Report of Gaussian Elimination using MPI

● Algorithm Description & Correctness Argument

To realize the Gaussian Elimination using MPI, we need to divide the original Matrix into several chunks, which is known as ranks in MPI.

Thus, the first thing we need to do is distributing data into different ranks.

For rank 0, we don't need to use MPI_Send and MPI_Recv, since we can reach the original matrix and vector in rank 0's memory, for other ranks, we have to use MPI_Send and MPI_Recv to distribute the data in Matrix A to local Matrix local_A.

```
If(rank == 0){
	/*set local A and local B for processor 0*/
	for(i = 0; i < chunkSize; i++)
		for(j = 0; j < MAXN; j++)
			local_A[i][j] = A[i][j];
	for(i = 0; i < chunkSize; i++)
		local_B[i] = B[i];

	/*distribute all the data to other processors*/
	for(i = chunkSize; i < MAXN; i++){
		MPI_Send(&A[i], MAXN, MPI_FLOAT, i/chunkSize, i%chunkSize, MPI_COMM_WORLD);
		MPI_Send(&B[i], 1, MPI_FLOAT, i/chunkSize, i%chunkSize, MPI_COMM_WORLD);
	}
}
else{
	/*other processors receive the data from processor 0, and store it in local_A and local_b*/
	for(i = 0; i < chunkSize; i++){
		MPI_Recv(&local_A[i], MAXN, MPI_FLOAT, 0, i, MPI_COMM_WORLD, &status);
		MPI_Recv(&local_B[i], 1, MPI_FLOAT, 0, i, MPI_COMM_WORLD, &status);
	}
}
```

After distributing data, we are going to do the Gaussian Elimination, I separate the serial code into two parts, computing the multipliers for each row and doing Gaussian Elimination using the multipliers.

Firstly, computing the multipliers: in the sequence code of Gauss, multiplier is changing according to row and column. However, in MPI, since the Matrix is divided into several ranks, we cannot get multiplier sequentially. I use a 2-D array buffer_A to store all the multipliers generated by different ranks, and use MPI_Bcast to broadcast it to all the ranks.

Secondly, doing the elimination: for each rank, the steps to eliminate is just as the sequence code, what's different is that we have to change original row's and column's number to new ones so as to fit the rank.

```
/*Gaussian elimination without pivoting*/
    for(i = 0; i < chunkSize; i++){
        /*the original row and col in Matrix A*/
        col = generalChunkSize*rank + i;
        row = col;
        /*for different rank, calculate its multiplier*/
        for(k = col; k < MAXN; k++)
            buffer_A[row][k] = local_A[i][k]/local_A[i][col];
        buffer_B[row] = local_B[i]/local_A[i][col];
        /*boardcast multiplier*/
        MPI_Bcast(&buffer_A[row], MAXN , MPI_FLOAT, rank, MPI_COMM_WORLD);
        MPI_Bcast(&buffer_B[row], 1, MPI_FLOAT, rank, MPI_COMM_WORLD);
    }

    /*for each chunk, doing gaussian elimination*/
    for(i = 0; i < chunkSize; i++){
        row = rank*generalChunkSize + i;
        /*j represents the row number of buffer,
        for rank n, it just need to count the buffer smaller than
        its row number*/
        for(j = 0; j < row; j++){
            for(k = j; k < MAXN; k++){
                local_A[i][k] -= local_A[i][j]*buffer_A[j][k];
            }
            local_B[i] -= local_B[i]*buffer_B[j];
        }
    }
```

To do the back substitution without MPI, we'd first gather all the local_A and local_B in different ranks into rank 0. I use MPI_Send and MPI_Recv, that's similar to the distribution of data.

