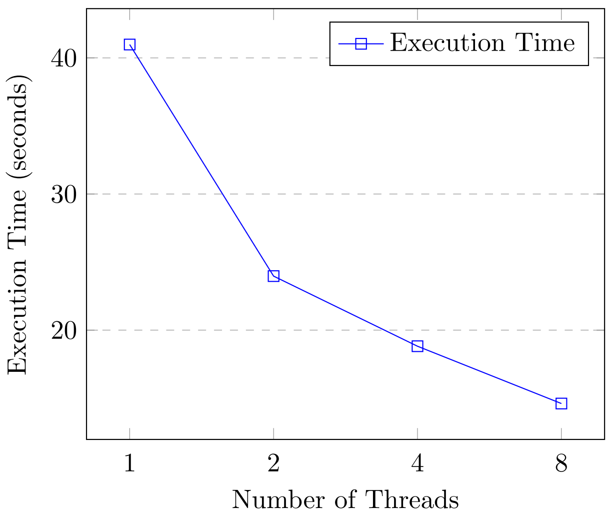


Figure 6: The execution time of the parallel Gauß-Seidel implementation using explicit tasks with dependencies.

1.5 Gauß–Seidel employing a do-across

Do-across is a parallelization pattern that can be applied to enable wave-front parallelism for data-dependent loop iterations.

```
#pragma omp parallel for private(diff, unew) ordered(2)
for (int ii=0; ii<nbx; ii++){
for (int jj=0; jj<nby; jj++){
```

Listing 9: Start of the parallel region of the do-across implementation

In addition to the pragma used to start the parallel region and define the private variables, we also use the `ordered(2)` clause to declare the iterations of the two following nested loops as ordered. As a result, the threads have to respect a specific order when computing a block. The respective order is defined using further pragmas which are explained in the following.

```
#pragma omp ordered depend(sink:ii-1,jj) depend(sink:ii,jj-1)
    {
    double local_sum=0;
    for (int i=1+ii*bx; i<=min((ii+1)*bx, sizex-2); i++){
    for (int j=1+jj*by; j<=min((jj+1)*by, sizey-2); j++)
```

Listing 10: Defining the dependencies in between iterations using sink clauses

Similar to the `in` clause seen in section 1.4, `sink` is used to define an input dependency. In this case, the clauses assure that for any given block, its left and top neighbors will have a lower index regarding the order in which the threads will pick up the blocks and run the inner loops.

```
#pragma omp atomic
sum+=local_sum;
     }
#pragma omp ordered depend(source)
}
```

Listing 11: Releasing a finished block as a potential source for other iterations that might depend on it.

The sum variable is used to detect convergence. Since it is shared among threads, `atomic` clause is used to prevent race conditions. The `depend(source)` marks the completion of a computed block. It is therefore located after the completion of the two inner loops and the summation of the global sum.

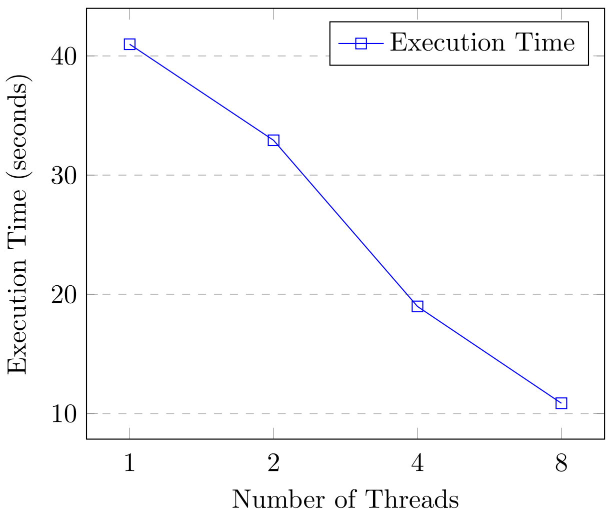


Figure 7: The execution time of the parallel Gauß-Seidel implementation using do-across and a dynamic amount of blocks.

