National and Kapodistrian University of Athens

Department of Informatics and Telecommunications

Parallel Systems

Course Project 2020-2021

Project implemented by :

CHARALAMPOS MARAZIARIS - 1115201800105
VISSARION MOUTAFIS - 1115201800119

Contents

1. Introduction

1.1 Problem

In this project we are assigned to numerically solve the following finite difference version of the *Poisson* equation:

$$(\nabla^2 - a)u = \frac{d^2}{dx^2}u + \frac{d^2}{dy^2}u + au = f$$

We are instructed to do so using the *Jacobi* method with successive over-relaxation.

1.2 Goal

Initially, we are given the source code of a serial solution to the problem. Our goal in this project is to improve the sequential code given, and then solve the problem by implementing and executing a parallel algorithm. By doing so, we hope to get the most out of the parallel computational power in hand, which we measure by speedup and efficiency rates. We implement and evaluate different parallelization approaches, namely:

- (Pure) MPI
- (Hybrid) MPI + OpenMP
- CUDA

1.3 Input

The user shall provide an `input` file, specifying a custom configuration to the problem. The input file is of the following structure:

```
n, m - rows/columns of u matrix
alpha - Helmholtz constant
relax - successive over-relaxation parameter
tol - error tolerance for the iterative solver
mits - maximum solver iterations
```

1.4 Code Structure

- ~`/Sequential/` : Contains the improved sequential source files, on which our parallel algorithm is based.
- ~`/ParallelMPI/` : Contains the files of our Pure MPI approach.
- ~`/HybridMPI/` : Contains the files of our Hybrid MPI+OpenMP approach.
- ~`/CUDA/` : Contains the files of our CUDA approach.

Each directory contains a **bash** script that submits the job using a configuration provided by the user.

1.5 Testing

The hardware we used to test this project is a cluster of 80 processors (cores) evenly split in 10 nodes. The cluster also provides 2 GPUs for our CUDA experiments. We use the **qsub** command to submit our jobs.

2. Sequential Program

2.1 Optimizations

The contents of the loop we had to optimize were:

```
while (iterationCount < maxIterationCount && error > maxAcceptableError)
{
    error = one_jacobi_iteration(...);
    iterationCount++;
    // Buffer swap
}
```

The nested loop of the **one_jacobi_iteration** function that was subjected to optimizations was:

```
#define SRC(XX,YY) src[(YY)*maxXCount+(XX)]
#define DST(XX,YY) dst[(YY)*maxXCount+(XX)]

for (y = 1; y < (maxYCount-1); y++)
{
    fY = yStart + (y-1)*deltaY;
    for (x = 1; x < (maxXCount-1); x++)
    {
        fX = xStart + (x-1)*deltaX;
        f = ...;
        updateVal = (... - f)/...;
        DST(x,y) = SRC(x,y) - omega*updateVal;
        error += updateVal*updateVal;
    }
}
```

Following the optimization instructions given, we tried to reduce the amount of assignments performed by removing variables - placeholders for middle results (such as variable **f**), and instead merging those calculations in a single one.

We achieved a major breakthrough in reducing the total time spent in the inner loop body, when we noticed that the value **fX** was independent of the outer loop (and the value of **y**), and thus we didn't have to calculate it over and over for every iteration of the outer loop. Instead, we calculated the values of **fX** before the nested loop, and stored them in a static array for faster access. We followed the same practice replacing variables **fY** and **(YY)*maxXCount** (heavily used in the defined values **SRC** and **DST**) with arrays computed beforehand.

Lastly, we *inlined* the function we optimized into the main `while`-loop body. The final result looks like this:

```
for (int x = 0; x < n; x++)
    fX_sq[x] = (xLeft + x * deltaX) * (xLeft + x * deltaX);

for (int y = 0; y < m; y++)
    fY_sq[y] = (yBottom + y * deltaY) * (yBottom + y * deltaY);

for (int i = 0; i < maxYCount; i++)
    indices[i] = i * maxXCount;

while (iterationCount < maxIterationCount && error > maxAcceptableError)
{
    iterationCount++;
    error = 0.0;
```

```
14      for (int y = 1; y < (maxYCount - 1); y++)
15      {
16          for (int x = 1; x < (maxXCount - 1); x++)
17          {
18              updateVal = ...
19              u[indices[y] + x] = u_old[indices[y] + x] - relax * updateVal;
20              error += updateVal * updateVal;
21          }
22      }
23      error = sqrt(error) / (n * m);
24      // Buffer swap
25  }
```

Furthermore, we changed the type of function, f, to decrease the number of multiplications and by assigning temp variables we make sure to avoid the use of division, the initial function type contained. In this manner, we managed to improve the benchmarks against the original program even more (check next section), while decreasing the comprehension and readability.

2.2 Benchmarks

First we will display the benchmarks of the challenge We begin by showing the comparison in the execution time of the Initial code versus the Optimized code.

Table 1: MPI Wall Time and Speedup

Prog/Array sz	840	1680	3360	6720	13440	26880
Initial	0.873 s	3.337 s	13.336 s	53.336 s	213.313 s	853.351 s
Optimized	0.338 s	1.338 s	5.334 s	21.267 s	85.197 s	340.596 s
Speedup	**2.58**	**2.49**	**2.50**	**2.51**	**2.50**	**2.51**

In an effort to explain the reason of this great speedup (about **2.50**), we showcase the difference in operations performed in the Initial versus the Optimized version.

Table 2: Assignment Operations

	Before `while`	Outter	Inner
Initial	3	1	5
Optimized	12 + 2m + n	0	4

Table 3: Mult/Div Operations

	Before `while`	Outter	Inner
Initial	6	1	**22**
Optimized	14 + 4m + 3n	0	**7**

Table 4: Add/Sub Operations

	Before `while`	Outter	Inner
Initial	2	2	23
Optimized	4 + 4m + 3n	0	25

As we can see, in the optimized version we shifted much of the work outside the `while` loop, and greatly reduced the amount of mult/div operations performed in the inner loop. It is very important to note that:

- `Before while` operations are just performed **once** in the program.
- `Outter` operations are performed at most $\boldsymbol{maxIterations \cdot m}$ times.
- `Inner` operations are performed at most $\boldsymbol{maxIterations \cdot m \cdot n}$ times.

For example, if $m = n = 6720$ and $maxIterations = 50$, the `Before while` section is executed 1 time, the outter loop is executed around 330K times, while the inner loop is executed 2.2B (!) times.

Thus, we mainly attribute the drastic fall of execution time to the difference in the mult/div operations between the Initial and the Optimized code. These operations are reduced to about **1/3** in the final version, as shown in table 3.

Another thing to note, is that the extensive use of additional arrays seemingly respects cache locality, and thus the bottleneck introduced from cache misses due to array items, is minimal.

3. MPI Parallel Design

3.1 Data Distribution

Suppose that our parallel MPI program is being ran by d processes in parallel.

Our initial problem consists of a $n \times m$ matrix (n, m are user-supplied values). Thus, we wish to split the matrix in d equal-sized blocks, distribute the blocks among the processes, perform the necessary computations and finally, present the collective result.

To do so, we refer to the *Process Topologies* MPI has to offer, and more specifically the **Cartesian Grid**.

```
dims[2] = { 0, 0 };
MPI_Dims_create(num_of_processes, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic = {0, 0}, 1, &cart);
```

We use *MPI_Dims_Create* to assist us in selecting the optimal 2D arrangement for the processes, which also helps us distribute the data:

```
local_n = n / dims[0];
local_m = m / dims[1];
```

Each process gets a $local\_n \times local\_m$ block of values to compute, which amounts to a total of:

$$local\_n \times local\_m = \frac{n}{dims[0]} \cdot \frac{m}{dims[1]} = \frac{n \cdot m}{dims[0] \cdot dims[1]} = \frac{n \cdot m}{d} \quad \text{values,}$$

proving that data are equally distributed among the d processes in the $dims[0] \times dims[1]$ Cartesian grid.

