Parallel computing

Project 1

Diogo Rocha 202501406

Francesco Arboresi 202509510

-Summary of the algorithm: detail on the idea, data structure, auxiliary functions, what communication

-Performance evaluation: times in milliseconds, comparison with sequential execution (at least 1, 4, 9,16,25 processes)

-difficulties found during the work, and suggestions

Data structure

Firstly, the matrix structure has been defined, that is simply a re-defined unsigned long pointer and conversely it's MPI counter part is defined as an MPI_UNSIGNED_LONG, this approach avoids having to create a new MPI_DATATYPE.

A grid type structure is defined by a series of global variables (int grid_size; MPI_Comm comm; int coords[2]; MPI_Comm col_comm; MPI_Comm row_comm;) to setup a grid_sizexgride_size Cartesian Grid

Auxiliary functions

A series of functions are declared outside the main:

- proc_init - Function to read the input file, allocate memory for the blocks and matrices and init the data;
- data_init - Function to initialize the matrix with the data inside the input file
- proc_end - Free all allocated memory;
- setup_grid - Function for creating the two-dimensional grid communicator and communicators for each row and each column of the grid;
- print_matrix - Self-explanatory function to print the matrix;
- distribute_data - Distribution of data among the processes;
- matrix_scatter - Function for checkerboard matrix decomposition;
- fox_algorithmn - Function for parallel execution of the Fox method;
- gather_results - Join final results in Matrix C;
- adjust_values - Set INT_MAX values back to 0 in the result matrix;
- check_results - Function to compare the result matrix with an expected output file;
- save_matrix - Save a Matrix to an output file;

The Fox function defines the Fox algorithm which calculates our sum-min operations with the parallel approach.

The arguments of this function are 3 local matrixes A, B, C. Where local_A and local_B start as the submatrix assigned to the specific processor elected by the grid system, and the local_C matrix is the outcome of the min-plus operation between local_A and local_B.

It operates on submatrices (blocks) of the larger matrices A, B, and C, which are distributed among processes arranged in a grid. Each process is identified by its grid coordinates and participates in row and column communicators. The algorithm proceeds in several iterations, one for each step in the grid. In every iteration, each process row determines a “pivot” process whose A-block will be broadcast to all processes in that row. If a process is the pivot, it copies its local portion of A into a temporary buffer, which is then broadcast using MPI_Bcast. After the broadcast, all processes in that row possess the correct A-block for that iteration. Each process then performs local computation on its received A-block and local B-block.

After computing partial results and updating its local C-block, each process shifts its B-block upward within its column using MPI_Sendrecv_replace, ensuring that every process receives a new B-block for the next iteration. This combination of broadcasting A-blocks along rows, shifting B-blocks along columns, and performing local computation ensures that all necessary submatrix pairs interact over the course of the algorithm, ultimately producing the correct distributed result in C.

Main and communicators

The program begins with the initialization of the MPI environment using MPI_Init, followed by MPI_Comm_size and MPI_Comm_rank to obtain the total number of processes (num_procs) and each process’s unique rank (my_rank). The processes then collectively validate that the number of processes forms a perfect square — a requirement for constructing the two-dimensional process grid used in Fox’s algorithm. If an error is detected, the program calls MPI_Finalize (or MPI_Abort for hard failure), signaling all processes to terminate synchronously.

Before proceeding, the code calls MPI_Barrier(MPI_COMM_WORLD) twice — once after reading command-line arguments and once after grid and matrix initialization. This ensures that all processes have successfully set up their local data structures and communicators before the computation starts.

During computation, the main communication work occurs inside the do–while loop. The function distribute_data() is responsible for scattering portions of the global matrices A, B, and C to the local process blocks. This uses MPI_Scatter in the sub function matrix_scatter() to distribute submatrices from the root process (rank 0) to all others in a checkerboard pattern. Similarly, gather_results() is expected to collect the computed local results from all processes back to the root, using MPI_Gather.

Inside each iteration, the fox_algorithmn() function performs other communications: it uses row-wise broadcasts (MPI_Bcast) and column-wise cyclic shifts (MPI_Sendrecv_replace) within(row_comm and col_comm. These are not global operations; instead, they happen within subgroups of processes.The broadcast operation shares one process’s block of A with all others in the same row communicator, while the send-receive call simultaneously exchanges B-blocks up and down the column communicator.

After each iteration, the main loop includes the broadcast:

MPI_Bcast(&iteration, 1, MPI_INT, 0, MPI_COMM_WORLD);

Here, the root process (rank 0) sends the updated variable iteration to all processes. This is a global broadcast, ensuring that every process has the same control..

Before finalizing we add a bit of Quality of Life to the program by printing the final output and how much time it took to complete, doing an optional check with an already pre-made file with the solution to check if it achieved the desired result and finally saving our output to a file happily named ‘output’.

Finally, after all computation steps, MPI_Finalize() is called to cleanly terminate the MPI environment.

Performance evaluation

Due to lack of time when testing the implementation we didn’t have access to the Labs computers so we could only gather results up to 9 processes. Never the less, they are as follows using the input300:

Num Procs	Attempt 1	Attempt 2	Attempt 3	Average	Increase Speed
1	0.851832	0.830566	0.860754	0.847717	1x
4	0.194618	0.202543	0.206323	0.201161	~4,21x
9	0.167712	0.168189	0.168963	0.168288	~5.04x

Main difficulties encountered

- Coding in C.
- Proper examples with fox’s algorithm. A User’s guide to MPI abstracts the handling of matrices and only focus on the parelisation part. Other examples found are non working.
- Proper debugging, when testing we found larger matrices start producing wrong results at around index 11 to 13, we weren’t able to discover why in time.

Bibliography/ Resources

- Pacheco, P. S. (n.d.). A User's Guide to MPI. DCC. https://www3.dcc.fc.up.pt/~miguel-areias/teaching/2526/cp/material/mpi_guide.pdf
- Yosbi. (n.d.). All Pairs Shortest Path Implementation. Github. https://github.com/Yosbi/All-Pair-Shortest-Path
- Salgado, C. (n.d.). MPI All Pairs Shortest Path Implementation. Github. https://github.com/CaioCSdev/mpi_all_pairs-_shortest_paths-