1.6 Gauß–Seidel and other code parts

This section includes the results of the parallel Gauß-Seidel implementations together with the parallelized `misc.c` file. The do-across variant turned out to be very efficient when adapting the number of blocks to the respective amount of available threads. Therefore, a total execution time of 6.28 seconds with 16 threads could be achieved, as can be seen in figure 9.

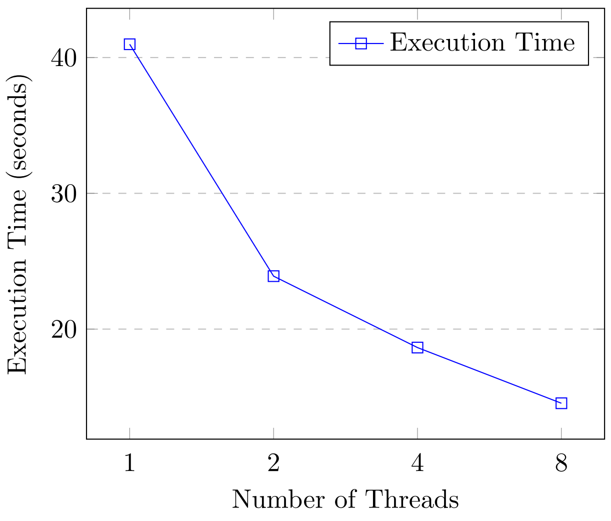


Figure 8: This plot shows the time needed to execute the parallelized Gauß-Seidel solver together with other code sections running in parallel.

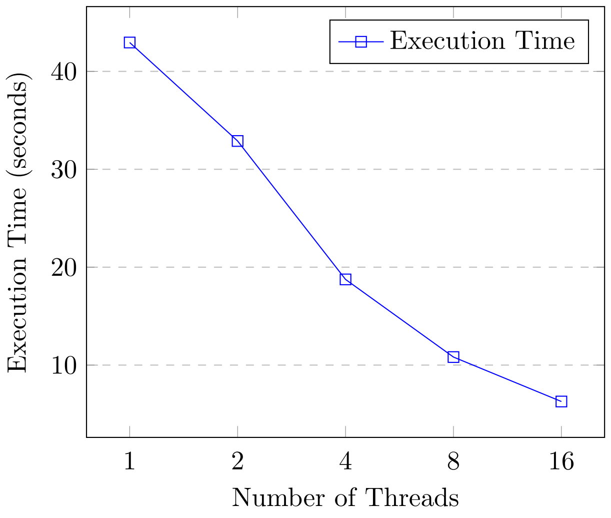


Figure 9: The execution time of the parallel Gauß-Seidel implementation using do-across schema. This implementation also includes other code parts running in parallel.

2 Message–passing parallelization with MPI

Now we will use MPI to parallelize the solving of the heat equation. The main difference is that now the communication between processes needs to be handled very carefully and correctly. First, we will start by talking about the Jacobi solver parallelization, and then continue with the Gauss-Seidel relaxation function.

2.1 Jacobi

In order to parallelize the Jacobi solver, we need to take into account some essential requirements that will make it execute correctly.

- **Data decomposition:** The master will be in charge of sending the initial data to each process. In this case, we are going to perform a decomposition by rows. The matrix will be broken down by rows and the information needed for each process to carry on the calculation will be sent to them before any calculations take place.

```
// Send data to workers
for (int i = 0; i < numprocs; i++) {
    if (i > 0) {
                                    .
        // Other parameters go here
        MPI_Send(&param.u.[i*np*rows], (rows+2) * (np),
    MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
        MPI_Send(&param.uhelpl[i*np*rows], (rows+2)*(np),
    MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
                                    .
        // Other parameters go here
    }
}
```

Listing 12: Example of a block of code that sends the data necessary for the workers to perform the calculations. In this case, we are sending the initial matrices.