3.2 Communication Design

Divided in *Idle* communication, which involves input broadcasting and result gathering, and *Computational* communication, which is done for the purpose of updating the matrix.

3.2.1 While Idle

Process 0 reads the input parameters specified by the user. We then use the *MPI_Broadcast* function to broadcast the input parameters from process 0 to every process participating in the parallel run.

```
MPI_Comm_rank(comm, &myRank);
if (myRank == 0) {
    scanf("%d,%d", &n, &m);
    ...
}
MPI_Bcast(&n, 1, MPI_INT, 0, comm);
...
```

In order to get the maximum *clock()* time and MPI time among every working process, and the global error (the summary of all the process-local errors), we use the *MPI_Reduce* function, with the appropriate MPI operators:

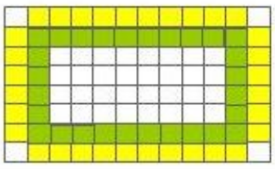
```
MPI_Reduce(&msec, &max_msec, 1, MPI_INT, MPI_MAX, 0, comm);
    ...
MPI_Reduce(&local_time, &global_time, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
    ...
MPI_Reduce(&error, &error_sum, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
error = sqrt(error_sum) / (n * m);
```

In order to **re-assemble** the full $n \times m$ matrix from the $local\_n \times local\_m$ local matrices, we use the *MPI_Gather* function, with some *post-processing* to bring the block values in their correct place in the global matrix. This is done, so that we can calculate the error of the global matrix instead of adding the individual errors of the local matrices.

```
MPI_Type_vector(local_m, local_n, local_n+2, MPI_DOUBLE, &block_t);
MPI_Type_commit(&block_t);
...
MPI_Gather(&u_old[local_n+3], 1, block_t, u_all, 1, block_t, 0, comm);
... (Post-Processing, u_all -> u_final)
absoluteError = checkSolution(..., u_final, ...);
absoluteError = sqrt(absoluteError) / (n * m);
```

3.2.2 While Computing

In order for our parallel program to perform the exact same computations performed in the sequential program, each process is required to communicate with at least 2 and at most 4 other processes. Suppose that the *green* values in the shape below are located on the "edge" of the block assigned to the process, that is, they reside in the first or last row or column of the block.



Thus, the presence of *yellow* values is mandatory for the correct update of the green values. The *yellow* values either belong to the neighboring block, or, if no neighboring block exists, their value is 0.

Implementing the concepts discussed above, each process finds its neighbors in the Cartesian Grid by calling *MPI_Cart_Shift* (which by default sets the variable to *MPI_PROC_NULL* if no neighbor exists in this direction):

```
MPI_Cart_shift(comm_cart, 0, 1, &north, &south);
MPI_Cart_shift(comm_cart, 1, 1, &west, &east);
```

Then proceeds to install *Persistent Communications* (discussed in the following section), with the neighboring processes, enabling them to exchange the desired values.

Notice that the *yellow* values are only found in 2 rows and 2 columns (called **halo points**). Thus, we decided to use the *MPI_Type* functionality provided by MPI in order to reduce the amoung of messages required to perform the exchange:

```
MPI_Datatype row_t, col_t;
MPI_Type_contiguous(local_m, MPI_DOUBLE, &row_t);
MPI_Type_vector(local_n, 1, local_m+2, MPI_DOUBLE, &col_t);
MPI_Type_commit(&row_t);
MPI_Type_commit(&col_t);
```

Additional storage of 2 rows and 2 columns is required for every process to store the values sent from its neighbors and use them in correctly computing the *green* values. Thus, every process stores a $(local\_n + 2) \times (local\_m + 2)$ matrix, which is initialized to 0s, to cover the case of no-neighbor in some directions.

```
u     = calloc(((local_n + 2) * (local_m + 2)), sizeof(double));
u_old = calloc(((local_n + 2) * (local_m + 2)), sizeof(double));
```

3.3 Foster's methodology in the current design

The foster methodology of using parallel design to improve serial programs is obvious in our project. In the following lines we will analyze each and every stage of Foster's Method.

3.3.1 Partitioning

At the very beginning of our code we can distinguish couple of lines that determine the dimensions of the Cartesian network we will use to parallelize the problem. These are also useful to divide the initial data board to smaller area-equal blocks, that will be sent over to each process, according to its place in the Cartesian topology. This data division will take place only in the beginning and then each process will execute the algorithm, on the data partition it had been assign to.

This data partition is actually called **domain decomposition** and it was used since the functions in each data part are overlapping, so there was no viable way to introduce functional decomposition into this problem.

From the memory-usage point, we will use smaller arrays for each process and we will apply the respective frees when and where necessary.

3.3.2 Communication

We will now proceed to point out all the communication related actions. Note that the communication is *structured* (Grid Topology) and *asynchronous* as requested by the project readings.

The so-called **Halo Points**, are in-need of constant updating at the start of every iteration of the main algorithm. These array parts are actually acquired from the neighbors of the process in the Cartesian Topology. The communication between the *local process neighbourhood* is accomplished with **persistent communicators**, to reduce overhead between the process and communication controller. In the same spirit we used non-blocking communication to avoid any kind of I/O blocking overhead, while we ensured that we will use the updated halo-points, by using appropriate *wait-all type routine*, after calculating the results in the inner parts of the data-block.

Other communication points are the error gathering code-blocks that are considered *global communication points* and are necessary for evaluating the overall algorithm precision on the problem considering the given input.

3.3.3 Agglomeration

The agglomeration stage is manifested at the end of the main algorithm, where we will gather all data blocks back to the original board and we will estimate the result error. There a comparison between the agglomerated method, where only the root process estimates the error and the iterative method where every process estimates its own error and then the root gathers everything. The agglomerated methods utilizes special block datatypes (see next section) that will reassure the original board is reconstructed properly. In this way we will reduce the communication time of gathering all error-per-process instances in the **convergence-check** variation of the program.

3.3.4 Mapping

The mapping of different processes to processors is left to the job scheduler that we specifically determine the *nodes, processors and processes/core* used in the submitted script. The configurations are fixed and are specified in the projects readings. Of course with different input work-load, we expect different mapping configurations to get the better results (check MPI-Profiling section), as the locality of the neighbours and the load-balancing in each core is crucial to the overall overhead caused by computations and communications respectively.

4. MPI Optimizations

4.1 Datatypes

We used the following data-types to reduce the communication overhead between processes:

- **row_t**, which is a contiguous type and interprets one row in the given data-block. This type is used in Halo Points send/receive calls.
- **col_t**, which is of type vector and interprets one column by skipping $sizeof(row)$ elements to get every row's respective element. This type is used in Halo Points send/receive calls.
- **block_t**, which is of type vector and interprets the inner part of current process' data-block (without the halo points in the outer section of the block). It is used in the gather call to reconstruct the final version of initial matrix and estimate the solution error.

By using all these data types the communications' overhead is decreased and the overall readability of our code improves

4.2 Persistent & Non-Blocking Communications

Using the appropriate calls we implement **persistent communication** busses between the neighbours of each process and itself. In that manner we decrease the overhead of creating a new bus each time we want to communicate with a neighbor to send/get the *halo points*.

Furthermore we use **non-blocking send/receive calls** to exchange *halo points*, so that no process has to wait for the other end to connect to the bus in order to complete the communication action. Nevertheless, there are specific calls in the code to wait for all *halo points* getter actions in order to continue computations that need this specific values. The actual gain here is that each process can make the requests, send its respective values and proceed to estimating the inner values (the white blocks in image above) without further due, then, when needed, it will wait for the new halos.

