Project Assignment I: All-Pairs Shortest Path Problem

Maroš Studenič, Weronika Trawińska

Building and running the program

`$ make`

(to clean) - removes builds components
`$ make clean`

to run
(mpi is not working on my machine idk why)

to run docker image that can run mpi
`$ sudo docker run -it -v $(pwd):/app orwel84/ubuntu-16-mpi bash`

to run program in docker
`mpirun -np 4 --allow-run-as-root /app/fox`

Structure - useful files
performance.sh - creates a performance test against all scenarios - uses sequential *floyd-warshal.c* and then performs tests on fox with 1,4,9,16 processes and on all test cases input6, input300, input600. input900, input1200

The output of this test is in *performance-result.txt*
Performance is transformed to table in *performance-table.txt*
And results are shown with *plot.py*, which turns the result into graphical comparison in the table.
And the graph is in *performance-table-pic.png.*

About the project

Process of project creation

1. We have created structure in which all source files are in src directory (components are in src/components) and all header files are in src/include directory
2. We have created / modified makefile that can build project and clean it
    - We used cookbook from https://makefiletutorial.com/ to create it (it can compile any project, that has source files in **src** directory)
    - We needed to change **$(CC)** to **mpicc** to use mpi compiler

About algorithm - fox algorithm

This report details the application of Fox's algorithm for solving the All-Pairs Shortest Path problem in a parallel computing context, highlighting its efficient use of matrix distribution and MPI communications to minimize computation time. Performance evaluations demonstrate substantial speedups across 1, 4, 9, and 16 processes compared to sequential execution, underscoring the algorithm's potential for scalable, high-performance computing solutions.

The All-Pairs Shortest Path (APSP) problem

The All-Pairs Shortest Path (APSP) problem involves finding the shortest paths between all pairs of nodes in a graph, a crucial challenge in fields like transportation and communication networks. Solving the APSP problem through parallel computing, specifically using Fox's algorithm, allows for leveraging multiple processors to accelerate the computation, making it feasible to handle large-scale graphs efficiently. Fox's algorithm optimizes this task by splitting the graph into smaller subgraphs, distributing them across processors, and minimizing inter-processor communication, thereby enhancing performance and scalability in distributed systems.

Algorithm

Base Idea

Fox's algorithm is designed to perform matrix multiplication in a parallel fashion by dividing the input matrices into blocks and distributing these blocks across a grid of processors. Its suitability for the min path problem lies in its ability to execute the min-plus matrix product, which is the operation at the heart of solving the APSP problem, with high efficiency in a parallel environment. By performing computation concurrently on different blocks of the matrices, it significantly reduces the overall execution time compared to serial approaches.

Auxiliary Functions

Key auxiliary functions include:

- **Matrix Splitting**: Divides the global matrix into smaller blocks that can be distributed among the processors.
- **Broadcasting Submatrices:** Within each row of the processor grid, a designated block is broadcasted to all other processors in the row.
- **Min-Plus Product Calculation:** A specialized function that, instead of performing standard multiplication, carries out the min-plus operation needed to determine the shortest paths.
- **Communications:** The algorithm employs two main patterns of communication:
    - **Row Broadcast:** At each step, one block of the matrix is broadcasted across a row of processors to align the necessary row of submatrices for the min-plus operation.
    - **Column Shift:** After each step, the B matrix blocks are shifted up one row in a circular manner, ensuring that each processor gets a new submatrix from the B matrix for the next step.

Algorithm Steps
The algorithm proceeds as follows:

1. Split the input matrices into submatrices.
2. At each iteration, select a pivot row and broadcast the corresponding submatrix across the processor row.
3. Perform the min-plus product on the current submatrices.
4. Shift the B submatrices up one row in a circular pattern.
5. Repeat the broadcast and min-plus product operations until all combinations of submatrices have been processed.
6. Gather the resulting submatrices to form the final matrix that represents the shortest paths between all pairs of nodes.
7. Repeat log_2 times of size of the input matrix because the path doubles the length for every iteration.

Why do we need to repeat it for log_2 times?

After 1. iteration we know the shortest path if we took 2 steps from each node -> in the initial we know the shortest path if we took 1 step.

After the second step we know the shortest path if we traveled by 4 edges in the graph.

After the third time we know the shortest path if we traveled by 8 edges...

After log_2th time, we know the shortest path if we traveled 2^log_2(size_of_matrix) > size_of_matrix. And we know that a shorter path is not possible because if we took all paths from one node to another we would took exactly size_of_matrix - 1 edges.

How did we implement the MPI broadcasting and shifting?

We used

```
MPI_Comm grid_comm;
int dims[2] = {m, m};
// we want periodic boundaries of columns
int periods[2] = {0, 1};
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &grid_comm);

// create row and column communicators
MPI_Comm row_comm;
MPI_Comm col_comm;
MPI_Comm_split(grid_comm, my_rank / m, my_rank % m, &row_comm);
MPI_Comm_split(grid_comm, my_rank % m, my_rank / m, &col_comm);

// save coords
int grid_coords[2];
MPI_Cart_coords(grid_comm, my_rank, DIM, grid_coords);
int my_row = grid_coords[0];
int my_col = grid_coords[1];
```

MPI_Cart_create with MPI_Comm_split - creates structure from a node that is like the table. (it can be more than 3 dimensional)

Challenges

We encountered a significant challenge in displaying the output on the console from the laboratory machine. However, when connecting through VSCode using SSH, the output was displayed correctly.

Later on, we noticed an error in the output when running tests with 9 cores; the issue was due to an incorrect operation in shifting row blocks upwards—instead, they were being shifted downwards. This error was also present in the tests with 4 processes, but it initially went unnoticed.

Performance

We developed a shell script to benchmark the sequential floyd-warshal.c against fox. The tests consistently showed that floyd-warshall outperformed fox, as evident in the graph titled 'performance-table-pic.png'.

This lag in performance is attributed to the overhead of message passing between processes, which incurs delays. Additionally, there is a synchronization cost since processes must wait for both the row shift and the broadcast to occur within the same row.