The figure above shows the sending of the information needed to the worker processes. They receive it into a buffer that was allocated with the right amount of memory.

```
// Allocate memory for worker
double *u = calloc(sizeof(double), (rows+2) * (np));
double *uhelp = calloc(sizeof(double), (rows+2 ) * (np));
    if ((!u) || (!uhelp)) {
        fprintf(stderr, "Error: Cannot allocate memory\n");
        return 0;
}

// Fill initial values with data received from the master
MPI_Recv(&u[0], (rows + 2) * (np), MPI_DOUBLE, 0, 0,
    MPI_COMM_WORLD, &status);
MPI_Recv(&uhelp[0], (rows + 2) * (np), MPI_DOUBLE, 0, 0,
    MPI_COMM_WORLD, &status);
```

Listing 13: Worker processes receiving the information and store it into their local buffers.

Now that the initial data has been decomposed and distributed, we have to handle communications.

- **Communication between processes:** Since we are not using a shared memory paradigm anymore, we need to have constant communication between processes to generate a correct solution. The implementation that we will follow in this case is basically the same that was followed in the previous laboratory assignment. Each process will have two extra rows which we will call "ghost rows". The iteration will happen only in the interior points of each process, however, the ghost rows will store the data from previous and posterior processes for the interior points to have access to their local neighbors in the global matrix. To manage this communication in a more organized and clean manner, the function `exchange_ghosts_jacobi` was implemented to be executed after each calculation. The function goes as follows:

```
void exchange_ghosts_jacobi(double *u, int np, int rows, int
    rank, int numprocs) {
    MPI_Status status;
    MPI_Barrier(MPI_COMM_WORLD);
    // Send to the next rank your last interior row, if you
    are not the last process.
    if (rank < numprocs - 1) {
        MPI_Send(&u[(rows) * np], np, MPI_DOUBLE, rank + 1,
    0, MPI_COMM_WORLD); }
    // Receive from the previous member their last interior
    row, if you are not the first process.
    if (rank > 0) {
        MPI_Recv(u, np, MPI_DOUBLE, rank - 1, 0,
    MPI_COMM_WORLD, &status); }
    // Send the first row to the previous rank only if you are
     not rank 0.
    if (rank > 0) {
        MPI_Send(&u[np], np, MPI_DOUBLE, rank - 1, 1,
    MPI_COMM_WORLD); }
    // Receive the first row from the next rank only if you are not the last rank.
    if (rank < numprocs - 1) {
        MPI_Recv(&u[(rows + 1) * np], np, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD, &status); }
}
```

Listing 14: Communication handling in parallelization of Jacobi with MPI.

This function is called by `do_calculations_master` and `do_calculations_worker`, which are just functions that condense the loop and communications within the main function.

- **Global convergence:** Each process has a criterion to follow to achieve convergence, which we can call local convergence. However, there are a couple of problems that could arise if convergence is not handled in a global manner. In the case that one of the processes arrives at local convergence while using blocking communication, the program will run into a deadlock. This will happen since other processes will be waiting for the information from that process, which has exited the iteration loop and will no longer send information. Another possible problem is incorrect results. If the residuals are not reduced correctly or an inappropriate reduction function is used (for example using MPI_MAX instead of MPI_SUM), the results will be different from the ones computed sequentially. To obtain global convergence and achieve the desired results, the MPI primitive `MPI_Allreduce` was used. This way all processes can have access to the total residual values, and stop when these go below a certain threshold.

```
MPI_Allreduce(&residual, &global_residual, 1, MPI_DOUBLE,
    MPI_SUM, MPI_COMM_WORLD);
```

Listing 15: Reduction to a global residual.

When all of the iterations have stopped, the processes send back the data to the master process to display it. We will show the results obtained by performing the Jacobi iteration here.

