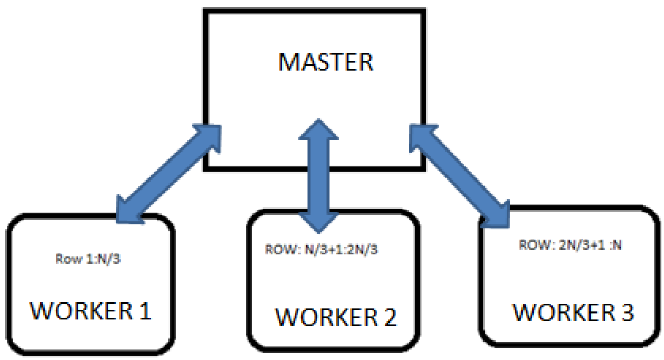
The characteristic of my Implementation of Gaussian Elimination Algorithm are:

- Parameterized over Data-Type :
  - Controlled by "define DATA_TYPE_SELECTED". Supported data types are:
    1. Float (DATA_TYPE_SELECTED=1)
    2. Double(DATA_TYPE_SELECTED=2)
- Parameterized over Matrix Size :
  - Controlled by "define MATRIX_SIZE". Since any system of linear equation of N variables need N equation to completely solve the problem so the matrix dimension is M[MATRIX_SIZE][MATRIX_SIZE+1] (" +1" for the right hand side of equation).
- I've tried **two communication pattern** to speed up my Gaussian Elimination Algorithm:
  - Master Synchronization dependent: In this communication pattern master process allocating the work to child process will synchronize the communication of all child process.
  - Algorithm as follow:
    - Master will not work itself. It just orchestrates the process and synchronizes the process.
    - Each Worker process will work on a portion of Constant Sub Portion of MATRIX.
    - No 2 Worker process will never communicate with one another.
    - Master will initialize the OUTPUT_MATRIX[0][0:MATRIX_SIZE+1]=MATRIX[0][0:MATRIX_SIZE+1] and COLUMN_INDEX=0; FACTOR_ROW=OUTPUT_MATRIX[COLUMN_INDEX][0:MATRIX_SIZE+1]
    - STEP1: Master will send a FACTOR_ROW and COLUMN_INDEX to Each Worker and Every Worker will use the scaled version of FACTOR_ROW to eliminate the COLUMN_INDEX i.e SUB_MATRIX [START_INDEX:MATRIZ_SIZE] [COLUMN_INDEX]=0. START_INDEX will depend on the COLUMN_INDEX. It's calculation depends on Gaussian

elimination algorithm itself. Explained below in the second algorithm.

- STEP2: Everyworker will send the complete signal.
- STEP3: Worker having the row=COLUMN_INDEX+1 will send the row=COLUMN_INDEX+1 back to MASTER.
- STEP4: MASTER will save the received row to OUTPUT_MATRIX [COLUMN+1]. This row will be the factor row for next iteration and so MASTER will send the new FACTOR_ROW and COLUMN_INDEX+1 to worker
- STEP 1:4 will be repeated until the COLUMN_INDEX reaches MATRIX_DIMENSION-1




Drawback:

- Lots of synchronization overheads.
- Master process does only synchronization task. So not full utilization of hardware resource.
- Lots of overhead

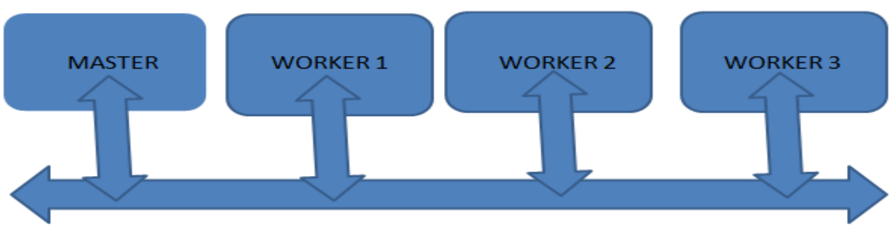
- Master Synchronization Independent: The above limitation was the source of this algorithm. In this algorithm also Master process sends the SUB_MATRIX to different process but it itself keeps one SUB_MATRIX to work. There is no Master process synchronization.
- Benefits:
  - Master process will also work.
  - Master process won't synchronize the work of worker process.
  - Every process will communicate with each other to get the task done.

