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

Ignacio Encinas Rubio, Adrián Jiménez González
{ignacio.encinas,adrian.jimenez.g}.estudiantat.upc.edu

December 22, 2022

1 Introduction

Some changes have been added to the document after the feedback received with the first submission.

- Speedup evaluation and report: We have introduced tables with the speedup obtained for every Parallel Model and size of the problem. Also, a brief comment of the resulting values have been added to the section.
- Modification of NB / for loops on Jacobi: We have modified the structure of the for loops. The 4 loops have been reduced to 2 for loops. This change deletes the blocks that uses NB so we don't need to assign any value depending the number of threads to it.
- Gauss-Seidel missing communication: As commented in the mail, this communication was already introduced and no modification is needed.
- For loops in solvers: The utilization of 4 loops is not needed in all the solvers. Only is needed for Gauss in OpenMP. Jacobi only needs 2 loops reducing overhead and facilitating the maintenance.

2 Parallelization

In this section we will explain the steps we followed to achieve the parallelization of both solvers, Jacobi and Gauss-Seidel. In the following sections, we are going to show how the parallel codes have been done in each Parallel Programming Model for each solver.

2.1 OpenMP

2.1.1 Jacobi

Parallelizing the Jacobi solver with OpenMP just requires to modifying the *solver-omp.c* file. The *heat-omp.c* file does not need any modification. Jacobi does not have any data dependencies inside an iteration, it just requires that the iterations themselves are carried out in order, so the parallelization can be achieved by simply adding a `#pragma omp` as shown below at Listing 1.

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

Listing 1: OpenMP pragma for Jacobi parallelization

Jacobi does not require performing the computation in "blocks" because it doesn't have internal dependencies like Gauss-Seidel. So we will fuse both for loops with the `collapse(2)` clause. The next clause we use is `private(diff)` in which we specify that each thread will have its own copy of the diff variable. Last, we have the `reduction(+:sum)` specifying the way we want to reduce the values obtained by each thread. In this case, we want to sum (+) the values for each `sum` when we finish the parallel section in order to return the residual of the solver.

To improve the execution of the solver, we have modified the copy of the matrices after the computation of the solver to a swap of pointers, improving the performance of our code. See Section 3 for the results.

2.1.2 Gauss-Seidel

The parallelization of Gauss-Seidel solver using OpenMP is quite different. In this case, we need tasks to parallelize while respecting the dependencies of that solver.

Explicit tasks

We introduce a new proxy variable `block`[1], used to indicate the dependencies between tasks and mark whenever they're fulfilled. See Listing 2 for more details.

```
int b[nbx][nby];

#pragma omp parallel
#pragma omp single
{
  for (int ii=0; ii<nbx; ii++) {
    for (int jj=0; jj<nby; jj++) {
    #pragma omp task depend(in: b[ii-1][jj], b[ii][jj-1]) depend(out: b[ii][jj]) private(diff, unew)
      {
        double omp_sum = 0.0;
        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++) {
              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];
              omp_sum += diff * diff;
              u[i*sizey+j]=unew;
          }
        }
        #pragma omp atomic
        sum += omp_sum;
      }
    }
  }
}
```

Listing 2: OpenMP pragma for Gauss-Seidel parallelization

In this case, not only the `#pragmas` have been added. Some code needs to be changed in order to obtain the same behaviour as sequential code. First, as we have mentioned, we have introduced a new variable `block` to manage the dependencies on the `#pragma omp task`. Each task will depend on the "top" and "left" blocks, indicated in the `depend(in:var_list)` clause. The task produces `block[ii][jj]` with the `depend(out:var_list)` clause. Then, when the tasks of the left and the top from a block are finished, this task will be able to start, allowing us to exploit wavefront parallelism.

Also, we need to indicate that region as parallel with `#pragma omp parallel` to create the threads to execute that region of code. Also, we need to specify `#pragma omp single` to indicate the region only is done by one thread, who will create the tasks for the rest of threads.

Inside the task region, we have a race condition in the `sum` variable. In order to solve it, we create a new variable where we make the operations for each thread, and then with a `#pragma omp atomic` we avoid that critical section, and we obtain the residual value correctly.

[1]Shown as `b` in the listing

Do-across (Implicit tasks)

To implement Gauss with implicit tasks, we have used `#pragma omp ordered`. In Listing 3, we can see how the parallelization has been achieved.