5. Pure MPI Benchmarks

In the following sections we will present the results of our experimenting with a pure MPI approach

5.1 Challenge Benchmarks

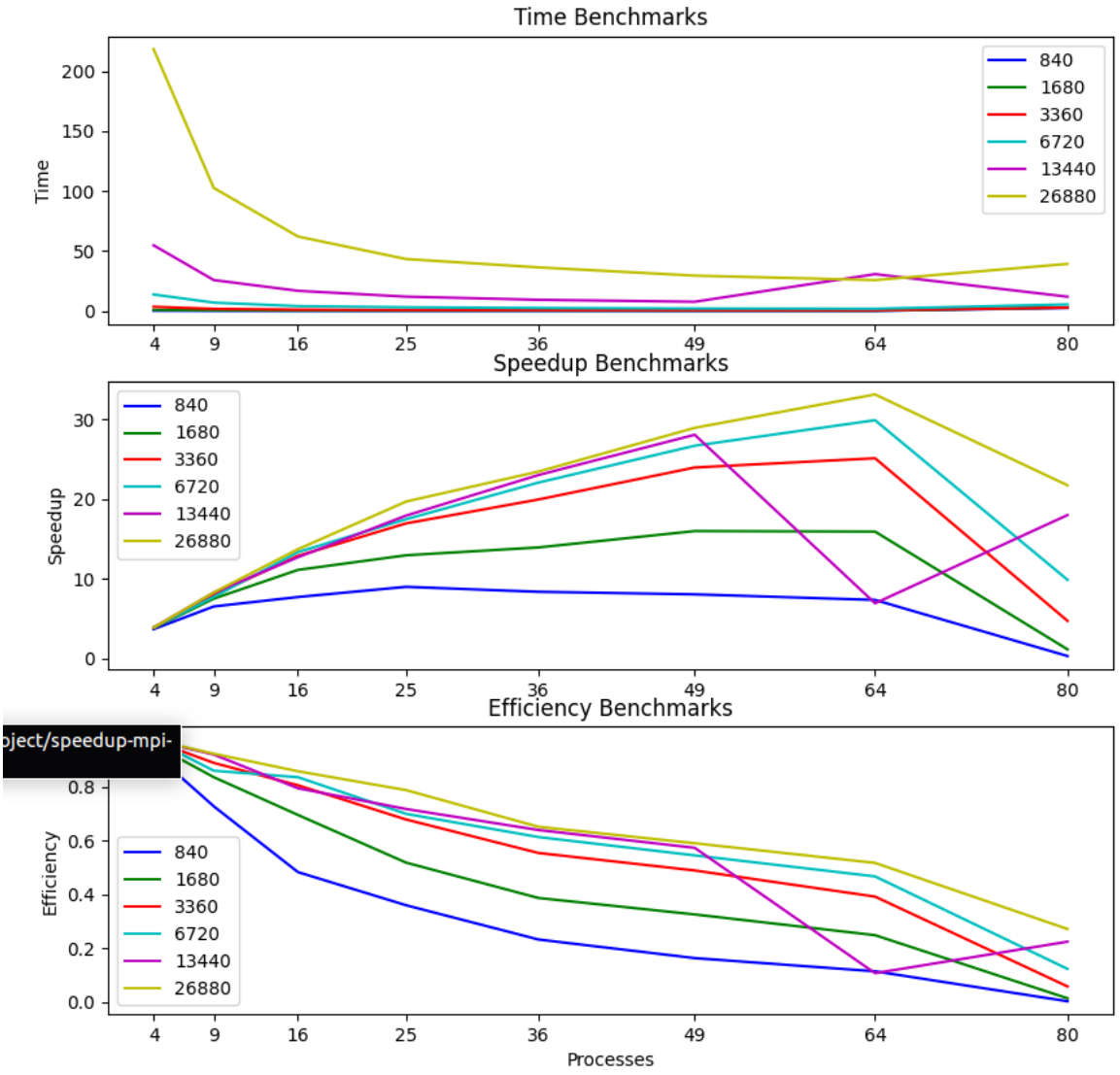


Table 5: Challenge Time Benchmark

Procs/Array sz	4	9	16	25	36	49	64	80
840	0.226s	0.127s	0.108s	0.0930s	0.099s	0.103s	0.113s	2.544s
1680	0.871s	0.443s	0.299s	0.257s	0.239s	0.208s	0.209s	2.854s
3360	3.429s	1.664s	1.032s	0.785s	0.667s	0.555s	0.529s	2.806s
6720	13.674s	6.881s	3.984s	3.051s	2.411s	1.994s	1.780s	5.383s
13440	54.724s	25.720s	16.746s	11.869s	9.249s	7.585s	30.684s	11.829s
26880	218.454s	102.626s	62.084s	43.248s	36.312s	29.436s	25.716s	39.190s

Table 6: Challenge Speedup Benchmark

Procs/Array sz	4	9	16	25	36	49	64	80
840	3.699	6.54	7.721	8.997	8.375	8.056	7.356	0.329
1680	3.824	7.518	11.121	12.954	13.938	15.986	15.922	1.167
3360	3.879	7.994	12.885	16.949	19.944	23.954	25.113	4.741
6720	3.891	7.732	13.353	17.474	22.064	26.684	29.881	9.884
13440	3.889	8.275	12.709	17.93	23.009	28.058	6.936	17.991
26880	3.897	8.296	13.714	19.687	23.446	28.924	33.108	21.725

Table 7: Challenge Efficiency Benchmark

Procs/Array sz	4	9	16	25	36	49	64	80
840	0.925	0.727	0.483	0.36	0.233	0.164	0.115	0.004
1680	0.956	0.835	0.695	0.518	0.387	0.326	0.249	0.015
3360	0.97	0.888	0.805	0.678	0.554	0.489	0.392	0.059
6720	0.973	0.859	0.835	0.699	0.613	0.545	0.467	0.124
13440	0.972	0.919	0.794	0.717	0.639	0.573	0.108	0.225
26880	0.974	0.922	0.857	0.787	0.651	0.59	0.517	0.272

