

Parallel Systems Semester Project

1-2. Introduction, optimization of the sequential program and preparation for parallelism:

To improve the program, all the functions were integrated in the main.c function as suggested in the instructions. Also, the temporary variables were removed, except of course those needed to avoid recalculation in the operations of each Jacobi step (coefficients, f_x / f_y etc).

- Serial calculations time (seconds):

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
Old	0.837	3.337	13.336	53.336	213.313	853.351	Leak
New	0.824	3.281	13.118	52.465	209,893	843.384	Leak

3-4. Design of MPI parallelization / Optimization:

Steps MPI implementation:

1. Initialize problem Variables / Tables
2. MPI_Init
3. Calculate the process variables (rank, neighbors, size of block, coefficients calculation etc.)
4. Create data types (rows, columns):

```

MPI_Datatype column;
MPI_Type_vector(block_height, 1, block_length, MPI_DOUBLE, &column);
MPI_Type_commit(&column);

MPI_Datatype row;
MPI_Type_contiguous(block_length, MPI_DOUBLE, &row);
MPI_Type_commit(&row);

```

5. MPI_Barrier
6. Main for loop:
 - a. Irecv from 2 to 4 neighbors, as many as needed depending on the location of the block (more efficient than just sending messages to MPI_PROC_NULL).
 - b. Likewise Isend to 2 to 4 neighbors.
 - c. Calculate white values-error with for loop.
 - d. 2-4 MPI_wait for recv
 - e. Calculate green values as follows:
 - Calculate the error value of all greens apart from the for loop corners.
 - Calculate the corner error value.

This way, we read directly from the neighbors' buffers and do not write values one by one in the "u table" to use them later for calculations!
 - f. 2-4 MPI_wait for send.
 - g. Change tables (with pointers).
 - h. MPI_Allreduce
7. MPI_Finalizesending
8. Return

Optimizations:

- Attempt to receive other messages before sending so that processes are ready to accept halos.
- No unnecessary recalculations in the main for loop (they have been calculated before).
- As mentioned before, for green calculations we read directly from the buffers.
- I could have used the "reduce" command instead of "allreduce", since we only need to have the total error once, but I did Allreduce because it was requested in the project.

5. Results:

With Allreduce active :

- Time (seconds):

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	0.225	0.852	3.317	13.143	52.431	210.489	Leak
9	0.12	0.407	1.521	5.923	23.475	93.665	Leak
16	0.092	0.263	0.936	3.602	14.131	56.026	Leak
25	0.082	0.188	0.596	2.225	8.633	34.071	Leak
36	0.075	0.171	0.425	1.588	6.044	23.78	Leak
49	0.073	0.142	0.388	1.231	4.565	17.763	Leak
64	0.074	0.128	0.305	0.992	3.668	14.206	Leak

- Speedup:

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	3.66	3.85	3.95	3.99	4.00	4.01	Leak
9	6.87	8.06	8.62	8.86	8.94	9.00	Leak
16	8.96	12.48	14.01	14.57	14.85	15.05	Leak
25	10.05	17.45	22.01	23.58	24.31	24.75	Leak
36	10.99	19.19	30.87	33.04	34.73	35.47	Leak
49	11.29	23.11	33.81	42.62	45.98	47.48	Leak
64	11.14	25.63	43.01	52.89	57.22	59.37	Leak

- Efficiency:

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	0.92	0.96	0.99	1.00	1.00	1.00	Leak
9	0.76	0.90	0.96	0.98	0.99	1.00	Leak
16	0.56	0.78	0.88	0.91	0.93	0.94	Leak
25	0.40	0.70	0.88	0.94	0.97	0.99	Leak
36	0.31	0.53	0.86	0.92	0.96	0.99	Leak
49	0.23	0.47	0.69	0.87	0.94	0.97	Leak
64	0.17	0.40	0.67	0.83	0.89	0.93	Leak

With Allreduce inactive :

- Time (seconds):