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

                double omp_sum = 0.0;
                #pragma omp ordered depend(sink: ii-1, jj) depend(sink: ii , jj-1)
                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++) {
                    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];
                    omp_sum += diff * diff;
                    u[i*sizey+j]=unew;
                    }
                }
                #pragma omp ordered depend(source)

                #pragma omp atomic
                sum += omp_sum;
        }
    }
```

Listing 3: OpenMP pragma for Gauss-Seidel parallelization

First, we have an `#pragma omp for ordered(2)`, in which we specify the number of loops within the doacross nest.

Then, we need to specify the dependencies. We use `depend(sink:var_list)` and `depend(source)` to indicate when a dependency starts and what iteration it depends on, and where the code has completed the computation from the current iteration. The way we avoid the race condition is equal to Gauss-Seidel implementation on OpenMP with explicit tasks. We use a local variable to add the diff over the iterations and once finished we do an atomic operation adding to *sum*.

2.2 MPI

2.2.1 Jacobi

For the parallelization of Jacobi using MPI, we need to manage all the boundaries or halos from each of the nodes. First, we need to split the data between all the nodes we use. For this, we need to send the part of `param.u` and `param.uhelp` to the corresponding children node[2]. Assuming $rows \equiv 0 \pmod{\text{numprocs}}$: $rowsWorkers = rows/numprocs + 2$. The $+2$ corresponds to the first and last rows, that are part of the halo.

```
/* ROOT NODE (rank == 0) */
for (int i=0; i<numprocs; i++) {
  MPI_Send(&param.u[np*rowsWorkers*i], np*(rowsWorkers+2), MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
  MPI_Send(&param.uhelp[np*rowsWorkers*i], np*(rowsWorkers+2), MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
}

/* (rank != 0) */
MPI_Recv(&u[0], (rows+2)*(columns+2), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&uhelp[0], (rows+2)*(columns+2), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
```

Listing 4: Sending/Receiving initial data to all nodes

The root node sends inside a for loop to all the nodes. The children nodes only need to receive once as it is shown above.

Inside the iteration part, we need to send and receiving the halo from the contiguous nodes. This is different for 3 types of nodes:

- **Node 0**: only sends to and receives from the next node.
- **Intermediate nodes**: send to and receive from previous and next nodes.
- **Last node**: only sends and receives from previous node.

```
/* ROOT NODE (rank == 0) */

MPI_Send(&param.u[np*rowsWorkers], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD);
MPI_Recv(&param.u[np*(rowsWorkers+1)], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD, &status);

residual = relax_jacobi(param.u, param.uhelp, rowsWorkers+2, np);

/* (rank != 0) */

MPI_Send(&u[columns+2], columns + 2, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD);

MPI_Recv(&u[0], columns + 2, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD, &status);

if(myid != numprocs - 1){
  MPI_Send(&u[rows*(columns+2)], columns+2, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD);
  MPI_Recv(&u[(rows+1)*(columns + 2)], columns+2, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD, &status);
}

residual = relax_jacobi(u, uhelp, rows+2, np);
```

Listing 5: Communications between nodes inside the iterations

After obtaining the residual of the heat equation, we need to sum the value from all the nodes to use it as break condition of the loop, so all the nodes could finish at the same time. This part is identical in all the nodes.

```
double res;
MPI_Allreduce(&residual, &res, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
residual = res;
```

Listing 6: Allreduce for residual value

Once we have finished the iterative part, we need to send each part of the image computed in the children to the parent.

```
/* ROOT NODE (rank == 0) */

for(int i = 1; i < numprocs; i++){
  MPI_Recv(&param.u[np*(rowsWorkers*i + 1)], np*(rowsWorkers), MPI_DOUBLE, i, i, MPI_COMM_WORLD, &status);
}
/* (rank != 0) */

MPI_Send(&u[np], np*rows, MPI_DOUBLE, 0, myid, MPI_COMM_WORLD);}

residual = relax_jacobi(u, uhelp, rows+2, np);
```

Listing 7: Communication of the computed image to the parent

[2]Some not modified code has been deleted from the listings in order to have the report more clear

2.2.2 Gauss-Seidel

For the parallelization of this solver, most of the work have been already done for Jacobi method. Parts like splitting the data between nodes, reduction of the residual for breaking condition and merging the computed image of children nodes in the parent node are common to Gauss-Seidel. Only the communications between nodes in the iterative part are different, as it is shown in Listing 8.