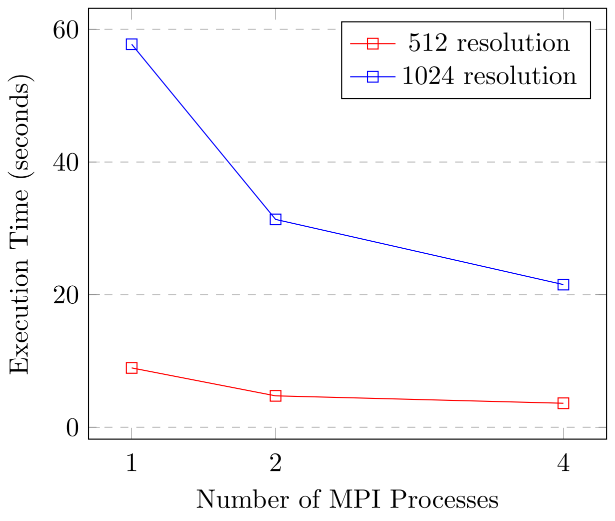


Figure 10: The execution time of the parallel Jacobi implementation using MPI processes for 512x512 resolution.

2.2 Gauß-Seidel

The Gauss-Seidel implementation follows a very similar pattern to the previous implementation. The difference is that now we have to take care of dependencies. Instead of solving each row independently and then exchanging ghost areas, each row will now be divided into blocks that will be calculated one after the other. For these blocks to start their iterations, they need the information from their top and left neighbors. Making the top left corner block (0,0) the first one to start performing calculations.

To minimize the amount of communication needed, which can increase the time of total execution, we are still dividing the data by rows and giving them to each process. Each process will execute its blocks in a for loop while waiting for the processes above to send their block. This ensures that all dependencies are met, and creates an iterative wavefront, which gets its name from how the calculations of the blocks expand across the matrix from one single point.

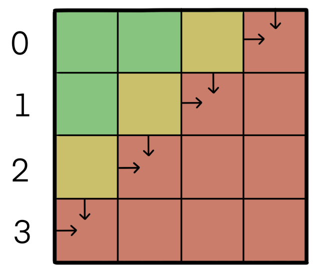


Figure 11: Example of data decomposition and wavefront iterations in MPI.

In the figure above we can observe how data is decomposed amongst the processes 0 to 3, and how the dependencies will be met. The green blocks have finished calculations and sent their information to the yellow blocks, which are performing their calculations. The red blocks will wait for the yellow blocks to finish and send their data. As we can see, the halo surrounding the matrix also provides the information necessary for the boundary blocks. In the code this is done like this:

```
case 2: // GAUSS
residual = 0.0;
for (int i = 0; i < numprocs; i++) {
    residual += relax_gauss(param->u, ghost_rows, np, numprocs, i);
    if (numprocs !=1){
        MPI_Isend(&param->u[(rows * np) + (i * block_size + 1)],
    block_size, MPI_DOUBLE, rank + 1, rank+1,MPI_COMM_WORLD,&
    r_master[i]);
}} break;
```

Listing 16: Code snippet of the master performing calculations

The master will always be capable of starting calculating since it's in the first row and has its dependencies covered by the halo. Therefore the master can:

1. Start the calculation.
2. Perform a non-blocking send which allows for the calculation to take place without interruptions, while also ensuring the data being sent is correct since it happened after the relaxation.

It is very important to add up all the residuals in the for loop using **+=**, since if this is not done, the value for the residual will be updated for each loop and greatly underestimated, making our iterations stop before they should.

The worker processes will follow this scheme:

```
case 2: // GAUSS
residual = 0.0;
for (int i = 0; i < numprocs; i++) {
    MPI_Irecv(&u[i * block_size + 1], block_size, MPI_DOUBLE,
    rank - 1, rank, MPI_COMM_WORLD, &r_recv[i]);
    MPI_Wait(&r_recv[i],&statuses[i]);
    residual += relax_gauss(u, ghost_rows, np, numprocs, i);

if (rank != numprocs - 1) {
    MPI_Isend(&u[(rows * np) + (i * block_size + 1)], block_size,
    MPI_DOUBLE, rank + 1, rank+1, MPI_COMM_WORLD,&r_send[i]);
}} break;
```

Listing 17: Code snippet of the worker performing calculations

The workers will, according to this code:

1. Receive the results of the ghost areas from their top neighbors. This communication is the only communication that needs a call to `MPI_Wait`, which will ensure that the workers get the information they need to calculate their block correctly.