- The process broadcasting the result will be responsible for synchronization.
- Algorithm as follow:
  - Master will allocate the SUB_MATRIX to every process including itself.
  - The allocation is such that Master process has SUB_MATRIX containing row 0 to MATRIX_SIZE/NUM_PROCESS. Similarly PROCESS_1 will have row MATRIX_SIZE/NUM_PROCESS to 2*MATRIX_SIZE/NUM_PROCESS.
  - NOTE IN ACTUAL CODE: If MATRIX_SIZE is not integer multiple of NUM_PROCESS then Balancing of row is done across every process. ROW_EXTRA=MATRIX_SIZE% NUM_PROCESS and every process whose PROCESS_INDEX< ROW_EXTRA will have extra row. Example MATRIX_SIZE 10 NUM_PROCESS 3(Including Master) then the allocation is as follow
    - Master – 4 rows
    - PROCESS_1- 3 rows
    - PROCESS_2- 3 rows
  - Every process will calculate the PROCESS_ROW_OFFSET and PROCESS_NUM_ROWS. Preceding Example
    - MASTER :
      - PROCESS_ROW_OFFSET = 0, PROCESS_NUM_ROWS = 4
    - PROCESS_1:
      - PROCESS_ROW_OFFSET=4
      - PROCESS_NUM_ROWS=3
    - PROCESS_2
      - PROCESS_ROW_OFFSET=7
      - PROCESS_NUM_ROWS=3
  - Every process will set the BROADCASTING_ROW=0.
  - STEP 1: Every PROCESS check the whether they have BROADCASTING_ROW but only the process having the BROADCASTING_ROW will broadcast the row while all other will be snooping over the broadcast.
  - STEP2: Every process whose BROADCASTING_ROW <= PROCESS_ROW_OFFSET+PROCESS_NUM_ROWS will use the

scaled version of RECEIVED_ROW to eliminate the SUB_MATRIX[START_INDEX:MATRIX_SIZE][BROADCASTING_ROW]=0. START_INDEX depends on the BROADCASTING_ROW

- IF BROADCASTING_ROW <PROCESS_ROW_OFFSET
  START_INDEX=0
  ELSE
  START_INDEX=BROADCASTING_ROW+1

- STEP3: Every process will increment the BROADCASTING_ROW.
  - IF BROADCASTING_ROW<MATRIX_SIZE-1:
    - LOOP to STEP 1.
  - ELSE:
    - EXIT




- Synchronization will be done by the process currently broadcasting.

**RESULTS:**

**BASELINE_ALGORITHM:**

**COMPILATION:**

gcc –O3 –DMATRIX_SIZE=<MATRIX_SIZE> -DDATA_TYPE_SELECTED=<FLOAT(1),DOUBLE(2)> basline_guassian_elimination_algorithm.c –o baseline_guassian_elimination_algorithm (OPTIONAL_FLAG –DDEBUG :to print the output of debug related information)

**RUN:**

./baseline_guassian_elimination_algorithm

## MPI IMPLEMENTATION:

**Compilation :**

mpicc –O3 –DMATRIX_SIZE=<MATRIX_SIZE> -DDATA_TYPE_SELECTED=<FLOAT(1),DOUBLE(2)> guassian_elimination_algorithm_submission.c –o guassian_elimination_algorithm (OPTIONAL_FLAG _DPRINT_OUT: to print the input and output matrix –DDEBUG: to print all the debug related information )

**RUN:**

mpirun –np <NUM_PROCESS> ./guassian_elimination_algorithm

EXPERIMENTS DONE:

1. The variation of compiler optimization: (MPI Launching 32 process)

| | MATRIX_SIZE:4096 | | | |
|---|---|---|---|---|
| | FLOAT Time in Milliseconds | | DOUBLE Time in Milliseconds | |
| | BASELINE | MPI | BASELINE | MPI |
| Without O3 Optimization | 113011 | 9186 | 116038 | 9160 |
| With O3 Optimization | 10959 | 1304 | 30834 | 3600 |