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	0.071	0.282	1.324	5.159	18.742	81.693	Leak
9	0.048	0.131	0.711	2.737	10.762	42.889	Leak
16	0.045	0.083	0.444	1.782	6.948	27.681	Leak
25	0.043	0.074	0.344	1.562	6.085	24.494	Leak
36	0.051	0.069	0.292	1.314	5.116	20.386	Leak
49	0.05	0.073	0.195	1.082	4.29	16.973	Leak
64	0.058	0.078	0.135	0.894	3.538	13.944	Leak

- Speedup:

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	11.61	11.63	9.91	10.17	11.20	10.32	Leak
9	17.17	25.05	18.45	19.17	19.50	19.66	Leak
16	18.31	39.53	29.55	29.44	30.21	30.47	Leak
25	19.16	44.34	38.13	33.59	34.49	34.43	Leak
36	16.16	47.55	44.92	39.93	41.03	41.37	Leak
49	16.48	44.95	67.27	48.49	48.93	49.69	Leak
64	14.21	42.06	97.17	58.69	59.33	60.48	Leak

- Efficiency:

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	2.90	2.91	2.48	2.54	2.80	2.58	Leak
9	1.91	2.78	2.05	2.13	2.17	2.18	Leak
16	1.14	2.47	1.85	1.84	1.89	1.90	Leak
25	0.77	1.77	1.53	1.34	1.38	1.38	Leak
36	0.45	1.32	1.25	1.11	1.14	1.15	Leak
49	0.34	0.92	1.37	0.99	1.00	1.01	Leak
64	0.22	0.66	1.52	0.92	0.93	0.95	Leak

Challenge executable:

- Time (seconds):

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	0.253	0.97	3.766	14.93	59.59		Leak
9	0.152	0.54	2.065	8.103	32.216	128.576	Leak
16	0.121	0.391	1.485	5.776	22.908	91.25	Leak
25	0.107	0.338	1.23	4.744	18.737	74.621	Leak
36	0.102	0.314	1.092	4.223	16.56	65.789	Leak
49	0.113	0.294	1.016	3.898	15.288	60.703	Leak
64	0.105	0.29	0.961	3.742	14.56	57.964	Leak

- Speedup:

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	3.26	3.38	3.48	3.51	3.52		Leak
9	5.42	6.08	6.35	6.47	6.52	6.56	Leak
16	6.81	8.39	8.83	9.08	9.16	9.24	Leak
25	7.70	9.71	10.67	11.06	11.20	11.30	Leak
36	8.08	10.45	12.01	12.42	12.67	12.82	Leak
49	7.29	11.16	12.91	13.46	13.73	13.89	Leak
64	7.85	11.31	13.65	14.02	14.42	14.55	Leak

- Efficiency:

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
4	0.81	0.85	0.87	0.88	0.88		Leak

9	0.60	0.68	0.71	0.72	0.72	0.73	Leak
16	0.43	0.52	0.55	0.57	0.57	0.58	Leak
25	0.31	0.39	0.43	0.44	0.45	0.45	Leak
36	0.22	0.29	0.33	0.35	0.35	0.36	Leak
49	0.15	0.23	0.26	0.27	0.28	0.28	Leak
64	0.12	0.18	0.21	0.22	0.23	0.23	Leak

Overall, we have much better results than the challenge!

6. MPI + OpenMp Hybrid:

- Create threads in the for loop.
- Parallelization of for loops with omp reduce and static scheduling (arguably the best, because each point of the table needs about the same calculation time).
- Send / Recv / Wait only from master threads.
- The angles of the table and some other values are calculated using omp single, so that the same calculations are not repeated from all threads.
- Omp barrier before the "green calculations" and before the end of the basic for, for better synchronization of threads.

Run times:

Run times with MPI + OpenMp are worse than MPI. The reason is that although it returns good results, the threads are created inside the for loop, which costs a lot of time.

7. CUDA:

- Programming Process:

1. Create Tables / Initialize Variables.
2. Transfer them to the GPU memory, and execute it:
`__global__ void jacobi (...)`
3. The above code runs in 420X420 threads where each thread takes over a block of the table as in the mpi implementation.
4. The GPU transfers the jacobi panel to the host memory.