2. Perform the relaxation.

3. Do a non-blocking send since they are guaranteed to send the correct information.

When executed locally, with 1,2 and 4 threads (limited by the local machine's capacity), the execution time depending on the number of MPI processes looks as follows:

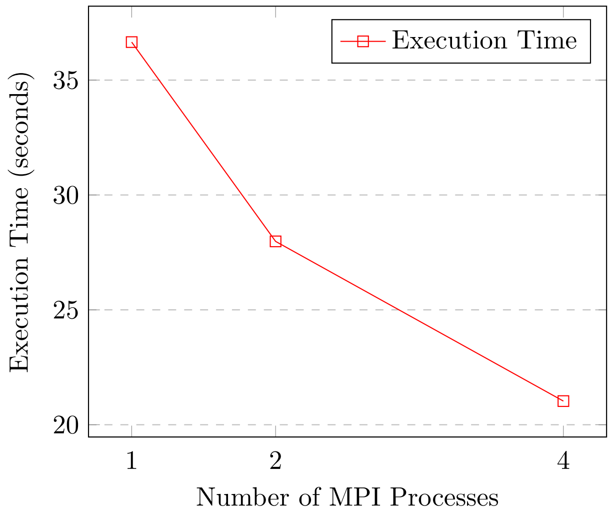


Figure 12: The execution time of the parallel Jacobi implementation using MPI processes locally, with a resolution of 512x512.

It is worth remarking on the **importance of waiting for the reception of data by performing an `MPI_Wait`** as we mentioned before. When this call was not implemented in the code, the results were much faster and appeared correct to the naked eye. However, when comparing these results to the ones obtained sequentially, it is clear that there are notable differences.

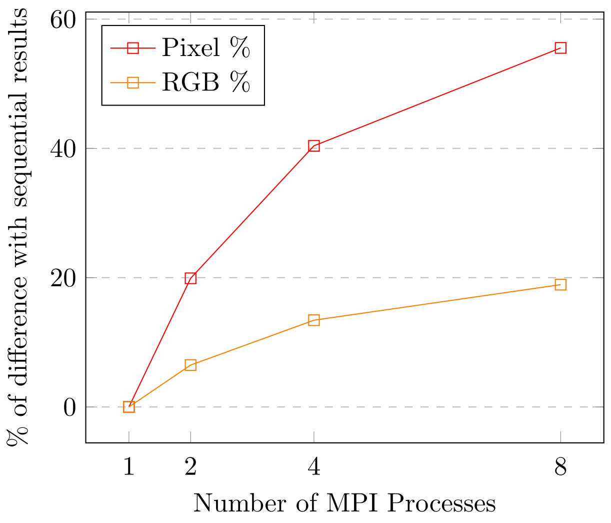


Figure 13: The difference in percentage with respect to the results calculated sequentially.

This is an interesting way to visualize how errors in communication and race conditions can propagate and generate great differences between the correct result and the obtained result. The RGB difference is always smaller, making it difficult to spot these mistakes.

3 Parallelization with CUDA

The parallelization with CUDA has proven to be by far the most efficient one as we were expecting. Because of the GPU's architecture, it has great potential to perform a large amount of calculations in parallel. The executed code has CPU and GPU components. Therefore, the same code will be executed and benchmarked in both of these. First of all, we needed to implement a kernel that runs on CUDA to calculate the values of each pixel through the Jacobi relaxation. After that, it would send those values back to the CPU where the residual would be computed to determine if we had achieved convergence.

The program needs to be prepared memory for the kernel before it is launched. The GPU needs to allocate memory for the arrays and different constants that will be used to perform calculations. It also needs the initial values of the arrays in order to start. The described steps can be seen in the following listing.

```
//CUDA MEMORY ALLOCATION
cudaMalloc((void**)&dev_u, sizeof(double)*(np*np));
cudaMalloc((void**)&dev_uhelp, sizeof(double)*(np*np));
//COPYING INITIAL VALUES FROM HOST TO DEVICE
cudaMemcpy(dev_u, param.u, sizeof(double)*(np*np),
    cudaMemcpyHostToDevice);
cudaMemcpy(dev_uhelp, param.uhelp, sizeof(double)*(np*np),
    cudaMemcpyHostToDevice);
```

Listing 18: Memory allocation and copying of initial values to device's memory.