```
/* ROOT NODE (rank == 0) */
residual = relax_gauss(param.u, rowsWorkers+2, np, myid, numprocs);

MPI_Recv(&param.u[(rowsWorkers + 1)*(np)], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD, &status);

/* (rank != 0) */

residual = relax_gauss(u, rows + 2, np, myid, numprocs);

MPI_Send(&u[np], np, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD);
if(myid != numprocs - 1){
  MPI_Recv(&u[(rows+1)*np], np, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD, &status);
}

residual = relax_jacobi(u, uhelp, rows+2, np);
```

Listing 8: Communications between nodes Gauss-Seidel

For Gauss-Seidel we have also different types of nodes in order to send or receive between iterations. This communications only appears when the Gauss solver has finished with all the blocks of its node:

- **Node 0**: only receives the lower boundary from next node.
- **Intermediate nodes**: receives from next node the lower boundary and sends to the previous node the first computed row.
- **Last node**: only sends to the previous node the first computed row.

The communications between nodes after a block is computed are done inside the Gauss-Seidel's kernel. We use the outer for loops of the kernel that compute the solver in blocks to send and receive the necessary information for each block. To be able to differentiate the behaviour of the different nodes that we have, *relax_gauss* definition has been modified to bring the solver the context of which node is executing and it make the different communications. We also need to initialize `&status` variable for the communications. See Listing 9.

```
double relax_gauss(double *u, unsigned sizex, unsigned sizey, int rank, int numprocs) //rank and numprocs
     passed as parameters
{
MPI_Status status;
for (int ii=0; ii<nbx; ii++)
   for (int jj=0; jj<nby; jj++) {
     if(ii == 0 && rank != 0){
       MPI_Recv(&u[jj * by], by, MPI_DOUBLE, rank-1, jj, MPI_COMM_WORLD, &status);
     }
     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++) {
         /* COMPUTE EQUATION */
       }
     if(ii == nbx - 1 && rank != numprocs - 1) {
       MPI_Send(&u[((sizex - 2) * sizey) + jj*by], by, MPI_DOUBLE, rank+1, jj, MPI_COMM_WORLD);
     }
   }
}
```

Listing 9: Communications between nodes Gauss-Seidel inside kernel

As we can see, we use ii to differentiate if the block is in the top side so the block must receive the upper boundary from previous node in case its rank is different to 0. Once we receive this data, we can compute the equation.

Blocks of the same node do not need to send or receive data between them as they are in shared memory. The blocks of the bottom side (`ii == nbx-1`) must send their last computed row to the next node in order to the upper block of the next node could start its execution. In this way we obtain the wavefront behaviour.

2.3 CUDA

In this section we'll study the implementation of a Jacobi solver in CUDA. We'll start by implementing the kernel of the solver on the GPU, then we'll integrate this kernel into our main function and discuss how the memory must be managed. Lastly implement the reduction on the GPU too, as performing it on the CPU poses a huge bottleneck.

2.3.1 Kernels

Listing 10 shows the Jacobi kernel implemented in CUDA. Here, we're relying on a square matrix of size NxN stored in row major order and in a 2D mapping of our blocks and threads. As we're required to avoid the border of our matrix the kernel itself becomes safe to use even when N is not divisible by the number of threads per block.

```
__global__ void gpu_Heat(float *h, float *g, int N) {

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < 1 || x > N - 2 || y < 1 || y > N - 2)
        return;

    g[N * x + y] = 0.25 * (h[N * x + y - 1] + h[N * x + y + 1] +
                           h[N * (x - 1) + y] + h[N * (x + 1) + y]);
}
```

Listing 10: Jacobi kernel

Listing 11 shows a minor modification where we also produce the difference vector needed to perform the reduction. Instead of copying back both matrices we'll just store the difference result in GPU memory which will later be reduced by another kernel. This translates into huge speedup and savings in memory bandwith that might be useful for other kernels running on the GPU.