- Run time measurements:

For run time measurements 2 differentEvent_t are created before copying the tables. Then, using the differentEventrecord () and differentEventElapsedTime we find the GPU time immediately after running `__global__ void jacobi (...)`.

Run times for different sizes table (seconds):

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
420x420	0.019	0.063	0.484	3.078	16.83	85.127	Leak

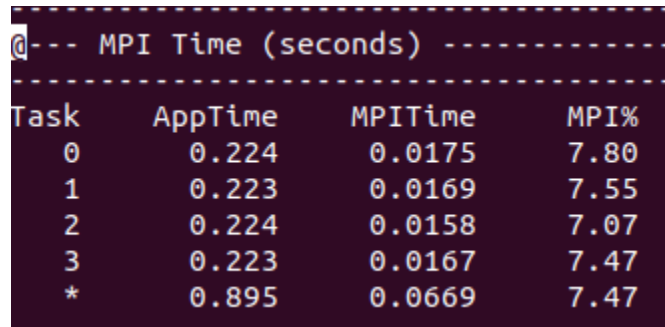
Speedup:

Μέγεθος Πλευράς	840x 840	1680x 1680	3360x 3360	6720x 6720	13440x 13440	26880x 26880	53760x 53760
420x420	43.37	52.08	27.10	17.05	12.47	9.91	Leak

- Note: I have also implemented code for the parallel finding of the error after the main for loop (it is commented out).

8. Conclusions:

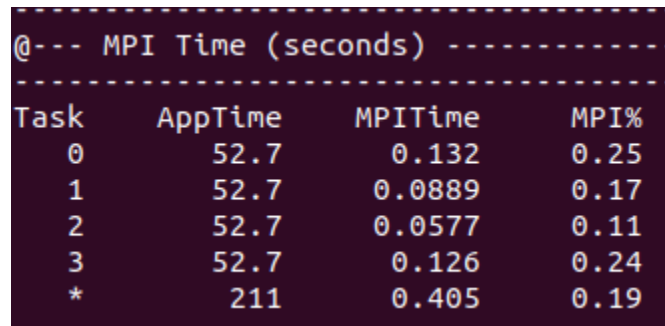
- The shortest execution time is given by CUDA, which is significantly faster than the rest.
- Profiling and MPITime in relation to the table size:
 - Run times with table size: 840X840:



A terminal screenshot showing MPI profiling data for a table size of 840X840. The output is a table with four columns: Task, AppTime, MPITime, and MPI%. The tasks are numbered 0, 1, 2, 3, and a summary row marked with an asterisk (*). The AppTime values are approximately 0.223-0.224 seconds, while MPITime values are very low, around 0.015-0.017 seconds. MPI% values range from 7.07 to 7.80.

Task	AppTime	MPITime	MPI%
0	0.224	0.0175	7.80
1	0.223	0.0169	7.55
2	0.224	0.0158	7.07
3	0.223	0.0167	7.47
*	0.895	0.0669	7.47

- Run times with table size: 26880X26880:



A terminal screenshot showing MPI profiling data for a table size of 26880X26880. The output is a table with four columns: Task, AppTime, MPITime, and MPI%. The tasks are numbered 0, 1, 2, 3, and a summary row marked with an asterisk (*). The AppTime values are significantly higher, around 52.7 seconds, while MPITime values are still low, around 0.08-0.13 seconds. MPI% values are very low, ranging from 0.11 to 0.25.

Task	AppTime	MPITime	MPI%
0	52.7	0.132	0.25
1	52.7	0.0889	0.17
2	52.7	0.0577	0.11
3	52.7	0.126	0.24
*	211	0.405	0.19

In short, the communication cost is relatively small for small tables, and very small for large (expected). The above screenshots are from profiling files in the mpi folder.

- Allreduce is quite demanding (of course, we could call it after the for loop is finished, since we do not need to calculate the error each time because we use for and not while loop with extra check(error> minerror).
- All implementations have generally good scaling, as:
 - Almost the entire code can be parallelized.
 - The number of messages does not increase as the size of the tables increases.