We have followed a naming convention to specify where the variables we are using reside. We will be calling the CPU our host, and the GPU our device.

The provided code calculates the dimensions of the blocks and grids in which we are going to organize our threads. The respective kernel launch is shown in the next listing.

```
gpu_Heat<<<Grid,Block>>>(dev_u, dev_uhelp, np);
cudaDeviceSynchronize();  // Wait for compute device to finish.
```

Listing 19: Kernel launch within the CPU

We need to call `cudaDeviceSynchronize()` as a barrier to ensure that all processes have finished in the GPU before continuing in the CPU. The kernel is just a simple implementation of the same calculation that we have been doing in the CPU, applied to GPU indexing.

```
__global__ void gpu_Heat(double *h, double *g, int N) {
  int i = blockIdx.y* blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;

  if (i >0 && i < N -1 && j> 0 && j< N-1){
    h[i*N+j]= 0.25 * (g[i * N + (j-1) ]+  // left
                g[ i * N + (j+1) ]+  // right
                  g[ (i-1) * N + j ]+  // top
                  g[ (i+1) * N + j ]); // bottom }}
```

Listing 20: Code snippet of the worker performing calculations

Notice that no for loops are used since the operations are all being executed in parallel by each of the threads we indexed with the variables `i` and `j`.

Once this was done, we brought everything back to the CPU and performed computations there.

```
//COPY RESULTS FROM GPU TO CPU TO CALCULATE RESIDUAL
cudaMemcpy(param.u, dev_u, sizeof(double)*(np*np),
    cudaMemcpyDeviceToHost);
cudaMemcpy(param.uhelp, dev_uhelp, sizeof(double)*(np*np),
    cudaMemcpyDeviceToHost);
residual = cpu_residual (param.u, param.uhelp, np, np);
```

Listing 21: Copying the initial results into the device's memory.

Obviously, this is not a good implementation, since it produces a bottleneck effect that creates a considerable slowdown. We can observe this when we comment everything regarding the calculation of the residual out in the host, and benchmark the times.

Without bringing the residual to the CPU:
`Time on GPU in ms.  = 2078.523193 (18.022 GFlop => 8670.77 MFlop/s)`
Bringin the residual back to CPU:
`Time on GPU in ms.  =1599.249268 (18.022 Glop => 11269.29 MFlop/s)`

This can be fixed by implementing new Kernels that calculate an array that consists of the differences between `u` and `utmp`, and another kernel that reduces this array into a single float through adding operations. We actually will need two reducing kernels because of the way that reductions work in CUDA. We will be performing a tree-based reduction, which works on arrays that have a length of a power of 2, although there are workarounds to make it work with other array lengths as well. This reduction works by adding to each thread, another thread which is a distance of a power of 2 away. We call this distance the stride.

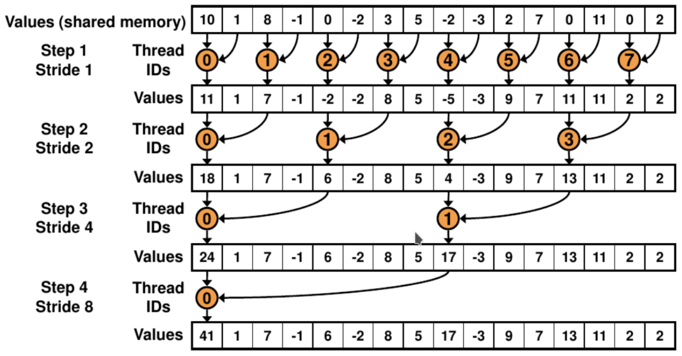


Figure 14: Tree-based reduction in CUDA

In order to do this, we use the following kernel:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
unsigned int gridSize = blockDim.x*2*gridDim.x;
  sdata[tid] = 0;
  while (i < N) {
    sdata[tid] += g_idata[i] + g_idata[i+blockDim.x];
    i += gridSize;}
  __syncthreads();
// Reduction of shared memory
for (s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
      sdata[tid] += sdata[tid + s];
    __syncthreads();