Then, we have all the data in the rank 0, and we can do the back substitution easily.

```
/*send local_A and local_B from different ranks into rank 0*/
if(rank == 0){
	/*set A and B in processor 0*/
	for(i = 0; i < chunkSize; i++)
		for(j = 0; j < MAXN; j++)
			A[i][j] = local_A[i][j];
	for(i = 0; i < chunkSize; i++)
		B[i] = local_B[i];

	/*receive all the data from other processors*/
	for(i = chunkSize; i < MAXN; i++){
		MPI_Recv(&A[i], MAXN, MPI_FLOAT, i/chunkSize, i%chunkSize, MPI_COMM_WORLD,
&status);
		MPI_Recv(&B[i], 1, MPI_FLOAT, i/chunkSize, i%chunkSize, MPI_COMM_WORLD,
&status);
	}
}
else{
	/*other processors send the data to processor 0, and store it in A and B*/
	for(i = 0; i < chunkSize; i++){
		MPI_Send(&local_A[i], MAXN, MPI_FLOAT, 0, i, MPI_COMM_WORLD);
		MPI_Send(&local_B[i], 1, MPI_FLOAT, 0, i, MPI_COMM_WORLD);
	}
}
MPI_Barrier(MPI_COMM_WORLD);
/* Back substitution */
if(rank == 0){
	for (i = MAXN - 1; i >= 0; i--) {
	    X[i] = B[i];
		for (j = MAXN - 1; j > i; j--) {
			X[i] -= A[i][j] * X[j];
		}
		X[i] /= A[i][i];
	}
}
```

- **Performance & Performance Analyze**

I've made several screen shot to demonstrate the performance, and they are shown blow.

Number of processor = 1

```
[hwang122@compute-0-12 assign4]$ mpiexec -machinefile mymachines -np 1 ./gauss_m
pi
Gaussian Elimination using MPI
Matrix dimension = 1000

Initializing...
Initializing finished...
Initialize the Matrix and Vector takes 0.056016 s.
Distributing data to each processor takes 0.012819 s.
Gaussian elimination takes 6.913654 s.
Gather data from all the processors takes 0.008804 s.
Back substitution takes 0.009460 s.
Total Running time of this program is 7.000753 s.
```

Number of processor = 2

```
[hwang122@compute-0-12 assign4]$ mpiexec -machinefile mymachines -np 2 ./gauss_m
pi
Gaussian Elimination using MPI
Matrix dimension = 1000

Initializing...
Initializing finished...
Initialize the Matrix and Vector takes 0.056275 s.
Distributing data to each processor takes 0.027479 s.
Gaussian elimination takes 4.753864 s.
Gather data from all the processors takes 0.045052 s.
Back substitution takes 0.009326 s.
Total Running time of this program is 4.891996 s.
```

Number of processor = 4

```
[hwang122@compute-0-12 assign4]$ mpiexec -machinefile mymachines -np 4 ./gauss_m
pi
Gaussian Elimination using MPI
Matrix dimension = 1000

Initializing...
Initializing finished...
Initialize the Matrix and Vector takes 0.055906 s.
Distributing data to each processor takes 0.035088 s.
Gaussian elimination takes 2.561542 s.
Gather data from all the processors takes 0.038589 s.
Back substitution takes 0.008026 s.
Total Running time of this program is 2.699151 s.
```

Number of processor = 8

```
[hwang122@compute-0-12 assign4]$ mpiexec -machinefile mymachines -np 8 ./gauss_m
pi
Gaussian Elimination using MPI
Matrix dimension = 1000

Initializing...
Initializing finished...
Initialize the Matrix and Vector takes 0.056009 s.
Distributing data to each processor takes 0.038832 s.
Gaussian elimination takes 1.547515 s.
Gather data from all the processors takes 0.039479 s.
Back substitution takes 0.006442 s.
Total Running time of this program is 1.688277 s.
```

Number of processor = 12

```
[hwang122@compute-0-12 assign4]$ mpiexec -machinefile mymachines -np 12 ./gauss_
mpi
Gaussian Elimination using MPI
Matrix dimension = 1000

Initializing...
Initializing finished...
Initialize the Matrix and Vector takes 0.055697 s.
Distributing data to each processor takes 0.037498 s.
Gaussian elimination takes 0.865871 s.
Gather data from all the processors takes 0.026177 s.
Back substitution takes 0.006169 s.
Total Running time of this program is 0.991412 s.
```

As we can see above, the total time of serial fraction is certain, it is about 0.0561s(initialize matrix and vector). And since the number of ranks increases, the time of distributing data increases normally.

The running time of Gaussian Elimination is obviously decreased. As the back substitution need much less time than the other steps, its running time is not changed apparently.

Here is the running time form:

Number of processor	1	2	4	8	12
Total running time	7.000	4.892	2.699	1.688	0.991
Gaussian Elimination	6.914	4.754	2.562	1.548	0.866

Here is the speedup form:

Number of processor	2	4	8	12
Total running time's speedup	1.431	2.594	4.147	7.064
Gaussian Elimination's speedup	1.454	2.699	4.466	7.984