5.2 MPI Benchmarks

5.2.1 With Convergence Check (with Allreduce)

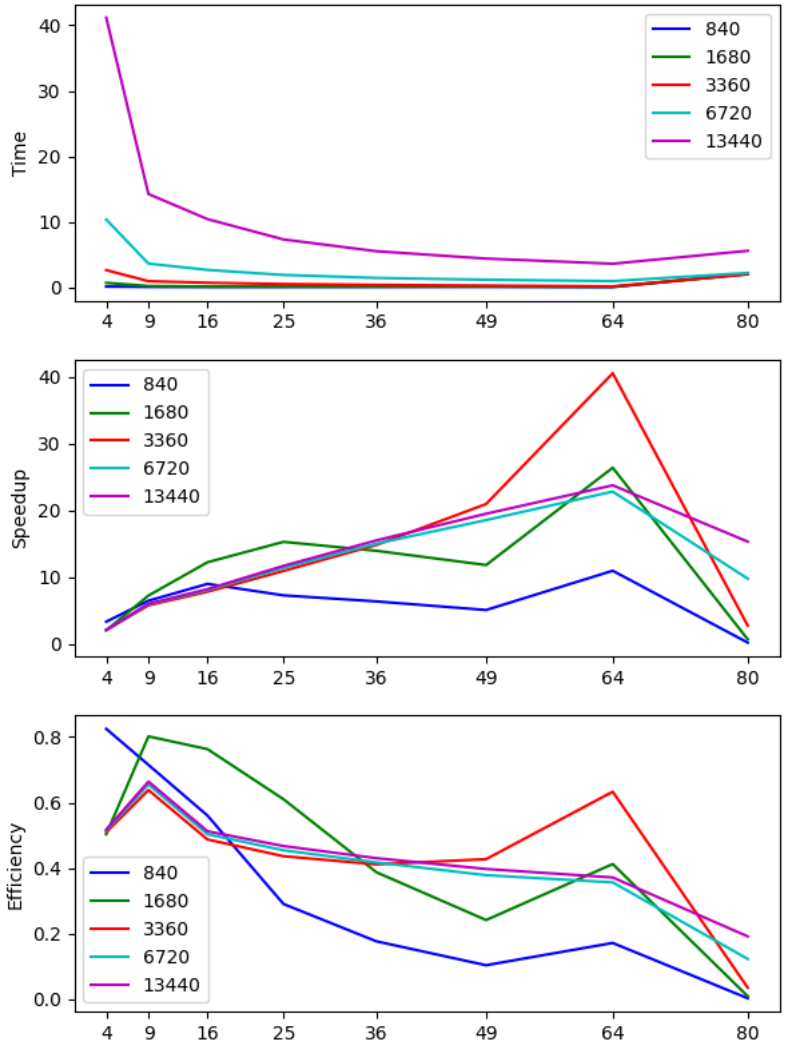


Table 8: MPI Wall Time with Convergence Check

Size/Procs	4	9	16	25	36	49	64	80
840	0.102s	0.052s	0.0378s	0.046s	0.053s	0.066s	0.033s	2.042s
1680	0.665s	0.185s	0.109s	0.087s	0.096s	0.113s	0.051s	2.051s
3360	2.614s	0.927s	0.683s	0.488s	0.359s	0.254s	0.131s	1.965s
6720	10.340s	3.607s	2.644s	1.873s	1.418s	1.148s	0.933s	2.183s
13440	41.180s	14.236s	10.401s	7.298s	5.501s	4.372s	3.587s	5.566s
26880	164.689s	56.807s	23.372s	0.937s	0.696s	0.533s	0.422s	0.534s

Table 9: Speedup with Convergence Check

Size/Procs	4	9	16	25	36	49	64	80
840	3.301	6.439	8.968	7.245	6.342	5.055	10.933	0.165
1680	2.011	7.215	12.205	15.26	13.921	11.786	26.362	0.652
3360	2.038	5.745	7.79	10.901	14.802	20.928	40.514	2.71
6720	2.057	5.898	8.045	11.357	14.995	18.521	22.79	9.743
13440	2.067	5.98	8.185	11.664	15.476	19.468	23.728	15.292
26880	2.068	5.994	14.57	363.107	488.566	638.334	806.494	636.533

Table 10: Efficiency with Convergence Check

Size/Procs	4	9	16	25	36	49	64	80
840	0.825	0.715	0.56	0.29	0.176	0.103	0.171	0.002
1680	0.503	0.802	0.763	0.61	0.387	0.241	0.412	0.008
3360	0.509	0.638	0.487	0.436	0.411	0.427	0.633	0.034
6720	0.514	0.655	0.503	0.454	0.417	0.378	0.356	0.122
13440	0.517	0.664	0.512	0.467	0.43	0.397	0.371	0.191
26880	0.517	0.666	0.911	14.524	13.571	13.027	12.601	7.957

5.2.2 Without Convergence check (w/out Allreduce)

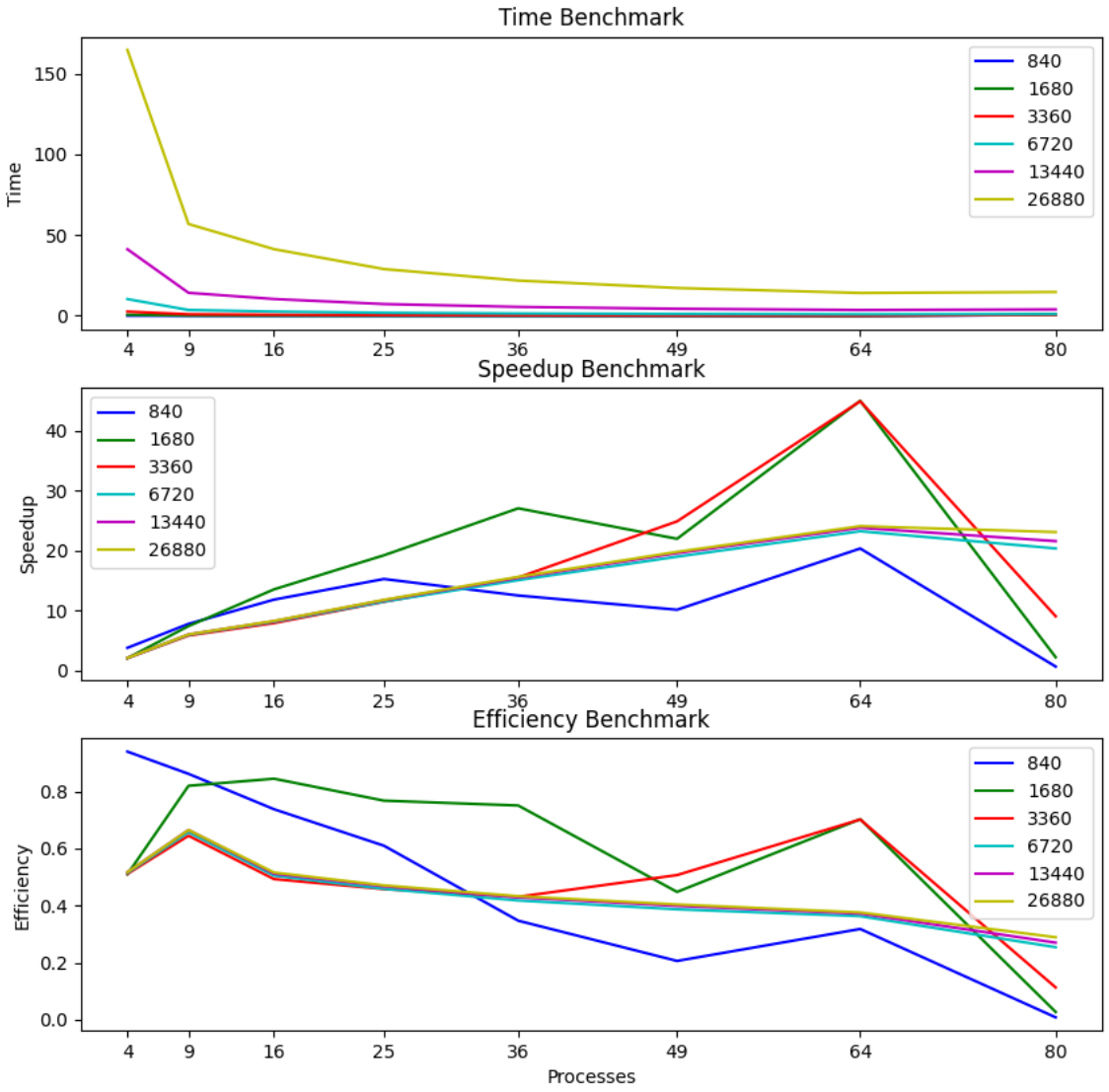


Table 11: MPI Wall Time without Convergence Check

Size/Procs	4	9	16	25	36	49	64	80
840	0.090s	0.043s	0.029s	0.022s	0.027s	0.033s	0.017s	0.530s
1680	0.655s	0.181s	0.098s	0.069s	0.049s	0.061s	0.029s	0.609s
3360	2.605s	0.91773s	0.676s	0.465s	0.344s	0.214s	0.119s	0.590s
6720	10.344s	3.594s	2.629s	1.853s	1.413s	1.121s	0.916s	1.045s
13440	41.190s	14.224s	10.386s	7.275s	5.513s	4.355s	3.578s	3.948s
26880	164.583s	56.833s	41.245s	28.914s	21.809s	17.207s	14.136s	14.739s