```
__global__ void gpu_Heat_diff(float *h, float *g, float *diff, int N) {

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < 1 || x > N - 2 || y < 1 || y > N - 2)
        return;

    g[N * x + y] =
        0.25 * (h[N * x + y - 1] + h[N * x + y + 1] + h[N * (x - 1) + y] + h[N * (x + 1) + y]);

    diff[(N-2) * (x-1) + y-1] = (g[N * x + y] - h[N * x + y]) * (g[N * x + y] - h[N * x + y]);
}
```

Listing 11: Jacobi with difference vector as output

We have to notice that the difference vector only makes sense for interior points, given that the borders do not change. Then, our difference vector will have $N - 2 \times N - 2$ elements.

2.3.2 Memory management

If we go to heat-CUDA.cu we'll notice some "TODO"s related with the necessary memory management needed to interact with CUDA. As the GPUs and CPUs use different memories we not only have to allocate memory on both of them but also copy it back and forth when needed[3].

Listing 12 shows the memory allocation and initialization. Listing 13 the copying of retrieval of results to compute the residual and Listing 13 the collection of results and freeing of resources.

```
    // TODO: Allocation on GPU for matrices u and uhelp
    cudaMalloc((void **)&dev_u, np * np * sizeof(float));
    cudaMalloc((void **)&dev_uhelp, np * np * sizeof(float));

    // TODO: Copy initial values in u and uhelp from host to GPU
    cudaMemcpy(dev_u, param.u, np * np * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_uhelp, param.uhelp, np * np * sizeof(float), cudaMemcpyHostToDevice);
```

Listing 12: Memory allocation and initialization

```
    // TODO: residual is computed on host, we need to get from GPU values computed in u and uhelp
    cudaMemcpy(param.u,       dev_u, np * np * sizeof(float), cudaMemcpyDeviceToHost);
    cudaMemcpy(param.uhelp, dev_uhelp, np * np * sizeof(float), cudaMemcpyDeviceToHost);
```

Listing 13: Memory copying

[3]CUDA Unified Memory is very ergonomic but sometimes noticeably slow. Using it also kind of defeats the purpouse when you're trying to learn how GPUs work

```
    // TODO: get result matrix from GPU
    cudaMemcpy(param.u, dev_u, np * np * sizeof(float), cudaMemcpyDeviceToHost);

    // TODO: free memory used in GPU
    cudaFree(dev_u);
    cudaFree(dev_uhelp);
```

Listing 14: Memory freeing and result retrieval

2.3.3 Reduction

For the GPU reduction we can use the implementations given to us in the *Hands on*.

For the first level reduction we will use the Kernel number 7, given that it supports that a thread previously reduces an arbitrary number of elements. This allows us to reduce huge vectors without having to necessarily spawn an enormous number of blocks.

For the second level reduction we will use Kernel06, given that the number of blocks will be somewhat small. We could use the same kernel twice but we thought it would be interesting to mix them a little bit.

```
__global__ void reduce(float *idata, float *odata, int N) {
    __shared__ float sdata[MAX_THREADS_PER_BLOCK];
    unsigned int s;
    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] += idata[i] + idata[i + blockDim.x];
        i += gridSize;
    }
    __syncthreads();
    for (s = blockDim.x / 2; s > 32; s >>= 1) {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    if (tid < 32) {
        volatile float *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) odata[blockIdx.x] = sdata[0];
}

__global__ void Kernel06(float *g_idata, float *g_odata) {
  __shared__ float sdata[MAX_THREADS_PER_BLOCK];
  unsigned int s;
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
  sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
  __syncthreads();

  for (s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
      sdata[tid] += sdata[tid + s];
    __syncthreads();
  }

 if (tid < 32) {
   volatile float *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 15: Reduction kernel

Obviously, we have to allocate and free the memory for the difference vector, the partial reduction for each block and for the final reduction.

The GPU reduction has to be enabled by specifying “gpu” as the fifth command line parameter. When it is enabled the following lines will be invoked:

```
    /* Arbitrary constants. They seem the fastest, at least for my 3060 */
    int elems_per_thread = 4;
    int num_threads_reduce = 128;
    int num_blocks_reduce = (np - 2) * (np - 2) / (elems_per_thread * num_threads_reduce);

    /* ... */

    gpu_Heat_diff<<<Grid, Block>>>(dev_u, dev_uhelp, dev_diff, np);
    reduce<<<num_blocks_reduce, num_threads_reduce>>>(dev_diff,
      dev_block_red, (np - 2) * (np - 2));
    /*
     * As every thread brings into shared memory the sum of 2
     * elemets and we're reducing num_block_reduce elements, we
     * have to invoke num_blocks_reduce/2 threads
     */
    Kernel06<<<1, num_blocks_reduce/2>>>(dev_block_red, dev_gpu_red);
    cudaMemcpy(&gpu_red, dev_gpu_red, sizeof(float), cudaMemcpyDeviceToHost);
```

Listing 16: Reduction

We have modified the code a little bit to measure the amount of time the CPU reduction takes. This allowed us to observe that great part of the overhead actually comes from copying memory from the GPU to the CPU in every iteration. It can be even bigger than the CPU reduction computation time.