2. The variation of matrix size : (MPI Launching 32 process)

| MATRIX_SIZE | FLOAT Time in Milliseconds | | DOUBLE Time in Milliseconds | |
|---|---|---|---|---|
| | BASELINE | MPI | BASELINE | MPI |
| 512 | 22 | 20 | 30 | 23 |
| 1024 | 123 | 75 | 383 | 102 |
| 2048 | 966 | 290 | 1906 | 428 |
| 4096 | 10189 | 1304 | 30051 | 3554 |

3. The variation with respect to num process launched:

| NUM_PROCESS | MATRIX_SIZE | |
|---|---|---|
| | FLOAT | DOUBLE |
| | Time in Milliseconds | |
| 1 | 11793 | 32807 |
| 2 | 8576 | 24772 |
| 4 | 4130 | 12444 |
| 8 | 2575 | 7203 |

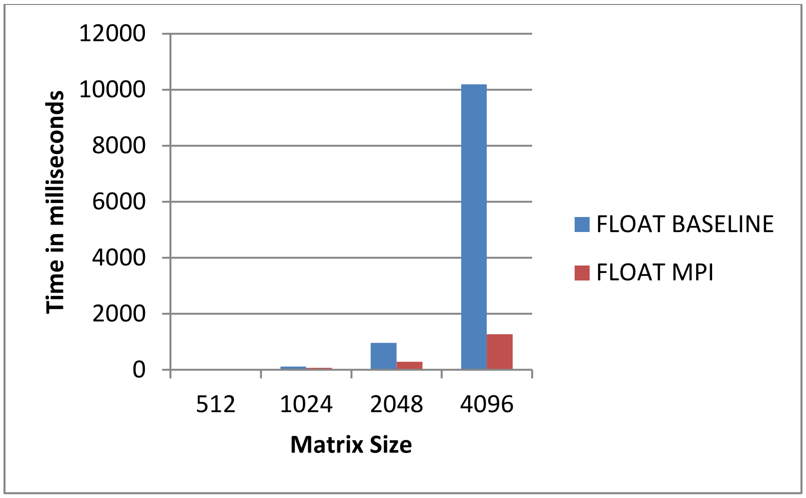
| | | |
|---|---|---|
| 12 | 2113 | 5237 |
| 16 | 1755 | 4572 |
| 24 | 1463 | 3995 |
| 32 | 1304 | 3580 |
| 64 | 1791 | 4389 |
| 88 | 1952 | 7602 |
| 128 | 1933 | 7607 |
| 150 | 2662 | 9651 |
| 200 | 2080 | 10645 |
| 245 | 1529 | 10358 |

COMMENT:

- There was some variation in the result observed in different run. In general when there is no work load working on the background of hydra2 then the variation is minimum and it's observed around maximum 1% But when there are workload running on background then the result fluctuate drastically and it depends on the load running.
- To perform the sanity check, I have done element wise comparison by developing custom float comparison function within a specified precision.To enable sanity check we can pass -DSANITY_CHECK as compile time argument.
- I observed rounding error while running this algorithm and specially this is prominent for larger matrix size because the number of approximation is O(n) so for larger n the number of approximation error is more and it becomes more prominent.

ANALYSIS:

- Enabling the compiler optimization O3 almost increases the performance almost by a factor of 3.
- Increasing the number of process beyond 32 increases the communication overhead so execution time increases.
- The variation of Float Matrix size : As the size of the matrix increases the time required to completely run the algorithm increases exponentially in Baseline algorithm whereas for MPI algorithm it's much less. For FLOAT MATRIX of size 4096 the MPI algorithm is around 8 time faster.(32 MPI process launched parallel)




The variation of Double Matrix size: Similarly as the size of double matrix increases the time required by baseline algorithm increases almost exponentially. For Double Matrix of size 4096 MPI algorithm is around 9 times faster.