Table 12: Speedup without Convergence Check

Size/Procs	4	9	16	25	36	49	64	80
840	3.758	7.755	11.808	15.249	12.497	10.118	20.335	0.634
1680	2.041	7.383	13.519	19.204	27.046	21.955	44.988	2.192
3360	2.046	5.806	7.883	11.45	15.501	24.865	44.918	9.028
6720	2.056	5.917	8.088	11.475	15.053	18.973	23.221	20.351
13440	2.067	5.987	8.199	11.705	15.448	19.554	23.799	21.571
26880	2.067	5.987	8.249	11.768	15.602	19.775	24.071	23.086

Table 13: Efficiency without Convergence Check

Size/Procs	4	9	16	25	36	49	64	80
840	0.94	0.862	0.738	0.61	0.347	0.206	0.318	0.008
1680	0.51	0.82	0.845	0.768	0.751	0.448	0.703	0.027
3360	0.511	0.645	0.493	0.458	0.431	0.507	0.702	0.113
6720	0.514	0.657	0.505	0.459	0.418	0.387	0.363	0.254
13440	0.517	0.665	0.512	0.468	0.429	0.399	0.372	0.27
26880	0.517	0.665	0.516	0.471	0.433	0.404	0.376	0.289

5.3 Comparison: Our MPI vs Challenge

Table 14: Challenge vs. our Optimal

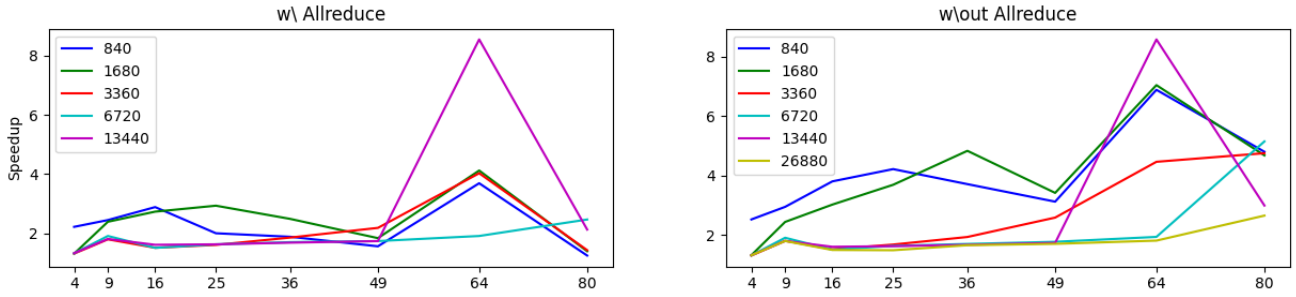


Table 15: Challenge/our Optimal with Allreduce

Procs/Array sz	4	9	16	25	36	49	64	80
840	2.217	2.446	2.885	2.0	1.881	1.559	3.691	1.246
1680	1.309	2.389	2.731	2.932	2.486	1.835	4.121	1.392
3360	1.312	1.795	1.51	1.606	1.854	2.182	4.029	1.428
6720	1.322	1.908	1.507	1.626	1.7	1.736	1.908	2.465
13440	1.329	1.807	1.61	1.626	1.682	1.735	8.553	2.125
26880	1.326	1.807	2.656	46.116	52.101	55.181	60.906	73.257

Table 16: Challenge/our Optimal w/out Allreduce

Procs/Array sz	4	9	16	25	36	49	64	80
840	2.53	2.953	3.809	4.221	3.716	3.128	**6.884**	4.803
1680	1.33	2.447	3.029	3.694	4.835	3.422	**7.041**	4.681
3360	1.317	1.814	1.528	1.687	1.941	2.592	**4.466**	**4.755**
6720	1.322	1.915	1.515	1.643	1.707	1.779	1.944	**5.151**
13440	1.329	1.808	1.612	1.632	1.678	1.742	**8.576**	2.997
26880	1.327	1.806	1.505	1.496	1.665	1.711	1.819	**2.659**

As we can see in table 14, our pure MPI program constantly **outperforms** the challenge program, with speedups varying from $\times 1.3$ to even $\times 8.5$ faster. Note that the greatest speedup is achieved while using 64 or 80 processes, meaning that Communication is done much more efficient from our program, which employs Persistent Communications and Cartesian Grid.

5.4 Notes

1. We conducted 3 experiments as the assignment readings suggested. The first one is consisted of the results for the challenge program. The second profiles our algorithm with Allreduce usage and convergence check in each iteration. The last one is consisted of benchmarks of our algorithm performance without any Allreduce usage and no convergence check.
2. The sequential starter program for the last 2 experiments is our optimized program, while for challenge program was the initial program given by the instructor.
3. We can clearly distinguish that our optimized program performs better than the challenge, in respect of time, at all parallel tests. This is displayed in the timing graphs of all 3 experiments.
4. We can also see that our program has bursts of speedup and efficiency for some inputs, especially in the Allreduce use case. Nevertheless we expected this to happen, since the Allreduce call, make calls in all processes and delays their execution. Graphs are more normal looking in the use case where we don't check convergence.
5. With convergence check we might even note some superlinear speedup and some extreme efficiency in the last input size use case.
6. Having a look at the profiler files we will actually watch the Allreduce call manifest the communication delay that we were earlier talking about and this event will play a huge roll in algorithm scaling especially when we add more processes. We see that the last 4 inputs are performing quite well in terms of speedup, that is, while it's changing very small, it is constantly elevating until it hits a max, at about the threshold of 64 processes.
7. Topology and fast sequential algorithm allowed our processes to run very fast and mainly focus on communicating and exchanging halos so there is a large part of execution time that will focus only in communication and synchronization of processes. This can easily be viewed by studying the speedup graphs, that very soon will diverge from their expected behaviour and could be verified by taking a look at the mpip profiling files.
8. In convergence check case, we exclude last input size from our scaling study, since the residual error is too small and it only applies for 1 iteration, hence the speedup and efficiency are enormous, as we might see if we study the respective benchmark tables.

6. Hybrid MPI+OpenMP

6.1 Design

For our "Hybrid" build, we preserved the core structure of our MPI program (described in the previous sections). On top of that, we added thread-level parallelization using **OpenMP** to parallelize the *for-loops* contained in the main `while-loop` of interest.

First of all, we used a "team" of parallel threads throughout the complete execution of the `while-loop`, to avoid the overhead introduced by continuously creating and terminating threads for every *for-loop*.

```
# pragma omp parallel shared(...) private(...)
{
    // while-loop related code
}
```

Next, we parallelized the initialization *for-loops* for the auxiliary arrays we used in our optimized solution.

```
# pragma omp for schedule(static)
for (int x = 0; x < local_n; x++) {
    fX_sq[x] = (xLeft + x * deltaX) * (xLeft + x * deltaX);
}
// fY_sq[], indices[] are initialized similarly
```

We will discuss the `schedule-clause` arguments later.
For the main part, the body of the `while-loop`, there are a couple things to point out:

6.1.1 Master thread

The **master** thread of each process executes every MPI-related call (via the `# pragma omp master` directive). That is, it begins/waits for the communications (*MPI_Startall/MPI_Waitall*) and contributes to the summary of the global error (*MPI_Reduce* statements).
The **master** thread is also responsible for swapping the buffers, calculating the summarized local error and maintaining the iteration counter.

6.1.2 Barriers

The `# pragma omp barrier` clauses are used to maintain the programs correctness.
They are used in the following cases:

1. To ensure that every thread performs the correct `while` check, that is: no thread can change the value of either the *iterationCount* or the *error*, before **every** thread has executed it.
2. To ensure that the process has finished exchanging **halo** rows/columns, thus the thread can proceed to calculate the inner values.