Performing a good measurement of the speedup is a hard task in itself, as it is not the purpouse of this course we will omit it. Just as a reference, for the default case of execution

```
./heat-CUDA test.dat -t 16 gpu
```

The observed speedup is approximately 7.

3 Parallel solution

In this section, we are going to show the results of each implementation after the execution on Boada. The results for the same size of image and same Programming Model are displayed together.

3.1 OpenMP

First, we have OpenMP implementations, we obtained results for the three different implementation we have. All of them have been tested with 256, 512 and 1024 square images. The tests have used 1, 2, 4 and 8 threads. We have also calculated the speedup for every size and solver. The results of Jacobi show an improvement over the secuential time, meanwhile the results of both Gauss solvers are almost the same in all executions.

Threads	Jacobi	Gauss task	Gauss doacross
1	2,376	5,321	5,325
2	1,146	5,319	5,255
4	0,987	5,614	4,797
8	0,892	5,792	4,553

Table 1: Time to compute an image of size 256x256

Threads	Jacobi	Gauss task	Gauss doacross
2	2,073298429	1,000376011	1,013320647
4	2,407294833	0,9478090488	1,110068793
8	2,66367713	0,9186809392	1,169558533

Table 2: Obtained speedup with OpenMP for image of size 256x256

Threads	Jacobi	Gauss task	Gauss doacross
1	19,744	42,79	42,8
2	10,718	42,95	42,659
4	5,582	45,36	41,926
8	4,558	48,182	39,98

Table 3: Time to compute an image of size 512x512

Threads	Jacobi	Gauss task	Gauss doacross
2	1,842134727	0,9962747381	1,003305281
4	3,537083483	0,9433421517	1,020846253
8	4,331724441	0,8880909883	1,070535268

Table 4: Obtained speedup with OpenMP for image of size 512x512

Threads	Jacobi	Gauss task	Gauss doacross
1	84,923	172	172,12
2	44,161	172	170
4	24,271	171	169
8	20,59	167	169

Table 5: Time to compute an image of size 1024x1024

Threads	Jacobi	Gauss task	Gauss doacross
2	1,923031634	1	1,012470588
4	3,498949363	1,005847953	1,018461538
8	4,124477902	1,02994012	1,018461538

Table 6: Obtained speedup with OpenMP for image of size 1024x1024

3.2 MPI

MPI implementations of Jacobi and Gauss-Seidel has been tested in Boada, using 1, 2, 4 and 8 nodes for images of size 256, 512, 1024 respectively. *Time Out* in the results means that the scheduler of Boada has killed the job for exceeding the limit of time execution. As we can see with the speedups, we obtain values lower than 1. That means that we have a slowdown in the execution.

Nodes	Jacobi	Gauss
1	2,291	4,156
2	5,161	5,214
4	10,146	10,508
8	20,79	21,21

Table 7: Time to compute an image of size 256x256

Nodes	Jacobi	Gauss
2	0,4439062197	0,7970847718
4	0,2258032722	0,3955081842
8	0,1101972102	0,1959453088

Table 8: Obtained speedup with MPI for image of size 256x256

Nodes	Jacobi	Gauss
1	17,314	39,911
2	33,635	43,74
4	70,319	87,57
8	158,598	178,97

Table 9: Time to compute an image of size 512x512

Nodes	Jacobi	Gauss
2	0,5147614092	0,9124599909
4	0,2462207938	0,4557611054
8	0,1091690942	0,2230038554

Table 10: Obtained speedup with MPI for image of size 512x512

Nodes	Jacobi	Gauss
1	87,77	167
2	180,705	181
4	365,47	370
8	Time Out	Time Out

Table 11: Time to compute an image of size 1024x1024

Nodes	Jacobi	Gauss
2	0,4857087518	0,02488622754
4	0,2401565108	0,02296132597
8	——	——

Table 12: Obtained speedup with MPI for image of size 1024x1024