**Gauss elimination using MPI:**

* The program reads 10000X10000 matrix from a file “matrix.txt”. For this purpose, a “filetype” of type MPI\_Filetype is created, and the contents of the file are read into the memory of the processes using MPI\_File\_read\_all(…) function.
* The matrix of elements is distributed across Processes. The number of elements in each of the processes is .
* The processes are mapped to a cartesian topology using MPI\_Cart\_create(…). The neighbouring processes are determined using MPI\_Cart\_shift(..).
* A pipelined version of the gauss elimination algorithm is implemented to solve the matrix, where communications are overlapped with computations.
* The processes send their rows and columns corresponding to the current iteration to the right and bottom processes and receive rows and columns from the top and left processes as long as the sending process or the receiving process is not MPI\_PROC\_NULL.
* When a process receives a row or column from the neighbouring process, it forwards it to the next process. While communication is happening, the process reduces its local submatrix.
* Non-blocking MPI\_Isend(..) is used for sending to overlap communication with computation. Whereas blocking communication call (MPI\_Recv(..)) is used for the receive operation because the receiving process cannot perform computations unless it receives the data from the neighbouring process.
* The matrix is solved from the top left to the bottom right, overlapping computations and communications.
* Each active process performs computations and communicates elements approximately. The parallel time of the algorithm is .
* The upper and lower triangular matrices are stored in the same matrix.
* MPI\_File\_write\_all(…) is used to write the contents of the matrix to the file “LU.txt”.

Fig1. Schematic of parallel gauss elimination for a matrix distributed across 9 processes. The figure corresponds to the 1st iteration.

**Strong scaling and Amdahl’s law:**

* Amdahl’s law states that the serial fraction of the program limits the speedup (or the amount of parallelization).
* As the number of processes , the speedup becomes . So, the parallelization efficiency reduces with an increase in the number of processes and reaches the limit .
* For strong scaling, the size of the matrix is kept constant and the number of processes is increased in each run.
* The speedup is measured using the expression .
* Table 1 and Figure 2 shows that with an increase in the number of processes, the speed is asymptotically approaching ~16. So, the serial fraction of the code is ~6.25% (i.e., 1/16).

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Num\_procs | Time(s) | Speedup |
| 10000X10000 | 9 | 280.872 | 4.06678487 |
| 10000X10000 | 16 | 174.785 | 6.535148897 |
| 10000X10000 | 36 | 99.2814 | 11.50513591 |
| 10000X10000 | 64 | 79.722 | 14.32786433 |

Table 1. Strong scaling

Fig2. The plot for strong scaling

**Weak scaling and Gustafson’s law:**

* Gustafson’s law states that the parallel part scales linearly with the number of processes.
* The problem size is increased with the number of processes by keeping the size per process constant.
* Table 2 and Figure 3 shows that the parallel part of the program scales linearly with respect to the number of processes.
* As can be seen from table 2, the time is not constant with scaling, which shouldn’t be the case. This is because of the way the parallel gauss elimination is implemented. As the iteration progresses, the load on the processes is not balanced (ie., the initial processes remain idle).

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Num\_procs | Time | Scaled speedup |
| 3000X3000 | 9 | 7.5321 | 1.361333493 |
| 4000X4000 | 16 | 10.7844 | 1.690293387 |
| 6000X6000 | 36 | 20.0174 | 2.048957407 |
| 8000X8000 | 64 | 30.7893 | 2.368199342 |

Table 2. Weak scaling

Fig3. The plot for weak scaling