6.1.3 Order of calculations

We first calculate the inner values, and then the outer values (since we need the halo points for the latter).
As expected, we parallelize the **nested-for** loops needed for the **inner** values.
However, we split the calculation of the **outer** values into **4 for-loops** (1st/last row/col), which we parallelize. We found that this split is more preferable than calculating the first-last row and the first-last column in 2 separate for-loops. This is partly attributed to the cache performance we gain when calculating its row separately (since there is strong spatial locality to the elements belonging in the same row).

6.2 Benchmarks

6.2.1 `for-loop` collapse(2)

We are going to investigate the effect of the `collapse(2)` directive in our parallel nested `for-loop` that calculates the inner values. The loop looks like this:

```
# pragma omp for schedule(SCHEDULE_TYPE) collapse(2)
for (int y = 2; y < (maxYCount - 2); y++)
    for (int x = 2; x < (maxXCount - 2); x++) {
        fX_dot_fY_sq = fX_sq[x - 1] * fY_sq[y - 1];
        updateVal = (u_old[indices[y] + x - 1] + u_old[indices[y] + x + 1])
        * cx_cc + (u_old[indices[y-1] + x] + u_old[indices[y+1] + x])
        * cy_cc + u_old[indices[y] + x] +
        c1 * (1.0 - fX_sq[x - 1] - fY_sq[y - 1] + fX_dot_fY_sq) -
        c2 * (fX_dot_fY_sq - 1.0);
        u[indices[y] + x] = u_old[indices[y] + x] - relax * updateVal;
        local_error0 += updateVal * updateVal;
    }
```

We experiment with different values of the SCHEDULE_TYPE constant. The results below are measured with an array size of 6720×6720, with 2 processes per node and 2 threads per process, operating in 1 node.

Table 17: Speedup with and without `collapse(2)`, for various schedule types

`collapse(2)`/SCHEDULE_TYPE	static	dynamic	dynamic, 840	guided
With	17.996 s	57.207 s	18.164 s	18.028 s
Without	11.307 s	11.360 s	11.324 s	11.319 s
Speedup	**1.59**	**5.06**	**1.59**	**1.59**

We notice a major decrease in performance when we use the `collapse(2)` directive in our *parallel for* clause, for every schedule type we tried.
We mainly attribute the result to the speculation that **with** `collapse(2)` enabled, we lose some of the cache performance gained from the fact that the *inner* loop is processing elements belonging to the same row of matrices **u, u_old**. Thus, the cache lines containing this row are used to the greatest extent, upon they are brought in the CPU from the main memory, since every subsequent iteration of the inner loop uses a value contained in that line (until the last value of the line). When we enable the directive, we lose the guarantee that every cache line will be used to the maximum all at once, thus this method is prone to scenarios where a thread replaces a cache line in the CPU, only to bring the exact same line later on, due to the rearrangement of the calculations caused by `collapse(2)`.
We strongly believe that these results are representative of other configurations (processes/threads) as well as bigger array sizes, since they eventually reduce to smaller array sizes, such as the one we experimented with.
In conclusion, the `collapse(2)` directive is **not** suitable for our parallel nested `for-loop`.

6.2.2 `for-loop` scheduling

In order to measure the performance of various `for-loop` scheduling types, we ran our optimal configuration (see next section) to 1 Node.

Table 18: for-loop scheduling: **auto** and **dynamic**

Config/Array sz	auto	dynamic, 1	dynamic, 8	dynamic, 512
13440	**41.49s**	**41.46s**	**41.75s**	**41.48s**
26880	**166.38s**	**166.31s**	169.63s	167.06s

Table 19: for-loop scheduling: **guided** and **static**

Config/Array sz	guided, 1	guided, 8	guided, 512	static, 1	static, 8	static, 512
13440	**41.48s**	**41.49s**	**41.49s**	**41.45s**	45.79s	**41.56s**
26880	**166.62s**	167.12s	**166.83s**	**166.56s**	183.64s	**166.79s**

From the results obtained above, it is evident that for the crucial input sizes (13440×13440, 26880×26880) we **cannot** conclude to a schedule type that outperforms the others, with **auto**, **guided with chunks 1 and 512**, **dynamic with chunk 1** and **static with chunks 1 and 512** performing roughly the same.

6.2.3 Optimal process-thread configuration per node

Table 20: Hybrid-MPI Wall Time for various configurations in 1 CPU/Node

Config/Array sz	840	1680	3360	6720	13440	26880
1 Process - 8 Threads	-	-	-	21.314 s	85.195 s	343.380 s
2P-4T	-	-	-	11.370 s	45.462 s	183.280 s
2P-8T	-	-	-	11.423 s	45.478 s	186.792 s
4P-2T	-	-	**2.616** s	**10.363** s	**41.595** s	**166.308** s
4P-4T	-	-	2.626 s	10.423 s	41.763 s	168.806 s

As we can see from the results obtained above, the optimal configuration is **4 processes per CPU node and 2 Threads per process**.

6.2.4 Benchmarks with the optimal configuration

We measure our benchmarks with the number of **Nodes** used. In each Node we ran 4 processes and 2 threads per process, thus **4 processes and 8 Threads per Node**.
With Convergence Check (with Allreduce)

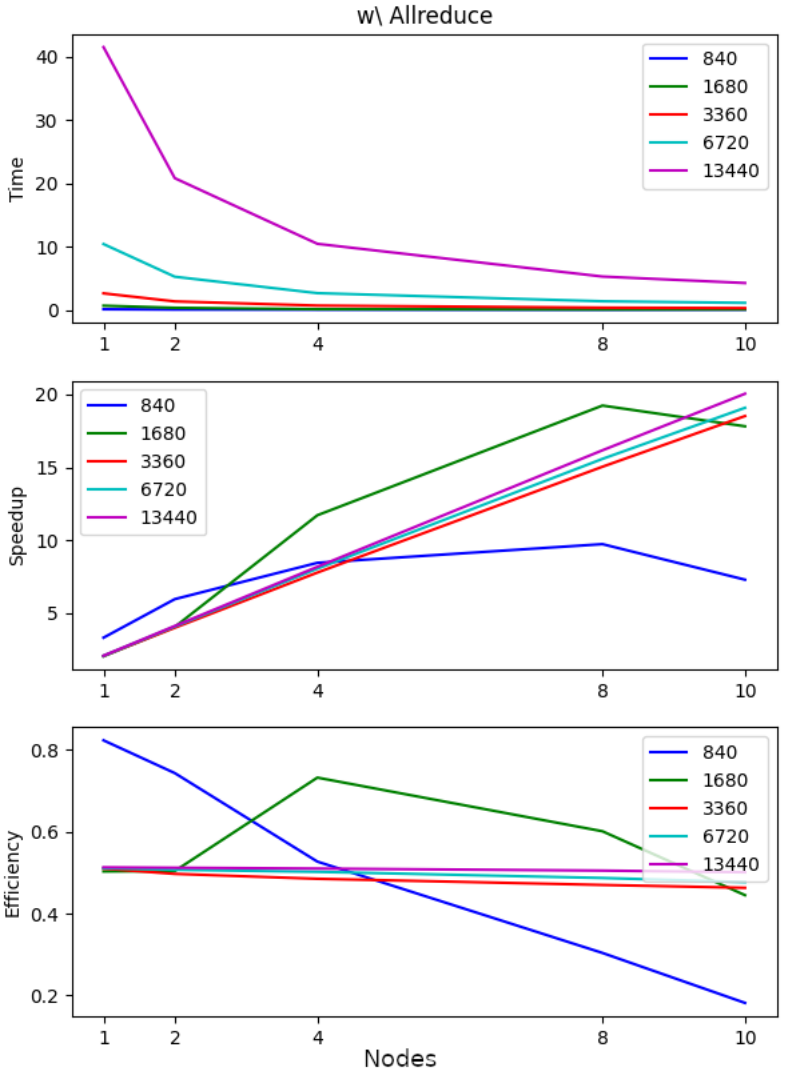


Table 21: MPI-Hybrid Wall Time with Convergence Check

Size/Nodes(Procs-Threads)	1(4-8)	2(8-16)	4(16-32)	8(32-64)	10(40-80)
840	0.10264s	0.05688s	0.04005s	0.03476s	0.04643s
1680	0.66544s	0.33195s	0.11432s	0.06954s	0.0751s
3360	2.61602s	1.3427s	0.68701s	0.35468s	0.28799s
6720	10.39968s	5.24058s	2.64763s	1.36515s	1.11409s
13440	41.48938s	20.7986s	10.4379s	5.26887s	4.24781s
26880	166.7651s	83.22221s	23.53783s	0.68048s	0.57932s