    // Operations in the last warp
    if (tid < 32) {
    volatile double *smem = sdata;
    smem[tid] += smem[tid + 32];
    smem[tid] += smem[tid + 16];
    smem[tid] += smem[tid + 8];
    smem[tid] += smem[tid + 4];
    smem[tid] += smem[tid + 2];
    smem[tid] += smem[tid + 1];
  }
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Listing 22: Code snippet from first reduction kernel.

Where we are using global indexing of threads (variable `i`), to add all the values that are one block away from each thread. We reduce that memory until our thread ID is no greater than 32. This is so that we operate within a warp, which has a capacity of 32 threads in NVIDIA architecture. Operating within a warp is very convenient since a warp is a group of threads that perform operations in lockstep, with no need to synchronize. Then the data is stored in an array in which each element is the reduction corresponding to each block. Once we have performed the first reduction, we call a second reduction kernel that will reduce the output of the first reduction. We can introduce the parameters as a one-dimensional array.

```
__global__ void finalReduceKernel(double *g_idata, double *g_odata,
     int N) {
    extern __shared__ double sdata[];
    unsigned int tid = threadIdx.x;
    // Load block sums from global memory to shared memory
    sdata[tid] = (tid < N) ? g_idata[tid] : 0;
    __syncthreads();
    // Perform final reduction in shared memory
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[0] = sdata[0];}
```

Listing 23: Code snippet from first reduction kernel.

Before we can finish the program and write out the image, we have to release the previously allocated GPU memory using the `cudaFree`.

```
cudaFree(dev_u);
cudaFree(dev_uhelp);
cudaFree(dev_residual);
cudaFree(dev_residuals_first);
cudaFree(dev_residuals_second);
cudaFree(dev_diff);
```

Listing 24: Releasing the GPU memory.

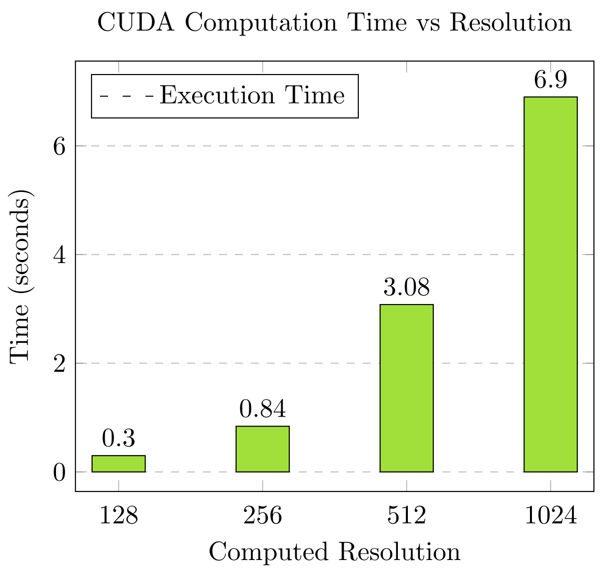


Figure 15: Execution time of CUDA operations as a function of the computed resolution.

The figure above shows the needed time to compute the relaxation on the GPU for the respective resolution of the image. By employing hardware acceleration using CUDA, we were able to significantly reduce the needed time to produce the image.

4 Summary

In this assignment, we analyzed the data dependencies of the heat equation, reasoned our data decomposition strategies, and applied three different techniques to compute the relaxation in parallel. We parallelized both the Jacobi and Gauß-Seidel solver in a shared–memory environment by employing OpenMP. In addition, we considered other aspects influencing performance, such as data locality and task scheduling. We utilized MPI to realize a message–passing parallelization for the Jacobi and Gauß-Seidel solver, both in a blocking and non-blocking manner. Lastly, we took advantage of hardware acceleration and used the CUDA framework to run the computation on a GPU.