Table 22: MPI-Hybrid Speedup with Convergence Check

Size/Nodes(Procs-Threads)	1(4-8)	2(8-16)	4(16-32)	8(32-64)	10(40-80)
840	3.292	5.94	8.436	9.72	7.277
1680	2.011	4.031	11.705	19.243	17.818
3360	2.039	3.973	7.765	15.04	18.523
6720	2.045	4.058	8.033	15.579	19.089
13440	2.053	4.096	8.162	16.17	20.057
26880	2.042	4.093	14.47	500.523	587.924

Table 23: MPI-Hybrid Efficiency with Convergence Check

Size/Nodes(Procs-Threads)	1(4-8)	2(8-16)	4(16-32)	8(32-64)	10(40-80)
840	0.823	0.743	0.527	0.304	0.182
1680	0.503	0.504	0.732	0.601	0.445
3360	0.51	0.497	0.485	0.47	0.463
6720	0.511	0.507	0.502	0.487	0.477
13440	0.513	0.512	0.51	0.505	0.501
26880	0.51	0.512	0.904	15.641	14.698

Without Convergence check (w/out Allreduce)

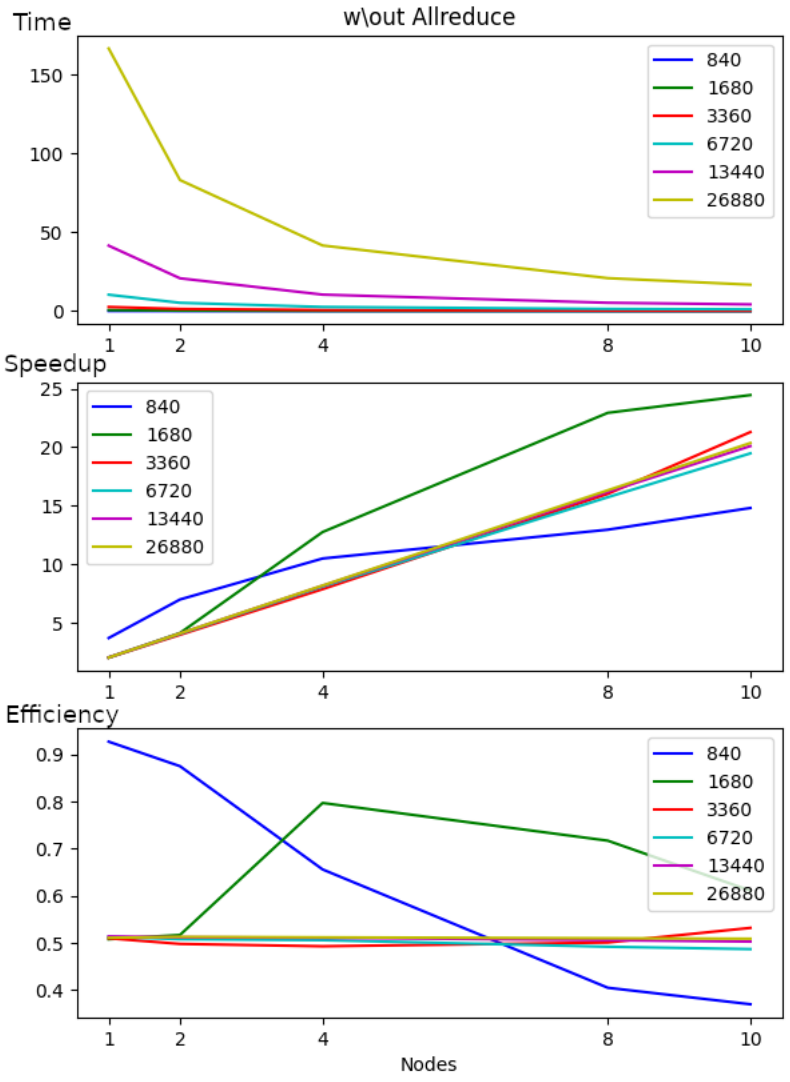


Table 24: MPI-Hybrid Wall Time without Convergence Check

Size/Nodes(Procs-Threads)	1(4-8)	2(8-16)	4(16-32)	8(32-64)	10(40-80)
840	0.09106s	0.04827s	0.03218s	0.02607s	0.02281s
1680	0.65775s	0.32351s	0.10488s	0.05834s	0.05472s
3360	2.61651s	1.33895s	0.67679s	0.33301s	0.2505s
6720	10.36924s	5.23313s	2.62602s	1.352s	1.09199s
13440	41.46707s	20.77977s	10.42776s	5.26134s	4.23861s
26880	166.46822s	83.05415s	41.57526s	20.86524s	16.72776s

Table 25: MPI-Hybrid Speedup without Convergence Check

Size/Nodes(Procs-Threads)	1(4-8)	2(8-16)	4(16-32)	8(32-64)	10(40-80)
840	3.71	7.0	10.499	12.96	14.812
1680	2.034	4.136	12.759	22.937	24.454
3360	2.039	3.984	7.882	16.019	21.295
6720	2.051	4.064	8.099	15.73	19.476
13440	2.055	4.1	8.17	16.193	20.1
26880	2.046	4.101	8.192	16.324	20.361

Table 26: MPI-Hybrid Efficiency without Convergence Check

Size/Nodes(Procs-Threads)	1(4-8)	2(8-16)	4(16-32)	8(32-64)	10(40-80)
840	0.927	0.875	0.656	0.405	0.37
1680	0.508	0.517	0.797	0.717	0.611
3360	0.51	0.498	0.493	0.501	0.532
6720	0.513	0.508	0.506	0.492	0.487
13440	0.514	0.512	0.511	0.506	0.503
26880	0.511	0.513	0.512	0.51	0.509

6.2.5 Notes

- Speedup and Efficiency rates are calculated in comparison to our **optimized Serial-Jacobi program**.
- We omitted the size 26880×26880 from the *Convergence-Check* graphs since it terminates in just 1 iteration, compared with 50 iterations in every other size, thus obtaining great speedup showcased in table 6.2.4. We attribute this super-fast convergence in the rounding errors which occur in the **MPI_SUM** reduction operation, caused by the extremely low residual errors we are dealing per process.
- We can see that for the largest inputs (3360×3360, 6720×6720, 13440×13440, 26880×26880) we achieve linear speedup as the number of nodes and the input size doubles, thus we could conclude that our problem is **Weakly Scalable**.
- We also notice that Efficiency is stable, around **0.5**, while the input size and the number of nodes doubles.

7. CUDA

The last part of the assignment is centered around CUDA threads to solve our problem.

7.1 Core Algorithm

The main algorithm parallelization method we use is based on the fact that GPUs can accommodate multiple threads in a cheap manner, so overhead caused by usual CPU issues will not occur. Each thread is assigned **only 1** element of the original matrix. The thread will acquire its needed neighboring elements and will calculate the new value for its respective element.

Every thread needs 4 additional neighboring elements, assigned to other threads, in order to calculate the new value. This would result in multiple access calls in the global GPU memory that are quite expensive. Most of these calls would also be unnecessary, since other threads would have probably already fetched their assigned element, which of course is needed by the neighboring threads, thus the neighboring threads need not fetch it as well from the global memory. To tackle this problem, we use the block-wide **shared**

memory as a cache for every element and their "halo" elements processed by the threads of each CUDA block. In this manner we will ensure that multiple access calls for a specific point (with number of max calls for a point to be exactly 5) will be made in respect to the shared block memory that is much faster than calls in the global GPU or the Unified CPU and GPU memory space.

The algorithm for both CPU and GPU has the following structure:

```
1. Set up arrays in global GPU memory
2. Set up appropriately BLOCK_SIZE, GRID_SIZE
3. while (iter < max_iter) {
   3.1 call kernel with appropriate args
   3.2 acquire result and estimate residual error (check next section)
}
4. Estimate iterative error for the solution
```

Listing 1: CPU Basic Algorithm

```
1. Set up respective shared memory elements in shared memory array
2. Wait for all threads in the same block
3. Calculate the assigned element of each thread based on the neighbors in
    the shared memory
4. Update original matrix in global GPU memory
5. Set error for calculation in the respective error_matrix array (check next
     section)
```

Listing 2: GPU Basic Kernel Algorithm

7.2 Error Reduction

As in the previous sections, we might want to check convergence of our algorithm to reduce the number of iterations and avoid pointless calculations that bind system resources. The algorithm for estimating the **residual** error in each one of the iterations is based on 3 simple steps:

1. Every thread calculates its local error, during the main algorithm, after estimating its assigned element.
2. The threads in each block cooperate to reduce their total error - thus each block ends up with a "block error".
3. We use a second kernel to reduce the summary of the "block errors" obtained collectively from each CUDA block.

As for the second kernel regarding error-reduction, each thread is assigned one element and a second one based on the *stride* set by the CPU. We continuously divide the error matrix in 2 sub-arrays and save the reduction results of each step in the elements of the first half. In the end we get the residual error in the very first element of the error matrix. By diminishing each thread's calculation to the very basic, we acquire very fast and parallel executions that will make up for multiple GPU kernel calls from the CPU.

7.3 Development using 2 GPUs

In this section we will explain the changes in data assignment that needed to be done to execute the parallel algorithm in 2 GPUs and acquire even better speed up.
The core algorithmic solution remains the same as in the code structure proposed for the 1 GPU setup. Thanks to the CUDA kernels' portability, all we need to do is just setup each of the two kernels with the necessary arguments to specify a data split, and the kernels will take care of the calculations and the error-reduction, as if 1 GPU was used.

The main difference is that in the 2 GPU setup, we take advantage of the extra memory provided by the second GPU and thus we use **Unified Memory** to store our input and auxiliary matrices. The usage of Unified Memory also provides an easy way of **communication** between the 2 GPUs, since "Halo"-point exchange is required. We also optimize the Unified Memory by **pre-fetching** the data assigned in its GPU, thus reducing drastically the **Page Fault** rate.

7.4 Benchmarks

Table 27: CUDA Time Benchmark

Array sz/Prog	840	1680	3360	6720	13440	26880
Serial	0.338 s	1.338 s	5.334 s	21.267 s	85.197 s	340.596 s
1 GPU	0.027 s	0.082 s	0.290 s	0.974 s	3.607 s	-
2 GPU	0.220 s	0.307 s	0.444 s	0.911 s	2.052 s	5.024 s

Table 28: CUDA Speedup Benchmark

Array sz/Prog	840	1680	3360	6720	13440	26880
Serial	1	1	1	1	1	1
1 GPU	**12.52**	**16.32**	**18.39**	21.83	23.62	-
2 GPU	1.54	4.36	12.01	**23.34**	**41.52**	**67.79**

We notice that by using only 1 GPU we obtain better speedup for the 3 lowest input matrix sizes (840×840, 1680×1680, 3360×3360), whereas 2 GPUs dominate in the speedup metric for the 3 greatest sizes (6720×6720, 13440×13440, 26880×26880). Thus, we conclude that the overhead introduced by the communication of the 2 GPUs plays an important role for small matrix sizes, but eventually its deficit is cancelled out by the overall gain in performance for the big matrix sizes.

8. Conclusions

To conclude our experimental research in these 3 types of parallel programming, we will reduce the case studies in the following points/conclusions.

Our optimizations in the **serialized** code with the extensive caching of common computations helped us achieve even lower runtimes when parallel methods were implemented, since every problem was already solved $\times 2.5$ faster than the original.

Our **pure MPI** implementation gave us huge performance improvements, most notably while running 64 processes (8 procs/node), where the speedup obtained was varied from $\times 10$ to $\times 40$ for every input size, with the biggest sizes scoring a very significant $\times 20$ increase in performance.

Our **Hybrid-MPI** approach was fun to develop and seemed to have a very good potential to achieve even lower runtimes than the pure MPI approach. However, our implementation didn't live up to our expectations since, despite obtaining a hugely respectable Speedup of around $\times 20$ for **every** input size ran in 10 Nodes with 40 total processes and 80 total threads, it didn't reach the sky-rocketed results of our pure MPI ran with 64 processes.

Moving on to **CUDA**, with just 2 GPUs at hand, we were able to mimic the performance of 49-64 MPI processes by obtaining roughly a $\times 18$ speedup for matrix sizes up to $6720 \times$

6720. What really took us by surprise was the $\times 41$ and $\times 67$ speedup obtained in the 13440×13440 and 26880×26880 sizes respectively. We should also note that adding the 2nd GPU helped us solve problems that were infeasible to be solved by 1 GPU, due to insufficient memory.

Thus we can safely conclude that CUDA and generally GPU programming is the future of Parallel Programming, especially in the Era of Big Data.

We can also conclude that MPI is very useful because of the customizability it provides to the programmer, in addition to achieving remarkable benchmarks. However, we could argue that for large scale problems with many modules and communication requirements, it might be quite hard to implement due to demanding features such as topologies, synchronization and inter-process communication which should be crafted very carefully.

Finally, MPI parallel programming paradigm, although being a quite familiar and comfortable programming prototype, scales fairly good in respect to timing, speed up and efficiency for small and moderately large input sizes.
However, poorly designed communication algorithms, will delay the overall execution because of intense process message exchanges, especially in large data - more procs scenarios. This event can be witnessed in the case study of challenge vs Pure MPI, especially in the Allreduce use case. We would notice that in almost every input the scaling of our program could not scale as good as the respective challenge program because of the extensive communications. Nevertheless convergence check applies to all sub-blocks and enhances performance when we use many processes, since the residual error is too small and the tasks diminish their execution time to the very minimum.

9. Appendix

9.1 global_run.sh script

Simple script to create input and PBS-submit files and auto-submit the task for us. Gives a specific and intuitive naming in each of the executables and output files, according to the input size and the procs used in each submission. CHECK RUN_INSTRUCTIONS before testing our code.

9.2 stats.py script

This is the script we used to extract LaTeXtables and graphs for our scaling study. To use it run $python3\ stats.py\ [mpi, hybrid, challenge]$
Note: MUST contain sequential run files with format $J\_seq\_1\_\{input\ size\}.o*$ and output files with name format $J\_\{program\ type\}\_\{procs\}\_\{input\ size\}.o*$. These formats are auto generated by our *global run* bash scripts that we put in each one of the files.
This script was adjusted for small issues between different output formats for better parsing. Better run with mpi records to display usage.

10. Contact

Team members (2)

Fullname: Charalampos Maraziaris / Χαράλαμπος Μαραζιάρης
ARGO username: argo203
NKUA ID: 1115201800105
email: sdi1800105@di.uoa.gr

Fullname: Vissarion Moutafis / Βησσαρίων Μουτάφης
ARGO username: argo210
NKUA ID: 1115201800119
email: sdi1800119@di.uoa.gr